

Parallel Programming in MPI and OpenMP

The Art of HPC, volume 2

Victor Eijkhout

2nd edition 2022, formatted March 7, 2025

Book and slides download: <https://tinyurl.com/vle335course>

Public repository: https://github.com/VictorEijkhout/TheArtOfHPC_vol2_parallelprogramming

HTML version: <https://theartofhpc.com/pcse/>

This book is published under the CC-BY 4.0 license.

The term ‘parallel computing’ means different things depending on the application area. In this book we focus on parallel computing – and more specifically parallel *programming*; we will not discuss a lot of theory – in the context of scientific computing.

Two of the most common software systems for parallel programming in scientific computing are MPI and OpenMP. They target different types of parallelism, and use very different constructs. Thus, by covering both of them in one book we can offer a treatment of parallelism that spans a large range of possible applications.

Finally, we also discuss the PETSc (Portable Toolkit for Scientific Computing) library, which offers an abstraction level higher than MPI or OpenMP, geared specifically towards parallel linear algebra, and very specifically the sort of linear algebra computations arising from Partial Differential Equation modeling.

The main languages in scientific computing are C/C++ and Fortran. We will discuss both MPI and OpenMP with many examples in these two languages. For MPI and the PETSc library we will also discuss the Python interfaces.

Comments This book is in perpetual state of revision and refinement. Please send comments of any kind to eijkhout@tacc.utexas.edu.

Contents

I MPI	5
1 Getting started with MPI	7
1.1 <i>Distributed memory and message passing</i>	7
1.2 <i>History</i>	8
1.3 <i>Basic model</i>	8
1.4 <i>Making and running an MPI program</i>	9
1.5 <i>Language bindings</i>	10
1.6 <i>Review</i>	14
2 MPI topic: Functional parallelism	15
2.1 <i>The SPMD model</i>	15
2.2 <i>Starting and running MPI processes</i>	17
2.3 <i>Processor identification</i>	21
2.4 <i>Functional parallelism</i>	27
2.5 <i>Distributed computing and distributed data</i>	28
2.6 <i>Review questions</i>	29
3 MPI topic: Collectives	30
3.1 <i>Working with global information</i>	30
3.2 <i>Reduction</i>	33
3.3 <i>Rooted collectives: broadcast, reduce</i>	39
3.4 <i>Scan operations</i>	47
3.5 <i>Rooted collectives: gather and scatter</i>	50
3.6 <i>All-to-all</i>	55
3.7 <i>Reduce-scatter</i>	57
3.8 <i>Barrier</i>	62
3.9 <i>Variable-size-input collectives</i>	62
3.10 <i>MPI Operators</i>	68
3.11 <i>Nonblocking collectives</i>	72
3.12 <i>Performance of collectives</i>	78
3.13 <i>Collectives and synchronization</i>	79
3.14 <i>Performance considerations</i>	80
3.15 <i>Review questions</i>	84

4 MPI topic: Point-to-point	86
4.1 <i>Blocking point-to-point operations</i>	86
4.2 <i>Nonblocking point-to-point operations</i>	102
4.3 <i>The Status object and wildcards</i>	116
4.4 <i>More about point-to-point communication</i>	124
4.5 <i>Review questions</i>	126
5 MPI topic: Communication modes	130
5.1 <i>Persistent communication</i>	130
5.2 <i>Partitioned communication</i>	135
5.3 <i>Synchronous and asynchronous communication</i>	137
5.4 <i>Local and nonlocal operations</i>	138
5.5 <i>Buffered communication</i>	139
6 MPI topic: Data types	143
6.1 <i>The MPI_Datatype data type</i>	143
6.2 <i>Predefined data types</i>	144
6.3 <i>Derived datatypes</i>	152
6.4 <i>Big data types</i>	172
6.5 <i>Type maps and type matching</i>	176
6.6 <i>Type extent</i>	176
6.7 <i>Reconstructing types</i>	185
6.8 <i>Packing</i>	186
6.9 <i>Review questions</i>	190
7 MPI topic: Communicators	191
7.1 <i>Basic communicators</i>	191
7.2 <i>Duplicating communicators</i>	192
7.3 <i>Sub-communicators</i>	196
7.4 <i>Splitting a communicator</i>	198
7.5 <i>Communicators and groups</i>	202
7.6 <i>Intercommunicators</i>	204
7.7 <i>Review questions</i>	209
8 MPI topic: Process management	210
8.1 <i>Process spawning</i>	210
8.2 <i>Socket-style communications</i>	214
8.3 <i>Sessions</i>	218
8.4 <i>Functionality available outside init/finalize</i>	222
9 MPI topic: One-sided communication	223
9.1 <i>Windows</i>	224
9.2 <i>Active target synchronization: epochs</i>	228
9.3 <i>Put, get, accumulate</i>	232

9.4	<i>Passive target synchronization</i>	243
9.5	<i>More about window memory</i>	247
9.6	<i>Assertions</i>	251
9.7	<i>Implementation</i>	252
9.8	<i>Review questions</i>	253
10	MPI topic: File I/O	254
10.1	<i>File handling</i>	255
10.2	<i>File reading and writing</i>	256
10.3	<i>Consistency</i>	263
10.4	<i>Constants</i>	263
10.5	<i>Error handling</i>	263
10.6	<i>Review questions</i>	265
11	MPI topic: Topologies	266
11.1	<i>Cartesian grid topology</i>	266
11.2	<i>Distributed graph topology</i>	273
12	MPI topic: Shared memory	282
12.1	<i>Recognizing shared memory</i>	282
12.2	<i>Shared memory for windows</i>	283
13	MPI topic: Hybrid computing	288
13.1	<i>MPI support for threading</i>	288
14	MPI topic: Tools interface	291
14.1	<i>Initializing the tools interface</i>	291
14.2	<i>Control variables</i>	291
14.3	<i>Performance variables</i>	293
14.4	<i>Categories of variables</i>	295
14.5	<i>Events</i>	296
15	MPI leftover topics	297
15.1	<i>Contextual information, attributes, etc.</i>	297
15.2	<i>Error handling</i>	304
15.3	<i>Fortran issues</i>	307
15.4	<i>Progress</i>	308
15.5	<i>Fault tolerance</i>	309
15.6	<i>Performance, tools, and profiling</i>	309
15.7	<i>Determinism</i>	313
15.8	<i>Subtleties with processor synchronization</i>	314
15.9	<i>Shell interaction</i>	314
15.10	<i>Leftover topics</i>	316
15.11	<i>Literature</i>	318

16 MPI Examples	319
16.1 Bandwidth and halfbandwidth	319
 II OpenMP	 323
17 Getting started with OpenMP	325
17.1 The OpenMP model	325
17.2 Logistics of an OpenMP program run	328
17.3 Your first OpenMP program	329
17.4 Thread data	331
17.5 Creating parallelism	333
 18 OpenMP topic: Parallel regions	 335
18.1 Creating parallelism with parallel regions	335
18.2 Nested parallelism	337
18.3 Cancel parallel construct	339
18.4 Review questions	340
 19 OpenMP topic: Loop parallelism	 341
19.1 Loop parallelism through directives	341
19.2 An example	345
19.3 Loop schedules	349
19.4 Timing experiments	351
19.5 Reductions	353
19.6 Nested loops	354
19.7 Ordered iterations	356
19.8 nowait	357
19.9 While loops	357
19.10 Review questions	358
 20 OpenMP topic: Reductions	 359
20.1 Reductions: why, what, how?	359
20.2 Built-in reduction	363
20.3 Initial value for reductions	364
20.4 User-defined reductions	364
20.5 Scan / prefix operations	368
20.6 Reductions and floating-point math	369
20.7 Reductions in C++ standard algorithms	369
 21 OpenMP topic: Work sharing	 371
21.1 Work sharing constructs	371
21.2 Sections	371
21.3 Single thread execution	372

21.4 Fortran array syntax parallelization	373
22 OpenMP topic: Controlling thread data	375
22.1 Shared data	375
22.2 Private data	375
22.3 Data in dynamic scope	377
22.4 Temporary variables in a loop	378
22.5 Default	379
22.6 First and last private	379
22.7 Array data	380
22.8 Persistent data through threadprivate	382
22.9 Allocators	384
23 OpenMP topic: Synchronization	386
23.1 Barrier	386
23.2 Mutual exclusion	388
23.3 Locks	390
23.4 Relaxed memory model	393
23.5 Example: Fibonacci computation	393
24 OpenMP topic: Tasks	396
24.1 Task generation	396
24.2 Task data	397
24.3 Task synchronization	398
24.4 Task dependencies	399
24.5 Task reduction	401
24.6 More	401
24.7 Examples	402
25 OpenMP topic: Affinity	404
25.1 OpenMP thread affinity control	404
25.2 First-touch	407
25.3 Affinity control outside OpenMP	410
25.4 Tests	410
26 OpenMP topic: SIMD processing	416
26.1 SIMD loops	416
26.2 SIMD function calls	416
27 OpenMP topic: Offloading	420
27.1 Data on the device	420
27.2 Execution on the device	421
28 OpenMP remaining topics	422
28.1 Runtime functions, environment variables, internal control variables	422

28.2	<i>Timing</i>	424
28.3	<i>Thread safety</i>	424
28.4	<i>Performance and tuning</i>	425
28.5	<i>Accelerators</i>	425
28.6	<i>Tools interface</i>	426
28.7	<i>OpenMP standards</i>	426
28.8	<i>Memory model</i>	427
29	OpenMP Exercises and examples	429
29.1	<i>Histograms</i>	429
29.2	<i>N-body problems</i>	429
29.3	<i>Tree traversal</i>	434
29.4	<i>Depth-first search</i>	435
29.5	<i>Filtering array elements</i>	438
29.6	<i>Thread synchronization</i>	441
III	PETSc	443
30	PETSc basics	444
30.1	<i>What is PETSc and why?</i>	444
30.2	<i>Basics of running a PETSc program</i>	446
30.3	<i>PETSc installation</i>	450
30.4	<i>External packages</i>	451
31	PETSc objects	452
31.1	<i>Distributed objects</i>	452
31.2	<i>Scalars</i>	453
31.3	<i>Vec: Vectors</i>	454
31.4	<i>Mat: Matrices</i>	464
31.5	<i>Index sets and Vector Scatters</i>	475
31.6	<i>AO: Application Orderings</i>	477
31.7	<i>Partitionings</i>	477
32	Grid support	478
32.1	<i>Grid definition</i>	478
32.2	<i>Constructing a vector on a grid</i>	483
32.3	<i>Vectors of a distributed array</i>	484
32.4	<i>Matrices of a distributed array</i>	484
33	Finite Elements support	486
33.1	<i>General Data Management</i>	486
34	PETSc solvers	489
34.1	<i>KSP: linear system solvers</i>	489

34.2 <i>Direct solvers</i>	498
34.3 <i>Control through command line options</i>	499
35 PETSc nonlinear solvers	500
35.1 <i>Nonlinear systems</i>	500
35.2 <i>Time-stepping</i>	502
36 PETSc GPU support	503
36.1 <i>Installation with GPUs</i>	503
36.2 <i>Setup for GPU</i>	503
36.3 <i>Distributed objects</i>	503
36.4 <i>Other</i>	504
37 PETSc tools	505
37.1 <i>Error checking and debugging</i>	505
37.2 <i>Program output</i>	507
37.3 <i>Commandline options</i>	511
37.4 <i>Timing and profiling</i>	513
37.5 <i>Memory management</i>	513
38 PETSc topics	515
38.1 <i>Communicators</i>	515
IV Other programming models	517
39 Co-array Fortran	519
39.1 <i>History and design</i>	519
39.2 <i>Compiling and running</i>	519
39.3 <i>Basics</i>	519
40 Kokkos	522
40.1 <i>Compilation</i>	522
40.2 <i>Parallel code execution</i>	522
40.3 <i>Data</i>	524
40.4 <i>Execution and memory spaces</i>	525
40.5 <i>Configuration</i>	527
40.6 <i>Stuff</i>	527
41 Sycl, OneAPI, DPC++	528
41.1 <i>Logistics</i>	528
41.2 <i>Platforms and devices</i>	529
41.3 <i>Queues</i>	529
41.4 <i>Kernels</i>	530
41.5 <i>Parallel operations</i>	531

41.6 Memory access	535
41.7 Parallel output	538
41.8 Other	538
41.9 DPCPP extensions	538
41.10 Intel devcloud notes	538
41.11 Examples	539
42 CUDA	540
42.1 Host and device	540
42.2 Architecture	541
42.3 Querying	543
42.4 Performance	544
43 Python multiprocessing	545
43.1 Software and hardware	545
43.2 Process	545
43.3 Pools and mapping	546
43.4 Shared data	547
V The Rest	549
44 Exploring computer architecture	550
44.1 Tools for discovery	550
45 Hybrid computing	551
45.1 Concurrency	552
45.2 Affinity	552
45.3 What does the hardware look like?	553
45.4 Affinity control	553
45.5 Discussion	553
45.6 Processes and cores and affinity	554
45.7 Practical specification	557
46 Support libraries	559
46.1 SimGrid	559
46.2 Other	559
VI Class projects	561
47 A Style Guide to Project Submissions	562
47.1 General approach	562
47.2 Style	562
47.3 Structure of your writeup	562

47.4	<i>The parallel part</i>	563
47.5	<i>Remarks</i>	565
48	Warmup Exercises	566
48.1	<i>Hello world</i>	566
48.2	<i>Collectives</i>	566
48.3	<i>Linear arrays of processors</i>	567
49	Mandelbrot set	569
49.1	<i>MPI solutions</i>	570
49.2	<i>OpenMP solutions</i>	574
50	Data parallel grids	578
50.1	<i>Description of the problem</i>	578
50.2	<i>Code basics</i>	578
51	N-body problems	581
51.1	<i>Solution methods</i>	581
51.2	<i>Shared memory approaches</i>	581
51.3	<i>Distributed memory approaches</i>	581
VII	Didactics	583
52	Teaching guide	584
53	Teaching from mental models	585
53.1	<i>Introduction</i>	585
53.2	<i>Implied mental models</i>	586
53.3	<i>Teaching MPI, the usual way</i>	588
53.4	<i>Teaching MPI, our proposal</i>	589
53.5	<i>'Parallel computer games'</i>	594
53.6	<i>Further course summary</i>	595
53.7	<i>Prospect for an online course</i>	596
53.8	<i>Evaluation and discussion</i>	597
53.9	<i>Summary</i>	597
VIII	Bibliography, index, and list of acronyms	599
54	Bibliography	600
55	List of acronyms	602
56	General Index	604

57 Lists of notes	620
57.1 MPI-4 notes	620
57.2 Fortran notes	621
57.3 C++ notes	622
57.4 The MPL C++ interface	623
57.5 Python notes	625
58 Index of MPI commands and keywords	627
58.1 From the standard document	627
58.2 MPI for Python	632
59 Index of OpenMP keywords	633
60 Index of PETSc keywords	634
61 Index of KOKKOS keywords	635
62 Index of SYCL keywords	636

PART I

MPI

This section of the book teaches MPI ('Message Passing Interface'), the dominant model for distributed memory programming in science and engineering. It will instill the following competencies.

Basic level:

- The student will understand the SPMD model and how it is realized in MPI (chapter 2).
- The student will know the basic collective calls, both with and without a root process, and can use them in applications (chapter 3).
- The student knows the basic blocking and non-blocking point-to-point calls, and how to use them (chapter 4).

Intermediate level:

- The student knows about derived datatypes and can use them in communication routines (chapter 6).
- The student knows about intra-communicators, and some basic calls for creating subcommunicators (chapter 7); also Cartesian process topologies (section 11.1).
- The student understands the basic design of MPI I/O calls and can use them in basic applications (chapter 10).
- The student understands about graph process topologies and neighborhood collectives (section 11.2).

Advanced level:

- The student understands one-sided communication routines, including window creation routines, and synchronization mechanisms (chapter 9).
- The student understands MPI shared memory, its advantages, and how it is based on windows (chapter 12).
- The student understands MPI process spawning mechanisms and inter-communicators (chapter 8).

Chapter 1

Getting started with MPI

In this chapter you will learn the use of the main tool for distributed memory programming: the Message Passing Interface (MPI) library. The MPI library has about 250 routines, many of which you may never need. Since this is a textbook, not a reference manual, we will focus on the important concepts and give the important routines for each concept. What you learn here should be enough for most common purposes. You are advised to keep a reference document handy, in case there is a specialized routine, or to look up subtleties about the routines you use.

1.1 Distributed memory and message passing

In its simplest form, a distributed memory machine is a collection of single computers hooked up with network cables. In fact, this has a name: a *Beowulf cluster*. As you recognize from that setup, each processor can run an independent program, and has its own memory without direct access to other processors' memory. MPI is the magic that makes multiple instantiations of the same executable run so that they know about each other and can exchange data through the network.

One of the reasons that MPI is so successful as a tool for high performance on clusters is that it is very explicit: the programmer controls many details of the data motion between the processors. Consequently, a capable programmer can write very efficient code with MPI. Unfortunately, that programmer will have to spell things out in considerable detail. For this reason, people sometimes call MPI ‘the assembly language of parallel programming’. If that sounds scary, be assured that things are not that bad. You can get started fairly quickly with MPI, using just the basics, and coming to the more sophisticated tools only when necessary.

Another reason that MPI was a big hit with programmers is that it does not ask you to learn a new language: it is a library that can be interfaced to C/C++ or Fortran; there are even bindings to Python and Java (not described in this course). This does not mean, however, that it is simple to ‘add parallelism’ to an existing sequential program. An MPI version of a serial program takes considerable rewriting; certainly more than shared memory parallelism through OpenMP, discussed later in this book.

MPI is also easy to install: there are free implementations that you can download and install on any computer that has a Unix-like operating system, even if that is not a parallel machine. However, if you are working on a supercomputer cluster, likely there will already be an MPI installation, tuned to that machine’s network.

1.2 History

Before the MPI standard was developed in 1993-4, there were many libraries for distributed memory computing, often proprietary to a vendor platform. MPI standardized the inter-process communication mechanisms. Other features, such as process management in *PVM*, or parallel I/O were omitted. Later versions of the standard have included many of these features.

Since MPI was designed by a large number of academic and commercial participants, it quickly became a standard. A few packages from the pre-MPI era, such as *Charmpp* [18], are still in use since they support mechanisms that do not exist in MPI.

1.3 Basic model

Here we sketch the two most common scenarios for using MPI. In the first, the user is working on an interactive machine, which has network access to a number of hosts, typically a network of workstations; see figure 1.1. The user types the command `mpiexec`¹ and supplies

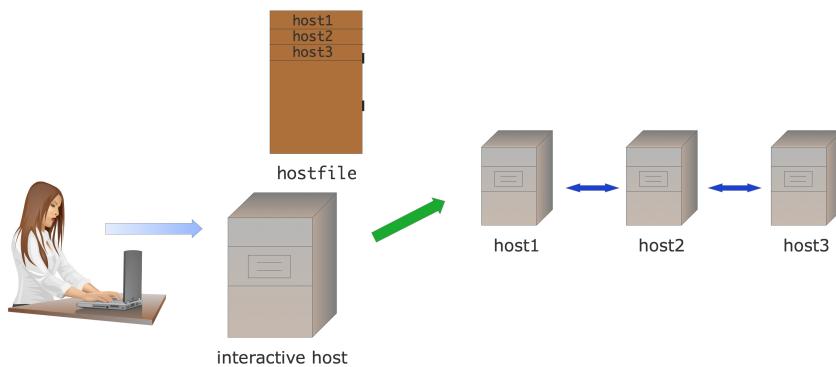


Figure 1.1: Interactive MPI setup

- The number of hosts involved,
- their names, possibly in a hostfile,
- and other parameters, such as whether to include the interactive host; followed by
- the name of the program and its parameters.

The `mpiexec` program then makes an ssh connection to each of the hosts, giving them sufficient information that they can find each other. All the output of the processors is piped through the `mpiexec` program, and appears on the interactive console.

In the second scenario (figure 1.2) the user prepares a *batch job* script with commands, and these will be run when the *batch scheduler* gives a number of hosts to the job. Now the batch script contains the `mpiexec` command, and the hostfile is dynamically generated when the job starts. Since the job now runs at a time when the user may not be logged in, any screen output goes into an output file.

1. A command variant is `mpirun`; your local cluster may have a different mechanism.

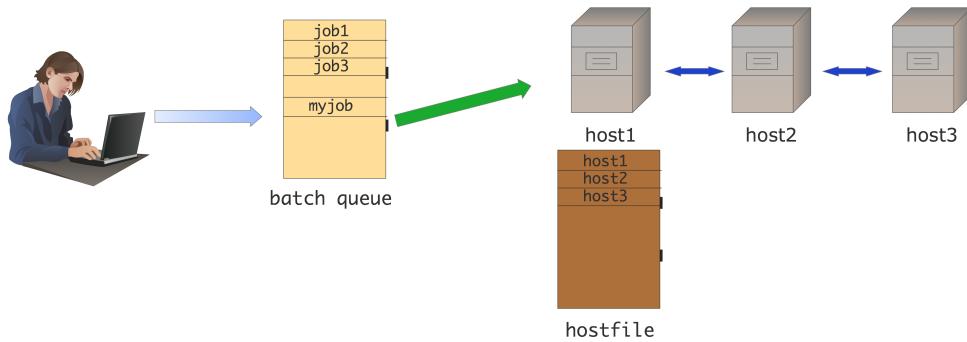


Figure 1.2: Batch MPI setup

You see that in both scenarios the parallel program is started by the `mpiexec` command using a Single Program Multiple Data (SPMD) mode of execution: all hosts execute the same program. It is possible for different hosts to execute different programs, but we will not consider that in this book.

There can be options and environment variables that are specific to some MPI installations, or to the network.

- `mpich` and its derivatives such as *Intel MPI* or *Cray MPI* have *mpiexec options*: <https://www.mpich.org/static/docs/v3.1/www1/mpiexec.html>

1.4 Making and running an MPI program

MPI is a library, called from programs in ordinary programming languages such as C/C++ or Fortran. To compile such a program you use your regular compiler:

```
gcc -c my_mpi_prog.c -I/path/to/mpi.h
gcc -o my_mpi_prog my_mpi_prog.o -L/path/to/mpi -lmpich
```

However, MPI libraries may have different names between different architectures, making it hard to have a portable makefile. Therefore, MPI typically has shell scripts around your compiler call, called `mpicc`, `mpicxx`, `mpif90` for C/C++/Fortran respectively.

```
mpicc -c my_mpi_prog.c
mpicc -o my_mpi_prog my_mpi_prog.o
```

If you want to know what `mpicc` does, there is usually an option that prints out its definition. On a Mac with the `clang` compiler:

```
## mpicc -show
clang -fPIC -fstack-protector -fno-stack-check -Qunused-arguments -g3 -O0 -Wno-implicit-functi
```

Remark In OpenMPI, these commands are binary executables by default, but you can make it a shell script by passing the `--enable-script-wrapper-compilers` option at configure time.

MPI programs can be run on many different architectures. Obviously it is your ambition (or at least your dream) to run your code on a cluster with a hundred thousand processors and a fast network. But maybe

you only have a small cluster with plain *ethernet*. Or maybe you're sitting in a plane, with just your laptop. An MPI program can be run in all these circumstances – within the limits of your available memory of course.

The way this works is that you do not start your executable directly, but you use a program, typically called *mpiexec* or something similar, which makes a connection to all available processors and starts a run of your executable there. So if you have a thousand nodes in your cluster, *mpiexec* can start your program once on each, and if you only have your laptop it can start a few instances there. In the latter case you will of course not get great performance, but at least you can test your code for correctness.

Python note 1: Running mpi4py programs. Load the TACC-provided python:

```
module load python  
and run it as:  
ibrun python-mpi yourprogram.py
```

1.5 Language bindings

1.5.1 C

The MPI library is written in C. However, the standard is careful to distinguish between MPI routines, versus their *C bindings*. In fact, as of MPI MPI-4, for a number of routines there are two bindings, depending on whether you want 4 byte integers, or larger. See section 6.4, in particular 6.4.1.

1.5.2 C++, including MPL

C++ bindings were defined in the standard at one point, but they were declared deprecated, and have been officially removed in the MPI-3 standard. Thus, MPI can be used from C++ by including

```
#include <mpi.h>
```

and using the C API.

The *boost* library has its own version of MPI, but it seems not to be under further development. A recent effort at idiomatic C++ support is *Message Passing Library (MPL)* <https://github.com/rabauke/mpl>. This book has an index of MPL notes and commands: section 57.4.

MPL note 1: Notes format. MPL is a C++ header-only library. Notes on MPI usage from MPL will be indicated like this.

1.5.3 Fortran

Fortran note 1: Formatting of Fortran notes. Fortran-specific notes will be indicated with a note like this.

Traditionally, *Fortran bindings* for MPI look very much like the C ones, except that each routine has a final *error return* parameter. You will find that a lot of MPI code in Fortran conforms to this.

However, in the *MPI 3* standard it is recommended that an MPI implementation providing a Fortran interface provide a module named *mpi_f08* that can be used in a Fortran program. This incorporates the following improvements:

- This defines MPI routines to have an optional final parameter for the error.
- There are some visible implications of using the `mpi_f08` module, mostly related to the fact that some of the ‘MPI datatypes’ such as `MPI_Comm`, which were declared as `Integer` previously, are now a Fortran Type. See the following sections for details: Communicator 7.1, Datatype 6.1, Info 15.1.1, Op 3.10.2, Request 4.2.1, Status 4.3, Window 9.1.
- The `mpi_f08` module solves a problem with previous *Fortran90 bindings*: Fortran90 is a strongly typed language, so it is not possible to pass argument by reference to their address, as C/C++ do with the `void*` type for send and receive buffers. This was solved by having separate routines for each datatype, and providing an Interface block in the MPI module. If you manage to request a version that does not exist, the compiler will display a message like
There is no matching specific subroutine for this generic subroutine call [MPI_Send]
For details see <http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node409.htm>.

1.5.4 Python

Python note 2: Python notes. Python-specific notes will be indicated with a note like this.

The `mpi4py` package [6, 5] of *python bindings* is not defined by the MPI standards committee. Instead, it is the work of an individual, *Lisandro Dalcin*.

In a way, the Python interface is the most elegant. It uses Object-Oriented (OO) techniques such as methods on objects, and many default arguments.

Notable about the Python bindings is that many communication routines exist in two variants:

- a version that can send arbitrary Python objects. These routines have lowercase names such as `bcast`; and
- a version that sends `numpy` objects; these routines have names such as `Bcast`. Their syntax can be slightly different.

The first version looks more ‘pythonic’, is easier to write, and can do things like sending python objects, but it is also decidedly less efficient since data is packed and unpacked with `pickle`. As a common sense guideline, use the `numpy` interface in the performance-critical parts of your code, and the pythonic interface only for complicated actions in a setup phase.

Codes with `mpi4py` can be interfaced to other languages through Swig or conversion routines.

Data in `numpy` can be specified as a simple object, or `[data, (count,displ), datatype]`.

1.5.5 How to read routine signatures

Throughout the MPI part of this book we will give the reference syntax of the routines. This typically comprises:

- The semantics: routine name and list of parameters and what they mean.
- C syntax: the routine definition as it appears in the `mpi.h` file.
- Fortran syntax: routine definition with parameters, giving in/out specification.
- Python syntax: routine name, indicating to what class it applies, and parameter, indicating which ones are optional.

These ‘routine signatures’ look like code but they are not! Here is how you translate them.

1. Getting started with MPI

1.5.5.1 C

The typically C routine specification in MPI looks like:

```
int MPI_Comm_size(MPI_Comm comm, int *nprocs)
```

This means that

- The routine returns an `int` parameter. Strictly speaking you should test against `MPI_SUCCESS` (for all error codes, see section 15.2.1):

```
MPI_Comm comm = MPI_COMM_WORLD;
int nprocs;
int errorcode;
errorcode = MPI_Comm_size( MPI_COMM_WORLD,&nprocs);
if (errorcode!=MPI_SUCCESS) {
    printf("Routine MPI_Comm_size failed! code=%d\n",
           errorcode);
    return 1;
}
```

However, the error codes are hardly ever useful, and there is not much your program can do to recover from an error. Most people call the routine as

```
MPI_Comm_size( /* parameter ... */ );
```

For more on error handling, see section 15.2.

- The first argument is of type `MPI_Comm`. This is not a C built-in datatype, but it behaves like one. There are many of these MPI_something datatypes in MPI. So you can write:

```
MPI_Comm my_comm =
    MPI_COMM_WORLD; // using a predefined value
MPI_Comm_size( comm, /* remaining parameters */ );
```

- Finally, there is a ‘star’ parameter. This means that the routine wants an address, rather than a value. You would typically write:

```
MPI_Comm my_comm = MPI_COMM_WORLD; // using a predefined value
int nprocs;
MPI_Comm_size( comm, &nprocs );
```

Seeing a ‘star’ parameter usually means either: the routine has an array argument, or: the routine internally sets the value of a variable. The latter is the case here.

1.5.5.2 Fortran

The Fortran specification looks like:

```
MPI_Comm_size(comm, size, ierror)
Type(MPI_Comm), Intent(In) :: comm
Integer, Intent(Out) :: size
Integer, Optional, Intent(Out) :: ierror
```

or for the Fortran90 legacy mode:

```
MPI_Comm_size(comm, size, ierror)
INTEGER, INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

The syntax of using this routine is close to this specification: you write

```
Type(MPI_Comm) :: comm = MPI_COMM_WORLD
! legacy: Integer :: comm = MPI_COMM_WORLD
Integer :: comm = MPI_COMM_WORLD
Integer :: size,ierr
CALL MPI_Comm_size( comm, size ) ! without the optional ierr
```

- Most Fortran routines have the same parameters as the corresponding C routine, except that they all have the error code as final parameter, instead of as a function result. As with C, you can ignore the value of that parameter. Just don't forget it.
- The types of the parameters are given in the specification.
- Where C routines have `MPI_Comm` and `MPI_Request` and such parameters, Fortran has `INTEGER` parameters, or sometimes arrays of integers.

1.5.5.3 Python

The Python interface to MPI uses classes and objects. Thus, a specification like:

```
MPI.Comm.Send(self, buf, int dest, int tag=0)
```

should be parsed as follows.

- First of all, you need the MPI class:

```
from mpi4py import MPI
```

- Next, you need a `Comm` object. Often you will use the predefined communicator

```
comm = MPI.COMM_WORLD
```

- The keyword `self` indicates that the actual routine `Send` is a method of the `Comm` object, so you call:

```
comm.Send( .... )
```

- Parameters that are listed by themselves, such as `buf`, as positional. Parameters that are listed with a type, such as `int dest` are keyword parameters. Keyword parameters that have a value specified, such as `int tag=0` are optional, with the default value indicated. Thus, the typical call for this routine is:

```
comm.Send(sendbuf, dest=other)
```

specifying the send buffer as positional parameter, the destination as keyword parameter, and using the default value for the optional tag.

Some python routines are ‘class methods’, and their specification lacks the `self` keyword. For instance:

```
MPI.Request.Waitall(type cls, requests, statuses=None)
```

would be used as

```
MPI.Request.Waitall(requests)
```

1.6 Review

Review 1.1. What determines the parallelism of an MPI job?

1. The size of the cluster you run on.
2. The number of cores per cluster node.
3. The parameters of the MPI starter (`mpexec`, `ibrun`,...)

Review 1.2. T/F: the number of cores of your laptop is the limit of how many MPI processes you can start up.

Review 1.3. Do the following languages have an object-oriented interface to MPI? In what sense?

1. C
2. C++
3. Fortran2008
4. Python

Chapter 2

MPI topic: Functional parallelism

2.1 The SPMD model

MPI programs conform largely to the Single Program Multiple Data (SPMD) model, where each processor runs the same executable. This running executable we call a *process*.

When MPI was first written, 20 years ago, it was clear what a processor was: it was what was in a computer on someone's desk, or in a rack. If this computer was part of a networked cluster, you called it a *node*. So if you ran an MPI program, each node would have one MPI process; figure 2.1. You could of course run



Figure 2.1: Cluster structure as of the mid 1990s

more than one process, using the *time slicing* of the Operating System (OS), but that would give you no extra performance.

These days the situation is more complicated. You can still talk about a node in a cluster, but now a node can contain more than one processor chip (sometimes called a *socket*), and each processor chip probably has multiple *cores*. Figure 2.2 shows how you could explore this using a mix of MPI between the nodes, and a shared memory programming system on the nodes.

However, since each core can act like an independent processor, you can also have multiple MPI processes per node. To MPI, the cores look like the old completely separate processors. This is the 'pure MPI' model

2. MPI topic: Functional parallelism



Figure 2.2: Hybrid cluster structure

of figure 2.3, which we will use in most of this part of the book. (Hybrid computing will be discussed in chapter 45.)



Figure 2.3: MPI-only cluster structure

This is somewhat confusing: the old processors needed MPI programming, because they were physically separated. The cores on a modern processor, on the other hand, share the same memory, and even some caches. In its basic mode MPI seems to ignore all of this: each core receives an MPI process and the programmer writes the same send/receive call no matter where the other process is located. In fact, you can't immediately see whether two cores are on the same node or different nodes. Of course, on the implementation level MPI uses a different communication mechanism depending on whether cores are

on the same socket or on different nodes, so you don't have to worry about lack of efficiency.

Remark In some rare cases you may want to run in an Multiple Programm Multiple Data (MPMD) mode, rather than SPMD. This can be achieved either on the OS level (see section 15.9.4), using options of the `mpiexec` mechanism, or you can use MPI's built-in process management; chapter 8. Like I said, this concerns only rare cases.

2.2 Starting and running MPI processes

The SPMD model may be initially confusing. Even though there is only a single source, compiled into a single executable, the parallel run comprises a number of independently started MPI processes (see section 1.3 for the mechanism).

The following exercises are designed to give you an intuition for this one-source-many-processes setup. In the first exercise you will see that the mechanism for starting MPI programs starts up independent copies. There is nothing in the source that says 'and now you become parallel'.

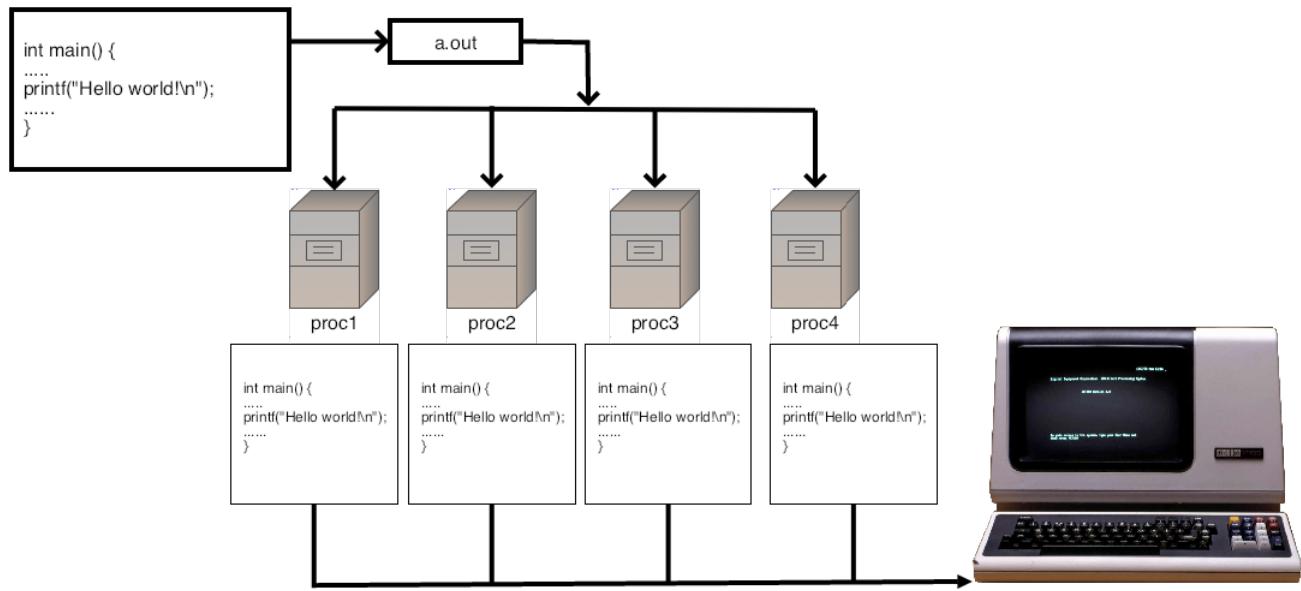


Figure 2.4: Running a hello world program in parallel

The following exercise demonstrates this point.

Exercise 2.1. Write a 'hello world' program, without any MPI in it, and run it in parallel with `mpiexec` or your local equivalent. Explain the output.
(There is a skeleton for this exercise under the name `hello`.)

This exercise is illustrated in figure 2.4.

2. MPI topic: Functional parallelism

Figure 2.1 MPI_Init

Name	Param name	Explanation	C type	F type	inout
MPI_Init	()				

2.2.1 Headers

If you use MPI commands in a program file, be sure to include the proper header file, `mpi.h` for C/C++.

```
#include "mpi.h" // for C
```

The internals of these files can be different between MPI installations, so you can not compile one file against one `mpi.h` file and another file, even with the same compiler on the same machine, against a different MPI.

Fortran note 2: MPI module. For MPI use from Fortran, use an MPI module.

```
use mpi      ! pre 3.0
use mpi_f08 ! 3.0 standard
```

New language developments, such as large counts; section 6.4.2 will only be included in the `mpi_f08` module, not in the earlier mechanisms.

The header file `mpif.h` is deprecated as of MPI-4.1: it may be supported by installations, but doing so is strongly discouraged.

Python note 3: Import mpi module. It's easiest to

```
from mpi4py import MPI
```

MPL note 2: Header file. To compile MPL programs, add a line

```
#include <mpl/mpl.hpp>
```

to your file. You need to add a path to your compile line:

```
mpicxx -o mpiprog -I${MPL_LOCATION}/include mympiprog.cpp
```

where `MPL_LOCATION` is system-dependent.

In CMake, MPL can be found:

```
find_package( mpl )
// this defines the target: mpl::mpl
```

2.2.2 Initialization / finalization

Every (useful) MPI program has to start with *MPI initialization* through a call to `MPI_Init` (figure 2.1), and have `MPI_Finalize` (figure 2.2) to finish the use of MPI in your program. The init call is different between the various languages.

In C, you can pass `argc` and `argv`, the arguments of a C language main program:

Figure 2.2 MPI_Finalize

Name	Param name	Explanation	C type	F type	inout
MPI_Finalize	()				

```
int main(int argc,char **argv) {
    ....
    return 0;
}
```

(It is allowed to pass `NULL` for these arguments.)

Fortran (before 2008) lacks this commandline argument handling, so `MPI_Init` lacks those arguments.

After `MPI_Finalize` no MPI routines (with a few exceptions such as `MPI_Finalized`) can be called. In particular, it is not allowed to call `MPI_Init` again. If you want to do that, use the *sessions model*; section 8.3.

Python note 4: Initialize/finalize. In many cases, no initialize and finalize calls are needed: the statement

```
## mpi.py
from mpi4py import MPI
```

performs the initialization. Likewise, the finalize happens at the end of the program.

However, for special cases, there is an `mpi4py.rc` object that can be set in between importing `mpi4py` and importing `mpi4py.MPI`:

```
import mpi4py
mpi4py.rc.initialize = False
mpi4py.rc.finalize = False
from mpi4py import MPI
MPI.Init()
# stuff
MPI.Finalize()
```

MPL note 3: Init, finalize. There is no initialization or finalize call.

Implementation note: Initialization is done at the first `mpl::environment` method call, such as `comm_world`.

This may look a bit like declaring ‘this is the parallel part of a program’, but that’s not true: again, the whole code is executed multiple times in parallel.

Exercise 2.2. Add the commands `MPI_Init` and `MPI_Finalize` to your code. Put three different print statements in your code: one before the init, one between init and finalize, and one after the finalize. Again explain the output.

Remark For hybrid MPI-plus-threads programming there is also a call `MPI_Init_thread`. For that, see section 13.1.

2. MPI topic: Functional parallelism

Figure 2.3 MPI_Abort

Name	Param name	Explanation	C type	F type	inout
MPI_Abort (

MPL:

```
void mpl::communicator::abort ( int ) const
```

Python:

```
MPI.Comm.Abort(self, int errorcode=0)
```

2.2.2.1 Aborting an MPI run

Apart from **MPI_Finalize**, which signals a successful conclusion of the MPI run, an abnormal end to a run can be forced by **MPI_Abort** (figure 2.3). This stop execution on all processes associated with the communicator, but many implementations will terminate all processes. The **value** parameter is returned to the environment.

Code:

```
// return.c
MPI_Abort(MPI_COMM_WORLD,17);
```

Output:

```
[code/mpi/c] return:
mpicc -o return return.o
mpirun -n 1 ./return ; \
    echo "MPI program return
    ↪code $?""
application called
↪MPI_Abort(MPI_COMM_WORLD,
↪17) - process 0
MPI program return code 17
```

2.2.2.2 Testing the initialized/finalized status

The commandline arguments **argc** and **argv** are only guaranteed to be passed to process zero, so the best way to pass commandline information is by a broadcast (section 3.3.3).

There are a few commands, such as **MPI_Get_processor_name**, that are allowed before **MPI_Init**.

If MPI is used in a library, MPI can have already been initialized in a main program. For this reason, one can test where **MPI_Init** has been called with **MPI_Initialized** (figure 2.4).

You can test whether **MPI_Finalize** has been called with **MPI_Finalized** (figure 2.5).

Figure 2.4 MPI_Initialized

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Initialized (</code>					

`flag` Flag is true if

`MPI_INIT` has been
called and false
otherwise

)

Figure 2.5 MPI_Finalized

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Finalized (</code>					

`flag` true if MPI was

finalized

)

2.2.2.3 Information about the run

Once MPI has been initialized, the `MPI_INFO_ENV` object contains a number of key/value pairs describing run-specific information; see section 15.1.1.1.

2.2.2.4 Commandline arguments

The `MPI_Init` routines takes a reference to `argc` and `argv` for the following reason: the `MPI_Init` calls filters out the arguments to `mpirun` or `mpiexec`, thereby lowering the value of `argc` and eliminating some of the `argv` arguments.

On the other hand, the commandline arguments that are meant for `mpiexec` wind up in the `MPI_INFO_ENV` object as a set of key/value pairs; see section 15.1.1.

2.3 Processor identification

Since all processes in an MPI job are instantiations of the same executable, you'd think that they all execute the exact same instructions, which would not be terribly useful. You will now learn how to distinguish processes from each other, so that together they can start doing useful work.

2.3.1 Processor name

In the following exercise you will print out the hostname of each MPI process with `MPI_Get_processor_name` (figure 2.6) as a first way of distinguishing between processes. This routine has a character buffer argument, which needs to be allocated by you. The length of the buffer is also passed, and on return that parameter has the actually used length. The maximum needed length is `MPI_MAX_PROCESSOR_NAME`.

2. MPI topic: Functional parallelism

Figure 2.6 MPI_Get_processor_name

Name	Param name	Explanation	C type	F type	inout
MPI_Get_processor_name		<pre> name A unique specifier for the actual (as opposed to virtual) node. resultlen Length (in printable characters) of the result returned in name) </pre>	char*	CHARACTER	OUT

Python:

`MPI.Get_processor_name()`

```
Code:  
  
// procname.c  
int name_length = MPI_MAX_PROCESSOR_NAME;  
char proc_name[name_length];  
MPI_Get_processor_name(proc_name,&name_length);  
printf("Process %d/%d is running on node <<%s>>\n",  
      procid,nprocs,proc_name);
```

(Underallocating the buffer will not lead to a runtime error.)

MPL note 4: Processor name. The processor name is a method of the environment class:

Code:

```
// procname.cxx
procno = comm_world.rank();
string procname =
    mpl::environment::processor_name();
stringstream ss;
ss << "[" << procno << "] "
    << " Running on: " << procname;
cout << ss.str() << '\n';
```

Output:

```
[examples/mpi/mp1] procname:
TACC: Starting up job 6051291
TACC: Starting parallel tasks...
[6] Running on:
    ↳c204-031.frontera.tacc.utexas.edu
[7] Running on:
    ↳c204-031.frontera.tacc.utexas.edu
[2] Running on:
    ↳c204-029.frontera.tacc.utexas.edu
[4] Running on:
    ↳c204-030.frontera.tacc.utexas.edu
[0] Running on:
    ↳c204-028.frontera.tacc.utexas.edu
[3] Running on:
    ↳c204-029.frontera.tacc.utexas.edu
[1] Running on:
    ↳c204-028.frontera.tacc.utexas.edu
[5] Running on:
    ↳c204-030.frontera.tacc.utexas.edu
TACC: Shutdown complete. Exiting.
```

Fortran note 3: Processor name. Allocate a **Character** variable with the appropriate length. The returned value of the length parameter can assist in printing the result:

Code:

```
!! procname.F90
Character(len=MPI_MAX_PROCESSOR_NAME)
  :: proc_name
Integer :: len
len = MPI_MAX_PROCESSOR_NAME
call MPI_Get_processor_name(proc_name
    ,len)
print *, "Proc", procid, "runs on ",
proc_name(1:len), ". "
```

Output:

```
[examples/mpi/f08] procname:
Proc           1 runs on
    ↳c202-010.frontera.tacc.utexas.edu.
Proc           3 runs on
    ↳c202-011.frontera.tacc.utexas.edu.
Proc           0 runs on
    ↳c202-010.frontera.tacc.utexas.edu.
Proc           2 runs on
    ↳c202-011.frontera.tacc.utexas.edu.
```

Exercise 2.3. Use the command **MPI_Get_processor_name**. Confirm that you are able to run a program that uses two different nodes.

2.3.2 Communicators

First we need to introduce the concept of *communicator*, which is an abstract description of a group of processes. For now you only need to know about the existence of the **MPI_Comm** data type, and that there is a pre-defined communicator **MPI_COMM_WORLD** which describes all the processes involved in your parallel run.

In the procedural languages C, a *communicator* is a *variable* that is passed to most routines:

2. MPI topic: Functional parallelism

```
#include <mpi.h>
MPI_Comm comm = MPI_COMM_WORLD;
MPI_Send( /* stuff */ comm );
```

Fortran note 4: Communicator type. In Fortran, pre-2008 a communicator was an *opaque handle*, stored in an `Integer`. With *Fortran 2008*, communicators are derived types:

```
use mpi_f08
Type(MPI_Comm) :: comm = MPI_COMM_WORLD
call MPI_Send( ... comm )
```

Python note 5: Communicator objects. In object-oriented languages, a communicator is an *object*, and rather than passing it to routines, the routines are often methods of the communicator object:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
comm.Send( buffer, target )
```

MPL note 5: World communicator. The naive way of declaring a communicator would be:

```
// commrank.cxx
mpl::communicator comm_world =
    mpl::environment::comm_world();
```

calling the predefined environment method `comm_world`.

However, if the variable will always correspond to the world communicator, it is better to make it `const` and declare it to be a reference:

```
const mpl::communicator &comm_world =
    mpl::environment::comm_world();
```

MPL note 6: Communicator copying. The communicator class has its copy operator deleted; however, copy initialization exists:

```
// commcompare.cxx
const mpl::communicator &comm =
    mpl::environment::comm_world();
cout << "same: " << boolalpha << (comm==comm) << endl;

mpl::communicator copy =
    mpl::environment::comm_world();
cout << "copy: " << boolalpha << (comm==copy) << endl;

mpl::communicator init = comm;
cout << "init: " << boolalpha << (init==comm) << endl;
```

(This outputs true/false/false respectively.)

Implementation note: The copy initializer performs an `MPI_Comm_dup`.

MPL note 7: Communicator passing. Pass communicators by reference to avoid communicator duplication:

Figure 2.7 MPI_Comm_size

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_size (

MPL:

```
int mpl::communicator::size () const
```

Python:

```
MPI.Comm.Get_size(self)
```

Figure 2.8 MPI_Comm_rank

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_rank (

MPL:

```
int mpl::communicator::rank () const
```

Python:

```
MPI.Comm.Get_rank(self)
```

```
// commpass.cxx
// BAD! this does a MPI_Comm_dup.
void comm_val( const mpl::communicator comm );

// correct!
void comm_ref( const mpl::communicator &comm );
```

You will learn much more about communicators in chapter 7.

2.3.3 Process and communicator properties: rank and size

To distinguish between processes in a communicator, MPI provides two calls

1. **MPI_Comm_size** (figure 2.7) reports how many processes there are in all; and
2. **MPI_Comm_rank** (figure 2.8) states what the number of the process is that calls this routine.

2. MPI topic: Functional parallelism

If every process executes the `MPI_Comm_size` call, they all get the same result, namely the total number of processes in your run. On the other hand, if every process executes `MPI_Comm_rank`, they all get a different result, namely their own unique number, an integer in the range from zero to the number of processes minus 1. See figure 2.5. In other words, each process can find out ‘I am process 5 out of a total of 20’.

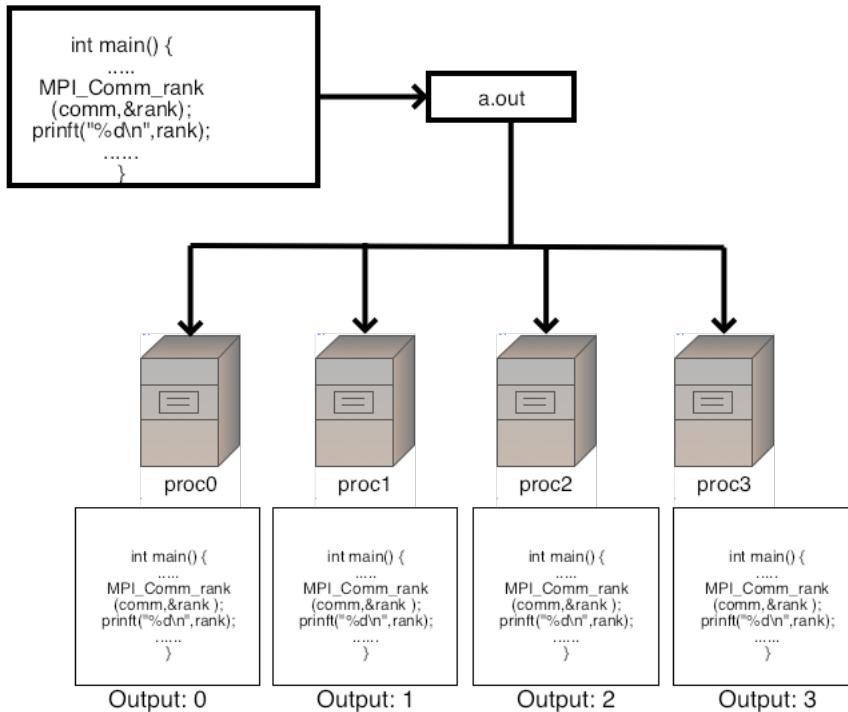


Figure 2.5: Parallel program that prints process rank

Exercise 2.4. Write a program where each process prints out a message reporting its number, and how many processes there are:

Hello from process 2 out of 5!

Write a second version of this program, where each process opens a unique file and writes to it. *On some clusters this may not be advisable if you have large numbers of processors, since it can overload the file system.*

(There is a skeleton for this exercise under the name `commrank`.)

Exercise 2.5. Write a program where only the process with number zero reports on how many processes there are in total.

In object-oriented approaches to MPI, that is, `mpi4py` and `MPL`, the `MPI_Comm_rank` and `MPI_Comm_size` routines are methods of the communicator class:

Python note 6: Communicator rank and size. Rank and size are methods of the communicator object. Note that their names are slightly different from the MPI standard names.

```

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()

```

MPL note 8: Rank and size. The rank of a process and the size of a communicator are both methods of the `communicator` class:

```
const mpl::communicator &comm_world =
    mpl::environment::comm_world();
int procid = comm_world.rank();
int nprocs = comm_world.size();
```

2.4 Functional parallelism

Now that processes can distinguish themselves from each other, they can decide to engage in different activities. In an extreme case you could have a code that looks like

```
// climate simulation:
if (procid==0)
    earth_model();
else if (procid==1)
    sea_model();
else
    air_model();
```

Practice is a little more complicated than this. But we will start exploring this notion of processes deciding on their activity based on their process number.

Being able to tell processes apart is already enough to write some applications, without knowing any other MPI. We will look at a simple parallel search algorithm: based on its rank, a processor can find its section of a search space. For instance, in *Monte Carlo codes* a large number of random samples is generated and some computation performed on each. (This particular example requires each MPI process to run an independent random number generator, which is not entirely trivial.)

Exercise 2.6. Is the number $N = 2,000,000,111$ prime? Let each process test a disjoint set

of integers, and print out any factor they find. You don't have to test all integers $< N$: any factor is at most $\sqrt{N} \approx 45,200$.

(Hint: `i%0` probably gives a runtime error.)

Can you find more than one solution?

(There is a skeleton for this exercise under the name `prime`.)

Remark Normally, we expect parallel algorithms to be faster than sequential. Now consider the above exercise. Suppose the number we are testing is divisible by some small prime number, but every process has a large block of numbers to test. In that case the sequential algorithm would have been faster than the parallel one. Food for thought.

As another example, in *Boolean satisfiability* problems a number of points in a search space needs to be evaluated. Knowing a process's rank is enough to let it generate its own portion of the search space. The computation of the *Mandelbrot set* can also be considered as a case of functional parallelism. However, the image that is constructed is data that needs to be kept on one processor, which breaks the symmetry of the parallel run.

Of course, at the end of a functionally parallel run you need to summarize the results, for instance printing out some total. The mechanisms for that you will learn next.

2.5 Distributed computing and distributed data

One reason for using MPI is that sometimes you need to work on a single object, say a vector or a matrix, with a data size larger than can fit in the memory of a single processor. With distributed memory, each processor then gets a part of the whole data structure and only works on that.

So let's say we have a large array, and we want to distribute the data over the processors. That means that, with p processes and n elements per processor, we have a total of $n \cdot p$ elements.

```
int n;
double data[n];
```

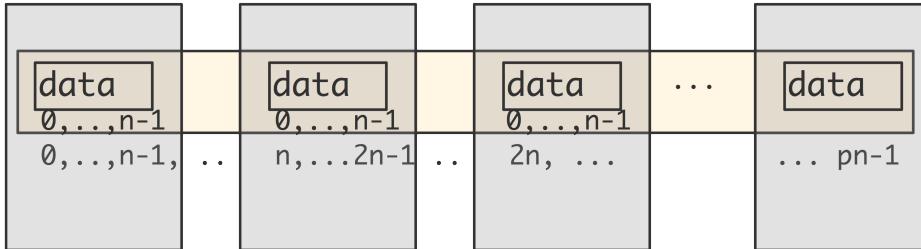


Figure 2.6: Local parts of a distributed array

In figure 2.6 we say that `data` is the local part of a *distributed array* with a total size of $n \cdot p$ elements. However, this array only exists conceptually: each processor has an array with lowest index zero, and you have to translate that yourself to an index in the global array. In other words, you have to write your code in such a way that it acts like you're working with a large array that is distributed over the processors, while actually manipulating only the local arrays on the processors.

Your typical code then looks like

```
int myfirst = ....;
for (int ilocal=0; ilocal<nlocal; ilocal++) {
    int iglobal = myfirst+ilocal;
    array[ilocal] = f(iglobal);
}
```

Exercise 2.7. Allocate on each process an array:

```
int my_ints[10];
```

and fill it so that process 0 has the integers 0 … 9, process 1 has 10 … 19, et cetera.

It may be hard to print the output in a non-messy way.

If the array size is not perfectly divisible by the number of processors, we have to come up with a division that is uneven, but not too much. You could for instance, write

```
int Nglobal, // is something large
Nlocal = Nglobal/ntids,
excess = Nglobal%ntids;
if (mytid==ntids-1)
    Nlocal += excess;
```

Exercise 2.8. Argue that this strategy is not optimal. Can you come up with a better distribution? Load balancing is further discussed in HPC book, section ??.

2.6 Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

Exercise 2.9. True or false: mpicc is a compiler.

Exercise 2.10. T/F?

1. In C, the result of `MPI_Comm_rank` is a number from zero to number-of-processes-minus-one, inclusive.
2. In Fortran, the result of `MPI_Comm_rank` is a number from one to number-of-processes, inclusive.

Exercise 2.11. What is the function of a hostfile?

Chapter 3

MPI topic: Collectives

A certain class of MPI routines are called ‘collective’, or more correctly: ‘collective on a communicator’. This means that if process one in that communicator calls that routine, they all need to call that routine. In this chapter we will discuss collective routines that are about combining the data on all processes in that communicator, but there are also operations such as opening a shared file that are collective, which will be discussed in a later chapter.

3.1 Working with global information

If all processes have individual data, for instance the result of a local computation, you may want to bring that information together, for instance to find the maximal computed value or the sum of all values. Conversely, sometimes one processor has information that needs to be shared with all. For this sort of operation, MPI has *collectives*.

There are various cases, illustrated in figure 3.1, which you can (sort of) motivate by considering some classroom activities:

- The teacher tells the class when the exam will be. This is a *broadcast*: the same item of information goes to everyone.
- After the exam, the teacher performs a *gather* operation to collect the individual exams.
- On the other hand, when the teacher computes the average grade, each student has an individual number, but these are now combined to compute a single number. This is a *reduction*.
- Now the teacher has a list of grades and gives each student their grade. This is a *scatter* operation, where one process has multiple data items, and gives a different one to all the other processes.

This story, while illustrative, is an imperfect analogy to what happens with MPI processes, because these are more symmetric. There is no ‘main’ or ‘master’ process: the process doing the reducing and broadcasting is no different from the others. Any process can function as the *root process* in a collective operation

Exercise 3.1. How would you realize the following scenarios with MPI collectives?

1. Let each process compute a random number. You want to print the maximum of these numbers to your screen.
2. Each process computes a random number again. Now you want to scale these numbers by their maximum.

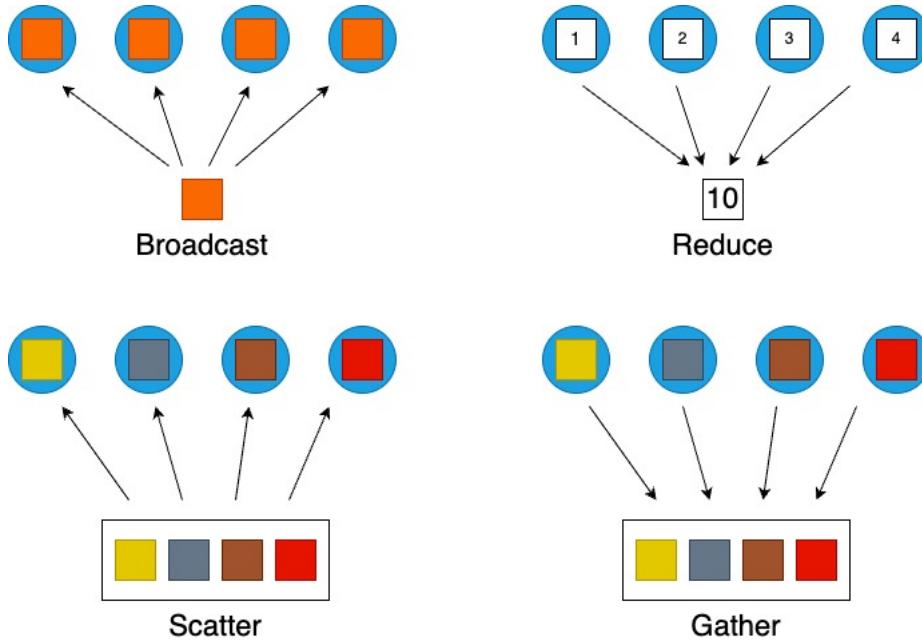


Figure 3.1: The four most common collective structures

3. Let each process compute a random number. You want to print on what processor the maximum value is computed.
Think about time and space complexity of your suggestions.

3.1.1 Practical use of collectives

Collectives are quite common in scientific applications. For instance, if one process reads data from disc or the commandline, it can use a broadcast or a gather to get the information to other processes. Likewise, at the end of a program run, a gather or reduction can be used to collect summary information about the program run.

However, a more common scenario is that the result of a collective is needed on all processes.

Consider the computation of the *standard deviation*:

$$\sigma = \sqrt{\frac{1}{N-1} \sum_i^N (x_i - \mu)^2} \quad \text{where} \quad \mu = \frac{\sum_i^N x_i}{N}$$

and assume that every process stores just one x_i value.

1. The calculation of the average μ is a reduction, since all the distributed values need to be added.
2. Now every process needs to compute $x_i - \mu$ for its value x_i , so μ is needed everywhere. You can compute this by doing a reduction followed by a broadcast, but it is better to use a so-called *allreduce* operation, which does the reduction and leaves the result on all processors.

3. The calculation of $\sum_i(x_i - \mu)$ is another sum of distributed data, so we need another reduction operation. Depending on whether each process needs to know σ , we can again use an allreduce.

3.1.2 Synchronization

Collectives are operations that involve all processes in a communicator. A collective is a single call, and it blocks on all processors, meaning that a process calling a collective cannot proceed until the other processes have similarly called the collective.

That does not mean that all processors exit the call at the same time: because of implementational details and network latency they need not be synchronized in their execution. However, semantically we can say that a process can not finish a collective until every other process has at least started the collective.

In addition to these collective operations, there are operations that are said to be ‘collective on their communicator’, but which do not involve data movement. Collective then means that all processors must call this routine; not to do so is an error that will manifest itself in ‘hanging’ code. One such example is [MPI_File_open](#).

3.1.3 Collectives in MPI

We will now explain the MPI collectives in the following order.

Allreduce We use the allreduce as an introduction to the concepts behind collectives; section 3.2. As explained above, this routines serves many practical scenarios.

Broadcast and reduce We then introduce the concept of a root in the reduce (section 3.3.1) and broadcast (section 3.3.3) collectives.

- Sometimes you want a reduction with partial results, where each processor computes the sum (or other operation) on the values of lower-numbered processors. For this, you use a *scan* collective (section 3.4).

Gather and scatter The gather/scatter collectives deal with more than a single data item; section 3.5.

There are more collectives or variants on the above.

- If every processor needs to broadcast to every other, you use an *all-to-all* operation (section 3.6).
- The reduce-scatter is a lesser known combination of collectives; section 3.7.
- A barrier is an operation that makes all processes wait until every process has reached the barrier (section 3.8).
- If you want to gather or scatter information, but the contribution of each processor is of a different size, there are ‘variable’ collectives; they have a v in the name (section 3.9).

Finally, there are some advanced topics in collectives.

- User-defined reduction operators; section 3.10.2.
- Nonblocking collectives; section 3.11.
- We briefly discuss performance aspects of collectives in section 3.12.
- We discuss synchronization aspects in section 3.13.

Figure 3.1 MPI_Allreduce

Name	Param name	Explanation	C type	F type	inout
MPI_Allreduce					
	MPI_Allreduce_c				
sendbuf		starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
recvbuf		starting address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
count		number of elements in send buffer	[int MPI_Count]	INTEGER	IN
datatype		datatype of elements of send buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
op		operation	MPI_Op	TYPE(MPI_Op)	IN
comm		communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)					

MPL:

```
template<typename T , typename F >
void mpl::communicator::allreduce
( F, const T &, T & ) const;
( F, const T *, T *,
  const contiguous_layout< T > & ) const;
( F, T & ) const;
( F, T *, const contiguous_layout< T > & ) const;
F : reduction function
T : type
```

Python:

```
MPI.Comm.Allreduce(self, sendbuf, recvbuf, Op op=SUM)
```

3.2 Reduction

3.2.1 Reduce to all

Above we saw a couple of scenarios where a quantity is reduced, with all processes getting the result. The MPI call for this is `MPI_Allreduce` (figure 3.1).

Example: we give each process a random number, and sum these numbers together. The result should be approximately 1/2 times the number of processes.

```
// allreduce.c
float myrandom,sumrandom;
myrandom = (float) rand()/(float)RAND_MAX;
// add the random variables together
MPI_Allreduce(&myrandom,&sumrandom,
             1,MPI_FLOAT,MPI_SUM,comm);
// the result should be approx nprocs/2:
if (procno==nprocs-1)
    printf("Result %.9f compared to .5\n",sumrandom/nprocs);
```

Or:

```
MPI_Count bufsize = 1000;
double *indata,*outdata;
indata = (double*) malloc( bufsize*sizeof(double) );
outdata = (double*) malloc( bufsize*sizeof(double) );
MPI_Allreduce_c(indata,outdata,bufsize,
                MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
```

3.2.1.1 Buffer description

This is the first example in this course that involves MPI data buffers: the `MPI_Allreduce` call contains two buffer arguments. In most MPI calls (with the one-sided ones as big exception) a buffer is described by three parameters:

1. a pointer to the data,
2. the number of items in the buffer, and
3. the datatype of the items in the buffer.

Each of these needs some elucidation.

1. The buffer specification depends on the programming languages. Defaults are in section 3.2.4.
2. The count was a 4-byte integer in MPI standard up to and including MPI-3. In the MPI-4 standard the `MPI_Count` data type become allowed. See section 6.4 for details.
3. Datatypes can be predefined, as in the above example, or user-defined. See chapter 6 for details.

Remark *Routines with both a send and receive buffer should not alias these. Instead, see the discussion of `MPI_IN_PLACE`; section 3.3.2.*

3.2.1.2 Examples and exercises

Exercise 3.2. Let each process compute a random number, and compute the sum of these numbers using the `MPI_Allreduce` routine.

$$\xi = \sum_i x_i$$

Each process then scales its value by this sum.

$$x'_i \leftarrow x_i / \xi$$

Compute the sum of the scaled numbers

$$\xi' = \sum_i x'_i$$

and check that it is 1.

(There is a skeleton for this exercise under the name `randommax`.)

Exercise 3.3. Extend the previous exercise to letting each process have an array.

Exercise 3.4. Implement a (very simple-minded) Fourier transform: if f is a function on the interval $[0, 1]$, then the n -th Fourier coefficient is

$$f_n \hat{=} \int_0^1 f(t) e^{-2\pi x} dx$$

which we approximate by

$$f_n \hat{=} \sum_{i=0}^{N-1} f(ih) e^{-in\pi/N}$$

- Make one distributed array for the e^{-inh} coefficients,
- make one distributed array for the $f(ih)$ values
- calculate a couple of coefficients

Exercise 3.5. In the previous exercise you worked with a distributed array, computing a local quantity and combining that into a global quantity. Why is it not a good idea to gather the whole distributed array on a single processor, and do all the computation locally?

MPL note 9: Allreduce operator. Versions with separate send/receive buffers, and reduction in place:

```
// separate recv buffer
comm_world.allreduce(mpl::plus<float>(), proc_data, reduce_data);
// in place
comm_world.allreduce(mpl::plus<float>(), proc_data);
```

Note the parentheses after the operator. Also note that the operator comes first, not last as in the C Application Programmer Interface (API).

Available: `max`, `min`, `plus`, `multiples`, `logical_and`, `logical_or`, `logical_xor`, `bit_and`, `bit_or`, `bit_xor`.

Implementation note: The reduction operator has to be compatible with $T(T, T)$.

For more about operators, see section 3.10.

3.2.2 Inner product as allreduce

One of the more common applications of the reduction operation is the *inner product* computation. Typically, you have two vectors x, y that have the same distribution, that is, where all processes store equal parts of x and y . The computation is then

```
local_inprod = 0;
for (i=0; i<localsize; i++)
    local_inprod += x[i]*y[i];
MPI_Allreduce( &local_inprod, &global_inprod, 1,MPI_DOUBLE ... )
```

Exercise 3.6. The *Gram-Schmidt* method is a simple way to orthogonalize two vectors:

$$u \leftarrow u - (u^t v) / (u^t u)$$

Implement this, and check that the result is indeed orthogonal.

Suggestion: fill v with the values $\sin 2nh\pi$ where $n = 2\pi/N$, and u with $\sin 2nh\pi + \sin 4nh\pi$. What does u become after orthogonalization?

3.2.3 Reduction operations

Several `MPI_Op` values are pre-defined. For the list, see section [3.10.1](#).

For use in reductions and scans it is possible to define your own operator.

```
MPI_Op_create( MPI_User_function *func, int commute, MPI_Op *op);
```

For more details, see section [3.10.2](#).

3.2.4 Data buffers

Collectives are the first example you see of MPI routines that involve transfer of user data. Here, and in every other case, you see that the data description involves:

- A buffer. This can be a scalar or an array.
- A datatype. This describes whether the buffer contains integers, single/double floating point numbers, or more complicated types, to be discussed later.
- A count. This states how many of the elements of the given datatype are in the send buffer, or will be received into the receive buffer.

These three together describe what MPI needs to send through the network.

In the various languages such a buffer/count/datatype triplet is specified in different ways.

First of all, in C the *buffer* is always an *opaque handle*, that is, a `void*` parameter to which you supply an address. This means that an MPI call can take two forms.

For scalars we need to use the ampersand operator to take the address:

```
float x,y;
MPI_Allreduce( &x,&y,1,MPI_FLOAT, ... );
```

But for arrays we use the fact that arrays and addresses are more-or-less equivalent in:

```
float xx[2],yy[2];
MPI_Allreduce( xx,yy,2,MPI_FLOAT, ... );
```

You could *cast* the buffers and write:

```
MPI_Allreduce( (void*)&xx,(void*)&y,1,MPI_FLOAT, ... );
MPI_Allreduce( (void*)xx,(void*)yy,2,MPI_FLOAT, ... );
```

but that is not necessary. The compiler will not complain if you leave out the cast.

C++ note 1: Buffer treatment. Treatment of scalars in C++ is the same as in C. However, for arrays you have the choice between C-style arrays, and `std::vector` or `std::array`. For the latter there are two ways of dealing with buffers:

```
vector<float> xx(25);
MPI_Send( xx.data(),25,MPI_FLOAT, ... );
MPI_Send( &xx[0],25,MPI_FLOAT, ... );
```

Fortran note 5: MPI send/recv buffers. In Fortran parameters are always passed by reference, so the *buffer* is treated the same way:

```
Real*4 :: x
Real*4,dimension(2) :: xx
call MPI_Allreduce( x,1,MPI_REAL4, ... )
call MPI_Allreduce( xx,2,MPI_REAL4, ... )
```

In discussing OO languages, we first note that the official C++ API has been removed from the standard.

Specification of the buffer/count/datatype triplet is not needed explicitly in OO languages.

Python note 7: Buffers from numpy. Most MPI routines in Python have both a variant that can send arbitrary Python data, and one that is based on *numpy* arrays. The former looks the most ‘pythonic’, and is very flexible, but is usually demonstrably inefficient.

```
## allreduce.py
random_number = random.randint(1,random_bound)
# native mode send
max_random = comm.allreduce(random_number,op=MPI.MAX)
```

In the numpy variant, all *buffers* are *numpy* objects, which carry information about their type and size. For scalar reductions this means we have to create an array for the receive buffer, even though only one element is used.

```
myrandom = np.empty(1,dtype=int)
myrandom[0] = random_number
allrandom = np.empty(nprocs,dtype=int)
# numpy mode send
comm.Allreduce(myrandom,allrandom[:1],op=MPI.MAX)
```

Python note 8: Buffers from subarrays. In many examples you will pass a whole Numpy array as send/receive buffer. Should want to use a buffer that corresponds to a subset of an array, you can use the following notation:

```
MPI_Whatever( buffer[... ,5] # more stuff
```

for passing the buffer that starts at location 5 of the array.

For even more complicated effects, use *numpy.frombuffer*:

3. MPI topic: Collectives

```
Code:
## bcastcolumn.py
datatype = np.intc
elementsizes = datatype().itemsize
typechar = datatype().dtype.char
buffer = np.zeros( [nprocs,nprocs] ,
                  dtype=datatype)
buffer[:, :] = -1
for proc in range(nprocs):
    if procid==proc:
        buffer[proc, :] = proc
comm.Bcast\(
    [ np.frombuffer\(
        ( buffer.data,
          dtype=datatype,
          offset=(proc*nprocs+proc)
          *elementsizes ),
        nprocs=proc, typechar ],
        root=proc )
```

```
Output:
[examples/mpi/p] buffer:
int size: 4
i
[[ 0  0  0  0  0  0]
 [-1  1  1  1  1  1]
 [-1 -1  2  2  2  2]
 [-1 -1 -1  3  3  3]
 [-1 -1 -1 -1  4  4]
 [ 5  5  5  5  5  5]]
```

MPL note 10: Scalar buffers. Buffer type handling is done through polymorphism (templating and ADL): no explicit indication of types.

Scalars are handled as such:

```
float x,y;
comm.bcast( 0,x ); // note: root first
```

MPL note 11: Vector buffers. If your buffer is a `std::vector` you need to take the `.data()` component of it:

```
vector<float> xx(2),yy(2);
comm.allreduce( mpl::plus<float>(),
    xx.data(), yy.data(), mpl::contiguous_layout<float>(2) );
```

The `contiguous_layout` is a ‘derived type’; this will be discussed in more detail elsewhere (see note 47 and later). For now, interpret it as a way of indicating the count/type part of a buffer specification.

MPL note 12: Array buffers. You can pass a C-style array as buffer, requiring a layout:

```
// vector of 50 floats
vector<float> ar(50);
auto root = 0;
auto data = ar.data(); // or &(ar[0]) or &ar.front()
auto layout = mpl::contiguous_layout<float>(50)
comm_world::bcast( root,data,layout );
```

MPL note 13: Iterator buffers. MPL point-to-point routines have a way of specifying the buffer(s) through a `begin` and `end` iterator.

```
// sendrange.cxx
vector<double> v(15);
comm_world.send(v.begin(), v.end(), 1); // send to rank 1
comm_world.recv(v.begin(), v.end(), 0); // receive from rank 0
```

Not available for collectives.

MPL note 14: Send vs recv buffer. There are separate variants for root vs non-root processes in rooted collectives:

```
// scangather.cxx
const int root=0;
if (procno==root) {
    comm_world.reduce
        ( mpl::plus<int>(),root,
        my_number_of_elements,total_number_of_elements );
} else {
    comm_world.reduce
        ( mpl::plus<int>(),root,my_number_of_elements );
}
```

3.3 Rooted collectives: broadcast, reduce

In some scenarios there is a certain process that has a privileged status.

- One process can generate or read in the initial data for a program run. This then needs to be communicated to all other processes.
- At the end of a program run, often one process needs to output some summary information.

This process is called the *root* process, and we will now consider routines that have a root.

3.3.1 Reduce to a root

In the broadcast operation a single data item was communicated to all processes. A reduction operation with `MPI_Reduce` (figure 3.2) goes the other way: each process has a data item, and these are all brought together into a single item.

Here are the essential elements of a reduction operation:

```
MPI_Reduce( senddata, recvdata..., operator,
            root, comm );
```

- There is the original data, and the data resulting from the reduction. It is a design decision of MPI that it will not by default overwrite the original data. The send data and receive data are of the same size and type: if every processor has one real number, the reduced result is again one real number.
- It is possible to indicate explicitly that a single buffer is used, and thereby the original data overwritten; see section 3.3.2 for this ‘in place’ mode.

3. MPI topic: Collectives

Figure 3.2 MPI_Reduce

Name	Param name	Explanation	C type	F type	inout
MPI_Reduce					
	MPI_Reduce_c				
	sendbuf	address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
	recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
	count	number of elements in send buffer	[int MPI_Count]	INTEGER	IN
	datatype	datatype of elements of send buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
	op	reduce operation	MPI_Op	TYPE(MPI_Op)	IN
	root	rank of root process	int	INTEGER	IN
	comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

MPL:

```
void mpl::communicator::reduce
// root, in place
( F f,int root_rank,T & sendrecvdata ) const
( F f,int root_rank,T * sendrecvdata,const contiguous_layout< T > & l ) const
// non-root
( F f,int root_rank,const T & senddata ) const
( F f,int root_rank,
    const T * senddata,const contiguous_layout< T > & l ) const
// general
( F f,int root_rank,const T & senddata,T & recvdata ) const
( F f,int root_rank,
    const T * senddata,T * recvdata,const contiguous_layout< T > & l ) const
```

Python:

```
comm.Reduce(self, sendbuf, recvbuf, Op op=SUM, int root=0)
native:
comm.reduce(self, sendobj=None, recvobj=None, op=SUM, int root=0)
```

- There is a reduction operator. Popular choices are `MPI_SUM`, `MPI_PROD` and `MPI_MAX`, but complicated operators such as finding the location of the maximum value exist. (For the full list, see section 3.10.1.) You can also define your own operators; section 3.10.2.
- There is a root process that receives the result of the reduction. Since the nonroot processes do not receive the reduced data, they can actually leave the receive buffer undefined.

```
// reduce.c
float myrandom = (float) rand()/(float)RAND_MAX,
      result;
int target_proc = nprocs-1;
// add all the random variables together
MPI_Reduce(&myrandom,&result,1,MPI_FLOAT,MPI_SUM,
            target_proc,comm);
// the result should be approx nprocs/2:
if (procno==target_proc)
    printf("Result %6.3f compared to nprocs/2=%5.2f\n",
           result,nprocs/2.);
```

Exercise 3.7. Write a program where each process computes a random number, and process 0 finds and prints the maximum generated value. Let each process print its value, just to check the correctness of your program.

Collective operations can also take an array argument, instead of just a scalar. In that case, the operation is applied pointwise to each location in the array.

Exercise 3.8. Create on each process an array of length 2 integers, and put the values 1, 2 in it on each process. Do a sum reduction on that array. Can you predict what the result should be? Code it. Was your prediction right?

3.3.2 Reduce in place

By default MPI will not overwrite the original data with the reduction result, but you can tell it to do so using the `MPI_IN_PLACE` specifier:

```
// allreduceinplace.c
for (int irand=0; irand<nrandoms; irand++)
    myrandoms[irand] = (float) rand()/(float)RAND_MAX;
// add all the random variables together
MPI_Allreduce(MPI_IN_PLACE,myrandoms,
              nrandoms,MPI_FLOAT,MPI_SUM,comm);
```

Now every process only has a receive buffer, so this has the advantage of saving half the memory. Each process puts its input values in the receive buffer, and these are overwritten by the reduced result.

The above example used `MPI_IN_PLACE` in `MPI_Allreduce`; in `MPI_Reduce` it's little tricky. The reasoning is as follows:

- In `MPI_Reduce` every process has a buffer to contribute, but only the root needs a receive buffer. Therefore, `MPI_IN_PLACE` takes the place of the receive buffer on any processor except for the root ...

3. MPI topic: Collectives

- ... while the root, which needs a receive buffer, has `MPI_IN_PLACE` takes the place of the send buffer. In order to contribute its value, the root needs to put this in the receive buffer.

Here is one way you could write the in-place version of `MPI_Reduce`:

```
if (procno==root)
    MPI_Reduce(MPI_IN_PLACE,myrandoms,
               nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
else
    MPI_Reduce(myrandoms,MPI_IN_PLACE,
               nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
```

However, as a point of style, having different versions of a collective in different branches of a condition is infelicitous. The following may be preferable:

```
float *sendbuf,*recvbuf;
if (procno==root) {
    sendbuf = MPI_IN_PLACE; recvbuf = myrandoms;
} else {
    sendbuf = myrandoms; recvbuf = MPI_IN_PLACE;
}
MPI_Reduce(sendbuf,recvbuf,
           nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
```

Fortran note 6: In-place operations. In Fortran you can not do these address calculations. You can use the solution with a conditional:

```
!! reduceinplace.F90
call random_number(mynumber)
target_proc = ntids-1;
! add all the random variables together
if (mytid.eq.target_proc) then
    result = mytid
    call MPI_Reduce(MPI_IN_PLACE,result,1,MPI_REAL,MPI_SUM,&
                   target_proc,comm)
else
    mynumber = mytid
    call MPI_Reduce(mynumber,result,1,MPI_REAL,MPI_SUM,&
                   target_proc,comm)
end if
```

but you can also solve this with pointers:

```
!! reduceinplaceptr.F90
in_place_val = MPI_IN_PLACE
if (mytid.eq.target_proc) then
    ! set pointers
    result_ptr => result
    mynumber_ptr => in_place_val
    ! target sets value in receive buffer
    result_ptr = mytid
else
    ! set pointers
    mynumber_ptr => mynumber
```

```
result_ptr => in_place_val
! non-targets set value in send buffer
mynumber_ptr = mytid
end if
call MPI_Reduce(mynumber_ptr,result_ptr,1,MPI_REAL,MPI_SUM,&
target_proc,comm,err)
```

Python note 9: In-place collectives. The value `MPI.IN_PLACE` can be used for the send buffer:

```
## allreduceinplace.py
myrandom = np.empty(1,dtype=int)
myrandom[0] = random_number

comm.Allreduce(MPI.IN_PLACE,myrandom,op=MPI.MAX)
```

MPL note 15: Reduce in place. The in-place variant is activated by specifying only one instead of two buffer arguments.

```
// separate recv buffer
comm_world.allreduce(mpl::plus<float>(), proc_data,reduce_data);
// in place
comm_world.allreduce(mpl::plus<float>(), proc_data);
```

Reducing a buffer requires specification of a `contiguous_layout`:

```
// collectbuffer.cxx
vector<float> rank2p2p1{ 2*xrank,2*xrank+1 },reduce2p2p1{0,0};
mpl::contiguous_layout<float> two_floats(rank2p2p1.size());
comm_world.allreduce
  (mpl::plus<float>(), rank2p2p1.data(),reduce2p2p1.data(),two_floats);
if ( iprint )
  cout << "Got: " << reduce2p2p1.at(0) << ","
    << reduce2p2p1.at(1) << endl;
```

Note that the buffers are of type `T *`, so it is necessary to take the `data()` of any `std::vector` and such.

3.3.3 Broadcast

A broadcast models the scenario where one process, the ‘root’ process, owns some data, and it communicates it to all other processes.

The broadcast routine `MPI_Bcast` (figure 3.3) has the following structure:

```
MPI_Bcast( data..., root , comm);
```

Here:

- There is only one buffer, the send buffer. Before the call, the root process has data in this buffer; the other processes allocate a same sized buffer, but for them the contents are irrelevant.
- The root is the process that is sending its data. Typically, it will be the root of a broadcast tree.

3. MPI topic: Collectives

Figure 3.3 MPI_Bcast

Name	Param name	Explanation	C type	F type	inout
MPI_Bcast					
	MPI_Bcast_c				
buffer		starting address of buffer	void*	TYPE(*), DIMENSION(..)	INOUT
count		number of entries in buffer	[int MPI_Count	INTEGER	IN
datatype		datatype of buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
root		rank of broadcast root	int	INTEGER	IN
comm		communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)					

MPL:

```
template<typename T >
void mpl::communicator::bcast
( int root, T & data ) const
( int root, T * data, const layout< T > & l ) const
```

Python:

```
MPI.Comm.Bcast(self, buf, int root=0)
```

Example: in general we can not assume that all processes get the *commandline arguments*, so we broadcast them from process 0.

```
// init.c
if (procno==0) {
    if ( argc==1 || // the program is called without parameter
        ( argc>1 && !strcmp(argv[1],"-h") ) // user asked for help
    ) {
        printf("\nUsage: init [0-9]+\n");
        MPI_Abort(comm,1);
    }
    input_argument = atoi(argv[1]);
}
MPI_Bcast(&input_argument,1,MPI_INT,0,comm);
```

Python note 10: Sending objects. In python it is both possible to send objects, and to send more C-like buffers. The two possibilities correspond (see section 1.5.4) to different routine names; the buffers have to be created as numpy objects.

We illustrate both the general Python and numpy variants. In the former variant the result is given as a function return; in the numpy variant the send buffer is reused.

```
## bcast.py
# first native
if procid==root:
```

```

    buffer = [ 5.0 ] * dsize
else:
    buffer = [ 0.0 ] * dsize
buffer = comm.bcast(obj=buffer,root=root)
if not reduce( lambda x,y:x and y,
               [ buffer[i]==5.0 for i in range(len(buffer)) ] ):
    print( "Something wrong on proc %d: native buffer <<%s>>" \
          % (procid,str(buffer)) )

# then with NumPy
buffer = np.arange(dsize, dtype=np.float64)
if procid==root:
    for i in range(dsize):
        buffer[i] = 5.0
comm.Bcast( buffer,root=root )
if not all( buffer==5.0 ):
    print( "Something wrong on proc %d: numpy buffer <<%s>>" \
          % (procid,str(buffer)) )
else:
    if procid==root:
        print("Success.")

```

MPL note 16: Broadcast. The broadcast call comes in two variants, with scalar argument and general layout:

```

template<typename T >
void mpl::communicator::bcast
( int root_rank, T &data ) const;
void mpl::communicator::bcast
( int root_rank, T *data, const layout< T > &l ) const;

```

Note that the root argument comes first in a departure from the C API.

For the following exercise, study figure 3.2.

Exercise 3.9. The *Gauss-Jordan algorithm* for solving a linear system with a matrix A (or computing its inverse) runs as follows:

for pivot $k = 1, \dots, n$

let the vector of scalings $\ell_i^{(k)} = A_{ik}/A_{kk}$
for row $r \neq k$

for column $c = 1, \dots, n$

$A_{rc} \leftarrow A_{rc} - \ell_r^{(k)} A_{kc}$

where we ignore the update of the righthand side, or the formation of the inverse. Let a matrix be distributed with each process storing one column. Implement the Gauss-Jordan algorithm as a series of broadcasts: in iteration k process k computes and broadcasts the scaling vector $\{\ell_i^{(k)}\}_i$. Replicate the right-hand side on all processors.

(There is a skeleton for this exercise under the name `jordan`.)

3. MPI topic: Collectives

Initial:

matrix	sol	rhs	action	scaling	matrix	sol	rhs	action	scaling	matrix	sol	rhs	action	scaling
2 2 13	1	17			2 2 13	1	17			2 0 1	1	3	minus $1 \times 1/3$	
4 5 32	1	41			0 1 6	1	7	take this row		0 1 6	1	7		
-2 -3 -16	1	-21			0 -1 -3	1	-4			↑↑↑	1	3	take this row	1/3

Step 1:

matrix	sol	rhs	action	scaling	matrix	sol	rhs	action	scaling	matrix	sol	rhs	action	scaling
2 2 13	1	17	take this row	1/2	2 2 13	1	17	minus 2×1		2 0 0	1	2		
4 5 32	1	41			↑↑↑	1	6	take this row		0 1 6	1	7		-1/3
-2 -3 -16	1	-21			0 -1 -3	1	-4			0 0 3	1	3	take this row	1/3

Step 2:

matrix	sol	rhs	action	scaling	matrix	sol	rhs	action	scaling	matrix	sol	rhs	action	scaling
2 2 13	1	17	take this row	1/2	2 0 1	1	3			2 0 0	1	2		
↓ ↓ ↓	↓	↓	minus $4 \times (1/2)$		0 1 6	1	7	take this row		0 1 6	1	7	minus $6 \times 1/3$	
4 5 32	1	41			0 -1 -3	1	-4			0 0 3	1	3	take this row	1/3

Step 3:

matrix	sol	rhs	action	scaling	matrix	sol	rhs	action	scaling	matrix	sol	rhs	action	scaling
2 2 13	1	17	take this row	1/2	2 0 1	1	3			2 0 0	1	2		
0 1 6	1	7		-2	0 1 6	1	7	take this row		0 1 6	1	7	minus $6 \times 1/3$	
-2 -3 -16	1	-21			↓↓↓	-1 -3	1	-4	minus $(-1) \times 1$	0 3	1	3	take this row	1/3

Step 4:

matrix	sol	rhs	action	scaling	matrix	sol	rhs	action	scaling	matrix	sol	rhs	action	scaling
2 2 13	1	17	take this row	1/2	2 0 1	1	3			2 0 0	1	2		
↓ ↓ ↓	↓	↓	minus $(-2) \times (1/2)$		0 1 6	1	7	take this row		0 3	1	3	take this row	
0 1 6	1	7		-2	0 1 6	1	7			0 0 3	1	3		

Step 5:

matrix	sol	rhs	action	scaling	matrix	sol	rhs	action	scaling	matrix	sol	rhs	action	scaling
2 2 13	1	17	take this row	1/2	2 0 1	1	3			2 0 0	1	2		
↓ ↓ ↓	↓	↓	minus $(-2) \times (1/2)$		0 1 6	1	7	take this row		0 0 3	1	3	take this row	
0 1 6	1	7		-2	0 1 6	1	7			0 0 3	1	3		

Step 6:

matrix	sol	rhs	action	scaling	matrix	sol	rhs	action	scaling	matrix	sol	rhs	action	scaling
2 2 13	1	17	take this row	1/2	2 0 1	1	3			2 0 0	1	2		
0 1 6	1	7		-2	0 1 6	1	7	second column done	1	0 0 3	1	3	third column done	1/3
0 -1 -3	1	-4		+1	0 0 3	1	3			0 0 3	1	3		

Finished:

Step 11:

Figure 3.2: Gauss-Jordan elimination example

Figure 3.4 MPI_Scan

Name	Param name	Explanation	C type	F type	inout
MPI_Scan (
MPI_Scan_c (
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN	
recvbuf	starting address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT	
count	number of elements in input buffer	[int MPI_Count]	INTEGER	IN	
datatype	datatype of elements of input buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN	
op	operation	MPI_Op	TYPE(MPI_Op)	IN	
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN	
)					

MPL:

```
template<typename T , typename F >
void mpl::communicator::scan
( F, const T &, T & ) const;
( F, const T *, T *,
  const contiguous_layout< T > & ) const;
( F, T & ) const;
( F, T *, const contiguous_layout< T > & ) const;
```

F : reduction function

T : type

Python:

```
res = Intracomm.scan( sendobj=None,recvobj=None,op=MPI.SUM)
```

Exercise 3.10. Add partial pivoting to your implementation of Gauss-Jordan elimination.

Change your implementation to let each processor store multiple columns, but still do one broadcast per column. Is there a way to have only one broadcast per processor?

3.4 Scan operations

The `MPI_Scan` operation also performs a reduction, but it keeps the partial results. That is, if processor i contains a number x_i , and \oplus is an operator, then the scan operation leaves $x_0 \oplus \dots \oplus x_i$ on processor i . This type of operation is often called a *prefix operation*; see HPC book, section ??.

The `MPI_Scan` (figure 3.4) routine is an *inclusive scan* operation, meaning that it includes the data on the process itself; `MPI_Exscan` (see section 3.4.1) is exclusive, and does not include the data on the calling process.

process :	0	1	2	...	$p - 1$
data :	x_0	x_1	x_2	...	x_{p-1}
inclusive :	x_0	$x_0 \oplus x_1$	$x_0 \oplus x_1 \oplus x_2$...	$\bigoplus_{i=0}^{p-1} x_i$
exclusive :	unchanged	x_0	$x_0 \oplus x_1$...	$\bigoplus_{i=0}^{p-2} x_i$

```
// scan.c
// add all the random variables together
MPI_Scan(&myrandom,&result,1,MPI_FLOAT,MPI_SUM,comm);
// the result should be approaching nprocs/2:
if (procno==nprocs-1)
    printf("Result %6.3f compared to nprocs/2=%5.2f\n",
           result,nprocs/2.);
```

In python mode the result is a function return value, with numpy the result is passed as the second parameter.

```
## scan.py
mycontrib = 10+random.randint(1,nprocs)
myfirst = 0
mypartial = comm.scan(mycontrib)
sbuf = np.empty(1,dtype=int)
rbuf = np.empty(1,dtype=int)
sbuf[0] = mycontrib
comm.Scan(sbuf,rbuf)
```

You can use any of the given reduction operators, (for the list, see section 3.10.1), or a user-defined one. In the latter case, the `MPI_Op` operations do not return an error code.

MPL note 17: Scan operations. As in the C/F interfaces, MPL interfaces to the scan routines have the same calling sequences as the ‘Allreduce’ routine.

3.4.1 Exclusive scan

Often, the more useful variant is the *exclusive scan* `MPI_Exscan` (figure 3.5) with the same signature.

The result of the exclusive scan is undefined on processor 0 (None in python), and on processor 1 it is a copy of the send value of processor 1. In particular, the `MPI_Op` need not be called on these two processors.

Exercise 3.11. The exclusive definition, which computes $x_0 \oplus x_{i-1}$ on processor i , can be derived from the inclusive operation for operations such as `MPI_SUM` or `MPI_PROD`. Are there operators where that is not the case?

3.4.2 Use of scan operations

The `MPI_Scan` operation is often useful with indexing data. Suppose that every processor p has a local vector where the number of elements n_p is dynamically determined. In order to translate the local numbering

Figure 3.5 MPI_Exscan

Name	Param name	Explanation	C type	F type	inout
MPI_Exscan					
	MPI_Exscan_c				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN	
recvbuf	starting address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT	
count	number of elements in input buffer	[int MPI_Count]	INTEGER	IN	
datatype	datatype of elements of input buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN	
op	operation	MPI_Op	TYPE(MPI_Op)	IN	
comm	intra-communicator	MPI_Comm	TYPE (MPI_Comm)	IN	
)					

MPL:

```
template<typename T , typename F >
void mpl::communicator::exscan
( F, const T &, T & ) const;
( F, const T *, T *,
  const contiguous_layout< T > & ) const;
( F, T & ) const;
( F, T *, const contiguous_layout< T > & ) const;
F : reduction function
T : type
```

Python:

```
res = Intracomm.exscan( sendobj=None,recvobj=None,op=MPI.SUM)
```

$0 \dots n_p - 1$ to a global numbering one does a scan with the number of local elements as input. The output is then the global number of the first local variable.

As an example, setting *Fast Fourier Transform (FFT)* coefficients requires this translation. If the local sizes are all equal, determining the global index of the first element is an easy calculation. For the irregular case, we first do a scan:

```
// fft.c
MPI_Allreduce( &localsize,&globalsize,1,MPI_INT,MPI_SUM, comm );
globalsize += 1;
int myfirst=0;
MPI_Exscan( &localsize,&myfirst,1,MPI_INT,MPI_SUM, comm );

for (int i=0; i<localsize; i++)
    vector[i] = sin( pi*freq* (i+1+myfirst) / globalsize );
```

Exercise 3.12.



Figure 3.3: Local arrays that together form a consecutive range

- Let each process compute a random value n_{local} , and allocate an array of that length. Define

$$N = \sum n_{\text{local}}$$

- Fill the array with consecutive integers, so that all local arrays, laid end-to-end, contain the numbers $0 \dots N - 1$. (See figure 3.3.)

(There is a skeleton for this exercise under the name `scangather`.)

Exercise 3.13. Did you use `MPI_Scan` or `MPI_Exscan` for the previous exercise? How would you describe the result of the other scan operation, given the same input?

It is possible to do a *segmented scan*. Let x_i be a series of numbers that we want to sum to X_i as follows. Let y_i be a series of booleans such that

$$\begin{cases} X_i = x_i & \text{if } y_i = 0 \\ X_i = X_{i-1} + x_i & \text{if } y_i = 1 \end{cases}$$

(This is the basis for the implementation of the *sparse matrix vector product* as prefix operation; see HPC book, section ??.) This means that X_i sums the segments between locations where $y_i = 0$ and the first subsequent place where $y_i = 1$. To implement this, you need a user-defined operator

$$\begin{pmatrix} X \\ x \\ y \end{pmatrix} = \begin{pmatrix} X_1 \\ x_1 \\ y_1 \end{pmatrix} \bigoplus \begin{pmatrix} X_2 \\ x_2 \\ y_2 \end{pmatrix} : \begin{cases} X = x_1 + x_2 & \text{if } y_2 == 1 \\ X = x_2 & \text{if } y_2 == 0 \end{cases}$$

This operator is not commutative, and it needs to be declared as such with `MPI_Op_create`; see section 3.10.2

3.5 Rooted collectives: gather and scatter

In the `MPI_Scatter` operation, the root spreads information to all other processes. The difference with a broadcast is that it involves individual information from/to every process. Thus, the gather operation typically has an array of items, one coming from each sending process, and scatter has an array, with an individual item for each receiving process; see figure 3.5.

These gather and scatter collectives have a different parameter list from the broadcast/reduce. The broadcast/reduce involves the same amount of data on each process, so it was enough to have a single datatype-/size specification; for one buffer in the broadcast, and for both buffers in the reduce call. In the gather/scatter calls you have

- a large buffer on the root, with a datatype and size specification, and

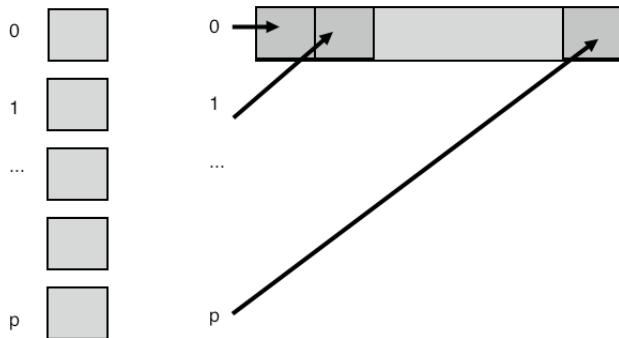


Figure 3.4: Gather collects all data onto a root

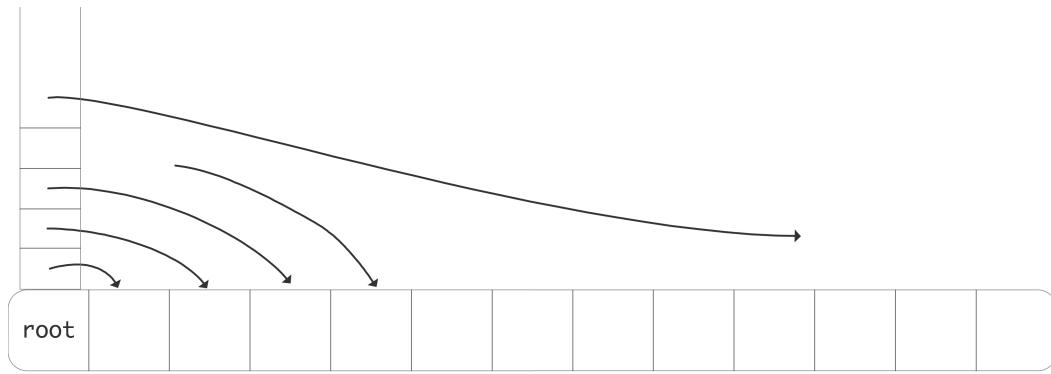


Figure 3.5: A scatter operation

- a smaller buffer on each process, with its own type and size specification.

In the gather and scatter calls, each processor has n elements of individual data. There is also a root processor that has an array of length np , where p is the number of processors. The gather call collects all this data from the processors to the root; the scatter call assumes that the information is initially on the root and it is spread to the individual processors.

Here is a small example:

```
// gather.c
// we assume that each process has a value "localsize"
// the root process collects these values

if (procno==root)
    localsizes = (int*) malloc( nprocs*sizeof(int) );

// everyone contributes their info
MPI_Gather(&localsize,1,MPI_INT,
           localsizes,1,MPI_INT,root,comm);
```

This will also be the basis of a more elaborate example in section 3.9.

Exercise 3.14. Let each process compute a random number. You want to print the

3. MPI topic: Collectives

Figure 3.6 MPI_Gather

Name	Param name	Explanation	C type	F type	inout
MPI_Gather					
	MPI_Gather_c				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN	
sendcount	number of elements in send buffer	[int MPI_Count]	INTEGER	IN	
sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN	
recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT	
recvcount	number of elements for any single receive	[int MPI_Count]	INTEGER	IN	
recvtype	datatype of recv buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN	
root	rank of receiving process	int	INTEGER	IN	
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN	
)					

MPL:

```
void mpl::communicator::gather
( int root_rank, const T * senddata, const layout< T > & sendl ) const
( int root_rank, const T * senddata, const layout< T > & sendl,
    T * recvdata, const layout< T > & recvl ) const
// non-root versions:
( int root_rank, const T & senddata ) const
( int root_rank, const T & senddata, T * recvdata ) const
```

Python:

```
MPI.Comm.Gather
(self, sendbuf, recvbuf, int root=0)
```

maximum value and on what processor it is computed. What collective(s) do you use? Write a short program.

The **MPI_Scatter** operation is in some sense the inverse of the gather: the root process has an array of length np where p is the number of processors and n the number of elements each processor will receive.

```
int MPI_Scatter
(void* sendbuf, int sendcount, MPI_Datatype sendtype,
 void* recvbuf, int recvcount, MPI_Datatype recvtype,
 int root, MPI_Comm comm)
```

Two things to note about these routines:

- The signature for **MPI_Gather** (figure 3.6) has two ‘count’ parameters, one for the length of the

individual send buffers, and one for the receive buffer. However, confusingly, the second parameter (which is only relevant on the root) does not indicate the total amount of information coming in, but rather the size of *each* contribution. Thus, the two count parameters will usually be the same (at least on the root); they can differ if you use different `MPI_Datatype` values for the sending and receiving processors.

- While every process has a sendbuffer in the gather, and a receive buffer in the scatter call, only the root process needs the long array in which to gather, or from which to scatter. However, because in SPMD mode all processes need to use the same routine, a parameter for this long array is always present. Nonroot processes can use a *null pointer* here.
- More elegantly, the `MPI_IN_PLACE` option can be used for buffers that are not applicable, such as the receive buffer on a sending process. See section 3.3.2.

MPL note 18: Gather/scatter. Gathering (by `communicator::gather`)

or scattering (by `communicator::scatter`)

a single scalar takes a scalar argument and a raw array:

```
vector<float> v;
float x;
comm_world.scatter(0, v.data(), x);
```

If more than a single scalar is gathered, or scattered into, it becomes necessary to specify a layout:

```
vector<float> vrecv(2),vsend(2*nprocs);
mpl::contiguous_layout<float> twonums(2);
comm_world.scatter
(0, vsend.data(),twonums, vrecv.data(),twonums );
```

MPL note 19: Gather on nonroot. Logically speaking, on every nonroot process, the gather call only has a send buffer. MPL supports this by having two variants that only specify the send data.

```
if (procno==root) {
    vector<int> size_buffer(nprocs);
    comm_world.gather
    (
        root,my_number_of_elements,size_buffer.data()
    );
} else {
    /*
     * If you are not the root, do versions with only send buffers
     */
    comm_world.gather
    ( root,my_number_of_elements );
```

3.5.1 Examples

In some applications, each process computes a row or column of a matrix, but for some calculation (such as the determinant) it is more efficient to have the whole matrix on one process. You should of course

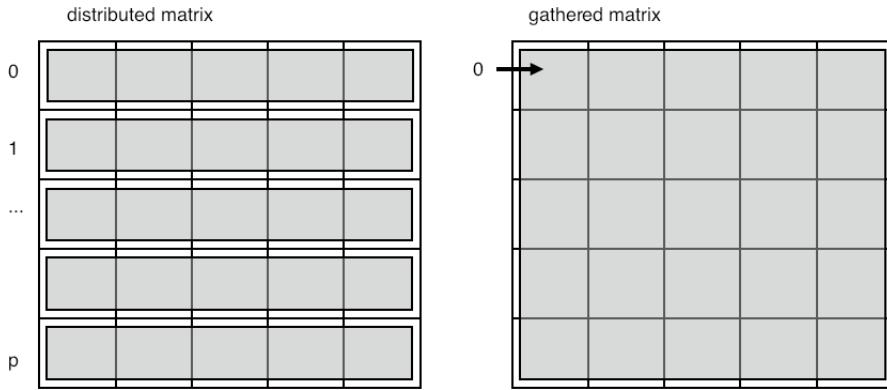


Figure 3.6: Gather a distributed matrix onto one process

only do this if this matrix is essentially smaller than the full problem, such as an interface system or the last coarsening level in multigrid.

Figure 3.6 pictures this. Note that conceptually we are gathering a two-dimensional object, but the buffer is of course one-dimensional. You will later see how this can be done more elegantly with the ‘subarray’ datatype; section 6.3.4.

Another thing you can do with a distributed matrix is to transpose it.

```
// itransposeblock.c
for (int iproc=0; iproc<nprocs; iproc++) {
    MPI_Scatter( regular, 1, MPI_DOUBLE,
                 &(transpose[iproc]), 1, MPI_DOUBLE,
                 iproc, comm);
}
```

In this example, each process scatters its column. This needs to be done only once, yet the scatter happens in a loop. The trick here is that a process only originates the scatter when it is the root, which happens only once. Why do we need a loop? That is because each element of a process’ row originates from a different scatter operation.

Exercise 3.15. Can you rewrite this code so that it uses a gather rather than a scatter? Does that change anything essential about structure of the code?

Exercise 3.16. Take the code from exercise 3.12 and extend it to gather all local buffers onto rank zero. Since the local arrays are of differing lengths, this requires `MPI_Gatherv`. How do you construct the lengths and displacements arrays?
(There is a skeleton for this exercise under the name `scangather`.)

3.5.2 Allgather

The `MPI_Allgather` (figure 3.7) routine does the same gather onto every process: each process winds up with the totality of all data; figure 3.7.

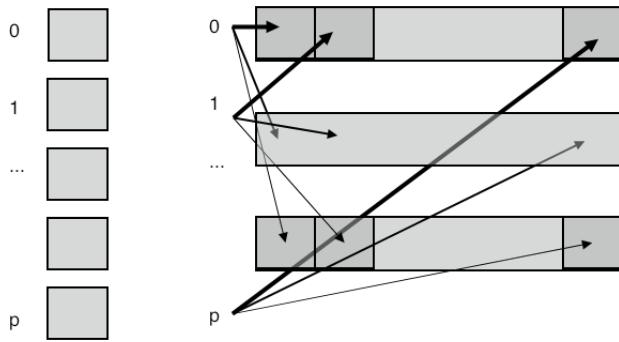


Figure 3.7: All gather collects all data onto every process

This routine can be used in the simplest implementation of the *dense matrix-vector product* to give each processor the full input; see HPC book, section ??.

The `MPI_IN_PLACE` keyword can be used with an all-gather:

1. Use `MPI_IN_PLACE` for the send buffer;
2. send count and datatype are ignored by MPI;
3. each process needs to put its ‘send content’ in the correct location of the gather buffer.

Some cases look like an all-gather but can be implemented more efficiently. Let’s consider as an example the set difference of two distributed vectors. That is, you have two distributed vectors, and you want to create a new vector, again distributed, that contains those elements of the one that do not appear in the other. You could implement this by gathering the second vector on each processor, but this may be prohibitive in memory usage.

Exercise 3.17. Can you think of another algorithm for taking the set difference of two distributed vectors. Hint: look up *bucket brigade* algorithm; section 4.1.5. What is the time and space complexity of this algorithm? Can you think of other advantages beside a reduction in workspace?

3.6 All-to-all

The all-to-all operation `MPI_Alltoall` (figure 3.8) can be seen as a collection of simultaneous broadcasts or simultaneous gathers. The parameter specification is much like an allgather, with a separate send and receive buffer, and no root specified. As with the gather call, the receive count corresponds to an individual receive, not the total amount.

Unlike the gather call, the send buffer now obeys the same principle: with a send count of 1, the buffer has a length of the number of processes.

3.6.1 All-to-all as data transpose

The all-to-all operation can be considered as a data transpose. For instance, assume that each process knows how much data to send to every other process. If you draw a connectivity matrix of size $P \times P$,

3. MPI topic: Collectives

Figure 3.7 MPI_Allgather

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Allgather</code>					
<code>MPI_Allgather_c</code>					
<code>sendbuf</code>	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN	
<code>sendcount</code>	number of elements in send buffer	[int MPI_Count]	INTEGER	IN	
<code>sendtype</code>	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN	
<code>recvbuf</code>	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT	
<code>recvcount</code>	number of elements received from any process	[int MPI_Count]	INTEGER	IN	
<code>recvtype</code>	datatype of receive buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN	
<code>comm</code>	communicator	MPI_Comm	TYPE (MPI_Comm)	IN	
)					

MPL:

```
void allgather
  ( const T & send_data, T * recv_data ) const
  ( const T * send_data, const layout< T > & sendl,
    T * recv_data, const layout< T > & recvl ) const
```

denoting who-sends-to-who, then the send information can be put in rows:

$$\forall_i : C[i, j] > 0 \quad \text{if process } i \text{ sends to process } j.$$

Conversely, the columns then denote the receive information:

$$\forall_j : C[i, j] > 0 \quad \text{if process } j \text{ receives from process } i.$$

The typical application for such data transposition is in the FFT algorithm, where it can take tens of percents of the running time on large clusters.

We will consider another application of data transposition, namely *radix sort*, but we will do that in a couple of steps. First of all:

Exercise 3.18. In the initial stage of a *radix sort*, each process considers how many elements to send to every other process. Use `MPI_Alltoall` to derive from this how many elements they will receive from every other process.

3.6.2 All-to-all-v

The major part of the *radix sort* algorithm consist of every process sending some of its elements to each of the other processes. The routine `MPI_Alltoallv` (figure 3.9) is used for this pattern:

Figure 3.8 MPI_Alltoall

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Alltoall (</code>					
<code> MPI_Alltoall_c (</code>					
<code>sendbuf</code>		starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
<code>sendcount</code>		number of elements sent to each process	[int MPI_Count]	INTEGER	IN
<code>sendtype</code>		datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
<code>recvbuf</code>		address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
<code>recvcount</code>		number of elements received from any process	[int MPI_Count]	INTEGER	IN
<code>recvtype</code>		datatype of receive buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
<code>comm</code>		communicator	MPI_Comm	TYPE (MPI_Comm)	IN
<code>)</code>					

- Every process scatters its data to all others,
- but the amount of data is different per process.

Exercise 3.19. The actual data shuffle of a *radix sort* can be done with `MPI_Alltoallv`. Finish the code of exercise 3.18.

3.7 Reduce-scatter

There are several MPI collectives that are functionally equivalent to a combination of others. You have already seen `MPI_Allreduce` which is equivalent to a reduction followed by a broadcast. Often such combinations can be more efficient than using the individual calls; see HPC book, section ??.

Here is another example: `MPI_Reduce_scatter` is equivalent to a reduction on an array of data (meaning a pointwise reduction on each array location) followed by a scatter of this array to the individual processes.

We will discuss this routine, or rather its variant `MPI_Reduce_scatter_block` (figure 3.10), using an important example: the *sparse matrix-vector product* (see HPC book, section ?? for background information). Each process contains one or more matrix rows, so by looking at indices the process can decide what other processes it needs to receive data from, that is, each process knows how many messages it will receive, and from which processes. The problem is for a process to find out what other processes it needs to send data to.

Let's set up the data:

3. MPI topic: Collectives

Figure 3.9 MPI_Alltoallv

Name	Param name	Explanation	C type	F type	inout
	MPI_Alltoallv (
	MPI_Alltoallv_c (
	sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank	[const int[] MPI_Count[]]	INTEGER(*)	IN
	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j	[const int[] MPI_Aint[]]	INTEGER(*)	IN
	sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
	recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank	[const int[] MPI_Count[]]	INTEGER(*)	IN
	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	[const int[] MPI_Aint[]]	INTEGER(*)	IN
	recvtype	datatype of receive buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
	comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

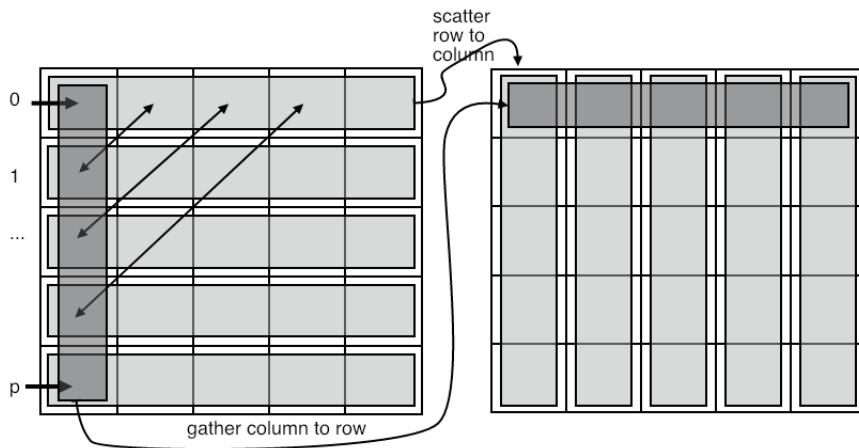


Figure 3.8: All-to-all transposes data

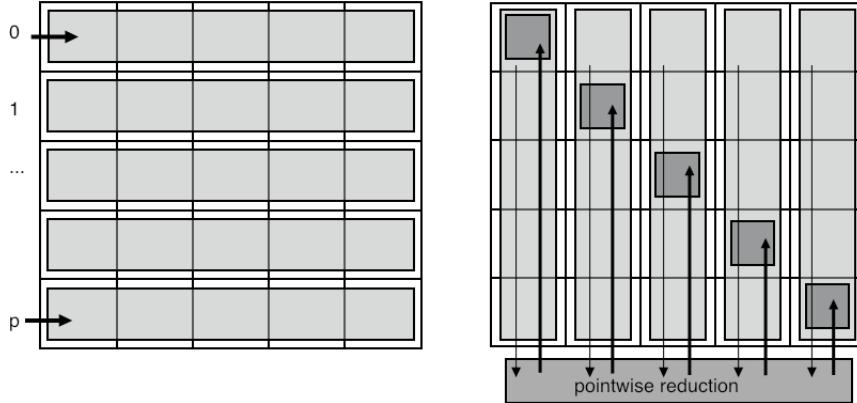


Figure 3.9: Reduce scatter

```
// reducescatter.c
int
// data that we know:
*i_recv_from_proc = (int*) malloc(nprocs*sizeof(int)),
*procs_to_recv_from, nprocs_to_recv_from=0,
// data we are going to determine:
*procs_to_send_to,nprocs_to_send_to;
```

Each process creates an array of ones and zeros, describing who it needs data from. Ideally, we only need the array `procs_to_recv_from` but initially we need the (possibly much larger) array `i_recv_from_proc`.

Next, the `MPI_Reduce_scatter_block` call then computes, on each process, how many messages it needs to send.

```
MPI_Reduce_scatter_block
(i_recv_from_proc,&nprocs_to_send_to,1,MPI_INT,
```

3. MPI topic: Collectives

```
MPI_SUM,comm);
```

We do not yet have the information to which processes to send. For that, each process sends a zero-size message to each of its senders. Conversely, it then does a receive to with `MPI_ANY_SOURCE` to discover who is requesting data from it. The crucial point to the `MPI_Reduce_scatter_block` call is that, without it, a process would not know how many of these zero-size messages to expect.

```
/*
 * Send a zero-size msg to everyone that you receive from,
 * just to let them know that they need to send to you.
 */
MPI_Request send_requests[nprocs_to_recv_from];
for (int iproc=0; iproc<nprocs_to_recv_from; iproc++) {
    int proc=procs_to_recv_from[iproc];
    double send_buffer=0.;
    MPI_Isend(&send_buffer,0,MPI_DOUBLE, /*to:*/ proc,0,comm,
              &(send_requests[iproc]));
}

/*
 * Do as many receives as you know are coming in;
 * use wildcards since you don't know where they are coming from.
 * The source is a process you need to send to.
 */
procs_to_send_to = (int*)malloc( nprocs_to_send_to * sizeof(int) );
for (int iproc=0; iproc<nprocs_to_send_to; iproc++) {
    double recv_buffer;
    MPI_Status status;
    MPI_Recv(&recv_buffer,0,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,comm,
             &status);
    procs_to_send_to[iproc] = status.MPI_SOURCE;
}
MPI_Waitall(nprocs_to_recv_from,send_requests,MPI_STATUSES_IGNORE);
```

The `MPI_Reduce_scatter` (figure 3.10) call is more general: instead of indicating the mere presence of a message between two processes, by having individual receive counts one can, for instance, indicate the size of the messages.

We can look at reduce-scatter as a limited form of the all-to-all data transposition discussed above (section 3.6.1). Suppose that the matrix C contains only 0/1, indicating whether or not a messages is send, rather than the actual amounts. If a receiving process only needs to know how many messages to receive, rather than where they come from, it is enough to know the column sum, rather than the full column (see figure 3.9).

Another application of the reduce-scatter mechanism is in the dense matrix-vector product, if a two-dimensional data distribution is used.

Exercise 3.20. Implement the dense matrix-vector product, where the matrix is distributed by columns: each process stores A_{*j} for a disjoint set of j -values. At the start and end of the algorithm each process should store a distinct part of the input and

Figure 3.10 MPI_Reduce_scatter

Name	Param name	Explanation	C type	F type	inout
MPI_Reduce_scatter (
MPI_Reduce_scatter_c (
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN	
recvbuf	starting address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT	
recvcounts	non-negative integer array (of length group size) specifying the number of elements of the result distributed to each process.	const int[] MPI_Count[]	INTEGER(*)	IN	
datatype	datatype of elements of send and receive buffers	MPI_Datatype	TYPE (MPI_Datatype)	IN	
op	operation	MPI_Op	TYPE(MPI_Op)	IN	
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN	
)					

output vectors. Argue that this can be done naively with an `MPI_Reduce` operation, but more efficiently using `MPI_Reduce_scatter`.

For discussion, see HPC book, section ??.

3.7.1 Examples

An important application of this is establishing an irregular communication pattern. Assume that each process knows which other processes it wants to communicate with; the problem is to let the other processes know about this. The solution is to use `MPI_Reduce_scatter` to find out how many processes want to communicate with you

```
MPI_Reduce_scatter_block
(i_recv_from_proc,&nprocs_to_send_to,1,MPI_INT,
MPI_SUM,comm);
```

and then wait for precisely that many messages with a source value of `MPI_ANY_SOURCE`.

```
/*
 * Send a zero-size msg to everyone that you receive from,
 * just to let them know that they need to send to you.
 */
MPI_Request send_requests[nprocs_to_recv_from];
for (int iproc=0; iproc<nprocs_to_recv_from; iproc++) {
    int proc=procs_to_recv_from[iproc];
    double send_buffer=0.;
```

3. MPI topic: Collectives

Figure 3.11 MPI_Barrier

Name	Param name	Explanation	C type	F type	inout
MPI_Barrier	comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN

```
MPI_Isend(&send_buffer,0,MPI_DOUBLE, /*to:*/ proc,0,comm,
           &(send_requests[iproc]));
}

/*
 * Do as many receives as you know are coming in;
 * use wildcards since you don't know where they are coming from.
 * The source is a process you need to send to.
 */
procs_to_send_to = (int*)malloc( nprocs_to_send_to * sizeof(int) );
for (int iproc=0; iproc<nprocs_to_send_to; iproc++) {
    double recv_buffer;
    MPI_Status status;
    MPI_Recv(&recv_buffer,0,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,comm,
             &status);
    procs_to_send_to[iproc] = status.MPI_SOURCE;
}
MPI_Waitall(nprocs_to_recv_from,send_requests,MPI_STATUSES_IGNORE);
```

Use of `MPI_Reduce_scatter` to implement the two-dimensional matrix-vector product. Set up separate row and column communicators with `MPI_Comm_split`, use `MPI_Reduce_scatter` to combine local products.

```
MPI_Allgather(&my_x,1,MPI_DOUBLE,
              local_x,1,MPI_DOUBLE,environ.col_comm);
MPI_Reduce_scatter(local_y,&my_y,&ione,MPI_DOUBLE,
                    MPI_SUM,environ.row_comm);
```

3.8 Barrier

A barrier call, `MPI_Barrier` (figure 3.11) is a routine that blocks all processes until they have all reached the barrier call. Thus it achieves time synchronization of the processes.

This call's simplicity is contrasted with its usefulness, which is very limited. It is almost never necessary to synchronize processes through a barrier: for most purposes it does not matter if processors are out of sync. Conversely, collectives (except the new nonblocking ones; section 3.11) introduce a barrier of sorts themselves.

3.9 Variable-size-input collectives

In the gather and scatter call above each processor received or sent an identical number of items. In many cases this is appropriate, but sometimes each processor wants or contributes an individual number of items.

Let's take the gather calls as an example. Assume that each processor does a local computation that produces a number of data elements, and this number is different for each processor (or at least not the same for all). In the regular `MPI_Gather` call the root processor had a buffer of size nP , where n is the number of elements produced on each processor, and P the number of processors. The contribution from processor p would go into locations $pn, \dots, (p + 1)n - 1$.

For the variable case, we first need to compute the total required buffer size. This can be done through a simple `MPI_Reduce` with `MPI_SUM` as reduction operator: the buffer size is $\sum_p n_p$ where n_p is the number of elements on processor p . But you can also postpone this calculation for a minute.

The next question is where the contributions of the processor will go into this buffer. For the contribution from processor p that is $\sum_{q < p} n_p, \dots, \sum_{q \leq p} n_p - 1$. To compute this, the root processor needs to have all the n_p numbers, and it can collect them with an `MPI_Gather` call.

We now have all the ingredients. All the processors specify a send buffer just as with `MPI_Gather`. However, the receive buffer specification on the root is more complicated. It now consists of:

```
outbuffer, array-of-outcounts, array-of-displacements, outtype
```

and you have just seen how to construct that information.

For example, in an `MPI_Gatherv` (figure 3.12) call each process has an individual number of items to contribute. To gather this, the root process needs to find these individual amounts with an `MPI_Gather` call, and locally construct the offsets array. Note how the offsets array has size `ntids+1`: the final offset value is automatically the total size of all incoming data. See the example below.

There are various calls where processors can have buffers of differing sizes.

- In `MPI_Scatterv` (figure 3.13) the root process has a different amount of data for each recipient.
- In `MPI_Gatherv`, conversely, each process contributes a different sized send buffer to the received result; `MPI_Allgatherv` (figure 3.14) does the same, but leaves its result on all processes; `MPI_Alltoallv` does a different variable-sized gather on each process.

3.9.1 Example of Gatherv

We use `MPI_Gatherv` to do an irregular gather onto a root. We first need an `MPI_Gather` to determine offsets.

3. MPI topic: Collectives

Figure 3.12 MPI_Gatherv

Name	Param name	Explanation	C type	F type	inout
MPI_Gatherv (
MPI_Gatherv_c (
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN	
sendcount	number of elements in send buffer	[int MPI_Count]	INTEGER	IN	
sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN	
recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT	
recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process	[const int[] MPI_Count[]]	INTEGER(*)	IN	
displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i	[const int[] MPI_Aint[]]	INTEGER(*)	IN	
recvtype	datatype of recv buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN	
root	rank of receiving process	int	INTEGER	IN	
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN	
)					

MPL:

```
template<typename T>
void gatherv
(int root_rank, const T *senddata, const layout<T> &sendl,
 T *recvdata, const layouts<T> &recvls, const displacements &recvdispls) const
(int root_rank, const T *senddata, const layout<T> &sendl,
 T *recvdata, const layouts<T> &recvls) const
(int root_rank, const T *senddata, const layout<T> &sendl ) const
```

Python:

```
Gatherv(self, sendbuf, [recvbuf,counts], int root=0)
```

Figure 3.13 MPI_Scatterv

Name	Param name	Explanation	C type	F type	inout
MPI_Scatterv (
MPI_Scatterv_c (
sendbuf	address of send buffer		const void*	TYPE(*), DIMENSION(..)	IN
sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank		[const int[] MPI_Count[]]	INTEGER(*)	IN
displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i		[const int[] MPI_Aint[]]	INTEGER(*)	IN
sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN	
recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT	
recvcount	number of elements in receive buffer	[int MPI_Count]	INTEGER	IN	
recvtype	datatype of receive buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN	
root	rank of sending process	int	INTEGER	IN	
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN	
)					

Code:

```
// gatherv.c
// we assume that each process has an
// array "localdata"
// of size "localsize"

// the root process decides how much
// data will be coming:
// allocate arrays to contain size and
// offset information
if (procno==root) {
    localsizes = (int*) malloc( nprocs*
        sizeof(int) );
    offsets = (int*) malloc( nprocs*
        sizeof(int) );
}

Victor Eijkhout
// everyone contributes their local
// size info
MPI_Gather(&localsize,1,MPI_INT,
           localsizes,1,MPI_INT,root,
           comm);
// the root constructs the offsets
// array
```

Output:

```
[examples/mpi/c] gatherv:
make[3]: `gatherv' is up to date.
TACC: Starting up job 4328411
TACC: Starting parallel tasks...
Local sizes: 13, 12, 13, 14, 11,
→12, 14, 6, 12, 8,
Collected:
0:1,1,1,1,1,1,1,1,1,1,1,1,1,1,1;
1:2,2,2,2,2,2,2,2,2,2,2,2,2,2,2;
2:3,3,3,3,3,3,3,3,3,3,3,3,3,3,3;
3:4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4;
4:5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5;
5:6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6;
6:7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7;
7:8,8,8,8,8,8;
8:9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9;
9:10,10,10,10,10,10,10,10,10,10,10;
TACC: Shutdown complete. Exiting.
```

3. MPI topic: Collectives

Figure 3.14 MPI_Allgatherv

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Allgatherv (</code>					
<code> MPI_Allgatherv_c (</code>					
<code>sendbuf</code>		starting address of send buffer	<code>const void*</code>	<code>TYPE(*), DIMENSION(..)</code>	<code>IN</code>
<code>sendcount</code>		number of elements in send buffer	<code>[int MPI_Count]</code>	<code>INTEGER</code>	<code>IN</code>
<code>sendtype</code>		datatype of send buffer elements	<code>MPI_Datatype</code>	<code>TYPE (MPI_Datatype)</code>	<code>IN</code>
<code>recvbuf</code>		address of receive buffer	<code>void*</code>	<code>TYPE(*), DIMENSION(..)</code>	<code>OUT</code>
<code>recvcounts</code>		non-negative integer array (of length group size) containing the number of elements that are received from each process	<code>[const int[] MPI_Count[]]</code>	<code>INTEGER(*)</code>	<code>IN</code>
<code>displs</code>		integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	<code>[const int[] MPI_Aint[]]</code>	<code>INTEGER(*)</code>	<code>IN</code>
<code>recvtype</code>		datatype of receive buffer elements	<code>MPI_Datatype</code>	<code>TYPE (MPI_Datatype)</code>	<code>IN</code>
<code>comm</code>		communicator	<code>MPI_Comm</code>	<code>TYPE (MPI_Comm)</code>	<code>IN</code>
<code>)</code>					

Python:

```
MPI.Comm.Allgatherv(self, sendbuf, recvbuf)
    where recvbuf = "[ array, counts, displs, type]"
```

```
## gatherv.py
# implicitly using root=0
globalsize = comm.reduce(localsize)
if procid==0:
    print("Global size=%d" % globalsize)
collecteddata = np.empty(globalsize,dtype=int)
counts = comm.gather(localsize)
comm.Gatherv(localdata, [collecteddata, counts])
```

3.9.2 Example of Allgatherv

Prior to the actual gatherv call, we need to construct the count and displacement arrays. The easiest way is to use a reduction.

```
// allgatherv.c
MPI_Allgather
( &my_count,1,MPI_INT,
  recv_counts,1,MPI_INT, comm );
int accumulate = 0;
for (int i=0; i<nprocs; i++) {
  recv_displs[i] = accumulate; accumulate += recv_counts[i]; }
int *global_array = (int*) malloc(accumulate*sizeof(int));
MPI_Allgatherv
( my_array,procno+1,MPI_INT,
  global_array,recv_counts,recv_displs,MPI_INT, comm );
```

In python the receive buffer has to contain the counts and displacements arrays.

```
## allgatherv.py
mycount = procid+1
my_array = np.empty(mycount,dtype=np.float64)

my_count = np.empty(1,dtype=int)
my_count[0] = mycount
comm.Allgather( my_count,recv_counts )

accumulate = 0
for p in range(nprocs):
  recv_displs[p] = accumulate; accumulate += recv_counts[p]
global_array = np.empty(accumulate,dtype=np.float64)
comm.Allgatherv( my_array, [global_array,recv_counts,recv_displs,MPI.DOUBLE] )
```

3.9.3 Variable all-to-all

The variable all-to-all routine `MPI_Alltoallv` is discussed in section 3.6.2.

3.10 MPI Operators

MPI *operators*, that is, objects of type `MPI_Op`, are used in reduction operators. Most common operators, such as sum or maximum, have been built into the MPI library; see section 3.10.1. It is also possible to define new operators; see section 3.10.2.

3.10.1 Pre-defined operators

The following is the list of *pre-defined operators* `MPI_Op` values.

MPI type	meaning	applies to
<code>MPI_MAX</code>	maximum	integer, floating point
<code>MPI_MIN</code>	minimum	
<code>MPI_SUM</code>	sum	integer, floating point, complex, multilanguage types
<code>MPI_REPLACE</code>	overwrite	
<code>MPI_NO_OP</code>	no change	
<code>MPI_PROD</code>	product	
<code>MPI_LAND</code>	logical and	C integer, logical
<code>MPI_LOR</code>	logical or	
<code>MPI_LXOR</code>	logical xor	
<code>MPI_BAND</code>	bitwise and	integer, byte, multilanguage types
<code>MPI_BOR</code>	bitwise or	
<code>MPI_BXOR</code>	bitwise xor	
<code>MPI_MAXLOC</code>	max value and location	<code>MPI_DOUBLE_INT</code> and such
<code>MPI_MINLOC</code>	min value and location	

3.10.1.1 Minloc and maxloc

The `MPI_MAXLOC` and `MPI_MINLOC` operations yield both the maximum and the rank on which it occurs. Their result is a `struct` of the data over which the reduction happens, and an int.

In C, the types to use in the reduction call are: `MPI_FLOAT_INT`, `MPI_LONG_INT`, `MPI_DOUBLE_INT`, `MPI_SHORT_INT`, `MPI_2INT`, `MPI_LONG_DOUBLE_INT`. Likewise, the input needs to consist of such structures: the input should be an array of such struct types, where the int is the rank of the number.

These types may have some unusual size properties:

Figure 3.15 MPI_Op_create

Name	Param name	Explanation	C type	F type	inout
MPI_Op_create					
MPI_Op_create_c					
user_fn	user defined function		[MPI_User_function*	PROCEDURE	IN
commute	true if commutative; false otherwise.	int	MPI_User_function_c*	(MPI_User_function)	
op	operation	MPI_Op*		LOGICAL	IN
)					

Python:

```
MPI.Op.create(cls,function,bool commute=False)
```

Code:

```
// longint.c
MPI_Type_size( MPI_LONG_INT,&s );
printf("MPI_LONG_INT size=%d\n",s);
MPI_Aint ss;
MPI_Type_extent( MPI_LONG_INT,&ss );
printf("MPI_LONG_INT extent=%ld\n",ss);
```

Output:

```
[examples/mpi/c] longint:
MPI_LONG_INT size=12
MPI_LONG_INT extent=16
```

Fortran note 7: Min/maxloc types. The original Fortran interface to MPI was designed around Fortran77 features, so it is not using Fortran derived types (`Type` keyword). Instead, all integer indices are stored in whatever the type is that is being reduced. The available result types are then `MPI_2REAL`, `MPI_2DOUBLE_PRECISION`, `MPI_2INTEGER`.

Likewise, the input needs to be arrays of such type. Consider this example:

```
Real*8,dimension(2,N) :: input,output
call MPI_Reduce( input,output, N, MPI_2DOUBLE_PRECISION, &
    MPI_MAXLOC, root, comm )
```

MPL note 20: Operators. Arithmetic: `plus`, `multiplies`, `max`, `min`.

Logic: `logical_and`, `logical_or`, `logical_xor`.

Bitwise: `bit_and`, `bit_or`, `bit_xor`.

3.10.2 User-defined operators

In addition to predefined operators, MPI has the possibility of *user-defined operators* to use in a reduction or scan operation.

The routine for this is `MPI_Op_create` (figure 3.15), which takes a user function and turns it into an object of type `MPI_Op`, which can then be used in any reduction:

3. MPI topic: Collectives

```
MPI_Op rwz;
MPI_Op_create(reduce_without_zero,1,&rwz);
MPI_Allreduce(data+procno,&positive_minimum,1,MPI_INT,rwz,comm);
```

Python note 11: Define reduction operator. In python, `Op.Create` is a class method for the `MPI` class.

```
rwz = MPI.Op.Create(reduceWithoutZero)
positive_minimum = np.zeros(1,dtype=intc)
comm.Allreduce(data[procid],positive_minimum,rwz);
```

The user function needs to have the following signature:

```
typedef void MPI_User_function
( void *invec, void *inoutvec, int *len,
  MPI_Datatype *datatype);

FUNCTION USER_FUNCTION( INVEC(*), INOUTVEC(*), LEN, TYPE)
<type> INVEC(LEN), INOUTVEC(LEN)
INTEGER LEN, TYPE
```

For example, here is an operator for finding the smallest nonzero number in an array of nonnegative integers:

```
*(int*)inout = m;
}
```

Python note 12: Reduction function. The python equivalent of such a function receives bare buffers as arguments. Therefore, it is best to turn them first into NumPy arrays using `np.frombuffer`:

```
## reductpositive.py
def reduceWithoutZero(in_buf, inout_buf, datatype):
    typecode = MPI._typecode(datatype)
    assert typecode is not None ## check MPI datatype is built-in
    dtype = np.dtype(typecode)

    in_array = np.frombuffer(in_buf, dtype)
    inout_array = np.frombuffer(inout_buf, dtype)

    n = in_array[0]; r = inout_array[0]
    if n==0:
        m = r
    elif r==0:
        m = n
    elif n < r:
        m = n
    else:
        m = r
    inout_array[0] = m
```

The `assert` statement accounts for the fact that this mapping of MPI datatype to NumPy dtype only works for built-in MPI datatypes.

MPL note 21: User-defined operators. A user-defined operator can be a templated class with an `operator()`.

Example:

```
// reduceuser.cxx
template<typename T>
class lcm {
public:
    T operator()(T a, T b) {
        T zero=T();
        T t((a/gcd(a, b))*b);
        if (t<zero)
            return -t;
        return t;
    }
}

comm_world.reduce(lcm<int>(), 0, v, result);
```

(The templated class can be a lambda expression)

MPL note 22: Lambda operator. You can also do the reduction by lambda:

```
comm_world.reduce
( [] (int i,int j) -> int { return i+j; },
  0,data );
```

The function has an array length argument `len`, to allow for pointwise reduction on a whole array at once. The `inoutvec` array contains partially reduced results, and is typically overwritten by the function.

There are some restrictions on the user function:

- It may not call MPI functions, except for `MPI_Abort`.
- It must be associative; it can be optionally commutative, which fact is passed to the `MPI_Op_create` call.

Exercise 3.21. Write the reduction function to implement the *one-norm* of a vector:

$$\|x\|_1 \equiv \sum_i |x_i|.$$

(There is a skeleton for this exercise under the name `onenorm`.)

The operator can be destroyed with a corresponding `MPI_Op_free`.

```
int MPI_Op_free(MPI_Op *op)
```

This sets the operator to `MPI_OP_NULL`. This is not necessary in OO languages, where the destructor takes care of it.

You can query the commutativity of an operator with `MPI_Op_commutative` (figure 3.16).

3.10.3 Local reduction

The application of an `MPI_Op` can be performed with the routine `MPI_Reduce_local` (figure 3.17). Using this routine and some send/receive scheme you can build your own global reductions. Note that this routine does not take a communicator because it is purely local.

3. MPI topic: Collectives

Figure 3.16 MPI_Op_commutative

Name	Param name	Explanation	C type	F type	inout
MPI_Op_commutative		(op operation commute true if op is commutative, false otherwise)	MPI_Op int*	TYPE(MPI_Op) LOGICAL	IN OUT

Figure 3.17 MPI_Reduce_local

Name	Param name	Explanation	C type	F type	inout
MPI_Reduce_local		(
MPI_Reduce_local_c		(
inbuf	input buffer		const void*	TYPE(*), DIMENSION(..)	IN
inoutbuf	combined input and output buffer		void*	TYPE(*), DIMENSION(..)	INOUT
count	number of elements in inbuf and inoutbuf buffers		[int MPI_Count	INTEGER	IN
datatype	datatype of elements of inbuf and inoutbuf buffers		MPI_Datatype	TYPE (MPI_Datatype)	IN
op	operation		MPI_Op	TYPE(MPI_Op)	IN
)					

3.11 Nonblocking collectives

Above you have seen how the ‘Isend’ and ‘Irecv’ routines can overlap communication with computation. This is not possible with the collectives you have seen so far: they act like blocking sends or receives. However, there are also *nonblocking collectives*, introduced in MPI-3.

Such operations can be used to increase efficiency. For instance, computing

$$y \leftarrow Ax + (x^t x)y$$

involves a matrix-vector product, which is dominated by computation in the *sparse matrix* case, and an inner product which is typically dominated by the communication cost. You would code this as

```
MPI_Iallreduce( .... x ... , &request);
// compute the matrix vector product
MPI_Wait(request);
// do the addition
```

This can also be used for 3D FFT operations [15]. Occasionally, a nonblocking collective can be used for nonobvious purposes, such as the **MPI_Ibarrier** in [16].

These have roughly the same calling sequence as their blocking counterparts, except that they output an `MPI_Request`. You can then use an `MPI_Wait` call to make sure the collective has completed.

Nonblocking collectives offer a number of performance advantages:

- Do two reductions (on the same communicator) with different operators simultaneously:

$$\begin{aligned}\alpha &\leftarrow x^t y \\ \beta &\leftarrow \|z\|_\infty\end{aligned}$$

which translates to:

```
MPI_Allreduce( &local_xy, &global_xy, 1, MPI_DOUBLE, MPI_SUM, comm );
MPI_Allreduce( &local_xinf, &global_xin, 1, MPI_DOUBLE, MPI_MAX, comm );
```

- do collectives on overlapping communicators simultaneously;
- overlap a nonblocking collective with a blocking one.

Exercise 3.22. Revisit exercise 7.1. Let only the first row and first column have certain data, which they broadcast through columns and rows respectively. Each process is now involved in two simultaneous collectives. Implement this with nonblocking broadcasts, and time the difference between a blocking and a nonblocking solution. (There is a skeleton for this exercise under the name `procgridnonblock`.)

Remark *Blocking and nonblocking don't match: either all processes call the nonblocking or all call the blocking one. Thus the following code is incorrect:*

```
if (rank==root)
    MPI_Reduce( &x /* ... */ root,comm );
else
    MPI_Ireduce( &x /* ... */ );
```

This is unlike the point-to-point behavior of nonblocking calls: you can catch a message with `MPI_Irecv` that was sent with `MPI_Send`.

Remark *Unlike sends and received, collectives have no identifying tag. With blocking collectives that does not lead to ambiguity problems. With nonblocking collectives it means that all processes need to issue them in identical order.*

List of nonblocking collectives:

- `MPI_Igather`, `MPI_Igatherv`, `MPI_Iallgather` (figure 3.18), `MPI_Iallgatherv`,
- `MPI_Iscatter`, `MPI_Iscaterv`,
- `MPI_Ireduce`, `MPI_Iallreduce` (figure 3.19), `MPI_Ireduce_scatter`, `MPI_Ireduce_scatter_block`,
- `MPI_Ialldtoall`, `MPI_Ialldtoallv`, `MPI_Ialltoallw`,
- `MPI_IBarrier`; section 3.11.2,
- `MPI_Ibcast`,
- `MPI_Iexscan`, `MPI_Iscan`,

MPL note 23: Nonblocking collectives. Nonblocking collectives have the same argument list as the corresponding blocking variant, except that instead of a `void` result, they return an `irequest`. (See 32)

Wait calls are methods of the `irequest` object.

3. MPI topic: Collectives

Figure 3.18 MPI_Iallgather

Name	Param name	Explanation	C type	F type	inout
MPI_Iallgather					
	MPI_Iallgather_c				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN	
sendcount	number of elements in send buffer	[int MPI_Count]	INTEGER	IN	
sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN	
recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT	
recvcount	number of elements received from any process	[int MPI_Count]	INTEGER	IN	
recvtype	datatype of receive buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN	
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN	
request	communication request	MPI_Request*	TYPE (MPI_Request)	OUT	
)					

Figure 3.19 MPI_Iallreduce

Name	Param name	Explanation	C type	F type	inout
MPI_Iallreduce					
	MPI_Iallreduce_c				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN	
recvbuf	starting address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT	
count	number of elements in send buffer	[int MPI_Count]	INTEGER	IN	
datatype	datatype of elements of send buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN	
op	operation	MPI_Op	TYPE(MPI_Op)	IN	
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN	
request	communication request	MPI_Request*	TYPE (MPI_Request)	OUT	
)					

```
// ireducescalar.cxx
float x{1.},sum;
auto reduce_request =
    comm_world.ireduce(mpl::plus<float>(), 0, x, sum);
reduce_request.wait();
if (comm_world.rank()==0) {
    std::cout << "sum = " << sum << '\n';
}
```

3.11.1 Examples

3.11.1.1 Array transpose

To illustrate the overlapping of multiple nonblocking collectives, consider transposing a data matrix. Initially, each process has one row of the matrix; after transposition each process has a column. Since each row needs to be distributed to all processes, algorithmically this corresponds to a series of scatter calls, one originating from each process.

```
// itransposeblock.c
for (int iproc=0; iproc<nprocs; iproc++) {
    MPI_Scatter( regular,1,MPI_DOUBLE,
                 &(transpose[iproc]),1,MPI_DOUBLE,
                 iproc,comm);
}
```

Introducing the nonblocking `MPI_Iscatter` call, this becomes:

```
MPI_Request scatter_requests[nprocs];
for (int iproc=0; iproc<nprocs; iproc++) {
    MPI_Iscatter( regular,1,MPI_DOUBLE,
                  &(transpose[iproc]),1,MPI_DOUBLE,
                  iproc,comm,scatter_requests+iproc);
}
MPI_Waitall(nprocs,scatter_requests,MPI_STATUSES_IGNORE);
```

Exercise 3.23. Can you implement the same algorithm with `MPI_Igatherv`?

3.11.1.2 Stencils

The ever-popular *five-point stencil* evaluation does not look like a collective operation, and indeed, it is usually evaluated with (nonblocking) send/recv operations. However, if we create a subcommunicator on each subdomain that contains precisely that domain and its neighbors, (see figure 3.10) we can formulate the communication pattern as a gather on each of these. With ordinary collectives this can not be formulated in a *deadlock-free* manner, but nonblocking collectives make this feasible.

We will see an even more elegant formulation of this operation in section 11.2.

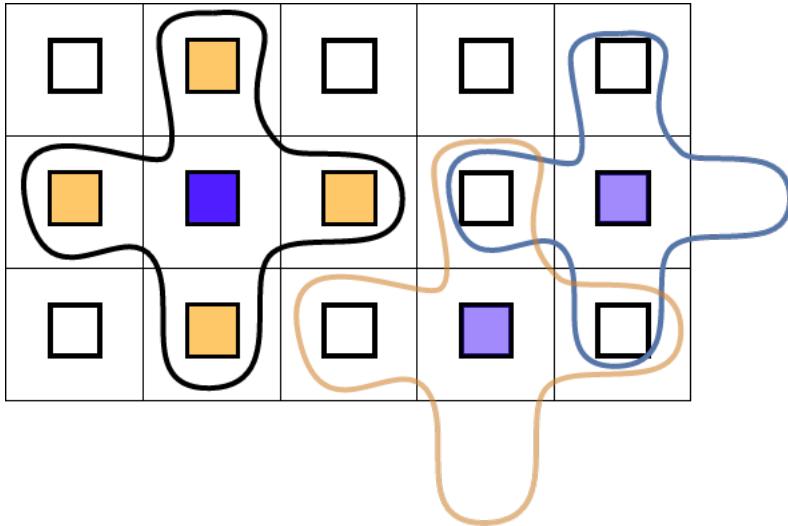


Figure 3.10: Illustration of five-point stencil gather

Figure 3.20 MPI_Ibarrier

Name	Param name	Explanation	C type	F type	inout
MPI_Ibarrier (
comm	communicator		MPI_Comm	TYPE (MPI_Comm)	IN
request	communication request		MPI_Request*	TYPE (MPI_Request)	OUT
)					

3.11.2 Nonblocking barrier

Probably the most surprising nonblocking collective is the *nonblocking barrier* **MPI_Ibarrier** (figure 3.20). The way to understand this is to think of a barrier not in terms of temporal synchronization, but state agreement: reaching a barrier is a sign that a process has attained a certain state, and leaving a barrier means that all processes are in the same state. The ordinary barrier is then a blocking wait for agreement, while with a nonblocking barrier:

- Posting the barrier means that a process has reached a certain state; and
- the request being fulfilled means that all processes have reached the barrier.

One scenario would be *local refinement*, where some processes decide to refine their subdomain, which fact they need to communicate to their neighbors. The problem here is that most processes are not among these neighbors, so they should not post a receive of any type. Instead, any refining process sends to its neighbors, and every process posts a barrier.

```
// ibarrierpoke.c
if (i_do_send) {
/*
 * Pick a random process to send to,
```

```

    * not yourself.
    */
int receiver = rand()%nprocs;
MPI_Ssend(&data,1,MPI_FLOAT,receiver,0,comm);
}
/*
 * Everyone posts the non-blocking barrier
 * and gets a request to test/wait for
 */
MPI_Request barrier_request;
MPI_Ibarrier(comm,&barrier_request);

```

Now every process alternately probes for messages and tests for completion of the barrier. Probing is done through the nonblocking `MPI_Iprobe` call, while testing completion of the barrier is done through `MPI_Test`.

```

for ( ; ; step++) {
    int barrier_done_flag=0;
    MPI_Test(&barrier_request,&barrier_done_flag,
             MPI_STATUS_IGNORE);
    //stop if you're done!
    if (barrier_done_flag) {
        break;
    } else {
        // if you're not done with the barrier:
        int flag; MPI_Status status;
        MPI_Iprobe
        ( MPI_ANY_SOURCE,MPI_ANY_TAG,
          comm, &flag, &status );
        if (flag) {
            // absorb message!

```

We can use a nonblocking barrier to good effect, utilizing the idle time that would result from a blocking barrier. In the following code fragment processes test for completion of the barrier, and failing to detect such completion, perform some local work.

```

// findbarrier.c
MPI_Request final_barrier;
MPI_Ibarrier(comm,&final_barrier);

int global_finish=mysleep;
do {
    int all_done_flag=0;
    MPI_Test(&final_barrier,&all_done_flag,MPI_STATUS_IGNORE);
    if (all_done_flag) {
        break;
    } else {
        int flag; MPI_Status status;
        // force progress
        MPI_Iprobe
        ( MPI_ANY_SOURCE,MPI_ANY_TAG,
          comm, &flag, MPI_STATUS_IGNORE );
        printf("[%d] going to work for another second\n",procid);

```

```

sleep(1);
global_finish++;
}
} while (1);

```

3.12 Performance of collectives

It is easy to visualize a broadcast as in figure 3.11: see figure 3.11. the root sends all of its data directly to

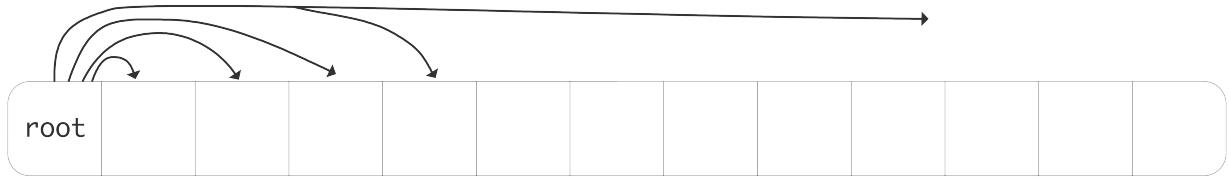


Figure 3.11: A simple broadcast

every other process. While this describes the semantics of the operation, in practice the implementation works quite differently.

The time that a message takes can simply be modeled as

$$\alpha + \beta n,$$

where α is the *latency*, a one time delay from establishing the communication between two processes, and β is the time-per-byte, or the inverse of the *bandwidth*, and n the number of bytes sent.

Under the assumption that a processor can only send one message at a time, the broadcast in figure 3.11 would take a time proportional to the number of processors.

Exercise 3.24. What is the total time required for a broadcast involving p processes? Give α and β terms separately.

One way to ameliorate that is to structure the broadcast in a tree-like fashion. This is depicted in fig-

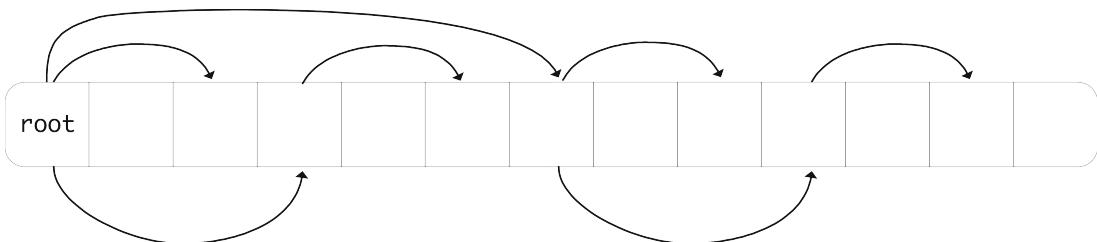


Figure 3.12: A tree-based broadcast

ure 3.12.

Exercise 3.25. How does the communication time now depend on the number of processors, again α and β terms separately.

What would be a lower bound on the α, β terms?

The theory of the complexity of collectives is described in more detail in HPC book, section ??; see also [3].

3.13 Collectives and synchronization

Collectives, other than a barrier, have a synchronizing effect between processors. For instance, in

```
MPI_Bcast( ....data.... root);
MPI_Send(....);
```

the send operations on all processors will occur after the root executes the broadcast. Conversely, in a

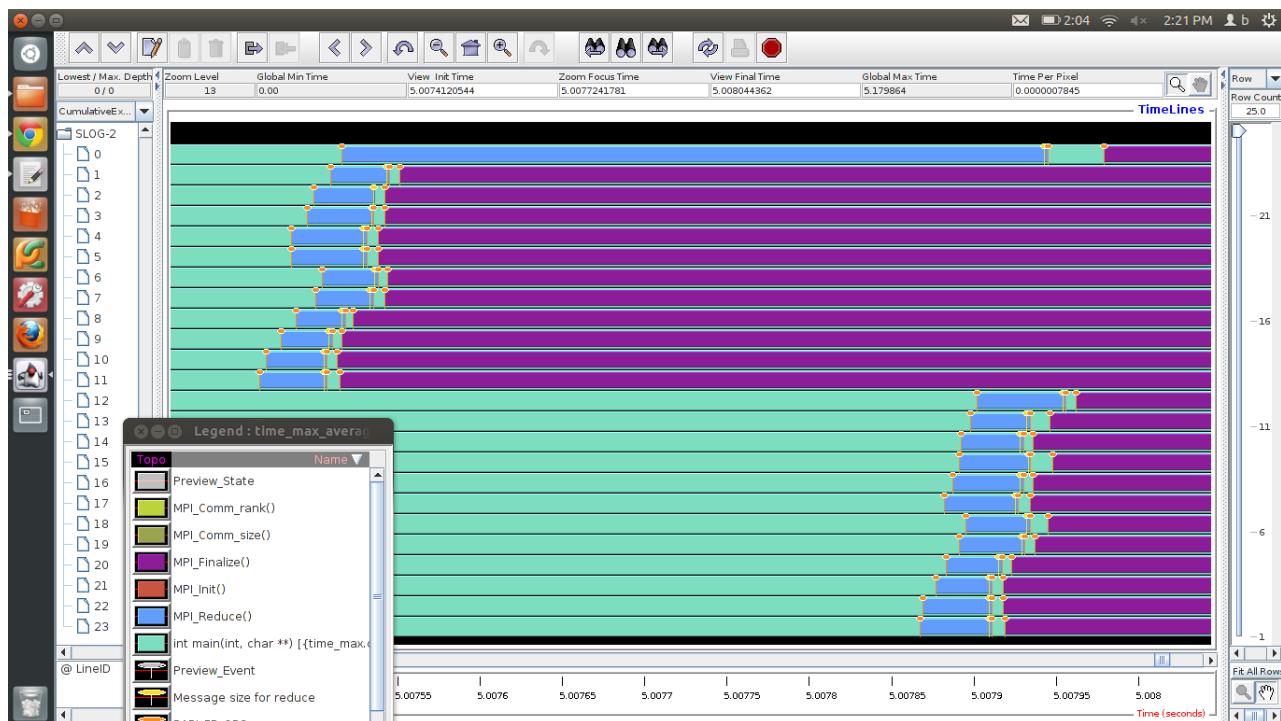


Figure 3.13: Trace of a reduction operation between two dual-socket 12-core nodes

reduce operation the root may have to wait for other processors. This is illustrated in figure 3.13, which gives a TAU trace of a reduction operation on two nodes, with two six-core sockets (processors) each. We see that¹:

- In each socket, the reduction is a linear accumulation;

1. This uses mvapich version 1.6; in version 1.9 the implementation of an on-node reduction has changed to simulate shared memory.

3. MPI topic: Collectives

- on each node, cores zero and six then combine their result;
- after which the final accumulation is done through the network.

We also see that the two nodes are not perfectly in sync, which is normal for MPI applications. As a result, core 0 on the first node will sit idle until it receives the partial result from core 12, which is on the second node.

While collectives synchronize in a loose sense, it is not possible to make any statements about events before and after the collectives between processors:

```
...event 1...
MPI_Bcast(...);
...event 2....
```

Consider a specific scenario:

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, &status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, &status);
        break;
    case 2:
        MPI_Send(buf2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

Note the `MPI_ANY_SOURCE` parameter in the receive calls on processor 1. One obvious execution of this would be:

1. The send from 2 is caught by processor 1;
2. Everyone executes the broadcast;
3. The send from 0 is caught by processor 1.

However, it is equally possible to have this execution:

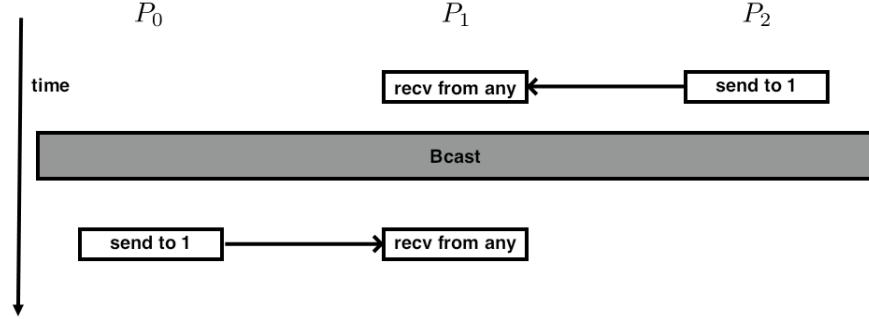
1. Processor 0 starts its broadcast, then executes the send;
2. Processor 1's receive catches the data from 0, then it executes its part of the broadcast;
3. Processor 1 catches the data sent by 2, and finally processor 2 does its part of the broadcast.

This is illustrated in figure 3.14.

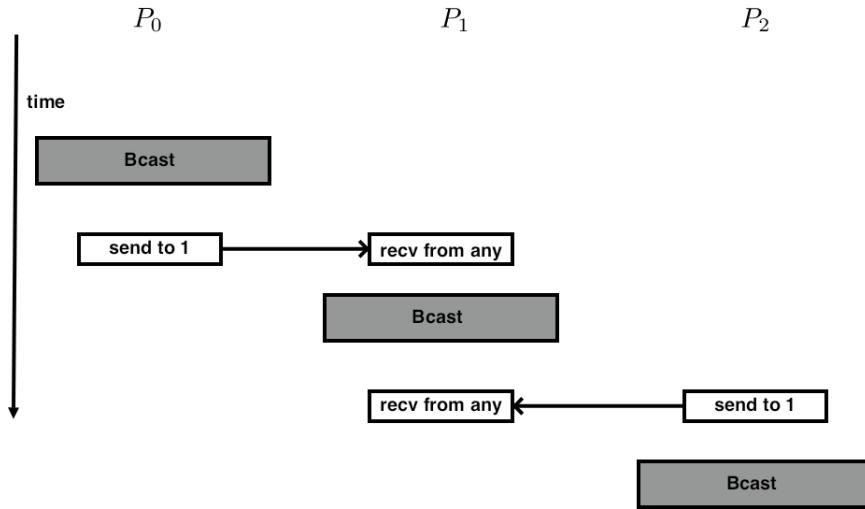
3.14 Performance considerations

In this section we will consider how collectives can be implemented in multiple ways, and the performance implications of such decisions. You can test the algorithms described here using *SimGrid* (section *Tutorials book, section ??*).

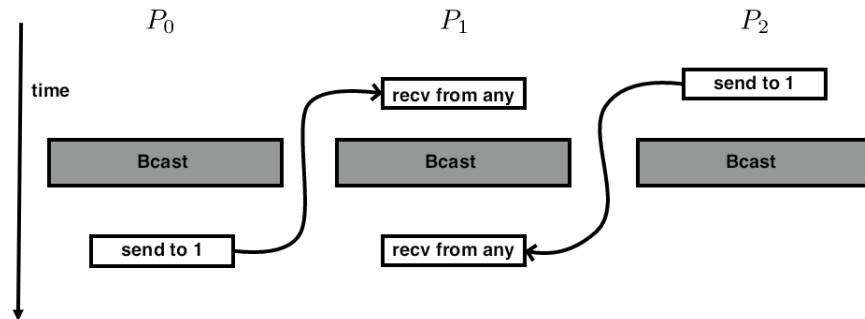
The most logical execution is:



However, this ordering is allowed too:



Which looks from a distance like:



In other words, one of the messages seems to go ‘back in time’.

Figure 3.14: Possible temporal orderings of send and collective calls

3.14.1 Scalability

We are motivated to write parallel software from two considerations. First of all, if we have a certain problem to solve which normally takes time T , then we hope that with p processors it will take time T/p . If this is true, we call our parallelization scheme *scalable in time*. In practice, we often accept small extra terms: as you will see below, parallelization often adds a term $\log_2 p$ to the running time.

Exercise 3.26. Discuss scalability of the following algorithms:

- You have an array of floating point numbers. You need to compute the sine of each
- You have a two-dimensional array, denoting the interval $[-2, 2]^2$. You want to make a picture of the *Mandelbrot set*, so you need to compute the color of each point.
- The primality test of exercise 2.6.

There is also the notion that a parallel algorithm can be *scalable in space*: more processors gives you more memory so that you can run a larger problem.

Exercise 3.27. Discuss space scalability in the context of modern processor design.

3.14.2 Complexity and scalability of collectives

3.14.2.1 Broadcast

Naive broadcast Write a broadcast operation where the root does an `MPI_Send` to each other process.

What is the expected performance of this in terms of α, β ?

Run some tests and confirm.

Simple ring Let the root only send to the next process, and that one send to its neighbor. This scheme is known as a *bucket brigade*; see also section 4.1.5.

What is the expected performance of this in terms of α, β ?

Run some tests and confirm.

Pipelined ring In a ring broadcast, each process needs to receive the whole message before it can pass it on. We can increase the efficiency by breaking up the message and sending it in multiple parts. (See figure 3.15.) This will be advantageous for messages that are long enough that the bandwidth cost dominates the latency.

Assume a send buffer of length more than 1. Divide the send buffer into a number of chunks. The root sends the chunks successively to the next process, and each process sends on whatever chunks it receives.

What is the expected performance of this in terms of α, β ? Why is this better than the simple ring?

Run some tests and confirm.

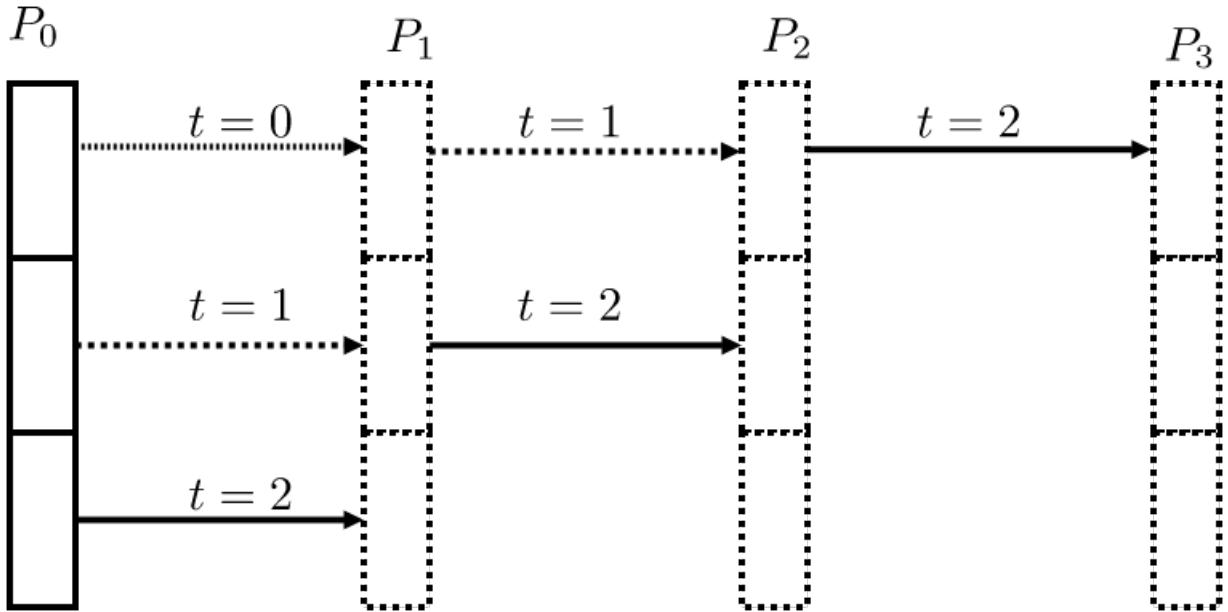


Figure 3.15: A pipelined bucket brigade

Recursive doubling Collectives such as broadcast can be *implemented* through *recursive doubling*, where the root sends to another process, then the root and the other process send to two more, those four send to four more, et cetera. However, in an actual physical architecture this scheme can be realized in multiple ways that have drastically different performance.

First consider the implementation where process 0 is the root, and it starts by sending to process 1; then they send to 2 and 3; these four send to 4–7, et cetera. If the architecture is a linear array of processors, this will lead to *contention*: multiple messages wanting to go through the same wire. (This is also related to the concept of *bisection bandwidth*.)

In the following analyses we will assume *wormhole routing*: a message sets up a path through the network, reserving the necessary wires, and performing a send in time independent of the distance through the network. That is, the send time for any message can be modeled as

$$T(n) = \alpha + \beta n$$

regardless source and destination, as long as the necessary connections are available.

Exercise 3.28. Analyze the running time of a recursive doubling broadcast as just described, with wormhole routing.

Implement this broadcast in terms of blocking MPI send and receive calls. If you have SimGrid available, run tests with a number of parameters.

The alternative, that avoids contention, is to let each doubling stage divide the network into separate halves. That is, process 0 sends to $P/2$, after which these two repeat the algorithm in the two halves of the network, sending to $P/4$ and $3P/4$ respectively.

Exercise 3.29. Analyze this variant of recursive doubling. Code it and measure runtimes on SimGrid.

Exercise 3.30. Revisit exercise 3.28 and replace the blocking calls by nonblocking `MPI_Isend / MPI_Irecv` calls.

Make sure to test that the data is correctly propagated.

MPI implementations often have multiple algorithms, which they dynamically switch between. Sometimes you can determine the choice yourself through environment variables.

TACC note. For Intel MPI, see <https://software.intel.com/en-us/mpi-developer-reference-linux-i-mpi-adjust-f>

3.15 Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

Review 3.31. How would you realize the following scenarios with MPI collectives?

- Let each process compute a random number. You want to print the maximum of these numbers to your screen.
- Each process computes a random number again. Now you want to scale these numbers by their maximum.
- Let each process compute a random number. You want to print on what processor the maximum value is computed.

Review 3.32. MPI collectives can be sorted in at least the following categories

1. rooted vs rootless
2. using uniform buffer lengths vs variable length buffers
3. blocking vs nonblocking.

Give examples of each type.

Review 3.33. True or false: collective routines are all about communicating user data between the processes.

Review 3.34. True or false: an `MPI_Scatter` call puts the same data on each process.

Review 3.35. True or false: using the option `MPI_IN_PLACE` you only need space for a send buffer in `MPI_Reduce`.

Review 3.36. True or false: using the option `MPI_IN_PLACE` you only need space for a send buffer in `MPI_Gather`.

Review 3.37. Given a distributed array, with every processor storing

```
double x[N]; // N can vary per processor
```

give the approximate MPI-based code that computes the maximum value in the array, and leaves the result on every processor.

Review 3.38.

```
double data[Nglobal];
int myfirst = /* something */ , mylast = /* something */;
for (int i=myfirst; i<mylast; i++) {
    if (i>0 && i<N-1) {
        process_point( data,i,Nglobal );
    }
}
void process_point( double *data,int i,int N ) {
    data[i-1] = g(i-1); data[i] = g(i); data[i+1] = g(i+1);
    data[i] = f(data[i-1],data[i],data[i+1]);
}
```

Is this scalable in time? Is this scalable in space? What is the missing MPI call?

Review 3.39.

```
double data[Nlocal+2]; // include left and right neighbor
int myfirst = /* something */ , mylast = myfirst+Nlocal;
for (int i=0; i<Nlocal; i++) {
    if (i>0 && i<N-1) {
        process_point( data,i,Nlocal );
    }
}
void process_point( double *data,int i0,int n ) {
    int i = i0+1;
    data[i-1] = g(i-1); data[i] = g(i); data[i+1] = g(i+1);
    data[i] = f(data[i-1],data[i],data[i+1]);
}
```

Is this scalable in time? Is this scalable in space? What is the missing MPI call?

Review 3.40. With data as in the previous question, given the code for normalizing the array, that is, scaling each element so that $\|x\|_2 = 1$.

Review 3.41. Just like `MPI_Allreduce` is equivalent to `MPI_Reduce` following by `MPI_Bcast`, `MPI_Reduce_scatter` is equivalent to at least one of the following combinations. Select those that are equivalent, and discuss differences in time or space complexity:

1. `MPI_Reduce` followed by `MPI_Scatter`;
2. `MPI_Gather` followed by `MPI_Scatter`;
3. `MPI_Allreduce` followed by `MPI_Scatter`;
4. `MPI_Allreduce` followed by a local operation (which?);
5. `MPI_Allgather` followed by a local operation (which?).

Review 3.42. Think of at least two algorithms for doing a broadcast. Compare them with regards to asymptotic behavior.

Chapter 4

MPI topic: Point-to-point

4.1 Blocking point-to-point operations

Suppose you have an array of numbers $x_i : i = 0, \dots, N$ and you want to compute

$$y_i = (x_{i-1} + x_i + x_{i+1})/3 : i = 1, \dots, N - 1.$$

As seen in figure 2.6, we give each processor a contiguous subset of the x_i s and y_i s. Let's define i_p as the first index of y that is computed by processor p . (What is the last index computed by processor p ? How many indices are computed on that processor?)

We often talk about the *owner computes* model of parallel computing: each processor ‘owns’ certain data items, and it computes their value. The values used for this computation need of course not be local, and this is where the need for communication arises.

Let's investigate how processor p goes about computing y_i for the i -values it owns. Let's assume that process p also stores the values x_i for these same indices. Now, for many values i it can evaluate the computation

$$y_i = (x_{i-1} + x_i + x_{i+1})/3$$

locally (figure 4.1).

However, there is a problem with computing y in the first index i_p on processor p :

$$y_{i_p} = (x_{i_p-1} + x_{i_p} + x_{i_p+1})/3$$

The point to the left, x_{i_p-1} , is not stored on process p (it is stored on $p-1$), so it is not immediately available for use by process p . (figure 4.2). There is a similar story with the last index that p tries to compute: that involves a value that is only present on $p+1$.

You see that there is a need for processor-to-processor, or technically *point-to-point*, information exchange. MPI realizes this through matched send and receive calls:

- One process does a send to a specific other process;
- the other process does a specific receive from that source.

We will now discuss the send and receive routines in detail.

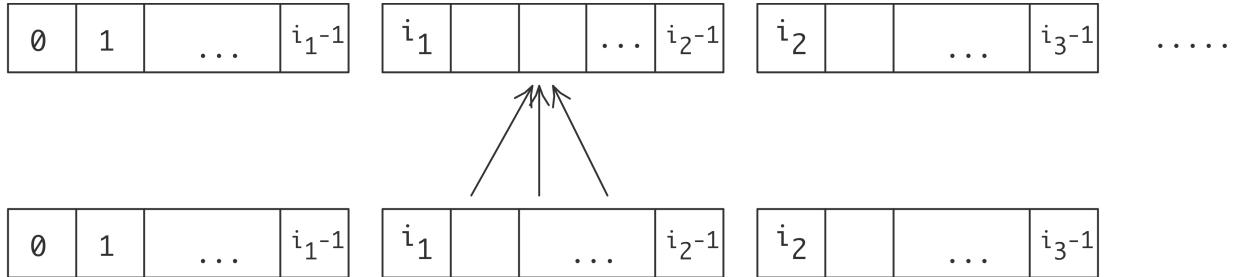


Figure 4.1: Three point averaging in parallel, case of interior points

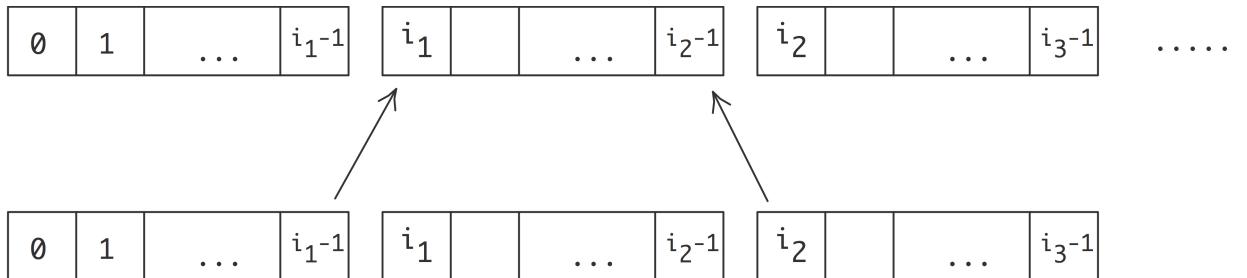


Figure 4.2: Three point averaging in parallel, case of edge points

4.1.1 Example: ping-pong

A simple scenario for information exchange between just two processes is the *ping-pong*: process A sends data to process B, which sends data back to A. This is not an operation that is particularly relevant to applications, although it is often used as a benchmark: half the time of a ping-pong is the time for a single sent message. (Why can you generally not time a single message?) Here we discuss it to explain basic ideas.

This means that process A executes the code

```
MPI_Send( /* to: */ B .... );
MPI_Recv( /* from: */ B ... );
```

while process B executes

```
MPI_Recv( /* from: */ A ... );
MPI_Send( /* to: */ A .... );
```

Since we are programming in SPMD mode, this means our program looks like:

```
if ( /* I am process A */ ) {
    MPI_Send( /* to: */ B .... );
    MPI_Recv( /* from: */ B ... );
} else if ( /* I am process B */ ) {
    MPI_Recv( /* from: */ A ... );
    MPI_Send( /* to: */ A .... );
}
```

4. MPI topic: Point-to-point

Figure 4.1 MPI_Send

Name	Param name	Explanation	C type	F type	inout
MPI_Send (
MPI_Send_c (
buf	initial address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN	
count	number of elements in send buffer	[int MPI_Count]	INTEGER	IN	
datatype	datatype of each send buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN	
dest	rank of destination	int	INTEGER	IN	
tag	message tag	int	INTEGER	IN	
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN	
)					

MPL:

```
template<typename T >
void mpl::communicator::send
( const T scalar&,int dest,tag = tag(0) ) const
( const T *buffer,const layout< T > &,int dest,tag = tag(0) ) const
( iterT begin,iterT end,int dest,tag = tag(0) ) const
T : scalar type
begin : begin iterator
end : end iterator
```

Python:

```
Python native:
MPI.Comm.send(self, obj, int dest, int tag=0)
Python numpy:
MPI.Comm.Send(self, buf, int dest, int tag=0)
```

Remark The structure of the send and receive calls shows the symmetric nature of MPI: every target process is reached with the same send call, no matter whether it's running on the same multicore chip as the sender, or on a computational node halfway across the machine room, taking several network hops to reach. Of course, any self-respecting MPI implementation optimizes for the case where sender and receiver have access to the same shared memory. This means that a send/recv pair is realized as a copy operation from the sender buffer to the receiver buffer, rather than a network transfer.

4.1.2 Send call

The blocking send command is MPI_Send (figure 4.1). Example:

```
// sendandrecv.c
double send_data = 1.;

MPI_Send
( /* send buffer/count/type: */ &send_data,1,MPI_DOUBLE,
/* to: */ receiver, /* tag: */ 0,
/* communicator: */ comm);
```

The send call has the following elements.

Buffer The *send buffer* is described by a trio of buffer/count/datatype. See section 3.2.4 for discussion.

Target The *message target* is an explicit process rank to send to. This rank is a number from zero up to the result of `MPI_Comm_size`. It is allowed for a process to send to itself, but this may lead to a runtime *deadlock*; see section 4.1.4 for discussion. The value `MPI_PROC_NULL` is allowed: using that as a target causes no message to be sent or received.

MPL note 24: Null processor. Use `mpl::proc_null`.

Tag Next, a message can have a *tag*. Many applications have each sender send only one message at a time to a given receiver. For the case where there are multiple simultaneous messages between the same sender / receiver pair, the tag can be used to disambiguate between the messages.

Often, a tag value of zero is safe to use. Indeed, OO interfaces to MPI typically have the tag as an optional parameter with value zero. If you do use tag values, you can use the key `MPI_TAG_UB` to query what the maximum value is that can be used; see section 15.1.2.

Communicator Finally, in common with the vast majority of MPI calls, there is a communicator argument that provides a context for the send transaction. In order to match a send and receive operation, they need to be in the same communicator.

MPL note 25: Buffer type safety.

- Scalar data type is handled through templating (and ‘argument-dependent-lookup’): derived by the compiler.
- Count > 1 is declared in the layout datatype.

MPL note 26: Blocking send and receive. In its scalar form, send and receive calls specify the data to be communicated and the source and target processes. MPL uses a default value for the tag, and it can deduce the type of the buffer. Sending a scalar becomes:

```
// sendscalar.cxx
if (comm_world.rank()==0) {
    double pi=3.14;
    comm_world.send(pi, 1); // send to rank 1
    cout << "sent: " << pi << '\n';
} else if (comm_world.rank()==1) {
    double pi=0;
    comm_world.recv(pi, 0); // receive from rank 0
    cout << "got : " << pi << '\n';
}
```

MPL note 27: Sending automatic arrays. MPL can send *static arrays* without further layout specification:

```
// sendarray.cxx
double v[2][2][2];
comm_world.send(v, 1); // send to rank 1
comm_world.recv(v, 0); // receive from rank 0
```

MPL note 28: Sending arrays. Sending vector data uses a pointer and a layout:

```
// sendbuffer.cxx
std::vector<double> v(8);
mpl::contiguous_layout<double> v_layout(v.size());
comm_world.send(v.data(), v_layout, 1); // send to rank 1
comm_world.recv(v.data(), v_layout, 0); // receive from rank 0
```

MPL note 29: Iterator layout. Noncontiguous iterable objects can be send with a `iterator_layout`:

```
std::list<int> v(20, 0);
mpl::iterator_layout<int> l(v.begin(), v.end());
comm_world.recv(&(*v.begin()), l, 0);
```

4.1.3 Receive call

The basic blocking receive command is `MPI_Recv` (figure 4.2).

An example:

```
double recv_data;
MPI_Recv
( /* recv buffer/count/type: */ &recv_data, 1, MPI_DOUBLE,
/* from: */ sender, /* tag: */ 0,
/* communicator: */ comm,
/* recv status: */ MPI_STATUS_IGNORE);
```

This is similar in structure to the send call, with some exceptions.

Buffer The *receive buffer* has the same buffer/count/data parameters as the send call. However, the `count` argument here indicates the size of the buffer, rather than the actual length of a message. This sets an upper bound on the length of the incoming message.

- For receiving messages with unknown length, use `MPI_Probe`; section 4.4.1.
- A message longer than the buffer size will give an overflow error, either returning an error, or ending your program; see section 15.2.2.

The length of the received message can be determined from the status object; see section 4.3 for more detail.

Figure 4.2 MPI_Recv

Name	Param name	Explanation	C type	F type	inout
MPI_Recv (
MPI_Recv_c (
buf		initial address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
count		number of elements in receive buffer	[int MPI_Count	INTEGER	IN
datatype		datatype of each receive buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
source		rank of source or MPI_ANY_SOURCE	int	INTEGER	IN
tag		message tag or MPI_ANY_TAG	int	INTEGER	IN
comm		communicator	MPI_Comm	TYPE (MPI_Comm)	IN
status		status object	MPI_Status*	TYPE (MPI_Status)	OUT
)					

MPL:

```
template<typename T >
status mpl::communicator::recv
( T &,int,tag = tag(0) ) const inline
( T *,const layout< T > &,int,tag = tag(0) ) const
( iterT begin,iterT end,int source, tag t = tag(0) ) const
```

Python:

```
Comm.Recv(self, buf, int source=ANY_SOURCE, int tag=ANY_TAG,
          Status status=None)
Python native:
recvbuf = Comm.recv(self, buf=None, int source=ANY_SOURCE, int tag=ANY_TAG,
                     Status status=None)
```

Source Mirroring the target argument of the `MPI_Send` call, `MPI_Recv` has a *message source* argument. This can be either a specific rank, or it can be the `MPI_ANY_SOURCE` wildcard. In the latter case, the actual source can be determined after the message has been received; see section 4.3. A source value of `MPI_PROC_NULL` is also allowed, which makes the receive succeed immediately with no data received.

MPL note 30: Any source. The constant `mpl::any_source` equals `MPI_ANY_SOURCE` (by `constexpr`).

Tag Similar to the messsage source, the message tag of a receive call can be a specific value or a wildcard, in this case `MPI_ANY_TAG`.

Python note 13: Message tags. Python calls sensible use a default `tag=0`, but you can specify your own tag value. On the receive call, the tag wildcard is `MPI.ANY_TAG`.

Communicator The communicator argument almost goes without remarking.

Status The `MPI_Recv` command has one parameter that the send call lacks: the `MPI_Status` object, describing the *message status*. This gives information about the message received, for instance if you used wildcards for source or tag. See section 4.3 for more about the status object.

Remark If you're not interested in the status, as is the case in many examples in this book, you can specify the constant `MPI_STATUS_IGNORE`. Note that the signature of `MPI_Recv` lists the status parameter as 'output'; this 'direction' of the parameter of course only applies if you do not specify this constant.

Exercise 4.1. Implement the ping-pong program. Add a timer using `MPI_Wtime`. For the status argument of the receive call, use `MPI_STATUS_IGNORE`.

- Run multiple ping-pongs (say a thousand) and put the timer around the loop. The first run may take longer; try to discard it.
- Run your code with the two communicating processes first on the same node, then on different nodes. Do you see a difference?
- Then modify the program to use longer messages. How does the timing increase with message size?

For bonus points, can you do a regression to determine α, β ?

(There is a skeleton for this exercise under the name `pingpong`.)

Exercise 4.2. Take your `pingpong` program and modify it to let half the processors be source and the other half the targets. Does the pingpong time increase? Does the observed behavior depend on how you choose the two sets?

4.1.4 Problems with blocking communication

You may be tempted to think that the send call puts the data somewhere in the network, and the sending code can progress after this call, as in figure 4.3, left. But this ideal scenario is not realistic: it assumes that

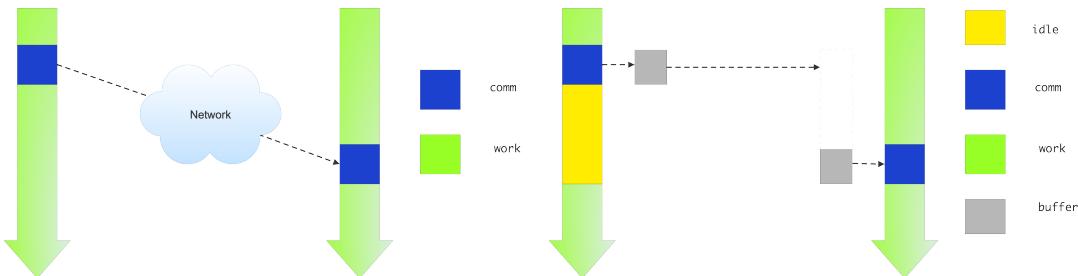


Figure 4.3: Illustration of an ideal (left) and actual (right) send-receive interaction

somewhere in the network there is buffer capacity for all messages that are in transit. This is not the case: data resides on the sender, and the sending call blocks, until the receiver has received all of it. (There is an exception for small messages, as explained in the next section.)

The use of `MPI_Send` and `MPI_Recv` is known as *blocking communication*: when your code reaches a send or receive call, it blocks until the call is successfully completed. Technically, blocking operations are called *non-local* since their execution depends on factors that are not local to the process. See section 5.4.

4.1.4.1 Deadlock

Suppose two processes need to exchange data, and consider the following pseudo-code, which purports to exchange data between processes 0 and 1:

```
other = 1-mytid; /* if I am 0, other is 1; and vice versa */
receive(source=other);
send(target=other);
```

Imagine that the two processes execute this code. They both issue the send call... and then can't go on, because they are both waiting for the other to issue the send call corresponding to their receive call. This is known as *deadlock*.

4.1.4.2 Eager vs rendezvous protocol

Messages can be sent using (at least) two different *protocols*:

1. Rendezvous protocol, and
2. Eager protocol.

The *rendezvous protocol* is the most general. Sending a message takes several steps:

1. the sender sends a header, typically containing the *message envelope*: metadata describing the message;
2. the receiver returns a 'ready-to-send' message;
3. the sender sends the actual data.

The purpose of this is to prepare the receiver buffer space for large messages. However, it implies that the sender has to wait for some return message from the receiver, making the behavior a *synchronous message*.

For the eager protocol, consider the example:

```
other = 1-mytid; /* if I am 0, other is 1; and vice versa */
send(target=other);
receive(source=other);
```

With a synchronous protocol you should get deadlock, since the send calls will be waiting for the receive operation to be posted.

In practice, however, this code will often work. The reason is that MPI implementations sometimes send small messages regardless of whether the receive has been posted. This is known as an *eager send*, and it relies on the availability of some amount of available buffer space. The size under which this behavior is used is sometimes referred to as the *eager limit*.

To illustrate eager and blocking behavior in `MPI_Send`, consider an example where we send gradually larger messages. From the screen output you can see what the largest message was that fell under the eager limit; after that the code hangs because of a deadlock.

4. MPI topic: Point-to-point

```
// sendblock.c
other = 1-procno;
/* loop over increasingly large messages */
for (int size=1; size<2000000000; size*=10) {
    sendbuf = (int*) malloc(size*sizeof(int));
    recvbuf = (int*) malloc(size*sizeof(int));
    if (!sendbuf || !recvbuf) {
        printf("Out of memory\n"); MPI_Abort(comm,1);
    }
    MPI_Send(sendbuf,size,MPI_INT,other,0,comm);
    MPI_Recv(recvbuf,size,MPI_INT,other,0,comm,&status);
    /* If control reaches this point, the send call
     did not block. If the send call blocks,
     we do not reach this point, and the program will hang.
    */
    if (procno==0)
        printf("Send did not block for size %d\n",size);
    free(sendbuf); free(recvbuf);
}

!! sendblock.F90
other = 1-mytid
size = 1
do
    allocate(sendbuf(size)); allocate(recvbuf(size))
    print *,size
    call MPI_Send(sendbuf,size,MPI_INTEGER,other,0,comm,err)
    call MPI_Recv(recvbuf,size,MPI_INTEGER,other,0,comm,status,err)
    if (mytid==0) then
        print *, "MPI_Send did not block for size",size
    end if
    deallocate(sendbuf); deallocate(recvbuf)
    size = size*10
    if (size>2000000000) goto 20
end do
20 continue

## sendblock.py
size = 1
while size<2000000000:
    sendbuf = np.empty(size, dtype=int)
    recvbuf = np.empty(size, dtype=int)
    comm.Send(sendbuf,dest=other)
    comm.Recv(recvbuf,source=other)
    if procid<other:
        print("Send did not block for",size)
    size *= 10
```

If you want a code to exhibit the same blocking behavior for all message sizes, you force the send call to be blocking by using `MPI_Ssend`, which has the same calling sequence as `MPI_Send`, but which does not allow eager sends.

```
// ssendblock.c
other = 1-procno;
sendbuf = (int*) malloc(sizeof(int));
recvbuf = (int*) malloc(sizeof(int));
size = 1;
MPI_Ssend(sendbuf, size, MPI_INT, other, 0, comm);
MPI_Recv(recvbuf, size, MPI_INT, other, 0, comm, &status);
printf("This statement is not reached\n");
```

Formally you can describe deadlock as follows. Draw up a graph where every process is a node, and draw a directed arc from process A to B if A is waiting for B. There is deadlock if this directed graph has a loop.

The solution to the deadlock in the above example is to first do the send from 0 to 1, and then from 1 to 0 (or the other way around). So the code would look like:

```
if ( /* I am processor 0 */ ) {
    send(target=other);
    receive(source=other);
} else {
    receive(source=other);
    send(target=other);
}
```

Eager sends also influences *non-blocking* sends. The wait call after a non-blocking send will return immediately, regardless any receive call, if the message is under the eager limit:

Code:
<pre>// eageri.c printf("Sending %lu elements\n",n); MPI_Request request; MPI_Isend(buffer,n,MPI_DOUBLE,processB ,0,comm,&request); MPI_Wait(&request,MPI_STATUS_IGNORE); printf(.. concluded\n");</pre>

Output:
<pre>[code/mpi/c] eageri: Setting eager limit to 5000 bytes TACC: Starting up job 4049189 TACC: Starting parallel tasks... Sending 1 elements .. concluded Sending 10 elements .. concluded Sending 100 elements .. concluded Sending 1000 elements ^C[mpiexec@c207-029.frontera.tacc.utexas.edu] ↳ Sending Ctrl-C to processes ↳ as requested</pre>

The eager limit is implementation-specific. For instance, for *Intel MPI* there is a variable *I_MPI_EAGER_THRESHOLD* (old versions) or *I_MPI_SHM_EAGER_THRESHOLD*; for *mvapich2* it is *MV2_IBA_EAGER_THRESHOLD*, and for *OpenMPI* the --mca options *btl_openib_eager_limit* and *btl_openib_rndv_eager_limit*.

4.1.4.3 *Serialization*

There is a second, even more subtle problem with blocking communication. Consider the scenario where every processor needs to pass data to its successor, that is, the processor with the next higher rank. The basic idea would be to first send to your successor, then receive from your predecessor. Since the last processor does not have a successor it skips the send, and likewise the first processor skips the receive. The pseudo-code looks like:

```
successor = mytid+1; predecessor = mytid-1;
if ( /* I am not the last processor */ )
    send(target=successor);
if ( /* I am not the first processor */ )
    receive(source=predecessor)
```

Exercise 4.3. (Classroom exercise) Each student holds a piece of paper in the right hand

– keep your left hand behind your back – and we want to execute:

1. Give the paper to your right neighbor;
2. Accept the paper from your left neighbor.

Including boundary conditions for first and last process, that becomes the following program:

1. If you are not the rightmost student, turn to the right and give the paper to your right neighbor.
2. If you are not the leftmost student, turn to your left and accept the paper from your left neighbor.

This code does not deadlock. All processors but the last one block on the send call, but the last processor executes the receive call. Thus, the processor before the last one can do its send, and subsequently continue to its receive, which enables another send, et cetera.

In one way this code does what you intended to do: it will terminate (instead of hanging forever on a deadlock) and exchange data the right way. However, the execution now suffers from unexpected *serialization*: only one processor is active at any time, so what should have been a parallel operation becomes a sequential one. This is illustrated in figure 4.4.

Exercise 4.4. Implement the above algorithm using `MPI_Send` and `MPI_Recv` calls. Run the code, and use TAU to reproduce the trace output of figure 4.4. If you don't have TAU, can you show this serialization behavior using timings, for instance running it on an increasing number of processes?
(There is a skeleton for this exercise under the name `rightsend`.)

It is possible to orchestrate your processes to get an efficient and deadlock-free execution, but doing so is a bit cumbersome.

Exercise 4.5. The above solution treated every processor equally. Can you come up with a solution that uses blocking sends and receives, but does not suffer from the serialization behavior?

There are better solutions which we will explore in the next section.

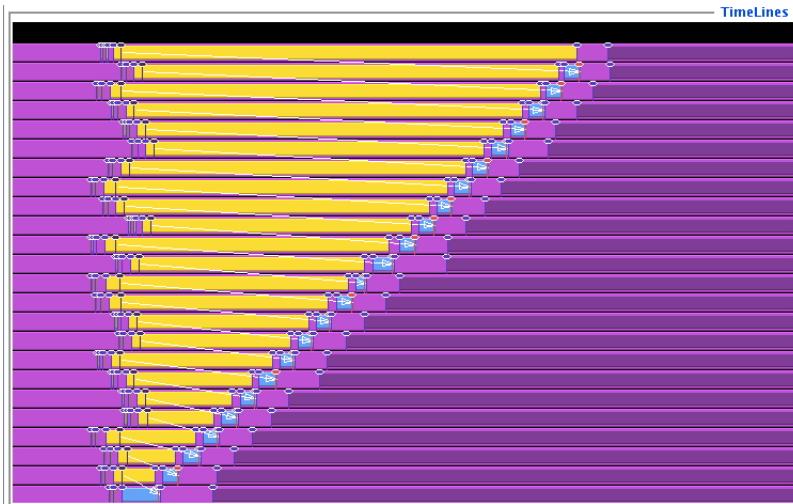


Figure 4.4: Trace of a simple send-recv code

4.1.5 Bucket brigade

The problem with the previous exercise was that an operation that was conceptually parallel became serial in execution. On the other hand, sometimes the operation is actually serial in nature. One example is the *bucket brigade* operation, where a piece of data is successively passed down a sequence of processors.

Exercise 4.6. Take the code of exercise 4.4 and modify it so that the data from process zero gets propagated to every process. Specifically, compute all partial sums $\sum_{i=0}^p i^2$:

$$\begin{cases} x_0 = 1 & \text{on process zero} \\ x_p = x_{p-1} + (p+1)^2 & \text{on process } p \end{cases}$$

Use `MPI_Send` and `MPI_Recv`; make sure to get the order right.

Food for thought: all quantities involved here are integers. Is it a good idea to use the integer datatype here?

(There is a skeleton for this exercise under the name `bucketblock`.)

Remark There is an `MPI_Scan` routine (section 3.4) that performs the same computation, but computationally more efficiently. Thus, this exercise only serves to illustrate the principle.

4.1.6 Pairwise exchange

Above you saw that with blocking sends the precise ordering of the send and receive calls is crucial. Use the wrong ordering and you get either deadlock, or something that is not efficient at all in parallel. MPI has a way out of this problem that is sufficient for many purposes: the combined send/recv call `MPI_Sendrecv` (figure 4.3).

The sendrecv call works great if every process is paired with precisely one sender and one receiver. You would then write

4. MPI topic: Point-to-point

Figure 4.3 MPI_Sendrecv

Name	Param name	Explanation	C type	F type	inout
MPI_Sendrecv (
MPI_Sendrecv_c (
sendbuf	initial address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN	
sendcount	number of elements in send buffer	[int MPI_Count]	INTEGER	IN	
sendtype	type of elements in send buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN	
dest	rank of destination	int	INTEGER	IN	
sendtag	send tag	int	INTEGER	IN	
recvbuf	initial address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT	
recvcount	number of elements in receive buffer	[int MPI_Count]	INTEGER	IN	
recvtype	type of elements receive buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN	
source	rank of source or MPI_ANY_SOURCE	int	INTEGER	IN	
recvtag	receive tag or MPI_ANY_TAG	int	INTEGER	IN	
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN	
status	status object	MPI_Status*	TYPE (MPI_Status)	OUT	
)					

MPL:

```
template<typename T >
status mpl::communicator::sendrecv
( const T & senddata, int dest,  tag sendtag,
  T & recvdata, int source, tag recvtag
) const
( const T * senddata, const layout< T > & sendl, int dest,  tag sendtag,
  T * recvdata, const layout< T > & recvl, int source, tag recvtag
) const
( iterT1 begin1, iterT1 end1, int dest,  tag sendtag,
  iterT2 begin2, iterT2 end2, int source, tag recvtag
) const
```

Python:

```
Sendrecv(self,
    sendbuf, int dest, int sendtag=0,
    recvbuf=None, int source=ANY_SOURCE, int recvtag=ANY_TAG,
    Status status=None)
```

```
sendrecv( ....from.... .to.... );
```

with the right choice of source and destination. For instance, to send data to your right neighbor:

```
MPI_Comm_rank(comm,&procno);
MPI_Sendrecv( ....
    /* from: */ procno-1
    ...
    /* to: */ procno+1
    ... );
```

This scheme is correct for all processes but the first and last. In order to use the sendrecv call on these processes, we use `MPI_PROC_NULL` for the non-existing processes that the endpoints communicate with.

```
MPI_Comm_rank( .... &procno );
if ( /* I am not the first processor */
    predecessor = procno-1;
else
    predecessor = MPI_PROC_NULL;
if ( /* I am not the last processor */
    successor = procno+1;
else
    successor = MPI_PROC_NULL;
sendrecv(from=predecessor,to=successor);
```

where the sendrecv call is executed by all processors.

All processors but the last one send to their neighbor; the target value of `MPI_PROC_NULL` for the last processor means a ‘send to the null processor’: no actual send is done.

Likewise, receiving from `MPI_PROC_NULL` succeeds without altering the receive buffer. The corresponding `MPI_Status` object has source `MPI_PROC_NULL`, tag `MPI_ANY_TAG`, and count zero.

Remark The `MPI_Sendrecv` can inter-operate with the normal send and receive calls, both blocking and non-blocking. Thus it would also be possible to replace the `MPI_Sendrecv` calls at the end points by simple sends or receives.

MPL note 31: Send-recv call. The send-recv call in MPL has the same possibilities for specifying the send and receive buffer as the separate send and recv calls: scalar, layout, iterator. However, out of the nine conceivably possible routine signatures, only the versions are available where the send and receive buffer are specified the same way. Also, the send and receive tag need to be specified; they do not have default values.

```
// sendrecv.cxx                                         // sendrecvarray.cxx
mpl::tag_t t0(0);                                     mpl::tag_t t0(0);
comm_world.sendrecv                                    mpl::contiguous_layout<double> twofloats(2);
    ( mydata,sendto,t0,
    leftdata,recvfrom,t0 );                           comm_world.sendrecv
                                                    ( mydata,twofloats,sendto,t0,
                                                    leftdata,twofloats,recvfrom,t0 );
```

Exercise 4.7. Revisit exercise 4.3 and solve it using `MPI_Sendrecv`.

If you have TAU installed, make a trace. Does it look different from the serialized send/recv code? If you don't have TAU, run your code with different numbers of processes and show that the runtime is essentially constant.

This call makes it easy to exchange data between two processors: both specify the other as both target and source. However, there need not be any such relation between target and source: it is possible to receive from a predecessor in some ordering, and send to a successor in that ordering; see figure 4.5.

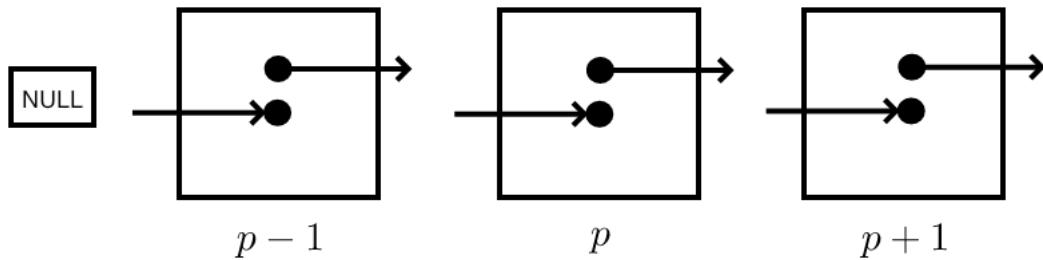


Figure 4.5: An MPI Sendrecv call

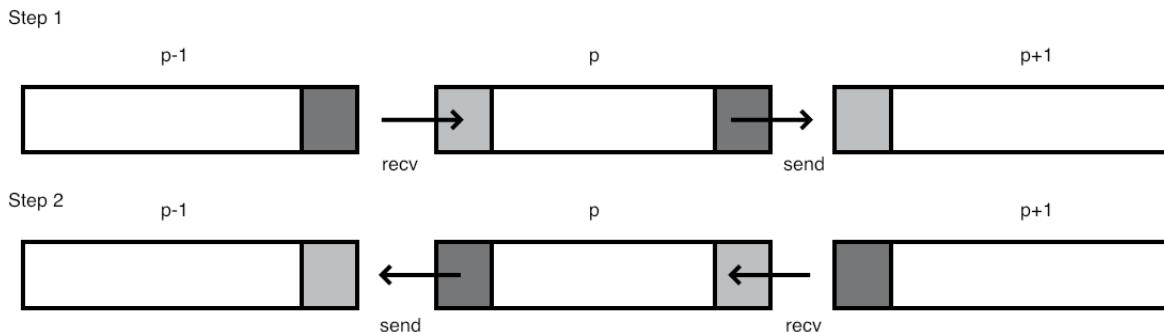


Figure 4.6: Two steps of send/recv to do a three-point combination

For the above three-point combination scheme you need to move data both left right, so you need two `MPI_Sendrecv` calls; see figure 4.6.

Exercise 4.8. Implement the above three-point combination scheme using `MPI_Sendrecv`; every processor only has a single number to send to its neighbor.
(There is a skeleton for this exercise under the name `sendrecv`.)

Hints for this exercise:

- Each process does one send and one receive; if a process needs to skip one or the other, you can specify `MPI_PROC_NULL` as the other process in the send or receive specification. In that case the corresponding action is not taken.
- As with the simple send/recv calls, processes have to match up: if process p specifies p' as the destination of the send part of the call, p' needs to specify p as the source of the recv part.

The following exercise lets you implement a sorting algorithm with the send-receive call¹.

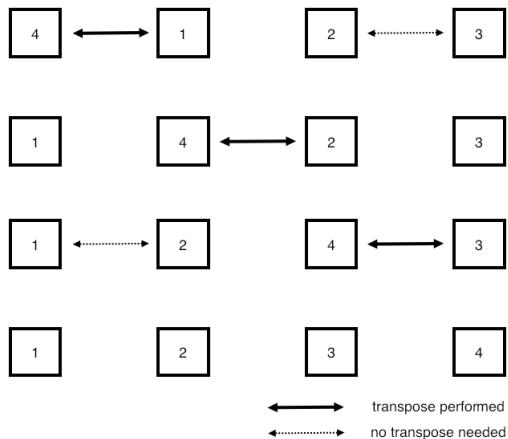


Figure 4.7: Odd-even transposition sort on 4 elements.

Exercise 4.9. A very simple sorting algorithm is *swap sort* or *odd-even transposition sort*:

pairs of processors compare data, and if necessary exchange. The elementary step is called a *compare-and-swap*: in a pair of processors each sends their data to the other; one keeps the minimum values, and the other the maximum. For simplicity, in this exercise we give each processor just a single number.

The transposition sort algorithm is split in even and odd stages, where in the even stage processors $2i$ and $2i + 1$ compare and swap data, and in the odd stage processors $2i + 1$ and $2i + 2$ compare and swap. You need to repeat this $P/2$ times, where P is the number of processors; see figure 4.7.

Implement this algorithm using `MPI_Sendrecv`. (Use `MPI_PROC_NULL` for the edge cases if needed.) Use a gather call to print the global state of the distributed array at the beginning and end of the sorting process.

Remark It is not possible to use `MPI_IN_PLACE` for the buffers, as in section 3.3.2. Instead, the routine `MPI_Sendrecv_replace` (figure 4.4) has only one buffer, used as both send and receive buffer. Of course, this requires the send and receive messages to fit in that one buffer.

Exercise 4.10. Extend this exercise to the case where each process hold an equal number of elements, more than 1. Consider figure 4.8 for inspiration. Is it coincidence that the algorithm takes the same number of steps as in the single scalar case?

The following material is for the recently released MPI-4 standard and may not be supported yet.

There are *non-blocking* and *persistent* versions of `MPI_Sendrecv`: `MPI_Isendrecv`, `MPI_Sendrecv_init`, `MPI_Isendrecv_replace`, `MPI_Sendrecv_replace_init`.

End of MPI-4 material

1. There is an `MPI_Compare_and_swap` call. Do not use that.

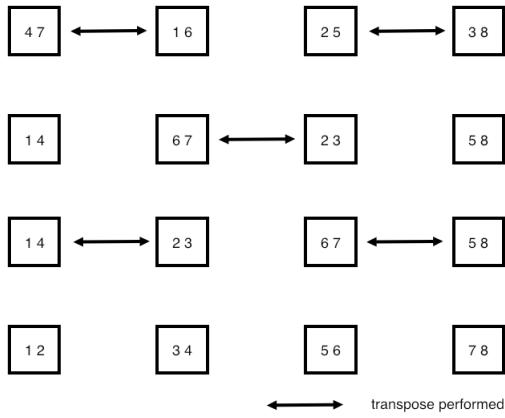


Figure 4.8: Odd-even transposition sort on 4 processes, holding 2 elements each.

4.2 Nonblocking point-to-point operations

The structure of communication is often a reflection of the structure of the operation. With some regular applications we also get a regular communication pattern. Consider again the above operation:

$$y_i = x_{i-1} + x_i + x_{i+1} : i = 1, \dots, N - 1$$

Doing this in parallel induces communication, as pictured in figure 4.1.

We note:

- The data is one-dimensional, and we have a linear ordering of the processors.
- The operation involves neighboring data points, and we communicate with neighboring processors.

Above you saw how you can use information exchange between pairs of processors

- using `MPI_Send` and `MPI_Recv`, if you are careful; or
- using `MPI_Sendrecv`, as long as there is indeed some sort of pairing of processors.

However, there are circumstances where it is not possible, not efficient, or simply not convenient, to have such a deterministic setup of the send and receive calls. Figure 4.9 illustrates such a case, where processors

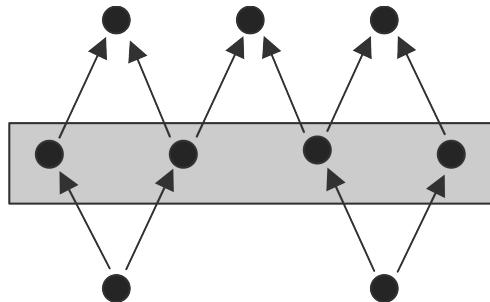


Figure 4.9: Processors with unbalanced send/receive patterns

Figure 4.4 MPI_Sendrecv_replace

Name	Param name	Explanation	C type	F type	inout
MPI_Sendrecv_replace					
	MPI_Sendrecv_replace_c				
buf		initial address of send and receive buffer	void*	TYPE(*), DIMENSION(..)	INOUT
count		number of elements in send and receive buffer	[int MPI_Count]	INTEGER	IN
datatype		type of elements in send and receive buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
dest		rank of destination	int	INTEGER	IN
sendtag		send message tag	int	INTEGER	IN
source		rank of source or MPI_ANY_SOURCE	int	INTEGER	IN
recvtag		receive message tag or MPI_ANY_TAG	int	INTEGER	IN
comm		communicator	MPI_Comm	TYPE (MPI_Comm)	IN
status		status object	MPI_Status*	TYPE (MPI_Status)	OUT
)					

are organized in a general graph pattern. Here, the numbers of sends and receive of a processor do not need to match.

In such cases, one wants a possibility to state ‘these are the expected incoming messages’, without having to wait for them in sequence. Likewise, one wants to declare the outgoing messages without having to do them in any particular sequence. Imposing any sequence on the sends and receives is likely to run into the serialization behavior observed above, or at least be inefficient.

4.2.1 Nonblocking send and receive calls

In the previous section you saw that blocking communication makes programming tricky if you want to avoid *deadlock* and performance problems. The main advantage of these routines is that you have full control about where the data is: if the send call returns the data has been successfully received, and the send buffer can be used for other purposes or de-allocated.

By contrast, the nonblocking calls `MPI_Isend` (figure 4.5) and `MPI_Irecv` (figure 4.6) (where the ‘I’ stands for ‘immediate’ or ‘incomplete’) do not wait for their counterpart: in effect they tell the runtime system ‘here is some data and please send it as follows’ or ‘here is some buffer space, and expect such-and-such data to come’. This is illustrated in figure 4.10.

```
// isendandirecv.c
double send_data = 1.;
MPI_Request request;
```

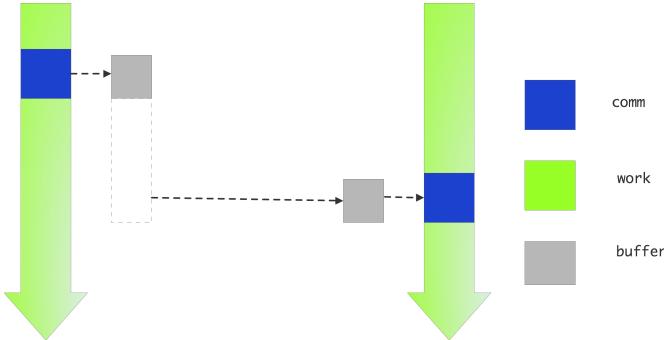


Figure 4.10: Nonblocking send

```

MPI_Isend
  ( /* send buffer/count/type: */ &send_data,1,MPI_DOUBLE,
    /* to: */ receiver, /* tag: */ 0,
    /* communicator: */ comm,
    /* request: */ &request);
MPI_Wait(&request,MPI_STATUS_IGNORE);

double recv_data;
MPI_Request request;
MPI_Irecv
  ( /* recv buffer/count/type: */ &recv_data,1,MPI_DOUBLE,
    /* from: */ sender, /* tag: */ 0,
    /* communicator: */ comm,
    /* request: */ &request);
MPI_Wait(&request,MPI_STATUS_IGNORE);

```

Issuing the **`MPI_Isend`** / **`MPI_Irecv`** call is sometimes referred to as *posting* a send/receive.

4.2.2 Request completion: wait calls

From the definition of **`MPI_Isend`** / **`MPI_Irecv`**, you seen that nonblocking routine yields an **`MPI_Request`** object. This request can then be used to query whether the operation has concluded. You may also notice that the **`MPI_Irecv`** routine does not yield an **`MPI_Status`** object. This makes sense: the status object describes the actually received data, and at the completion of the **`MPI_Irecv`** call there is no received data yet.

Waiting for the request is done with a number of routines. We first consider **`MPI_Wait`** (figure 4.7). It takes the request as input, and gives an **`MPI_Status`** as output. If you don't need the status object, you can pass **`MPI_STATUS_IGNORE`**.

```

// hangwait.c
if (procno==sender) {
  for (int p=0; p<nprocs-1; p++) {
    double send = 1.;
    MPI_Send( &send,1,MPI_DOUBLE,p,0,comm);
  }
}

```

Figure 4.5 MPI_Isend

Name	Param name	Explanation	C type	F type	inout
MPI_Isend (
MPI_Isend_c (
buf		initial address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
count		number of elements in send buffer	[int MPI_Count	INTEGER	IN
datatype		datatype of each send buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
dest		rank of destination	int	INTEGER	IN
tag		message tag	int	INTEGER	IN
comm		communicator	MPI_Comm	TYPE (MPI_Comm)	IN
request		communication request	MPI_Request*	TYPE (MPI_Request)	OUT
)					

MPL:

```
template<typename T >
irequest mpl::communicator::isend
( const T & data, int dest, tag t = tag(0) ) const;
( const T * data, const layout< T > & l, int dest, tag t = tag(0) ) const;
( iterT begin, iterT end, int dest, tag t = tag(0) ) const;
```

Python:

```
request = MPI.Comm.Isend(self, buf, int dest, int tag=0)
```

```
} else {
    double recv=0.;
    MPI_Request request;
    MPI_Irecv( &recv, 1, MPI_DOUBLE, sender, 0, comm, &request );
    do_some_work();
    MPI_Wait(&request, MPI_STATUS_IGNORE);
}
```

(Note that this example uses a mix of blocking and non-blocking operations: a blocking send is paired with a non-blocking receive.)

The request is passed by reference, so that the wait routine can free it:

- The wait call deallocates the request object, and
- sets the value of the variable to `MPI_REQUEST_NULL`.

(See section 4.2.4 for details.)

MPL note 32: Requests from nonblocking calls. Nonblocking routines have an `irequest` as function result.

Note: not a parameter passed by reference, as in the C interface. The various wait calls are methods of the `irequest` class.

4. MPI topic: Point-to-point

Figure 4.6 MPI_Irecv

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Irecv (</code>					
<code> MPI_Irecv_c (</code>					
buf		initial address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
count		number of elements in receive buffer	[int MPI_Count	INTEGER	IN
datatype		datatype of each receive buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
source		rank of source or MPI_ANY_SOURCE	int	INTEGER	IN
tag		message tag or MPI_ANY_TAG	int	INTEGER	IN
comm		communicator	MPI_Comm	TYPE (MPI_Comm)	IN
request		communication request	MPI_Request*	TYPE (MPI_Request)	OUT
<code>)</code>					

MPL:

```
template<typename T >
irequest mpl::communicator::irecv
( const T & data, int src, tag t = tag(0) ) const;
( const T * data, const layout< T > & l, int src, tag t = tag(0) ) const;
( iterT begin, iterT end, int src, tag t = tag(0) ) const;
```

Python:

```
recvbuf = Comm.irecv(self, buf=None, int source=ANY_SOURCE, int tag=ANY_TAG,
Request request=None)
Python numpy:
Comm.Irecv(self, buf, int source=ANY_SOURCE, int tag=ANY_TAG,
Request status=None)
```

Figure 4.7 MPI_Wait

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Wait (</code>					
request		request	MPI_Request*	TYPE (MPI_Request)	INOUT
status		status object	MPI_Status*	TYPE (MPI_Status)	OUT
<code>)</code>					

Python:

```
MPI.Request.Wait(type cls, request, status=None)
```

```
double recv_data;
mpl::irequest recv_request =
    comm_world.irecv( recv_data, sender );
recv_request.wait();
```

You can not default-construct the request variable:

```
// DOES NOT COMPILE:
mpl::irequest recv_request;
recv_request = comm.irecv( ... );
```

This means that the normal sequence of first declaring, and then filling in, the request variable is not possible.

Implementation note: The wait call always returns a `status_t` object; not assigning it means that the destructor is called on it.

Now we discuss in some detail the various wait calls. These are blocking; for the nonblocking versions see section 4.2.3.

4.2.2.1 Wait for one request

`MPI_Wait` waits for a single request. If you are indeed waiting for a single nonblocking communication to complete, this is the right routine. If you are waiting for multiple requests you could call this routine in a loop.

```
for (p=0; p<nrequests ; p++) // Not efficient!
    MPI_Wait(&request[p],&(status[p]));
```

However, this would be inefficient if the first request is fulfilled much later than the others: your waiting process would have lots of idle time. In that case, use one of the following routines.

4.2.2.2 Wait for all requests

`MPI_Waitall` (figure 4.8) allows you to wait for a number of requests, and it does not matter in what sequence they are satisfied. Using this routine is easier to code than the loop above, and it could be more efficient.

```
// irecvloop.c
MPI_Request requests =
    (MPI_Request*) malloc( 2*nprocs*sizeof(MPI_Request) );
recv_buffers = (int*) malloc( nprocs*sizeof(int) );
send_buffers = (int*) malloc( nprocs*sizeof(int) );
for (int p=0; p<nprocs; p++) {
    int
        left_p = (p-1+nprocs) % nprocs,
        right_p = (p+1) % nprocs;
    send_buffer[p] = nprocs-p;
    MPI_Isend(sendbuffer+p,1,MPI_INT, right_p,0, requests+2*p);
    MPI_Irecv(recvbuffer+p,1,MPI_INT, left_p,0, requests+2*p+1);
}
/* your useful code here */
MPI_Waitall(2*nprocs,requests,MPI_STATUSES_IGNORE);
```

Figure 4.8 MPI_Waitall

Name	Param name	Explanation	C type	F type	inout
MPI_Waitall (

Python:

```
MPI.Request.Waitall(type cls, requests, statuses=None)
```

The output argument is an array or `MPI_Status` object. If you don't need the status objects, you can pass `MPI_STATUSES_IGNORE`.

As an illustration, we realize exercise 4.4, and its trace in figure 4.4, with non-blocking execution and `MPI_Waitall`. Figure 4.11 shows the trace of this variant of the code.

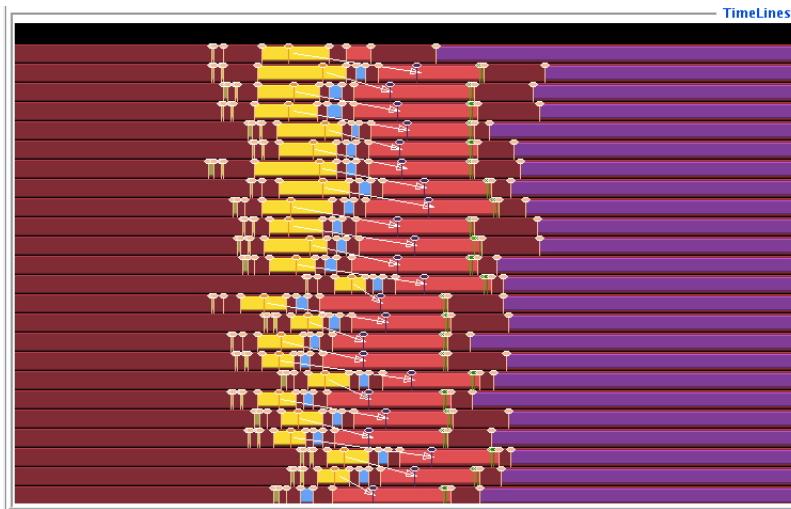


Figure 4.11: A trace of a nonblocking send between neighboring processors

Exercise 4.11. Revisit exercise 4.6 and consider replacing the blocking calls by nonblocking ones. How far apart can you put the `MPI_Isend` / `MPI_Irecv` calls and the corresponding `MPI_Waits`?
(There is a skeleton for this exercise under the name `bucketskipnonblock`.)

Exercise 4.12. Create two distributed arrays of positive integers. Take the set difference of the two: the first array needs to be transformed to remove from it those numbers that are in the second array.

How could you solve this with an `MPI_Allgather` call? Why is it not a good idea to do so? Solve this exercise instead with a circular bucket brigade algorithm.
(There is a skeleton for this exercise under the name `setdiff`.)

Python note 14: Handling a single request. Non-blocking routines such as `MPI_Isend` return a request object. The `MPI_Wait` is a class method, not a method of the request object:

```
## irecvsingle.py
sendbuffer = np.empty( nprocs, dtype=int )
recvbuffer = np.empty( nprocs, dtype=int )

left_p = (procid-1) % nprocs
right_p = (procid+1) % nprocs
send_request = comm.Isend\
    ( sendbuffer[procid:procid+1], dest=left_p )
recv_request = comm.Irecv\
    ( recvbuffer[procid:procid+1], source=right_p )
MPI.Request.Wait(send_request)
MPI.Request.Wait(recv_request)
```

Python note 15: Request arrays. An array of requests (for the waitall/some/any calls) is an ordinary Python list:

```
## irecvloop.py
requests = []
sendbuffer = np.empty( nprocs, dtype=int )
recvbuffer = np.empty( nprocs, dtype=int )

for p in range(nprocs):
    left_p = (p-1) % nprocs
    right_p = (p+1) % nprocs
    requests.append( comm.Isend\
        ( sendbuffer[p:p+1], dest=left_p ) )
    requests.append( comm.Irecv\
        ( recvbuffer[p:p+1], source=right_p ) )
MPI.Request.Waitall(requests)
```

The `MPI_Waitall` method is again a class method.

4.2.2.3 Wait for any requests

The ‘waitall’ routine is good if you need all nonblocking communications to be finished before you can proceed with the rest of the program. However, sometimes it is possible to take action as each request is satisfied. In that case you could use `MPI_Waitany` (figure 4.9) and write:

```
for (p=0; p<nrequests; p++) {
    MPI_Irecv(buffer+index, /* ... */, requests+index);
}
for (p=0; p<nrequests; p++) {
    MPI_Waitany(nrequests, request_array, &index, &status);
    // operate on buffer[index]
}
```

4. MPI topic: Point-to-point

Figure 4.9 MPI_Waitany

Name	Param name	Explanation	C type	F type	inout
MPI_Waitany (
count		list length	int	INTEGER	IN
array_of_requests		array of requests	MPI_Request []	TYPE (MPI_Request) (count)	INOUT
index		index of handle for operation that completed	int*	INTEGER	OUT
status		status object	MPI_Status*	TYPE (MPI_Status)	OUT
)					

Python:

```
MPI.Request.Waitany( requests,status=None )
class method, returns index
```

Note that this routine takes a single status argument, passed by reference, and not an array of statuses!

Fortran note 8: Index of requests. The index parameter is the index in the array of requests, which is a Fortran array, so it uses *1-based indexing*.

```
!! irecvsource.F90
if (mytid==ntids-1) then
    do p=1,ntids-1
        print *, "post"
        call MPI_Irecv(recv_buffer(p),1,MPI_INTEGER,p-1,0,comm,&
                      requests(p),err)
    end do
    do p=1,ntids-1
        call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE,err)
        write(*,'("Message from",i3,":",i5)') index,recv_buffer(index)
    end do

!! waitnull.F90
Type(MPI_Request),dimension(:),allocatable :: requests
allocate(requests(ntids-1))
call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE)
if ( .not. requests(index)==MPI_REQUEST_NULL) then
    print *, "This request should be null:",index

!! waitnull.F90
Type(MPI_Request),dimension(:),allocatable :: requests
allocate(requests(ntids-1))
call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE)
if ( .not. requests(index)==MPI_REQUEST_NULL) then
    print *, "This request should be null:",index
```

MPL note 33: Request pools. Instead of an array of requests, use an `irequest_pool` object, which acts like a vector of requests, meaning that you can *push* onto it.

```
// irecvsource.cxx
mpl::irequest_pool recv_requests;
for (int p=0; p<nprocs-1; p++) {
    recv_requests.push( comm_world.irecv( recv_buffer[p], p ) );
}
```

You can not declare a pool of a fixed size and assign elements. (Why not? Can you find a way around it?)

MPL note 34: Request pool status. Commands such as `waitall` are methods on the pool; they don't return statuses; statuses are found by indexing:

```
// irecvall.cxx
recv_requests.waitall();
auto status_last = recv_requests.get_status( recv_requests.size()-1 );
```

MPL note 35: Wait any. The `irequest_pool` class has methods `waitany`, `waitall`, `testany`, `testall`, `waitsome`, `testsome`.

The ‘any’ methods return a `std::pair<mpl::test_result, size_t>`, where the `test_result` is an `enum class` with values:

- `completed` (for any/some/all completions),
- `no_completed` (for none),
- `no_active_requests` (if no more requests active).

```
auto [success,index] = recv_requests.waitany();
if ( success==mpl::test_result::completed ) {
    auto recv_status = recv_requests.get_status(index);
```

MPL note 36: Request handling.

```
auto [success,index] = recv_requests.waitany();
if ( success==mpl::test_result::completed ) {
    auto recv_status = recv_requests.get_status(index);
```

4.2.2.4 Polling with MPI Wait any

The `MPI_Waitany` routine can be used to implement *polling*: occasionally check for incoming messages while other work is going on.

4. MPI topic: Point-to-point

```

Code:
// irecvsource.c
if (procno==nprocs-1) {
    int *recv_buffer;
    MPI_Request *request; MPI_Status
    status;
    recv_buffer = (int*) malloc((nprocs
        -1)*sizeof(int));
    request = (MPI_Request*) malloc
        ((nprocs-1)*sizeof(MPI_Request));

    for (int p=0; p<nprocs-1; p++) {
        ierr = MPI_Irecv(recv_buffer+p,1,
            MPI_INT, p,0,comm,
            request+p); CHK(
            ierr);
    }
    for (int p=0; p<nprocs-1; p++) {
        int index, sender;
        MPI_Waitany(nprocs-1,request,&index
            ,&status);
        if (index!=status.MPI_SOURCE)
            printf("Mismatch index %d vs
                source %d\n",
                index,status.MPI_SOURCE);
        printf("Message from %d: %d\n",
                index,recv_buffer[index]);
    }
} else {
    ierr = MPI_Send(&procno,1,MPI_INT,
        nprocs-1,0,comm);
}

```

```

Output:
[examples/mpi/c] irecvsource:
make[3]: `irecvsource' is up to
    date.
process 1 waits 6s before sending
process 2 waits 3s before sending
process 0 waits 13s before sending
process 3 waits 8s before sending
process 5 waits 1s before sending
process 6 waits 14s before sending
process 4 waits 12s before sending
Message from 5: 5
Message from 2: 2
Message from 1: 1
Message from 3: 3
Message from 4: 4
Message from 0: 0
Message from 6: 6

```

```

## irecvsource.py
if procid==nprocs-1:
    receive_buffer = np.empty(nprocs-1,dtype=int)
    requests = [ None ] * (nprocs-1)
    for sender in range(nprocs-1):
        requests[sender] = comm.Irecv(receive_buffer[sender:sender+1],source=sender)
    # alternatively: requests = [ comm.Irecv(s) for s in .... ]
    status = MPI.Status()
    for sender in range(nprocs-1):
        ind = MPI.Request.Waitany(requests,status=status)
        if ind!=status.Get_source():
            print("sender mismatch: %d vs %d" % (ind,status.Get_source()))
            print("received from",ind)
else:
    mywait = random.randint(1,2*nprocs)
    print("[%d] wait for %d seconds" % (procid,mywait))
    time.sleep(mywait)
    mydata = np.empty(1,dtype=int)

```

```
mydata[0] = procid
comm.Send([mydata,MPI.INT],dest=nprocs-1)
```

Each process except for the root does a blocking send; the root posts `MPI_Irecv` from all other processors, then loops with `MPI_Waitany` until all requests have come in. Use `MPI_SOURCE` to test the index parameter of the wait call.

Note the `MPI_STATUS_IGNORE` parameter: we know everything about the incoming message, so we do not need to query a status object. Contrast this with the example in section 4.3.1.

4.2.2.5 Wait for some requests

Finally, `MPI_Waitsome` is very much like `MPI_Waitany`, except that it returns multiple numbers, if multiple requests are satisfied. Now the status argument is an array of `MPI_Status` objects.

4.2.2.6 Receive status of the wait calls

The `MPI_Wait...` routines have the `MPI_Status` objects as output. If you are not interested in the status information, you can use the values `MPI_STATUS_IGNORE` for `MPI_Wait` and `MPI_Waitany`, or `MPI_STATUSES_IGNORE` for `MPI_Waitall`, `MPI_Waitsome`, `MPI_Testall`, `MPI_Testsome`.

Remark *The routines that can return multiple statuses, can return the error condition `MPI_ERR_IN_STATUS`, indicating that one of the statuses was in error. See section 4.3.3.*

Exercise 4.13.

(There is a skeleton for this exercise under the name `isendirecv`.) Now use nonblocking send/receive routines to implement the three-point averaging operation

$$y_i = (x_{i-1} + x_i + x_{i+1})/3 : i = 1, \dots, N - 1$$

on a distributed array. There are two approaches to the first and last process:

1. you can use `MPI_PROC_NULL` for the ‘missing’ communications;
2. you can skip these communications altogether, but now you have to count the requests carefully.

4.2.2.7 Latency hiding / overlapping communication and computation

There is a second motivation for the `Irecv` calls: if your hardware supports it, the communication can happen while your program can continue to do useful work:

```
// start nonblocking communication
MPI_Isend( ... ); MPI_Irecv( ... );
// do work that does not depend on incoming data
...
// wait for the Isend/Irecv calls to finish
MPI_Wait( ... );
// now do the work that absolutely needs the incoming data
...
```

This is known as *overlapping computation and communication*, or *latency hiding*. See also *asynchronous progress*; section 15.4.

Unfortunately, a lot of this communication involves activity in user space, so the solution would have been to let it be handled by a separate thread. Until recently, processors were not efficient at doing such multi-threading, so true overlap stayed a promise for the future. Some network cards have support for this overlap, but it requires a nontrivial combination of hardware, firmware, and MPI implementation.

Exercise 4.14.

(There is a skeleton for this exercise under the name `isendirecvarray`.) Take your code of exercise 4.13 and modify it to use latency hiding. Operations that can be performed without needing data from neighbors should be performed in between the `MPI_Isend` / `MPI_Irecv` calls and the corresponding `MPI_Wait` calls.

Remark You have now seen various send types: blocking, nonblocking, synchronous. Can a receiver see what kind of message was sent? Are different receive routines needed? The answer is that, on the receiving end, there is nothing to distinguish a nonblocking or synchronous message. The `MPI_Recv` call can match any of the send routines you have seen so far, and conversely a message sent with `MPI_Send` can be received by `MPI_Irecv`.

4.2.2.8 Buffer issues in nonblocking communication

While the use of nonblocking routines prevents deadlock, it introduces problems of its own.

- With a blocking send call, you could repeatedly fill the send buffer and send it off.

```
double *buffer;
for ( ... p ... ) {
    buffer = // fill in the data
    MPI_Send( buffer, ... /* to: */ p );
```

- On the other hand, when a nonblocking send call returns, the actual send may not have been executed, so the send buffer may not be safe to overwrite. Similarly, when the recv call returns, you do not know for sure that the expected data is in it. Only after the corresponding wait call are you sure that the buffer has been sent, or has received its contents.
- To send multiple messages with nonblocking calls you therefore have to allocate multiple buffers.

```
double **buffers;
for ( ... p ... ) {
    buffers[p] = // fill in the data
    MPI_Send( buffers[p], ... /* to: */ p );
}
MPI_Wait( /* the requests */ );

// irecvloop.c
MPI_Request requests =
    (MPI_Request*) malloc( 2*nprocs*sizeof(MPI_Request) );
recv_buffers = (int*) malloc( nprocs*sizeof(int) );
send_buffers = (int*) malloc( nprocs*sizeof(int) );
for (int p=0; p<nprocs; p++) {
    int
    left_p = (p-1+nprocs) % nprocs,
```

```

    right_p = (p+1) % nprocs;
    send_buffer[p] = nprocs-p;
    MPI_Isend(sendbuffer+p, 1, MPI_INT, right_p, 0, requests+2*p);
    MPI_Irecv(recvbuffer+p, 1, MPI_INT, left_p, 0, requests+2*p+1);
}
/* your useful code here */
MPI_Waitall(2*nprocs, requests, MPI_STATUSES_IGNORE);

```

4.2.3 Wait and test calls

The `MPI_Wait`... routines are blocking. Thus, they are a good solution if the receiving process can not do anything until the data (or at least *some* data) is actually received. The `MPI_Test`... calls are themselves nonblocking: they test for whether one or more requests have been fulfilled, but otherwise immediately return. It is also a *local operation*: it does not force progress.

Remark The `MPI_Test`... routines are similar to, but different from `MPI_Probe`, which is blocking and forces progress; see section 4.4.1.

The `MPI_Test` call can be used in the *manager-worker* model: the manager process creates tasks, and sends them to whichever worker process has finished its work. (This uses a receive from `MPI_ANY_SOURCE`, and a subsequent test on the `MPI_SOURCE` field of the receive status.) While waiting for the workers, the manager can do useful work too, which requires a periodic check on incoming message.

Pseudo-code:

```

while ( not done ) {
    // create new inputs for a while
    ...
    // see if anyone has finished
    MPI_Test( .... &index, &flag );
    if ( flag ) {
        // receive processed data and send new
    }
}

```

If the test is true, the request is deallocated and set to `MPI_REQUEST_NULL`, or, in the case of an active persistent request (section 5.1), set to inactive.

Analogous to `MPI_Wait`, `MPI_Waitany`, `MPI_Waitall`, `MPI_Waitsome`, there are `MPI_Test` (figure 4.10), `MPI_Testany`, `MPI_Testall`, `MPI_Testsome`.

Exercise 4.15. Read section HPC book, section ?? and give pseudo-code for the distributed sparse matrix-vector product using the above idiom for using `MPI_Test`... calls.

Discuss the advantages and disadvantages of this approach. The answer is not going to be black and white: discuss when you expect which approach to be preferable.

4.2.4 More about requests

Every nonblocking call allocates an `MPI_Request` object. Unlike `MPI_Status`, an `MPI_Request` variable is not actually an object, but instead it is an (opaque) pointer. This means that when you call, for instance, `MPI_Irecv`, MPI will allocate an actual request object, and return its address in the `MPI_Request` variable.

4. MPI topic: Point-to-point

Figure 4.10 MPI_Test

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Test (</code>					
	<code>request</code>	communication request	<code>MPI_Request*</code>	<code>TYPE (MPI_Request)</code>	<code>INOUT</code>
	<code>flag</code>	true if operation completed	<code>int*</code>	<code>LOGICAL</code>	<code>OUT</code>
	<code>status</code>	status object	<code>MPI_Status*</code>	<code>TYPE (MPI_Status)</code>	<code>OUT</code>
<code>)</code>					

Python:

```
request.Test()
```

Figure 4.11 MPI_Request_free

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Request_free (</code>					
	<code>request</code>	communication request	<code>MPI_Request*</code>	<code>TYPE (MPI_Request)</code>	<code>INOUT</code>
<code>)</code>					

Correspondingly, calls to `MPI_Wait` or `MPI_Test` free this object, setting the handle to `MPI_REQUEST_NULL`. (There is an exception for persistent communications where the request is only set to ‘inactive’; section 5.1.) Thus, it is wise to issue wait calls even if you know that the operation has succeeded. For instance, if all receive calls are concluded, you know that the corresponding send calls are finished and there is no strict need to wait for their requests. However, omitting the wait calls would lead to a *memory leak*.

Another way around this is to call `MPI_Request_free` (figure 4.11), which sets the request variable to `MPI_REQUEST_NULL`, and marks the object for deallocation after completion of the operation. Conceivably, one could issue a nonblocking call, and immediately call `MPI_Request_free`, dispensing with any wait call. However, this makes it hard to know when the operation is concluded and when the buffer is safe to reuse [26].

You can inspect the status of a request without freeing the request object with `MPI_Request_get_status` (figure 4.12). For multiple statuses use `MPI_Request_get_status_all`, `MPI_Request_get_status_some`, `MPI_Request_get_status_any` in MPI-4.1.

4.3 The Status object and wildcards

In section 4.1.1 you saw that `MPI_Recv` has a ‘status’ argument of type `MPI_Status` that `MPI_Send` lacks. (The various `MPI_Wait...` routines also have a status argument; see section 4.2.1.) Often you specify `MPI_STATUS_IGNORE` for this argument: commonly you know what data is coming in and where it is coming from.

Figure 4.12 MPI_Request_get_status

Name	Param name	Explanation	C type	F type	inout
MPI_Request_get_status (
request	request		MPI_Request	TYPE (MPI_Request)	IN
flag	boolean flag, same as from MPI_TEST		int*	LOGICAL	OUT
status	status object if flag is true		MPI_Status*	TYPE (MPI_Status)	OUT
)					

However, in some circumstances the recipient may not know all details of a message when you make the receive call, so MPI has a way of querying the *status* of the message:

- If you are expecting multiple incoming messages, it may be most efficient to deal with them in the order in which they arrive. So, instead of waiting for a specific message, you would specify `MPI_ANY_SOURCE` or `MPI_ANY_TAG` in the description of the receive message. Now you have to be able to ask ‘who did this message come from, and what is in it’.
- Maybe you know the sender of a message, but the amount of data is unknown. In that case you can overallocate your receive buffer, and after the message is received ask how big it was, or you can ‘probe’ an incoming message (see section 4.4.1) and allocate enough data when you find out how much data is being sent.

To do this, the receive call has a `MPI_Status` parameter. The `MPI_Status` object is a structure (in C a `struct`, in F90 an array, in F2008 a derived type) with freely accessible members:

- `MPI_SOURCE` gives the source of the message; see section 4.3.1.
- `MPI_TAG` gives the tag with which the message was received; see section 4.3.2.
- `MPI_ERROR` gives the error status of the receive call; see section 4.3.3.
- The number of items in the message can be deduced from the status object, not as a structure member, but through a function call to `MPI_Get_count`; see section 4.3.4.

Fortran note 9: Status object in f08. The `mpi_f08` module turns many handles (such as communicators) from Fortran `Integers` into `Types`. Retrieving the integer from the type is usually done through the `%val` member, but for the status object this is more difficult. The routines `MPI_Status_f2f08` and `MPI_Status_f082f` convert between these. (Remarkably, these routines are even available in C, where they operate on `MPI_Fint`, `MPI_F08_Status` arguments.)

The following material is for the recently released MPI-4 standard and may not be supported yet.

The `SOURCE`, `TAG`, `ERROR` fields can also be accessed, both to get and to set, with routines

- `MPI_Status_get_source`, `MPI_Status_set_source`;
- `MPI_Status_get_tag`, `MPI_Status_set_tag`;
- `MPI_Status_get_error`, `MPI_Status_set_error`;

End of MPI-4 material

Python note 16: Status object. The status object is explicitly created before being passed to the receive routine. It has the usual query method for the message count:

```
## pingpongbig.py
status = MPI.Status()
comm.Recv( rdata, source=0, status=status)
count = status.Get_count(MPI.DOUBLE)
```

(The count function without argument returns a result in bytes.)

However, unlike in C/F where the fields of the status object are directly accessible, Python has query methods for these too:

```
status.Get_source()
status.Get_tag()
status.Get_elements()
status.Get_error()
status.Is_cancelled()
```

Should you need them, there are even *Set* variants of these. <https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Status.html>

MPL note 37: Status object. The `mpl::status_t` object is created by the receive (or wait) call:

```
mpl::contiguous_layout<double> target_layout(count);
mpl::status_t recv_status =
    comm_world.recv(target.data(),target_layout, the_other);
recv_count = recv_status.get_count<double>();
```

4.3.1 Source

In some applications it makes sense that a message can come from one of a number of processes. In this case, it is possible to specify `MPI_ANY_SOURCE` as the source. To find out the *source* where the message actually came from, you would use the `MPI_SOURCE` field of the status object that is delivered by `MPI_Recv` or the `MPI_Wait...` call after an `MPI_Irecv`.

```
MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm,
          &status);
sender = status.MPI_SOURCE;
```

There are various scenarios where receiving from ‘any source’ makes sense. One is that of the *manager-worker* model. The manager task would first send data to the worker tasks, then issues a blocking wait for the data of whichever process finishes first.

In Fortran2008 style, the source is a member of the `Status` type.

```
!! anysource.F90
Type(MPI_Status) :: status
allocate(recv_buffer(ntids-1))
do p=0,ntids-2
    call MPI_Recv(recv_buffer(p+1),1,MPI_INTEGER,&
                  MPI_ANY_SOURCE,0,comm,status)
    sender = status%MPI_SOURCE
```

In Fortran90 style, the source is an index in the `Status` array.

```
!! anysource.F90
integer :: status(MPI_STATUS_SIZE)
    allocate(recv_buffer(ntids-1))
    do p=0,ntids-2
        call MPI_Recv(recv_buffer(p+1),1,MPI_INTEGER,&
                      MPI_ANY_SOURCE,0,comm,status,err)
        sender = status(MPI_SOURCE)
```

MPL note 38: Status querying. The `status` object can be queried:

```
int source = recv_status.source();
```

Likewise the source:

```
mpl::tag_t t = recv_status.tag();
```

4.3.2 Tag

In some circumstances, a tag wildcard can be used on the receive operation: `MPI_ANY_TAG`. The actual *tag* of a message can be retrieved as the `MPI_TAG` member in the status structure.

There are not many cases where this is needed.

- Messages from a single source, even non-blocking, are *non-overtaking*. This means that messages can be distinguished by their order.
- Messages from multiple sources can be distinguished by the source field.
- Retrieving the message tag might be needed if information is encoded in it.
- The non-overtaking argument does not apply in the case of hybrid computing: two threads may send messages that do not have an MPI-imposed order. See the example in section 45.1.

MPL note 39: Message tag. MPL differs from other APIs in its treatment of tags: a tag is not directly an integer, but an object of class `tag_t`.

```
// sendrecv.cxx
mpl::tag_t t0(0);
comm_world.sendrecv
( mydata,sendto,t0,
  leftdata,recvfrom,t0 );
```

The `tag_t` class has a couple of methods such as `mpl::tag_t::any()` (for the `MPI_ANY_TAG` wildcard in receive calls) and `mpl::tag_t::up()` (maximal tag, found from the `MPI_TAG_UB` attribute).

MPL note 40: Tag types. Tag are `int` or an `enum` typ:

```
template<typename T>
tag_t (T t);
tag_t (int t);
```

Example:

4. MPI topic: Point-to-point

Figure 4.13 MPI_Get_count

Name	Param name	Explanation	C type	F type	inout
MPI_Get_count					
MPI_Get_count_c					
status		return status of receive operation	const MPI_Status*	TYPE (MPI_Status)	IN
datatype		datatype of each receive buffer entry	MPI_Datatype	TYPE (MPI_Datatype)	IN
count		number of received entries	[int* MPI_Count*]	INTEGER	OUT
)					

MPL:

```
template<typename T>
int mpl::status::get_count () const

template<typename T>
int mpl::status::get_count (const layout<T> &l) const
```

Python:

```
status.Get_count( Datatype datatype=BYTE )
```

```
// inttag.cxx
enum class pingpongtag : int { ping=1, pong=2 };
int pinger = 0, ponger = world.size()-1;
if (world.rank()==pinger) {
    world.send(x, ponger, pingpongtag::ping);
    world.recv(x, ponger, pingpongtag::pong);
} else if (world.rank()==ponger) {
    world.recv(x, pinger, pingpongtag::ping);
    world.send(x, pinger, pingpongtag::pong);
}
```

4.3.3 Error

For functions that return a single status, any error is returned as the function result. For a function returning multiple statuses, such as `MPI_Waitall`, the presence of an error in one of the receives is indicated by a result of `MPI_ERR_IN_STATUS`. Any *errors* during the receive operation can be found as the `MPI_ERROR` member of the status structure.

4.3.4 Count

If the amount of data received is not known a priori, the *count* of elements received can be found by `MPI_Get_count` (figure 4.13):

```
// count.c
if (procid==0) {
    int sendcount = (rand()>.5) ? N : N-1;
    MPI_Send( buffer,sendcount,MPI_FLOAT,target,0,comm );
} else if (procid==target) {
    MPI_Status status;
    int recvcount;
    MPI_Recv( buffer,N,MPI_FLOAT,0,0, comm, &status );
    MPI_Get_count(&status,MPI_FLOAT,&recvcount);
    printf("Received %d elements\n",recvcount);
}
```

Code:

```
!! count.F90
if (procid==0) then
    sendcount = N
    call random_number(fraction)
    if (fraction>.5) then
        print *, "One less" ; sendcount
        = N-1
    end if
    call MPI_Send( buffer,sendcount,
    MPI_REAL,target,0,comm )
else if (procid==target) then
    call MPI_Recv( buffer,N,MPI_REAL
    ,0,0, comm, status )
    call MPI_Get_count(status,MPI_FLOAT
    ,recvcount)
    print *, "Received",recvcount,"elements"
end if
```

Output:

```
[examples/mpi/f08] count:
make[3]: `count' is up to date.
TACC: Starting up job 4051425
TACC: Setting up parallel
    ↗environment for
    ↗MVAPICH2+mpispawn.
TACC: Starting parallel tasks...
One less
Received         9 elements
TACC: Shutdown complete. Exiting.
```

This may be necessary since the count argument to `MPI_Recv` is the buffer size, not an indication of the actually received number of data items.

Remarks.

- Unlike the source and tag, the message count is not directly a member of the status structure.
- The ‘count’ returned is the number of elements of the specified datatype. If this is a derived type (section 6.3) this is not the same as the number of predefined datatype elements. For that, use `MPI_Get_elements` (figure 4.14) or `MPI_Get_elements_x` which returns the number of basic elements.

MPL note 41: Receive count. The `get_count` function is a method of the status object. The argument type is handled through templating:

```
// recvstatus.cxx
double pi=0;
```

Figure 4.14 MPI_Get_elements

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Get_elements</code>					
	<code>MPI_Get_elements_c</code>				
	<code>status</code>	return status of receive operation	const <code>MPI_Status*</code>	<code>TYPE (MPI_Status)</code>	IN
	<code>datatype</code>	datatype used by receive operation	<code>MPI_Datatype</code>	<code>TYPE (MPI_Datatype)</code>	IN
	<code>count</code>	number of received basic elements	<code>[int* MPI_Count*]</code>	<code>INTEGER</code>	OUT

```
auto s = comm_world.recv(pi, 0); // receive from rank 0
int c = s.get_count<double>();
std::cout << "got : " << c << " scalar(s): " << pi << '\n';
```

4.3.5 Example: receiving from any source

Consider an example where the last process receives from every other process. We could implement this as a loop

```
for (int p=0; p<nprocs-1; p++)
    MPI_Recv( /* from source= */ p );
```

but this may incur idle time if the messages arrive out of order.

Instead, we use the `MPI_ANY_SOURCE` specifier to give a wildcard behavior to the receive call: using this value for the ‘source’ value means that we accept messages from any source within the communicator, and messages are only matched by tag value. (Note that size and type of the receive buffer are not used for message matching!)

We then retrieve the actual source from the `MPI_Status` object through the `MPI_SOURCE` field.

```
// anysource.c
if (procno==nprocs-1) {
/*
 * The last process receives from every other process
 */
int *recv_buffer;
recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));

/*
 * Messages can come in in any order, so use MPI_ANY_SOURCE
 */
MPI_Status status;
for (int p=0; p<nprocs-1; p++) {
    err = MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm,
        &status); CHK(err);
```

```

    int sender = status.MPI_SOURCE;
    printf("Message from sender=%d: %d\n",
           sender,recv_buffer[p]);
}
free(recv_buffer);
} else {
/*
 * Each rank waits an unpredictable amount of time,
 * then sends to the last process in line.
*/
float randomfraction = (rand() / (double)RAND_MAX);
int randomwait = (int) ( nprocs * randomfraction );
printf("process %d waits for %e/%d=%d\n",
       procno,randomfraction,nprocs,randomwait);
sleep(randomwait);
err = MPI_Send(&randomwait,1,MPI_INT, nprocs-1,0,comm); CHK(err);
}

## anysource.py
rstatus = MPI.Status()
comm.Recv(rbuf,source=MPI.ANY_SOURCE,status=rstatus)
print("Message came from %d" % rstatus.Get_source())

```

The *manager-worker* model is a design patterns that offers an opportunity for inspecting the `MPI_SOURCE` field of the `MPI_Status` object describing the data that was received. All workers processes model their work by waitin a random amount of time, and the manager process accepts messages from any source.

```

// anysource.c
if (procno==nprocs-1) {
/*
 * The last process receives from every other process
 */
int *recv_buffer;
recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));

/*
 * Messages can come in in any order, so use MPI_ANY_SOURCE
 */
MPI_Status status;
for (int p=0; p<nprocs-1; p++) {
    err = MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm,
                  &status); CHK(err);
    int sender = status.MPI_SOURCE;
    printf("Message from sender=%d: %d\n",
           sender,recv_buffer[p]);
}
free(recv_buffer);
} else {
/*
 * Each rank waits an unpredictable amount of time,
 * then sends to the last process in line.
*/

```

Figure 4.15 MPI_Probe

Name	Param name	Explanation	C type	F type	inout
MPI_Probe (
source		rank of source or MPI_ANY_SOURCE	int	INTEGER	IN
tag		message tag or MPI_ANY_TAG	int	INTEGER	IN
comm		communicator	MPI_Comm	TYPE (MPI_Comm)	IN
status		status object	MPI_Status*	TYPE (MPI_Status)	OUT
)					

```

float randomfraction = (rand() / (double)RAND_MAX);
int randomwait = (int) ( nprocs * randomfraction );
printf("process %d waits for %e/%d=%d\n",
       procno,randomfraction,nprocs,randomwait);
sleep(randomwait);
err = MPI_Send(&randomwait,1,MPI_INT, nprocs-1,0,comm); CHK(err);
}

```

In chapter ?? you can do programming project with this model.

4.4 More about point-to-point communication

4.4.1 Message probing

MPI receive calls specify a receive buffer, and its size has to be enough for any data sent. In case you really have no idea how much data is being sent, and you don't want to overallocate the receive buffer, you can use a 'probe' call.

The routines **MPI_Probe** (figure 4.15) and **MPI_Iprobe** (figure 4.16) (for which see also section 15.4) accept a message but do not copy the data. Instead, when probing tells you that there is a message, you can use **MPI_Get_count** (section 4.3.4) to determine its size, allocate a large enough receive buffer, and do a regular receive to have the data copied.

```

// probe.c
if (procno==receiver) {
    MPI_Status status;
    MPI_Probe(sender,0,comm,&status);
    int count;
    MPI_Get_count(&status,MPI_FLOAT,&count);
    float recv_buffer[count];
    MPI_Recv(recv_buffer,count,MPI_FLOAT, sender,0,comm,MPI_STATUS_IGNORE);
} else if (procno==sender) {
    float buffer[buffer_size];
    ierr = MPI_Send(buffer,buffer_size,MPI_FLOAT, receiver,0,comm); CHK(ierr);
}

```

Figure 4.16 MPI_Iprobe

Name	Param name	Explanation	C type	F type	inout
MPI_Iprobe (
source		rank of source or MPI_ANY_SOURCE	int	INTEGER	IN
tag		message tag or MPI_ANY_TAG	int	INTEGER	IN
comm		communicator	MPI_Comm	TYPE (MPI_Comm)	IN
flag		true if there is a matching message that can be received	int*	LOGICAL	OUT
status		status object	MPI_Status*	TYPE (MPI_Status)	OUT
)					

Figure 4.17 MPI_Mprobe

Name	Param name	Explanation	C type	F type	inout
MPI_Mprobe (
source		rank of source or MPI_ANY_SOURCE	int	INTEGER	IN
tag		message tag or MPI_ANY_TAG	int	INTEGER	IN
comm		communicator	MPI_Comm	TYPE (MPI_Comm)	IN
message		returned message	MPI_Message*	TYPE (MPI_Message)	OUT
status		status object	MPI_Status*	TYPE (MPI_Status)	OUT
)					

There is a problem with the **MPI_Probe** call in a multithreaded environment: the following scenario can happen.

1. A thread determines by probing that a certain message has come in.
2. It issues a blocking receive call for that message...
3. But in between the probe and the receive call another thread has already received the message.
4. ... Leaving the first thread in a blocked state with no message to receive.

This is solved by **MPI_Mprobe** (figure 4.17), which after a successful probe removes the message from the *matching queue*: the list of messages that can be matched by a receive call. The thread that matched the probe now issues an **MPI_Mrecv** (figure 4.18) call on that message through an object of type **MPI_Message**.

4.4.2 Errors

MPI routines return **MPI_SUCCESS** upon successful completion. The following error codes can be returned (see section 15.2.1 for details) for completion with error by both send and receive operations: **MPI_ERR_COMM**,

Figure 4.18 MPI_Mrecv

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Mrecv (</code>					
<code> MPI_Mrecv_c (</code>					
<code>buf</code>		initial address of receive buffer	<code>void*</code>	<code>TYPE(*), DIMENSION(..)</code>	<code>OUT</code>
<code>count</code>		number of elements in receive buffer	<code>[int MPI_Count]</code>	<code>INTEGER</code>	<code>IN</code>
<code>datatype</code>		datatype of each receive buffer element	<code>MPI_Datatype</code>	<code>TYPE (MPI_Datatype)</code>	<code>IN</code>
<code>message</code>		message	<code>MPI_Message*</code>	<code>TYPE (MPI_Message)</code>	<code>INOUT</code>
<code>status</code>		status object	<code>MPI_Status*</code>	<code>TYPE (MPI_Status)</code>	<code>OUT</code>
<code>)</code>					

`MPI_ERR_COUNT, MPI_ERR_TYPE, MPI_ERR_TAG, MPI_ERR_RANK.`

4.4.3 Message envelope

Apart from its bare data, each message has a *message envelope*. This has enough information to distinguish messages from each other: the source, destination, tag, communicator.

4.5 Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

Review 4.16. Describe a deadlock scenario involving three processors.

Review 4.17. True or false: a message sent with `MPI_Isend` from one processor can be received with an `MPI_Recv` call on another processor.

Review 4.18. True or false: a message sent with `MPI_Send` from one processor can be received with an `MPI_Irecv` on another processor.

Review 4.19. Why does the `MPI_Irecv` call not have an `MPI_Status` argument?

Review 4.20. Suppose you are testing ping-pong timings. Why is it generally not a good idea to use processes 0 and 1 for the source and target processor? Can you come up with a better guess?

Review 4.21. What is the relation between the concepts of ‘origin’, ‘target’, ‘fence’, and ‘window’ in one-sided communication.

Review 4.22. What are the three routines for one-sided data transfer?

Review 4.23. In the following fragments assume that all buffers have been allocated with sufficient size. For each fragment note whether it deadlocks or not. Discuss performance issues.

```

for (int p=0; p<nprocs; p++)
    if (p!=procid)
        MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);
for (int p=0; p<nprocs; p++)
    if (p!=procid)
        MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE);

~

for (int p=0; p<nprocs; p++)
    if (p!=procid)
        MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE);
for (int p=0; p<nprocs; p++)
    if (p!=procid)
        MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);

int ireq = 0;
for (int p=0; p<nprocs; p++)
    if (p!=procid)
        MPI_Isend(sbuffers[p],buflen,MPI_INT,p,0,comm,&(requests[ireq++]));
for (int p=0; p<nprocs; p++)
    if (p!=procid)
        MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE);
MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE);

int ireq = 0;
for (int p=0; p<nprocs; p++)
    if (p!=procid)
        MPI_Irecv(rbuffers[p],buflen,MPI_INT,p,0,comm,&(requests[ireq++]));
for (int p=0; p<nprocs; p++)
    if (p!=procid)
        MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);
MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE);

int ireq = 0;
for (int p=0; p<nprocs; p++)
    if (p!=procid)
        MPI_Irecv(rbuffers[p],buflen,MPI_INT,p,0,comm,&(requests[ireq++]));
MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE);
for (int p=0; p<nprocs; p++)
    if (p!=procid)
        MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);

```

4. MPI topic: Point-to-point

Fortran codes:

```
do p=0,nprocs-1
  if (p/=procid) then
    call MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm,ierr)
  end if
end do
do p=0,nprocs-1
  if (p/=procid) then
    call MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE,ierr)
  end if
end do

do p=0,nprocs-1
  if (p/=procid) then
    call MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE,ierr)
  end if
end do
do p=0,nprocs-1
  if (p/=procid) then
    call MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm,ierr)
  end if
end do

ireq = 0
do p=0,nprocs-1
  if (p/=procid) then
    call MPI_Isend(sbuffers(1,p+1),buflen,MPI_INT,p,0,comm,&
                  requests(ireq+1),ierr)
    ireq = ireq+1
  end if
end do
do p=0,nprocs-1
  if (p/=procid) then
    call MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE,ierr)
  end if
end do
call MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE,ierr)

ireq = 0
do p=0,nprocs-1
  if (p/=procid) then
    call MPI_Irecv(rbuffers(1,p+1),buflen,MPI_INT,p,0,comm,&
                  requests(ireq+1),ierr)
    ireq = ireq+1
  end if
end do
do p=0,nprocs-1
  if (p/=procid) then
    call MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm,ierr)
  end if
end do
call MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE,ierr)
```

```
// block5.F90
ireq = 0
do p=0,nprocs-1
  if (p/=procid) then
    call MPI_Irecv(rbuffers(1,p+1),buflen,MPI_INT,p,0,comm,&
      requests(ireq+1),ierr)
    ireq = ireq+1
  end if
end do
call MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE,ierr)
do p=0,nprocs-1
  if (p/=procid) then
    call MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm,ierr)
  end if
end do
```

Review 4.24. Consider a ring-wise communication where

```
int
next = (mytid+1) % ntids,
prev = (mytid+ntids-1) % ntids;
```

and each process sends to *next*, and receives from *prev*.

The normal solution for preventing deadlock is to use both `MPI_Isend` and `MPI_Irecv`.

The send and receive complete at the wait call. But does it matter in what sequence you do the wait calls?

<pre>// ring3.c MPI_Request req1,req2; MPI_Irecv(&y,1,MPI_DOUBLE,prev,0,comm,&req1); MPI_Isend(&x,1,MPI_DOUBLE,next,0,comm,&req2); MPI_Wait(&req1,MPI_STATUS_IGNORE); MPI_Wait(&req2,MPI_STATUS_IGNORE);</pre>	<pre>// ring4.c MPI_Request req1,req2; MPI_Irecv(&y,1,MPI_DOUBLE,prev,0,comm,&req1); MPI_Isend(&x,1,MPI_DOUBLE,next,0,comm,&req2); MPI_Wait(&req2,MPI_STATUS_IGNORE); MPI_Wait(&req1,MPI_STATUS_IGNORE);</pre>
--	--

Can we have one nonblocking and one blocking call? Do these scenarios block?

<pre>// ring1.c MPI_Request req; MPI_Isend(&x,1,MPI_DOUBLE,next,0,comm,&req); MPI_Recv(&y,1,MPI_DOUBLE,prev,0,comm, MPI_STATUS_IGNORE); MPI_Wait(&req,MPI_STATUS_IGNORE);</pre>	<pre>// ring2.c MPI_Request req; MPI_Irecv(&y,1,MPI_DOUBLE,prev,0,comm,&req); MPI_Ssend(&x,1,MPI_DOUBLE,next,0,comm); MPI_Wait(&req,MPI_STATUS_IGNORE);</pre>
--	---

Chapter 5

MPI topic: Communication modes

5.1 Persistent communication

You can imagine that setting up a communication carries some overhead, and if the same communication structure is repeated many times, this overhead may be avoided.

Persistent communication is a mechanism for dealing with a repeating communication transaction, where the parameters of the transaction, such as sender, receiver, tag, root, and buffer address /type / size, stay the same. Only the contents of the buffers involved may change between the transactions.

1. For nonblocking communications `MPI_Ixxx` (both point-to-point and collective) there is a persistent variant `MPI_Xxx_init` with the same calling sequence. The ‘init’ call produces an `MPI_Request` output parameter, which can be used to test for completion of the communication.
2. The ‘init’ routine does not start the actual communication: that is done in `MPI_Start`, or `MPI_Startall` for multiple requests.
3. Any of the MPI ‘wait’ calls can then be used to conclude the communication.
4. The communication can then be restarted with another ‘start’ call.
5. The wait call does not release the request object, since it can be used for repeat occurrences of this transaction. The request object is only freed with `MPI_Request_free`.

```
MPI_Send_init( /* ... */ &request);
while ( /* ... */ ) {
    MPI_Start( request );
    MPI_Wait( request, &status );
}
MPI_Request_free( & request );
```

MPL note 42: Persistent requests. MPL returns a `request` from persistent ‘init’ routines, rather than an `irequest` (MPL note 32):

```
template<typename T>
request send_init( const T &data, int dest, tag t=tag(0)) const;
```

Likewise, there is a `request_pool` instead of an `irequest_pool` (note 33).

Figure 5.1 MPI_Send_init

Name	Param name	Explanation	C type	F type	inout
MPI_Send_init					
	MPI_Send_init_c				
buf		initial address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
count		number of elements sent	[int MPI_Count]	INTEGER	IN
datatype		type of each element	MPI_Datatype	TYPE (MPI_Datatype)	IN
dest		rank of destination	int	INTEGER	IN
tag		message tag	int	INTEGER	IN
comm		communicator	MPI_Comm	TYPE (MPI_Comm)	IN
request		communication request	MPI_Request*	TYPE (MPI_Request)	OUT
)					

Python:

```
MPI.Comm.Send_init(self, buf, int dest, int tag=0)
```

Figure 5.2 MPI_Startall

Name	Param name	Explanation	C type	F type	inout
MPI_Startall					
	count	list length	int	INTEGER	IN
	array_of_requests	array of requests	MPI_Request[]	TYPE (MPI_Request) (count)	INOUT
)					

Python:

```
MPI.Request.Startall(type cls, requests)
```

5.1.1 Persistent point-to-point communication

The main persistent point-to-point routines are [MPI_Send_init](#) (figure 5.1), which has the same calling sequence as [MPI_Isend](#), and [MPI_Recv_init](#), which has the same calling sequence as [MPI_Irecv](#).

In the following example a ping-pong is implemented with persistent communication. Since we use persistent operations for both send and receive on the ‘ping’ process, we use [MPI_Startall](#) (figure 5.2) to start both at the same time, and [MPI_Waitall](#) to test their completion. (There is [MPI_Start](#) for starting a single persistent transfer.)

5. MPI topic: Communication modes

```

Code:
// persist.c
if (procno==src) {
/*
 * Send ping, receive pong
 */
MPI_Send_init
(send,s,MPI_DOUBLE,tgt,0,comm,
 requests+0);
MPI_Recv_init
(recv,s,MPI_DOUBLE,tgt,0,comm,
 requests+1);
for (int n=0; n<NEXPERIMENTS; n++) {
fill_buffer(send,s,n);
MPI_Startall(2,requests);
MPI_Waitall(2,requests,
MPI_STATUSES_IGNORE);
int r = chck_buffer(send,s,n);
if (!r) printf("buffer problem %d\n",
",s);
}
} else if (procno==tgt) {
/*
 * Receive ping, send pong
 */
MPI_Send_init
(recv,s,MPI_DOUBLE,src,0,comm,
 requests+0);
MPI_Recv_init
(recv,s,MPI_DOUBLE,src,0,comm,
 requests+1);
for (int n=0; n<NEXPERIMENTS; n++) {
// receive
MPI_Start(requests+1);
MPI_Wait(requests+1,
MPI_STATUS_IGNORE);
// send
MPI_Start(requests+0);
MPI_Wait(requests+0,
MPI_STATUS_IGNORE);
}
MPI_Request_free(requests+0);
MPI_Request_free(requests+1);
}

```

```

Output:
[examples/mpi/c] persist:
make[3]: `persist' is up to date.
TACC: Starting up job 4328411
TACC: Starting parallel tasks...
Pingpong size=1: t=1.2123e-04
Pingpong size=10: t=4.2826e-06
Pingpong size=100: t=7.1507e-06
Pingpong size=1000: t=1.2084e-05
Pingpong size=10000: t=3.7668e-05
Pingpong size=100000: t=3.4415e-04
Persistent size=1: t=3.8177e-06
Persistent size=10: t=3.2410e-06
Persistent size=100: t=4.0468e-06
Persistent size=1000: t=1.1525e-05
Persistent size=10000: t=4.1672e-05
Persistent size=100000:
→t=2.8648e-04
TACC: Shutdown complete. Exiting.

```

(Ask yourself: why does the sender use `MPI_Startall` and `MPI_Waitall`, but the receiver uses `MPI_Start` and `MPI_Wait` twice?)

```

## persist.py
requests = [None] * 2
sendbuf = np.ones(size,dtype=int)

```

```

recvbuf = np.ones(size,dtype=int)
if procid==src:
    print("Size:",size)
    times[isize] = MPI.Wtime()
    for n in range(nexperiments):
        requests[0] = comm.Isend(sendbuf[0:size],dest=tgt)
        requests[1] = comm.Irecv(recvbuf[0:size],source=tgt)
        MPI.Request.Waitall(requests)
        sendbuf[0] = sendbuf[0]+1
    times[isize] = MPI.Wtime()-times[isize]
elif procid==tgt:
    for n in range(nexperiments):
        comm.Recv(recvbuf[0:size],source=src)
        comm.Send(recvbuf[0:size],dest=src)

```

As with ordinary send commands, there are persistent variants of the other send modes:

- `MPI_Bsend_init` for buffered communication, section 5.5;
- `MPI_Ssend_init` for synchronous communication, section 5.3.1;
- `MPI_Rsend_init` for ready sends, section 15.8.

5.1.2 Persistent collectives

The following material is for the recently released MPI-4 standard and may not be supported yet.

For each collective call, there is a persistent variant. As with persistent point-to-point calls (section 5.1.1), these have largely the same calling sequence as the nonpersistent variants, except for:

- an `MPI_Info` parameter that can be used to pass system-dependent hints; and
- an added final `MPI_Request` parameter.

(See for instance `MPI_Allreduce_init` (figure 5.3).) This request (or an array of requests from multiple calls) can then be used by `MPI_Start` (or `MPI_Startall`) to initiate the actual communication.

```

// powerpersist1.c
double localnorm,globalnorm=1.;
MPI_Request reduce_request;
MPI_Allreduce_init
( &localnorm,&globalnorm,1,MPI_DOUBLE,MPI_SUM,
  comm,MPI_INFO_NULL,&reduce_request);
for (int it=0; ; it++) {
/*
 * Matrix vector product
 */
matmult(indata,outdata,buffersize);

// start computing norm of output vector
localnorm = local_l2_norm(outdata,buffersize);
double old_globalnorm = globalnorm;
MPI_Start( &reduce_request );

// end computing norm of output vector
MPI_Wait( &reduce_request,MPI_STATUS_IGNORE );
globalnorm = sqrt(globalnorm);

```

5. MPI topic: Communication modes

Figure 5.3 MPI_Allreduce_init

Name	Param name	Explanation	C type	F type	inout
MPI_Allreduce_init					
	MPI_Allreduce_init_c				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN	
recvbuf	starting address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT	
count	number of elements in send buffer	[int MPI_Count]	INTEGER	IN	
datatype	datatype of elements of send buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN	
op	operation	MPI_Op	TYPE(MPI_Op)	IN	
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN	
info	info argument	MPI_Info	TYPE (MPI_Info)	IN	
request	communication request	MPI_Request*	TYPE (MPI_Request)	OUT	
)					

```
// now `globalnorm' is the L2 norm of `outdata'
scale(outdata, indata, buffersize, 1./globalnorm);
}
MPI_Request_free( &reduce_request );
```

Some points.

- Metadata arrays, such as of counts and datatypes, must not be altered until the `MPI_Request_free` call.
- The initialization call is nonlocal (for this particular case of persistent collectives), so it can block until all processes have performed it.
- Multiple persistent collective can be initialized, in which case they satisfy the same restrictions as ordinary collectives, in particular on ordering. Thus, the following code is incorrect:

```
// WRONG
if (procid==0) {
    MPI_Reduce_init( /* ... */ &req1);
    MPI_Bcast_init( /* ... */ &req2);
} else {
    MPI_Bcast_init( /* ... */ &req2);
    MPI_Reduce_init( /* ... */ &req1);
}
```

However, after initialization the start calls can be in arbitrary order, and in different order among the processes.

Available persistent collectives are: `MPI_Barrier_init` `MPI_Bcast_init` `MPI_Reduce_init` `MPI_Allreduce_init` `MPI_Reduce_scatter_init` `MPI_Reduce_scatter_block_init` `MPI_Gather_init` `MPI_Gatherv_init`

Figure 5.4 MPI_Psend_init

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Psend_init (</code>					
<code>buf</code>		initial address of send buffer	<code>const void*</code>	<code>TYPE(*), DIMENSION(..)</code>	<code>IN</code>
<code>partitions</code>		number of partitions	<code>int</code>	<code>INTEGER</code>	<code>IN</code>
<code>count</code>		number of elements sent per partition	<code>MPI_Count</code>	<code>INTEGER (KIND=MPI_COUNT_KIND)</code>	<code>IN</code>
<code>datatype</code>		type of each element	<code>MPI_Datatype</code>	<code>TYPE (MPI_Datatype)</code>	<code>IN</code>
<code>dest</code>		rank of destination	<code>int</code>	<code>INTEGER</code>	<code>IN</code>
<code>tag</code>		message tag	<code>int</code>	<code>INTEGER</code>	<code>IN</code>
<code>comm</code>		communicator	<code>MPI_Comm</code>	<code>TYPE (MPI_Comm)</code>	<code>IN</code>
<code>info</code>		info argument	<code>MPI_Info</code>	<code>TYPE (MPI_Info)</code>	<code>IN</code>
<code>request</code>		communication request	<code>MPI_Request*</code>	<code>TYPE (MPI_Request)</code>	<code>OUT</code>
<code>)</code>					

`MPI_Allgather_init MPI_Allgatherv_init MPI_Scatter_init MPI_Scatterv_init MPI_Alltoall_init
MPI_Alltoally_init MPI_Alltoallw_init MPI_Scan_init MPI_Exscan_init`

Remark Persistent operations can be started in any order. However, system-dependent optimizations are possible if all processes start persistent collectives in the same order. This can be declared in MPI-4.1 by setting the info key `mpi_assert_strict_persistent_collective_ordering` to true. (See section 15.1.1 for info objects.) This value needs to be set identically on all processes of the communicator.

End of MPI-4 material

5.1.3 Persistent neighbor communications

The following material is for the recently released MPI-4 standard and may not be supported yet.

There are persistent version of the neighborhood collectives; section 11.2.2.

`MPI_Neighbor_allgather_init, MPI_Neighbor_allgatherv_init, MPI_Neighbor_alltoall_init,
MPI_Neighbor_alltoally_init, MPI_Neighbor_alltoallw_init,`

End of MPI-4 material

5.2 Partitioned communication

The following material is for the recently released MPI-4 standard and may not be supported yet.

Partitioned communication is a variant on *persistent communication*, in the sense that we use the init / start / wait sequence. There difference is that now a message can be constructed in bit-by-bit.

- The normal `MPI_Send_init` is replaced by `MPI_Psend_init` (figure 5.4). Note the presence of an `MPI_Info` argument, as in persistent collectives, but unlike in persistent sends and receives.

5. MPI topic: Communication modes

Figure 5.5 MPI_Pready

Name	Param name	Explanation	C type	F type	inout
MPI_Pready (

- After this, the `MPI_Start` does not actually start the transfer; instead:
- Each partition of the message is separately declared as ready-to-be-sent with `MPI_Pready`.
- An `MPI_Wait` call completes the operation, indicating that all partitions have been sent.

A common scenario for this is in multi-threaded environments, where each thread can construct its own part of a message. Having partitioned messages means that partially constructed message buffers can be sent off without having to wait for all threads to finish.

Indicating that parts of a message are ready for sending is done by one of the following calls:

- `MPI_Pready` (figure 5.5) for a single partition;
- `MPI_Pready_range` for a range of partitions; and
- `MPI_Pready_list` for an explicitly enumerated list of partitions.

The `MPI_Psend_init` call yields an `MPI_Request` object that can be used to test for completion (see sections 4.2.2 and 4.2.3) of the full operation.

```

MPI_Request send_request;
MPI_Psend_init
  (sendbuffer,nparts,SIZE,MPI_DOUBLE,tgt,0,
   comm,MPI_INFO_NULL,&send_request);
for (int it=0; it<ITERATIONS; it++) {
  MPI_Start(&send_request);
  for (int ip=0; ip<nparts; ip++) {
    fill_buffer(sendbuffer,partitions[ip],partitions[ip+1],ip);
    MPI_Pready(ip,send_request);
  }
  MPI_Wait(&send_request,MPI_STATUS_IGNORE);
}
MPI_Request_free(&send_request);

```

The receiving side is largely the mirror image of the sending side:

```

double *recvbuffer = (double*)malloc(bufsize*sizeof(double));
MPI_Request recv_request;
MPI_Precv_init
  (recvbuffer,nparts,SIZE,MPI_DOUBLE,src,0,
   comm,MPI_INFO_NULL,&recv_request);
for (int it=0; it<ITERATIONS; it++) {
  MPI_Start(&recv_request); int r=1,flag;
  for (int ip=0; ip<nparts; ip++) // cycle this many times

```

Figure 5.6 MPI_Parrived

Name	Param name	Explanation	C type	F type	inout
MPI_Parrived					
	request	partitioned communication request	MPI_Request	TYPE (MPI_Request)	IN
	partition	partition to be tested	int	INTEGER	IN
	flag	true if operation completed on the specified partition, false if not	int*	LOGICAL	OUT
)				

```

for (int ap=0; ap<nparts; ap++) { // check specific part
    MPI_Parrived(recv_request,ap,&flag);
    if (flag) {
        r *= chck_buffer
        (recvbuffer,partitions[ap],partitions[ap+1],ap);
        break;
    }
    MPI_Wait(&recv_request,MPI_STATUS_IGNORE);
}
MPI_Request_free(&recv_request);

```

- a partitioned send can only be matched with a partitioned receive, so we start with an **MPI_Precv_init**.
- Arrival of a partition can be tested with **MPI_Parrived** (figure 5.6).
- A call to **MPI_Wait** completes the operation, indicating that all partitions have arrived.

Again, the **MPI_Request** object from the receive-init call can be used to test for completion of the full receive operation.

End of MPI-4 material

5.3 Synchronous and asynchronous communication

It is easiest to think of blocking as a form of synchronization with the other process, but that is not quite true. Synchronization is a concept in itself, and we talk about *synchronous* communication if there is actual coordination going on with the other process, and *asynchronous* communication if there is not. Blocking then only refers to the program waiting until the user data is safe to reuse; in the synchronous case a blocking call means that the data is indeed transferred, in the asynchronous case it only means that the data has been transferred to some system buffer. The four possible cases are illustrated in figure 5.1.

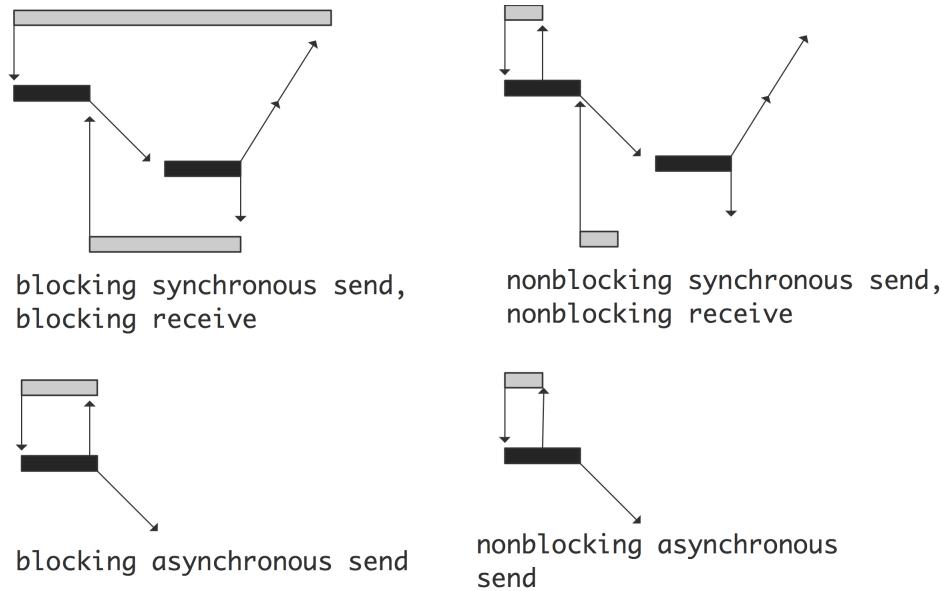


Figure 5.1: Blocking and synchronicity

5.3.1 Synchronous send operations

MPI has a number of routines for synchronous communication, such as `MPI_Ssend`. Driving home the point that nonblocking and asynchronous are different concepts, there is a routine `MPI_Issend`, which is synchronous but nonblocking. These routines have the same calling sequence as their not-explicitly synchronous variants, and only differ in their semantics.

See section 4.1.4.2 for examples.

5.4 Local and nonlocal operations

The MPI standard does not dictate whether communication is buffered. If a message is buffered, a send call can complete, even if no corresponding send has been posted yet. See section 4.1.4.2. Thus, in the standard communication, a send operation is *nonlocal*: its completion may depend on whether the corresponding receive has been posted. A *local operation* is one that is not nonlocal.

On the other hand, *buffered communication* (routines `MPI_Bsend`, `MPI_Ibsend`, `MPI_Bsend_init`; section 5.5) is *local*: the presence of an explicit buffer means that a send operation can complete no matter whether the receive has been posted.

The *synchronous send* (routines `MPI_Ssend`, `MPI_Issend`, `MPI_Ssend_init`; section 15.8) is again nonlocal (even in the nonblocking variant) since it will only complete when the receive call has completed.

Finally, the *ready mode send* (`MPI_Rsend`, `MPI_Irsend`) is nonlocal in the sense that its only correct use is when the corresponding receive has been issued.

Figure 5.7 MPI_Bsend

Name	Param name	Explanation	C type	F type	inout
MPI_Bsend (
MPI_Bsend_c (
buf		initial address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
count		number of elements in send buffer	[int MPI_Count]	INTEGER	IN
datatype		datatype of each send buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
dest		rank of destination	int	INTEGER	IN
tag		message tag	int	INTEGER	IN
comm		communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)					

5.5 Buffered communication

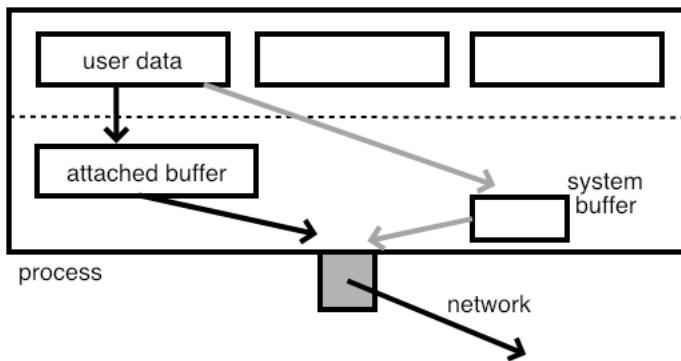


Figure 5.2: User communication routed through an attached buffer

By now you have probably got the notion that managing buffer space in MPI is important: data has to be somewhere, either in user-allocated arrays or in system buffers. Using *buffered communication* is yet another way of managing buffer space.

1. You allocate your own buffer space, and you attach it to your process. This buffer is not a send buffer: it is a replacement for buffer space used inside the MPI library or on the network card; figure 5.2. If high-bandwidth memory is available, you could create your buffer there.

The following material is for the recently released MPI-4 standard and may not be supported yet.

2. A buffer can also be attached directly to a communicator or session; see below.

End of MPI-4 material

3. You use the `MPI_Bsend` (figure 5.7) (or its *local* variant `MPI_Ibsend`) call for sending, using otherwise normal send and receive buffers;
4. You detach the buffer when you're done with the buffered sends.

One advantage of buffered sends is that they are nonblocking: since there is a guaranteed buffer long

Figure 5.8 MPI_Buffer_attach

Name	Param name	Explanation	C type	F type	inout
MPI_Buffer_attach					
	MPI_Buffer_attach_c				
	buffer	initial buffer address	void*	TYPE(*), DIMENSION(..)	IN
	size	buffer size, in bytes	[int MPI_Count]	INTEGER	IN
)				

Figure 5.9 MPI_Comm_attach_buffer

enough to contain the message, it is not necessary to wait for the receiving process.

We illustrate the use of buffered sends:

```
// bufring.c
int bsize = BUFLEN*sizeof(float);
float
    *sbuf = (float*) malloc( bsize ),
    *rbuf = (float*) malloc( bsize );
MPI_Pack_size( BUFLEN,MPI_FLOAT,comm,&bsize );
bsize += MPI_BSEND_OVERHEAD;
float
    *buffer = (float*) malloc( bsize );

MPI_Buffer_attach( buffer,bsize );
err = MPI_Bsend(sbuf,BUFLEN,MPI_FLOAT,next,0,comm);
MPI_Recv (rbuf,BUFLEN,MPI_FLOAT,prev,0,comm,MPI_STATUS_IGNORE);
MPI_Buffer_detach( &buffer,&bsize );
```

5.5.1 Buffer treatment

If you attach the buffer directly to the MPI process with `MPI_Buffer_attach` (figure 5.8) there can be only one buffer per process.

The following material is for the recently released MPI-4 standard and may not be supported yet.

The calls `MPI_Comm_attach_buffer` (figure 5.9) and `MPI_Comm_detach_buffer` (figure 5.10) (as of MPI-4.1) can be used to have a buffer per communicator. Likewise `MPI_Session_attach_buffer` and `MPI_Session_detach_buffer`

Also: `MPI_Comm_flush_buffer`, `MPI_Session_flush_buffer`, and a global function `MPI_Buffer_flush`.

End of MPI-4 material

The buffer size should be enough for all `MPI_Bsend` calls that are simultaneously outstanding. You can compute the needed size of the buffer with `MPI_Pack_size`; see section 6.8. Additionally, a term of `MPI_BSEND_OVERHEAD` is needed. See the above code fragment.

Figure 5.10 MPI_Comm_detach_buffer

Figure 5.11 MPI_Buffer_detach

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Buffer_detach (</code>					

The following material is for the recently released MPI-4 standard and may not be supported yet.

specifying `MPI_BUFFER_AUTOMATIC` in any of the attach routines as the `buffer` argument. The `size` argument is then ignored.

End of MPI-4 material

The buffer is detached with `MPI_Buffer_detach` (figure 5.11). This returns the address and size of the buffer; the call blocks until all buffered messages have been delivered.

Note that both `MPI_Buffer_attach` and `MPI_Buffer_detach` have a `void*` argument for the buffer, but

- in the attach routine this is the address of the buffer,
- while the detach routine it is the address of the buffer pointer.

This is done so that the detach routine can zero the buffer pointer.

While the buffered send is nonblocking like an `MPI_Isend`, there is no corresponding wait call. You can force delivery by

```
MPI_Buffer_detach( &b, &n );
MPI_Buffer_attach( b, n );
```

MPL note 43: Buffered send. Creating and attaching a buffer is done through `bsend_buffer` and a support routine `bsend_size` helps in calculating the buffer size:

```
// bufring.cxx
vector<float> sbuf(BUFLEN), rbuf(BUFLEN);
int size{ comm_world.bsend_size<float>(mpl::contiguous_layout<float>(BUFLEN)) };
mpl::bsend_buffer buff(size);
comm_world.bsend(sbuf.data(),mpl::contiguous_layout<float>(BUFLEN), next);
```

Constant: `mpl::bsend_overhead` is `constexpr'd` to the MPI constant `MPI_BSEND_OVERHEAD`.

MPL note 44: Buffer attach and detach. There is a separate attach routine, but normally this is called by the constructor of the `bsend_buffer`. Likewise, the detach routine is called in the buffer destructor.

```
void mpl::environment::buffer_attach (void *buff, int size);
std::pair< void *, int > mpl::environment::buffer_detach ();
```

Figure 5.12 MPI_Bsend_init

Name	Param name	Explanation	C type	F type	inout
MPI_Bsend_init (
MPI_Bsend_init_c (
buf	initial address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN	
count	number of elements sent	[int MPI_Count]	INTEGER	IN	
datatype	type of each element	MPI_Datatype	TYPE (MPI_Datatype)	IN	
dest	rank of destination	int	INTEGER	IN	
tag	message tag	int	INTEGER	IN	
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN	
request	communication request	MPI_Request*	TYPE (MPI_Request)	OUT	
)					

5.5.2 Buffered send calls

The possible error codes are

- **MPI_SUCCESS** the routine completed successfully.
- **MPI_ERR_BUFFER** The buffer pointer is invalid; this typically means that you have supplied a null pointer.
- **MPI_ERR_INTERN** An internal error in MPI has been detected.

The asynchronous version is **MPI_Ibsend**, the persistent (see section 5.1) call is **MPI_Bsend_init** (figure 5.12).

5.5.3 Local behavior

The attach routines are local; the detach and flush routines are non-local.

Chapter 6

MPI topic: Data types

In the examples you have seen so far, every time data was sent it was as a contiguous buffer with elements of a single type. In practice you may want to send noncontiguous or heterogeneous data.

- As an example of noncontiguous data, communicating the real parts of an array of complex numbers means specifying every other number.
- Heterogeneous data is needed when communicating a C structure or Fortran type with more than one type of element.

The datatypes you have dealt with so far are known as *predefined datatypes*; the datatypes you create to deal with other data are known as *derived datatypes*.

6.1 The MPI_Datatype data type

Datatypes such as `MPI_INT` are values of the type `MPI_Datatype`. This type is handled differently in different languages.

In C you can declare variables as

```
MPI_Datatype mytype;
```

Fortran note 10: Derived types for handles. In Fortran before 2008, datatype variables are stored in `Integer` variables. With the Fortran2008 standard, datatypes are Fortran derived types:

```
!! vector.F90
Type(MPI_Datatype) :: newvectortype
```

Implementationwise speaking, these types have exactly one member, `MPI_VAL`, which is the same integer as was used for that datatype in the earlier Fortran version.

Python note 17: Data types. There is a class

```
mpi4py.MPI.Datatype
```

with predefined values such as

```
mpi4py.MPI.Datatype.DOUBLE
```

which are themselves objects with methods for creating derived types; see section 6.3.1.

MPL note 45: Datatype handling. MPL mostly handles datatypes through subclasses of the `layout` class. Layouts are MPL routines are templated over the data type.

```
// sendlong.cxx
mpl::contiguous_layout<long long> v_layout(v.size());
comm.send(v.data(), v_layout, 1); // send to rank 1
```

Also works with complex of float and double.

The data types, where MPL can infer their internal representation, are enumeration types, C arrays of constant size and the template classes `std::array`, `std::pair` and `std::tuple` of the C++ Standard Template Library. The only limitation is, that the C array and the mentioned template classes hold data elements of types that can be sent or received by MPL.

MPL note 46: Native MPI data types. Should you need the `MPI_Datatype` object contained in an MPL layout, there is an access function `native_handle`.

6.2 Predefined data types

MPI has a number of predefined data types of various kinds

- First of all there are the types corresponding to the simple data types of the host languages. The names are made to resemble the types of C and Fortran, for instance `MPI_FLOAT` and `MPI_DOUBLE` corresponding to `float` and `double` in C, versus `MPI_REAL` and `MPI_DOUBLE_PRECISION` corresponding to `Real` and `Double precision` in Fortran.
- The types `MPI_PACKED` and `MPI_BYTE` do not correspond to language types.
- The type `MPI_Aint` (and the Fortran kind `MPI_ADDRESS_KIND`) is used in Remote Memory Access (RMA) windows; see section 9.3.1.
- The type `MPI_Offset` (and the corresponding Fortran `MPI_OFFSET_KIND` kind) is used to define `MPI_Offset` quantities, used in file I/O; section 10.2.2.
- The type `MPI_Count` describes buffers; see section 6.4.
- The type `MPI_CHAR` corresponds to a character, which is not the same as a C `char`: it can be more than one byte. Also, MPI converts between native character representations when communicating between different architectures.

6.2.1 C/C++

Here we illustrate for C/C++ the correspondence between a type used to declare a variable, and how this type appears in MPI communication routines:

```
long int i;
MPI_Send(&i, 1, MPI_LONG, target, tag, comm);
```

See table 6.1.

- There is some, but not complete, support for C99 types; see table 6.2.

C type	MPI type
<code>char</code>	<code>MPI_CHAR</code>
<code>unsigned char</code>	<code>MPI_UNSIGNED_CHAR</code>
<code>char</code>	<code>MPI_SIGNED_CHAR</code>
<code>short</code>	<code>MPI_SHORT</code>
<code>unsigned short</code>	<code>MPI_UNSIGNED_SHORT</code>
<code>int</code>	<code>MPI_INT</code>
<code>unsigned int</code>	<code>MPI_UNSIGNED</code>
<code>long int</code>	<code>MPI_LONG</code>
<code>unsigned long int</code>	<code>MPI_UNSIGNED_LONG</code>
<code>long long int</code>	<code>MPI_LONG_LONG_INT</code>
<code>float</code>	<code>MPI_FLOAT</code>
<code>double</code>	<code>MPI_DOUBLE</code>
<code>long double</code>	<code>MPI_LONG_DOUBLE</code>
<code>unsigned char</code>	<code>MPI_BYTE</code>
(does not correspond to a C type)	<code>MPI_PACKED</code>

Table 6.1: Predefined datatypes in C

C99 type	MPI type
<code>_Bool</code>	<code>MPI_C_BOOL</code>
<code>float _Complex</code>	<code>MPI_C_COMPLEX</code>
	<code>MPI_C_FLOAT_COMPLEX</code>
<code>double _Complex</code>	<code>MPI_C_DOUBLE_COMPLEX</code>
<code>long double _Complex</code>	<code>MPI_C_LONG_DOUBLE_COMPLEX</code>

Table 6.2: C99 synonym types.

- There is support for C11 fixed width integer types; see table 6.3.
- The `MPI_LONG_INT` type is not an integer type, but rather a `long` and an `int` packed together; see section 3.10.1.1.
- See section 6.2.4 for `MPI_Aint` and more about byte counting.

6.2.2 Fortran

Table 6.4 lists standard Fortran types and common extensions. Not all the types in the right table need be supported; for instance `MPI_INTEGER16` may not exist, in which case it will be equivalent to `MPI_DATATYPE_NULL`.

The default integer type `MPI_INTEGER` is equivalent to `INTEGER(KIND=MPI_INTEGER_KIND)`.

6. MPI topic: Data types

C11 type	MPI type
<code>int8_t</code>	<code>MPI_INT8_T</code>
<code>int16_t</code>	<code>MPI_INT16_T</code>
<code>int32_t</code>	<code>MPI_INT32_T</code>
<code>int64_t</code>	<code>MPI_INT64_T</code>
<code>uint8_t</code>	<code>MPI_UINT8_T</code>
<code>uint16_t</code>	<code>MPI_UINT16_T</code>
<code>uint32_t</code>	<code>MPI_UINT32_T</code>
<code>uint64_t</code>	<code>MPI_UINT64_T</code>

Table 6.3: C11 fixed width integer types.

<code>MPI_CHARACTER</code>	Character(Len=1)	<code>MPI_INTEGER1</code>
<code>MPI_INTEGER</code>		<code>MPI_INTEGER2</code>
<code>MPI_REAL</code>		<code>MPI_INTEGER4</code>
<code>MPI_DOUBLE_PRECISION</code>		<code>MPI_INTEGER8</code>
<code>MPI_COMPLEX</code>		<code>MPI_INTEGER16</code>
<code>MPI_LOGICAL</code>		<code>MPI_REAL2</code>
<code>MPI_BYTE</code>		<code>MPI_REAL4</code>
<code>MPI_PACKED</code>		<code>MPI_REAL8</code>
		<code>MPI_DOUBLE_COMPLEX</code>
		<code>Complex(Kind=Kind(0.d0))</code>

Table 6.4: Standard Fortran types (left) and common extension (right)

6.2.2.1 Fortran90 kind-defined types

If your Fortran90 code uses `KIND` to define scalar types with specified precision, you need to use the following routines to make *MPI equivalences of Fortran scalar types*:

- `MPI_Type_create_f90_integer` (figure 6.1)
- `MPI_Type_create_f90_real` (figure 6.2)
- `MPI_Type_create_f90_complex` (figure 6.3).

Example of an integer kind;

```
INTEGER ( KIND = SELECTED_INT_KIND(15) ) , &
         DIMENSION(100) :: array
INTEGER :: root , error
Type(MPI_Datatype) :: integertype

CALL MPI_Type_create_f90_integer( 15 , integertype , error )
CALL MPI_Bcast ( array , 100 , &
                 integertype , root , MPI_COMM_WORLD , error )
! error parameter optional in f08, both routines.
```

Figure 6.1 MPI_Type_create_f90_integer

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Type_create_f90_integer</code>		<pre> r decimal exponent range, i.e., number of decimal digits newtype the requested MPI datatype) </pre>	int	INTEGER	IN

Figure 6.2 MPI_Type_create_f90_real

Name	Param name	Explanation	C type	F type	inout
		MPI_Type_create_f90_real (
	p	precision, in decimal digits	int	INTEGER	IN
	r	decimal exponent range	int	INTEGER	IN
	newtype	the requested MPI datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)				

Code :

```

! kindsend.F90
Integer,parameter :: digits=16
Integer,parameter :: ip =
     Selected_Int_Kind(digits)
Integer (kind=ip) :: data
Type(MPI_Datatype) :: mpi_ip
Call MPI_Type_create_f90_integer(
     digits,mpi_ip)
if (rank==0) then
    print *, "Fortran type has range",
    range(data)
    call MPI_Send( data,1,mpi_ip, 1,0,
    comm )
else if (rank==1) then
    call MPI_Recv( data,1,mpi_ip, 0,0,
    comm, MPI STATUS IGNORE )

```

Output:

```
[examples/mpi/f08] kindsend:  
Fortran type has range  
  Sending: 7290000000000000  
Received: 7290000000000000
```

Example of a real kind:

```

REAL ( KIND = SELECTED_REAL_KIND(15 ,300) ) , &
      DIMENSION(100) :: array
CALL MPI_Type_create_f90_real( 15 , 300 , realtype , error )

```

Example of a complex kind:

6. MPI topic: Data types

Figure 6.3 MPI_Type_create_f90_complex

Name	Param name	Explanation	C type	F type	inout
MPI_Type_create_f90_complex					
p	precision, in decimal digits		int	INTEGER	IN
r	decimal exponent range		int	INTEGER	IN
newtype	the requested MPI datatype		MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)					

```
COMPLEX ( KIND = SELECTED_REAL_KIND(15 ,300) ) , &
DIMENSION(100) :: array
CALL MPI_Type_create_f90_complex( 15 , 300 , complextyp , error )
```

Remark The MPI types thus created are predefined data types, so there is no need to commit or free them.

6.2.3 Python

Python note 18: Predefined data types. This section 6.2.3 discusses of predefined datatypes in Python.

In python, all buffer data comes from *Numpy*.

mpi4py type	NumPy type
MPI.INT	np.intc
	np.int32
MPI.LONG	np.int64
MPI.FLOAT	np.float32
MPI.DOUBLE	np.float64

In this table we see that Numpy has three integer types, one corresponding to C `ints`, and two with the number of bits explicitly indicated. There used to be a `np.int` type, but this is deprecated as of *Numpy 1.20*.

Examples:

```
Code:
## inttype.py
sizeofint = np.dtype('int32').itemsize
print("Size of numpy int32: {}".format(
    sizeofint))
sizeofint = np.dtype('intc').itemsize
print("Size of C int: {}".format(
    sizeofint))
```

```
Output:
[examples/mpi/p] intsize:
Size of numpy int32: 4
Size of C int: 4
```

```
## allgatherv.py
mycount = procid+1
my_array = np.empty(mycount,dtype=np.float64)
```

6.2.3.1 Type correspondences MPI / Python

Above we saw that the number of bytes of a Numpy type can be deduced from

```
sizeofint = np.dtype('intc').itemsize
```

It is possible to derive the Numpy type corresponding to an MPI type:

```
## typesize.py
datatype = MPI.FLOAT
typecode = MPI._typecode(datatype)
assert typecode is not None # check MPI datatype is built-in
dtype = np.dtype(typecode)
```

6.2.4 Byte addressing types

So far we have mostly been taking about datatypes in the context of sending them. The `MPI_Aint` type is not so much for sending, as it is for describing the size of objects, such as the size of an `MPI_Win` object (section 9.1) or byte displacements in `MPI_Type_create_hindexed`.

Addresses have type `MPI_Aint`. The start of the address range is given in `MPI_BOTTOM`. See also the `MPI_Sizeof` (section 6.2.5) and `MPI_Get_address` (section 6.3.6) routines.

Variables of type `MPI_Aint` can be sent as `MPI_AINT`:

```
MPI_Aint address;
MPI_Send( address,1,MPI_AINT, ... );
```

See section 9.5.3 for an example.

In order to prevent overflow errors in byte calculations there are support routines `MPI_Aint_add`

```
MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)
```

and similarly `MPI_Aint_diff`.

Fortran note 11: Byte counting types in Fortran. The equivalent of `MPI_Aint` in Fortran is an integer of kind `MPI_ADDRESS_KIND`:

```
integer(kind=MPI_ADDRESS_KIND) :: winsize
```

Using this integer kind to compute the size of a window also requires being able to query the size of the datatype in that window. See section 6.2.5 for details.

Example usage in `MPI_Win_create`:

6. MPI topic: Data types

Figure 6.4 MPI_Type_match_size

Name	Param name	Explanation	C type	F type	inout
MPI_Type_match_size		(
	typeclass	generic type specifier	int	INTEGER	IN
	size	size, in bytes, of representation	int	INTEGER	IN
	datatype	datatype with correct type, size	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)					

```
call MPI_Szef(windowdata,window_element_size,ierr)
window_size = window_element_size*500
call MPI_Win_create( windowdata,window_size,window_element_size,... )
```

Python note 19: Size of numpy types. Here is a good way for finding the size of *numpy* datatypes in bytes:

```
## putfence.py
intsize = np.dtype('int').itemsize
window_data = np.zeros(2,dtype=int)
win = MPI.Win.Create(window_data,intsize,comm=comm)
```

6.2.5 Matching language type to MPI type

In some circumstances you may want to find the MPI type that corresponds to a type in your programming language.

- In C++ functions and classes can be templated, meaning that the type is not fully known:

```
template<typename T> {
    class something<T> {
public:
    void dosend(T input) {
        MPI_Send( &input,1,/* ????? */ );
    };
};
```

(Note that in MPL this is hardly ever needed because MPI calls are templated there.)

- Petsc installations use a generic identifier `PetscScalar` (or `PetscReal`) with a configuration-dependent realization.
- The size of a datatype is not always statically known, for instance if the Fortran `KIND` keyword is used.

Here are some MPI mechanisms that address this problem.

6.2.5.1 Type matching in C

Datatypes in C can be translated to MPI types with `MPI_Type_match_size` (figure 6.4) where the `typeclass`

Figure 6.5 MPI_Type_size

Name	Param name	Explanation	C type	F type	inout
MPI_Type_size (

argument is one of MPI_TYPECLASS_REAL, MPI_TYPECLASS_INTEGER, MPI_TYPECLASS_COMPLEX.

Code:

```
// typematch.c
float x5;
double x10;
int s5,s10;
MPI_Datatype mpi_x5,mpi_x10;

MPI_Type_match_size
  (MPI_TYPECLASS_REAL,sizeof(x5),&
   mpi_x5);
MPI_Type_match_size
  (MPI_TYPECLASS_REAL,sizeof(x10),&
   mpi_x10);
MPI_Type_size(mpi_x5,&s5);
printf("float: size=%d, mpi size=%d\n"
      ,
      sizeof(x5),s5);
MPI_Type_size(mpi_x10,&s10);
printf("double: size=%d, mpi size=%d\n"
      ,
      sizeof(x10),s10);
```

Output:

```
[examples/mpi/c] typematch:
mpiexec -n 1 ./typematch
float: size=4, mpi size=4
double: size=8, mpi size=8
```

The space that MPI takes for a structure type can be queried in a variety of ways. First of all MPI_Type_size (figure 6.5) counts the *datatype size* as the number of bytes occupied by the data in a type. That means that in an *MPI vector datatype* it does not count the gaps.

```
// typesize.c
MPI_Type_vector(count,bs,stride,MPI_DOUBLE,&newtype);
MPI_Type_commit(&newtype);
MPI_Type_size(newtype,&size);
ASSERT( size==(count*bs)*sizeof(double) );
```

Figure 6.6 MPI_Sizeof

Name	Param name	Explanation	C type	F type	inout
MPI_Sizeof	()				

6.2.5.2 Type matching in Fortran

In Fortran, the size of the datatype in the language can be obtained with `MPI_Sizeof` (figure 6.6) (note the nonoptional error parameter!). This routine is deprecated in MPI-4: use of `storage_size` (which reports the number of bits) and/or `c_sizeof` (from the `iso_c_binding` module, which reports bytes) is recommended.

```
!! matchkind.F90
call MPI_Sizeof(x10,s10,ierr)
call MPI_Type_match_size(MPI_TYPECLASS_REAL,s10,mpi_x10)
call MPI_Type_size(mpi_x10,s10)
print *, "10 positions supported, MPI type size is",s10
```

Petsc has its own translation mechanism; see section 31.2.

6.3 Derived datatypes

MPI allows you to create your own data types, somewhat (but not completely...) analogous to defining structures in a programming language. MPI data types are mostly of use if you want to send multiple items in one message.

There are two problems with using only predefined datatypes as you have seen so far.

- MPI communication routines can only send multiples of a single data type: it is not possible to send items of different types, even if they are contiguous in memory. It would be possible to use the `MPI_BYTE` data type, but this is not advisable.
- It is also ordinarily not possible to send items of one type if they are not contiguous in memory. You could of course send a contiguous memory area that contains the items you want to send, but that is wasteful of bandwidth, and of memory space on the receiving side.

With MPI data types you can solve these problems in several ways.

- You can create a new *contiguous data type* consisting of an array of elements of another data type. There is no essential difference between sending one element of such a type and multiple elements of the component type.
- You can create a *vector data type* consisting of regularly spaced blocks of elements of a component type. This is a first solution to the problem of sending noncontiguous data.
- For not regularly spaced data, there is the *indexed data type*, where you specify an array of index locations for blocks of elements of a component type. The blocks can each be of a different size.
- The *struct data type* can accommodate multiple data types.

And you can combine these mechanisms to get irregularly spaced heterogeneous data, et cetera.

6.3.1 Basic calls

The typical sequence of calls for creating a new datatype is as follows:

- You need a variable for the datatype; this is of type `MPI_Datatype`.
- There is a create call, followed by a ‘commit’ call where MPI performs internal bookkeeping and optimizations; we will discuss this in great detail below.
- The type needs to be ‘committed’. After this:
- The datatype is used, possibly multiple times;
- When the datatype is no longer needed, it must be freed to prevent memory leaks; see section 6.3.1.2.

In code:

```
MPI_Datatype newtype;
MPI_Type_something( < oldtype specifications >, &newtype );
MPI_Type_commit( &newtype );
/* code that uses your new type */
MPI_Type_free( &newtype );
```

In Fortran2008:

```
Type(MPI_Datatype) :: newvectortype
call MPI_Type_something( <oldtype specification>, &
    newvectortype)
call MPI_Type_commit(newvectortype)
!! code that uses your type
call MPI_Type_free(newvectortype)
```

Python note 20: Derived type handling. The various type creation routines are methods of the datatype classes, after which commit and free are methods on the new type.

```
## vector.py
source = np.empty(stride*count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)
if procid==sender:
    newvectortype = MPI.DOUBLE.Create_vector(count,1,stride)
    newvectortype.Commit()
    comm.Send([source,1,newvectortype],dest=the_other)
    newvectortype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)
```

MPL note 47: Derived type handling. In MPL type creation routines are in the main namespace, templated over the datatypes.

```
// vector.cxx
vector<double>
source(stride*count);
if (procno==sender) {
    mpl::strided_vector_layout<double>
        newvectortype(count,1,stride);
    comm_world.send
```

6. MPI topic: Data types

```
(source.data(), newvectortype, the_other);  
}
```

The commit call is part of the type creation, and freeing is done in the destructor.

MPL note 48: Layouts.

```
namespace mpl {  
    template <typename T> class layout; // Basisklasse  
    template <typename T> class null_layout; // MPI_DATATYPE_NULL  
    template <typename T> class empty_layout; // leere Nachricht  
    template <typename T> class contiguous_layout; // MPI_Type_contiguous  
    template <typename T> class vector_layout; // MPI_Type_contiguous  
    template <typename T> class strided_vector_layout; // MPI_Type_vector  
    template <typename T> class indexed_layout; // MPI_Type_indexed  
    template <typename T> class hindexed_layout; // MPI_Type_create_hindex  
    template <typename T> class indexed_block_layout; // MPI_Type_create_indexed_block  
    template <typename T> class hindexed_block_layout; // MPI_Type_create_hindex  
    template <typename T> class iterator_layout; // MPI_Type_create_hindex  
    template <typename T> class subarray_layout; // MPI_Type_create_subarray  
    class heterogeneous_layout; // MPI_Type_create_struct  
}
```

6.3.1.1 Create calls

The `MPI_Datatype` variable gets its value by a call to one of the following routines:

- `MPI_Type_contiguous` for contiguous blocks of data; section 6.3.2;
- `MPI_Type_vector` for regularly strided data; section 6.3.3;
- `MPI_Type_create_subarray` for subsets out higher dimensional block; section 6.3.4;
- `MPI_Type_create_struct` for heterogeneous irregular data; section 6.3.7;
- `MPI_Type_indexed` and `MPI_Type_hindex` for irregularly strided data; section 6.3.6.

These calls take an existing type, whether predefined or also derived, and produce a new type.

6.3.1.2 Commit and free

It is necessary to call `MPI_Type_commit` (figure 6.7) on a new data type, which makes MPI do the indexing calculations for the data type.

When you no longer need the data type, you call `MPI_Type_free` (figure 6.8). (This is typically not needed in OO APIs.) This has the following effects:

- The definition of the datatype identifier will be changed to `MPI_DATATYPE_NULL`.
- Any communication using this data type, that was already started, will be completed successfully.
- Datatypes that are defined in terms of this data type will still be usable.

Figure 6.7 MPI_Type_commit

Name	Param name	Explanation	C type	F type	inout
MPI_Type_commit (

datatype datatype that is committed

MPI_Datatype*	TYPE	INOUT
	(MPI_Datatype)	

)

MPL:

Done as part of the type create call.

Figure 6.8 MPI_Type_free

Name	Param name	Explanation	C type	F type	inout
MPI_Type_free (

datatype datatype that is freed

MPI_Datatype*	TYPE	INOUT
	(MPI_Datatype)	

)

MPL:

Done in the destructor.

6.3.2 Contiguous type

The simplest derived type is the ‘contiguous’ type, constructed with `MPI_Type_contiguous` (figure 6.9).

A contiguous type describes an array of items of an predefined or earlier defined type. There is no difference between sending one item of a contiguous type and multiple items of the constituent type. This is illustrated in figure 6.1.

```
// contiguous.c
MPI_Datatype newvectortype;
if (procno==sender) {
    MPI_Type_contiguous(count,MPI_DOUBLE,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,receiver,0,comm);
    MPI_Type_free(&newvectortype);
} else if (procno==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,count,MPI_DOUBLE, sender,0,comm,
             &recv_status);
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    ASSERT(count==recv_count);
}

!! contiguous.F90
integer :: newvectortype
if (mytid==sender) then
    call MPI_Type_contiguous(count,MPI_DOUBLE_PRECISION,newvectortype)
```

6. MPI topic: Data types

Figure 6.9 MPI_Type_contiguous

Name	Param name	Explanation	C type	F type	inout
MPI_Type_contiguous					
	MPI_Type_contiguous_c				
	count	replication count	[int MPI_Count]	INTEGER	IN
	oldtype	old datatype	MPI_Datatype	TYPE (MPI_Datatype)	IN
	newtype	new datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT

Python:

```
Create_contiguous(self, int count)
```



Figure 6.1: A contiguous datatype is built up out of elements of a constituent type

```

call MPI_Type_commit(newvectortype)
call MPI_Send(source,1,newvectortype,receiver,0,comm)
call MPI_Type_free(newvectortype)
else if (mytid==receiver) then
    call MPI_Recv(target,count,MPI_DOUBLE_PRECISION, sender,0,comm,&
        recv_status)
    call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count)
    !ASSERT(count==recv_count);
end if

## contiguous.py
source = np.empty(count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)
if procid==sender:
    newcontiguoustype = MPI.DOUBLE.Create_contiguous(count)
    newcontiguoustype.Commit()
    comm.Send([source,1,newcontiguoustype],dest=the_other)
    newcontiguoustype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)

```

MPL note 49: Contiguous type. The MPL interface makes extensive use of `contiguous_layout`, as it is the main way to declare a nonscalar buffer.

MPL note 50: Contiguous composing. Contiguous layouts can only use predefined types or other contiguous layouts as their ‘old’ type. To make a contiguous type for other layouts, use `vector_layout`:

Figure 6.10 MPI_Type_vector

Name	Param name	Explanation	C type	F type	inout
MPI_Type_vector					
	MPI_Type_vector_c				
count		number of blocks	[int MPI_Count]	INTEGER	IN
blocklength		number of elements in each block	[int MPI_Count]	INTEGER	IN
stride		number of elements between start of each block	[int MPI_Count]	INTEGER	IN
oldtype		old datatype	MPI_Datatype	TYPE (MPI_Datatype)	IN
newtype		new datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)					

MPI:

```
template<typename T>
class strided_vector_layout : public mpl::layout<T>

strided_vector_layout(int count, int blocklength, int stride);
strided_vector_layout(int count, int blocklength, int stride, const layout<T> &l);
```

Python:

```
MPI.Datatype.Create_vector(self, int count, int blocklength, int stride)
```

```
// contiguous.cxx
mpl::contiguous_layout<int> type1(7);
mpl::vector_layout<int> type2(8,type1);
```

(Contrast this with `strided_vector_layout`; note 51.)

6.3.3 Vector type

The simplest noncontiguous datatype is the ‘vector’ type, constructed with `MPI_Type_vector` (figure 6.10). A vector type describes a series of blocks, all of equal size, spaced with a constant stride. This is illustrated in figure 6.2.

The vector datatype gives the first nontrivial illustration that datatypes can be *different on the sender and receiver*. If the sender sends b blocks of length l each, the receiver can receive them as b_1 contiguous elements, either as a contiguous datatype, or as a contiguous buffer of an predefined type; see figure 6.3. The receiver has no knowledge of the stride of the datatype on the sender.

In this example a vector type is created only on the sender, in order to send a strided subset of an array; the receiver receives the data as a contiguous block.

6. MPI topic: Data types

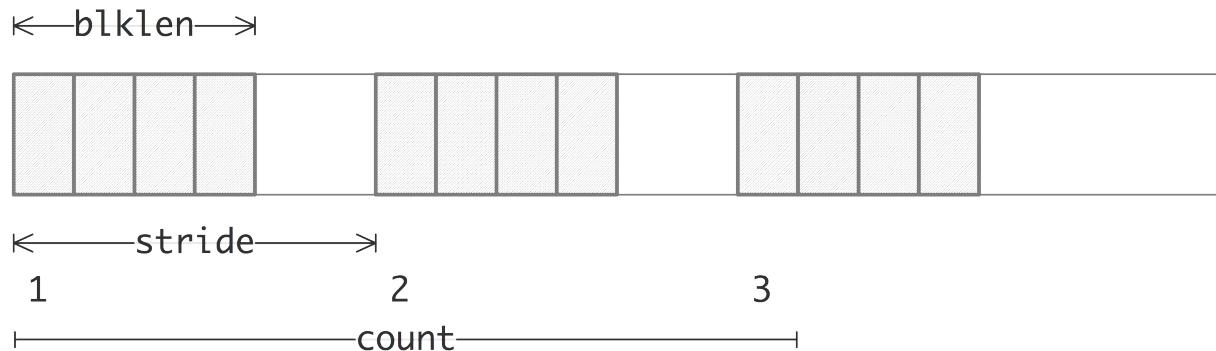


Figure 6.2: A vector datatype is built up out of strided blocks of elements of a constituent type

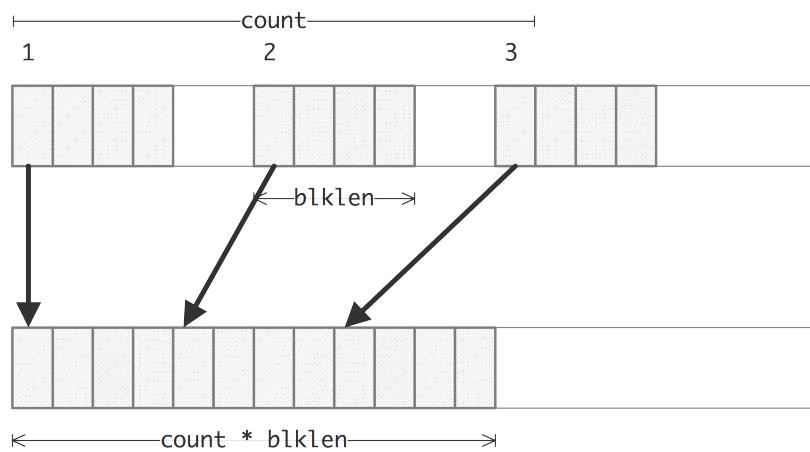


Figure 6.3: Sending a vector datatype and receiving it as predefined or contiguous

```
// vector.c
source = (double*) malloc(stride*count*sizeof(double));
target = (double*) malloc(count*sizeof(double));
MPI_Datatype newvectortype;
if (procno==sender) {
    MPI_Type_vector(count,1,stride,MPI_DOUBLE,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,the_other,0,comm);
    MPI_Type_free(&newvectortype);
} else if (procno==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
             &recv_status);
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    ASSERT(recv_count==count);
}
```

We illustrate Fortran2008:

```
if (mytid==sender) then
  call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
    newvectortype)
  call MPI_Type_commit(newvectortype)
  call MPI_Send(source,1,newvectortype,receiver,0,comm)
  call MPI_Type_free(newvectortype)
  if (.not. newvectortype==MPI_DATATYPE_NULL) then
    print *, "Trouble freeing datatype"
  else
    print *, "Datatype successfully freed"
  end if
else if (mytid==receiver) then
  call MPI_Recv(target,count,MPI_DOUBLE_PRECISION, sender,0,comm,&
    recv_status)
  call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count)
end if
```

In legacy mode Fortran90, code stays the same except that the type is declared as *Integer*:

```
!! vector.F90
integer :: newvectortype
integer :: recv_count
call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
  newvectortype,err)
call MPI_Type_commit(newvectortype,err)
```

Python note 21: Vector type. The vector creation routine is a method of the datatype class. For the general discussion, see section 6.3.1.

```
## vector.py
source = np.empty(stride*count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)
if procid==sender:
    newvectortype = MPI.DOUBLE.Create_vector(count,1,stride)
    newvectortype.Commit()
    comm.Send([source,1,newvectortype],dest=the_other)
    newvectortype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)
```

MPL note 51: Vector type. MPL has the `strided_vector_layout` class as equivalent of the vector type:

```
// vector.cxx
vector<double>
source(stride*count);
if (procno==sender) {
  mpl::strided_vector_layout<double>
    newvectortype(count,1,stride);
  comm_world.send
    (source.data(),newvectortype,the_other);
}
```

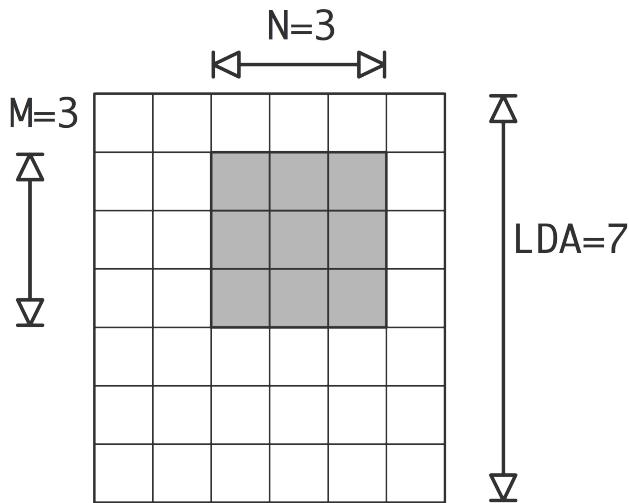


Figure 6.4: Memory layout of a row and column of a matrix in column-major storage

(See note 50 for nonstrided vectors.)

6.3.3.1 Two-dimensional arrays

Figure 6.4 indicates one source of irregular data: with a matrix on *column-major storage*, a column is stored in contiguous memory. However, a row of such a matrix is not contiguous; its elements being separated by a *stride* equal to the column length.

Exercise 6.1. How would you describe the memory layout of a submatrix, if the whole matrix has size $M \times N$ and the submatrix $m \times n$?

As an example of this datatype, consider the example of transposing a matrix, for instance to convert between C and Fortran arrays. Suppose that a processor has a matrix stored in C, row-major, layout, and it needs to send a column to another processor. If the matrix is declared as

```
int M, N; double mat[M][N]
```

then a column has M blocks of one element, spaced N locations apart. In other words:

```
MPI_Datatype MPI_column;
MPI_Type_vector(
    /* count= */ M, /* blocklength= */ 1, /* stride= */ N,
    MPI_DOUBLE, &MPI_column );
```

Sending the first column is easy:

```
MPI_Send( mat, 1, MPI_column, ... );
```

The second column is just a little trickier: you now need to pick out elements with the same stride, but starting at $A[0][1]$.

```
MPI_Send( &(mat[0][1]), 1, MPI_column, ... );
```

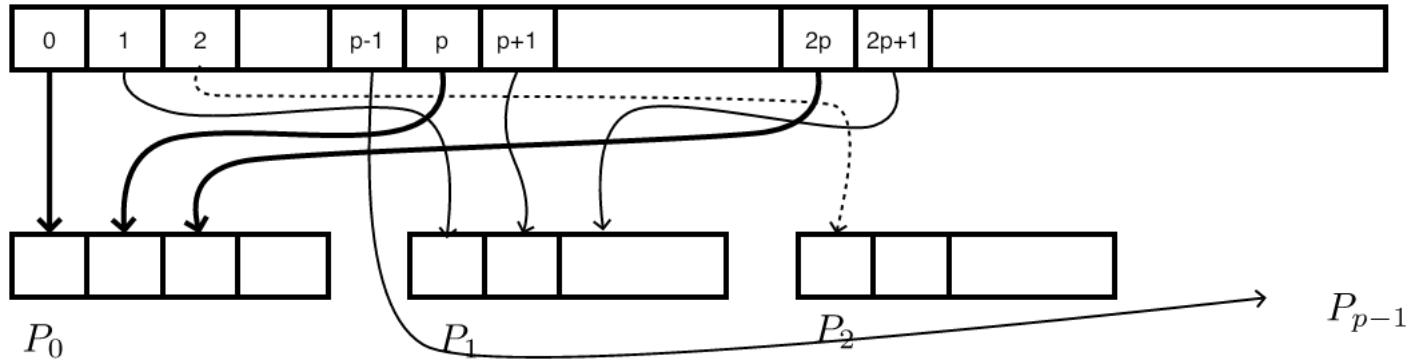


Figure 6.5: Send strided data from process zero to all others

You can make this marginally more efficient (and harder to read) by replacing the index expression by `mat+1`.

Exercise 6.2. Suppose you have a matrix of size $4N \times 4N$, and you want to send the elements $A[4*i][4*j]$ with $i, j = 0, \dots, N - 1$. How would you send these elements with a single transfer?

Exercise 6.3. Allocate a matrix on processor zero, using Fortran column-major storage. Using P `sendrecv` calls, distribute the rows of this matrix among the processors.

Python note 22: Sending from the middle of a matrix. In C and Fortran it's easy to apply a derived type to data in the middle of an array, for instance to extract an arbitrary column out of a C matrix, or row out of a Fortran matrix. While Python has no trouble describing sections from an array, usually it copies these instead of taking the address. Therefore, it is necessary to convert the matrix to a buffer and compute an explicit offset in bytes:

```
## rowcol.py
rowsize = 4; colszie = 5
coltype = MPI.INT.Create_vector(4, 1, 5)
coltype.Commit()
columntosend = 2
comm.Send\(
    [np.frombuffer(matrix.data, intc,
                   offset=columntosend*np.dtype('intc').itemsize),
     1,coltype],
    receiver)
```

Exercise 6.4. Let processor 0 have an array x of length $10P$, where P is the number of processors. Elements $0, P, 2P, \dots, 9P$ should go to processor zero, $1, P + 1, 2P + 1, \dots$ to processor 1, et cetera.

- Code this as a sequence of send/recv calls, using a vector datatype for the send, and a contiguous buffer for the receive.
- For simplicity, skip the send to/from zero. What is the most elegant solution if you want to include that case?
- For testing, define the array as $x[i] = i$.

6. MPI topic: Data types

Figure 6.11 MPI_Type_create_subarray

Name	Param name	Explanation	C type	F type	inout
MPI_Type_create_subarray					
MPI_Type_create_subarray_c					
ndims		number of array dimensions	int	INTEGER	IN
array_of_sizes		number of elements of type oldtype in each dimension of the full array	[const int[] MPI_Count[]] (ndims)	INTEGER	IN
array_of_subsizes		number of elements of type oldtype in each dimension of the subarray	[const int[] MPI_Count[]] (ndims)	INTEGER	IN
array_of_starts		starting coordinates of the subarray in each dimension	[const int[] MPI_Count[]] (ndims)	INTEGER	IN
order		array storage order flag	int	INTEGER	IN
oldtype		old datatype	MPI_Datatype	TYPE (MPI_Datatype)	IN
newtype		new datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)					

Python:

```
MPI.Datatype.Create_subarray
    (self, sizes, subsizes, starts, int order=ORDER_C)
```

(There is a skeleton for this exercise under the name `stridesend`.)

Exercise 6.5. Write code to compare the time it takes to send a strided subset from an array: copy the elements by hand to a smaller buffer, or use a vector data type. What do you find? You may need to test on fairly large arrays.

6.3.4 Subarray type

The vector datatype can be used for blocks in an array of dimension more than 2 by using it recursively. However, this gets tedious. Instead, there is an explicit subarray type `MPI_Type_create_subarray` (figure 6.11). This describes the dimensionality and extent of the array, and the starting point (the ‘upper left corner’) and extent of the subarray.

MPL note 52: Subarray layout. The templated `subarray_layout` class is constructed from a vector of triplets of global size / subblock size / first coordinate.

```
mpl::subarray_layout<int>(
  { {ny, ny_1, ny_0}, {nx, nx_1, nx_0} }
);
```

Exercise 6.6. Assume that your number of processors is $P = Q^3$, and that each process has an array of identical size. Use `MPI_Type_create_subarray` to gather all data onto a root process. Use a sequence of send and receive calls; `MPI_Gather` does not work here. (There is a skeleton for this exercise under the name `cubegather`.)

Fortran note 12: Subarrays. Subarrays are naturally supported in Fortran through array sections.

```
!! section.F90
integer,parameter :: siz=20
real,dimension(siz,siz) :: matrix = [ ((j+(i-1)*siz,i=1,siz),j=1,siz) ]
real,dimension(2,2) :: submatrix
if (procno==0) then
  call MPI_Send(matrix(1:2,1:2),4,MPI_REAL,1,0,comm)
else if (procno==1) then
  call MPI_Recv(submatrix,4,MPI_REAL,0,0,comm,MPI_STATUS_IGNORE)
  if (submatrix(2,2)==22) then
    print *, "Yay"
  else
    print *, "nay...."
  end if
end if
```

However, there is a subtlety with non-blocking operations: for a non-contiguous buffer a temporary is created, which is released after the MPI call. This is correct for blocking sends, but for non-blocking the temporary has to stay around till the wait call.

```
!! sectionisend.F90
integer :: siz
real,dimension(:, :, allocatable) :: matrix
real,dimension(2,2) :: submatrix
siz = 20
allocate( matrix(siz,siz) )
matrix = reshape( [ ((j+(i-1)*siz,i=1,siz),j=1,siz) ], (/siz,siz/) )
call MPI_Isend(matrix(1:2,1:2),4,MPI_REAL,1,0,comm,request)
call MPI_Wait(request,MPI_STATUS_IGNORE)
deallocate(matrix)
```

In MPI-3 the variable `MPI_SUBARRAYS_SUPPORTED` indicates support for this mechanism:

```
if ( .not. MPI_SUBARRAYS_SUPPORTED ) then
  print *, "This code will not work"
  call MPI_Abort(comm,0)
end if
```

The possibilities for the `order` parameter are `MPI_ORDER_C` and `MPI_ORDER_FORTRAN`. However, this has nothing to do with the order of traversal of elements; it determines how the bounds of the subarray are interpreted. As an example, we fill a 4×4 array in C order with the numbers $0 \dots 15$, and send the $[0, 1] \times [0 \dots 4]$ slice two ways, first C order, then Fortran order:

```
// row2col.c
#define SIZE 4
int
```

```
    sizes[2], subsizes[2], starts[2];
    sizes[0] = SIZE; sizes[1] = SIZE;
    subsizes[0] = SIZE/2;   subsizes[1] = SIZE;
    starts[0] = starts[1] = 0;
    MPI_Type_create_subarray
        (2,sizes,subsizes,starts,
         MPI_ORDER_C,MPI_DOUBLE,&rrowtype);
    MPI_Type_create_subarray
        (2,sizes,subsizes,starts,
         MPI_ORDER_FORTRAN,MPI_DOUBLE,&coltype);
```

The receiver receives the following, formatted to bring out where the numbers originate:

Received C order:

```
0.000 1.000 2.000 3.000
4.000 5.000 6.000 7.000
```

Received F order:

```
0.000 1.000
4.000 5.000
8.000 9.000
12.000 13.000
```

6.3.5 Distributed array type

Each dimension can independently be distributed as `MPI_DISTRIBUTE_BLOCK`, `MPI_DISTRIBUTE_CYCLIC`, `MPI_DISTRIBUTE_NONE`,

With the cyclic distribution, the amount of cyclicity can be indicated by setting `dargs[id]` to a certain number.

With the block distribution, blocks can be set explicitly in `dargs[id]`, but `MPI_DISTRIBUTE_DFLT_DARG` causes an even distribution to be found.

Ordering can be `MPI_ORDER_C` or `MPI_ORDER_FORTRAN`.

6.3.6 Indexed type

The indexed datatype, constructed with `MPI_Type_indexed` (figure 6.12) can send arbitrarily located elements from an array of a single datatype. You need to supply an array of index locations, plus an array of blocklengths with a separate blocklength for each index. The total number of elements sent is the sum of the blocklengths.

The following example picks items that are on prime number-indexed locations.

```
// indexed.c
displacements = (int*) malloc(count*sizeof(int));
blocklengths = (int*) malloc(count*sizeof(int));
source = (int*) malloc(totalcount*sizeof(int));
target = (int*) malloc(targetbuffersize*sizeof(int));
```

Figure 6.12 MPI_Type_indexed

Name	Param name	Explanation	C type	F type	inout
MPI_Type_indexed (
MPI_Type_indexed_c (
count		number of blocks---also number of entries in array_of_displacements and array_of_blocklengths	[int MPI_Count]	INTEGER	IN
array_of_blocklengths		number of elements per block	[const int[] MPI_Count[]]	INTEGER (count)	IN
array_of_displacements		displacement for each block, in multiples of oldtype	[const int[] MPI_Count[]]	INTEGER (count)	IN
oldtype		old datatype	MPI_Datatype	TYPE (MPI_Datatype)	IN
newtype		new datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)					

Python:

```
MPI.Datatype.Create_indexed(self, blocklengths,displacements )
```

```
MPI_Datatype newvectortype;
if (procno==sender) {
    MPI_Type_indexed(count,blocklengths,displacements,MPI_INT,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,the_other,0,comm);
    MPI_Type_free(&newvectortype);
} else if (procno==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,targetbuffersize,MPI_INT,the_other,0,comm,
             &recv_status);
    MPI_Get_count(&recv_status,MPI_INT,&recv_count);
    ASSERT(recv_count==count);
}
```

For Fortran we show the legacy syntax for once:

```
!! indexed.F90
integer :: newvectortype;
ALLOCATE(indices(count))
ALLOCATE(blocklengths(count))
ALLOCATE(source(totalcount))
ALLOCATE(target(count))
if (mytid==sender) then
```

6. MPI topic: Data types

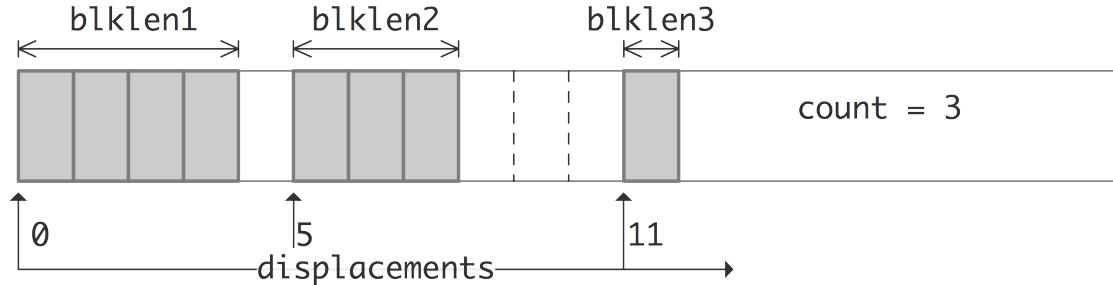


Figure 6.6: The elements of an MPI Indexed datatype

```

call MPI_Type_indexed(count,blocklengths,indices,MPI_INT,&
    newvectortype,err)
call MPI_Type_commit(newvectortype,err)
call MPI_Send(source,1,newvectortype,receiver,0,comm,err)
call MPI_Type_free(newvectortype,err)
else if (mytid==receiver) then
    call MPI_Recv(target,count,MPI_INT,source,0,comm,&
        recv_status,err)
    call MPI_Get_count(recv_status,MPI_INT,recv_count,err)
    ! ASSERT(recv_count==count);
end if

## indexed.py
displacements = np.empty(count,dtype=int)
blocklengths = np.empty(count,dtype=int)
source = np.empty(totalcount,dtype=np.float64)
target = np.empty(count,dtype=np.float64)
if procid==sender:
    newindextype = MPI.DOUBLE.Create_indexed(blocklengths,displacements)
    newindextype.Commit()
    comm.Send([source,1,newindextype],dest=the_other)
    newindextype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)

```

MPL note 53: Indexed type. In MPL, the `indexed_layout` is based on a vector of 2-tuples denoting block length / block location.

```

// indexed.cxx
const int count = 5;
mpl::contiguous_layout<int>
    fiveints(count);
mpl::indexed_layout<int>
    indexed_where{ { {1,2}, {1,3}, {1,5}, {1,7}, {1,11} } };

if (procno==sender) {
    comm_world.send( source_buffer.data(),indexed_where, receiver );
} else if (procno==receiver) {

```

```

auto recv_status =
    comm_world.recv( target_buffer.data(), fiveints, sender );
int recv_count = recv_status.get_count<int>();
assert(recv_count==count);
}

```

MPL note 54: Layouts for gatherv. The size/displacement arrays for `MPI_Gatherv` / `MPI_Alltoallv` are handled through a `layouts` object, which is basically a vector of `layout` objects.

```

mpl::layouts<int> receive_layout;
for ( int iproc=0, loc=0; iproc<nprocs; iproc++ ) {
    auto siz = size_buffer.at(iproc);
    receive_layout.push_back(
        ( mpl::indexed_layout<int>( {{ siz, loc } } ) );
    loc += siz;
}

```

MPL note 55: Indexed block type. For the case where all block lengths are the same, use `indexed_block_layout`:

```

// indexedblock.cxx
mpl::indexed_block_layout<int>
    indexed_where( 1, {2,3,5,7,11} );
comm_world.send( source_buffer.data(), indexed_where, receiver );

```

You can also `MPI_Type_create_hindexed` which describes blocks of a single old type, but with index locations in bytes, rather than in multiples of the old type.

```

int MPI_Type_create_hindexed
    (int count, int blocklens[], MPI_Aint indices[],
     MPI_Datatype old_type, MPI_Datatype *newtype)

```

A slightly simpler version, `MPI_Type_create_hindexed_block` (figure 6.13) assumes constant block length.

There is an important difference between the `hindexed` and the above `MPI_Type_indexed`: that one described offsets from a base location; these routines describes absolute memory addresses. You can use this to send for instance the elements of a linked list. You would traverse the list, recording the addresses of the elements with `MPI_Get_address` (figure 6.14). (The routine `MPI_Address` is deprecated.)

In C++ you can use this to send an `std::vector`, that is, a vector object from the *C++ standard library*, if the component type is a pointer.

6.3.7 Struct type

The structure type, created with `MPI_Type_create_struct` (figure 6.15), can contain multiple data types. (The routine `MPI_Type_struct` is deprecated with MPI-3.) The specification contains a ‘count’ parameter that specifies how many blocks there are in a single structure. For instance,

6. MPI topic: Data types

Figure 6.13 MPI_Type_create_hindexed_block

Name	Param name	Explanation	C type	F type	inout
MPI_Type_create_hindexed_block					
	MPI_Type_create_hindexed_block_c				
count		number of blocks---also number of entries in array_of_displacements	[int MPI_Count]	INTEGER	IN
blocklength		number of elements in each block	[int MPI_Count]	INTEGER	IN
array_of_displacements		byte displacement of each block	[const MPI_Aint[] MPI_Count[]]	INTEGER (KIND=MPI_ADDRESS_KIND) (count)	IN
oldtype		old datatype	MPI_Datatype	TYPE (MPI_Datatype)	IN
newtype		new datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)					

Figure 6.14 MPI_Get_address

Name	Param name	Explanation	C type	F type	inout
MPI_Get_address					
	location	location in caller memory	const void*	TYPE(*), DIMENSION(..)	IN
	address	address of location	MPI_Aint*	INTEGER (KIND=MPI_ADDRESS_KIND)	OUT
)					

```
struct {
    int i;
    float x,y;
} point;
```

has two blocks, one of a single integer, and one of two floats. This is illustrated in figure 6.7.

count The number of blocks in this datatype. The **blocklengths**, **displacements**, **types** arguments have to be at least of this length.

blocklengths array containing the lengths of the blocks of each datatype.

displacements array describing the relative location of the blocks of each datatype.

types array containing the datatypes; each block in the new type is of a single datatype; there can be multiple blocks consisting of the same type.

In this example, unlike the previous ones, both sender and receiver create the structure type. With structures it is no longer possible to send as a derived type and receive as a array of a simple type. (It would be possible to send as one structure type and receive as another, as long as they have the same *datatype*

Figure 6.15 MPI_Type_create_struct

Name	Param name	Explanation	C type	F type	inout
MPI_Type_create_struct					
MPI_Type_create_struct_c					
count		number of blocks---also number of entries in arrays	[int MPI_Count]	INTEGER	IN
array_of_types		array_of_displacements,			
array_of_displacements		and array_of_blocklengths			
array_of_blocklengths		number of elements in each block	[const int[] MPI_Count[]]	INTEGER (count)	IN
array_of_displacements		byte displacement of each block	[const MPI_Aint[] MPI_Count[]]	INTEGER (KIND=MPI_ADDRESS_KIND) (count)	IN
array_of_types		type of elements in each block	const MPI_Datatype[]	TYPE (MPI_Datatype) (count)	IN
newtype		new datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)					

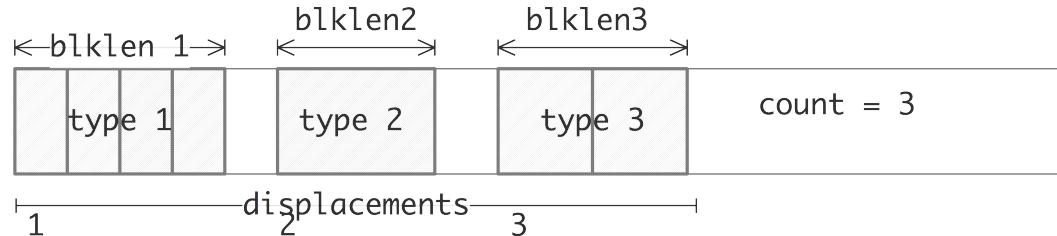


Figure 6.7: The elements of an MPI Struct datatype

signature.)

```
// struct.c
struct object {
    char c;
    double x[2];
    int i;
};

MPI_Datatype newstructuretype;
int structlen = 3;
int blocklengths[structlen]; MPI_Datatype types[structlen];
MPI_Aint displacements[structlen];

/*
```

6. MPI topic: Data types

```
* where are the components relative to the structure?  
*/  
MPI_Aint current_displacement=0;  
  
// one character  
blocklengths[0] = 1; types[0] = MPI_CHAR;  
displacements[0] = (size_t)&(myobject.c) - (size_t)&myobject;  
  
// two doubles  
blocklengths[1] = 2; types[1] = MPI_DOUBLE;  
displacements[1] = (size_t)&(myobject.x) - (size_t)&myobject;  
  
// one int  
blocklengths[2] = 1; types[2] = MPI_INT;  
displacements[2] = (size_t)&(myobject.i) - (size_t)&myobject;  
  
MPI_Type_create_struct(structlen,blocklengths,displacements,types,&newstructuretype);  
MPI_Type_commit(&newstructuretype);  
if (procno==sender) {  
    MPI_Send(&myobject,1,newstructuretype,the_other,0,comm);  
} else if (procno==receiver) {  
    MPI_Recv(&myobject,1,newstructuretype,the_other,0,comm,MPI_STATUS_IGNORE);  
}  
MPI_Type_free(&newstructuretype);
```

Note the displacement calculations in this example, which involve some not so elegant pointer arithmetic. The following Fortran code uses **MPI_Get_address**, which is more elegant, and in fact the only way address calculations can be done in Fortran.

```
!! struct.F90  
Type object  
    character :: c  
    real*8,dimension(2) :: x  
    integer :: i  
end type object  
type(object) :: myobject  
integer,parameter :: structlen = 3  
type(MPI_Datatype) :: newstructuretype  
integer,dimension(structlen) :: blocklengths  
type(MPI_Datatype),dimension(structlen) :: types;  
MPI_Aint,dimension(structlen) :: displacements  
MPI_Aint :: base_displacement, next_displacement  
if (procno==sender) then  
    myobject%c = 'x'  
    myobject%x(0) = 2.7; myobject%x(1) = 1.5  
    myobject%i = 37  
  
    !! component 1: one character  
    blocklengths(1) = 1; types(1) = MPI_CHAR  
    call MPI_Get_address(myobject,base_displacement)  
    call MPI_Get_address(myobject%c,next_displacement)  
    displacements(1) = next_displacement-base_displacement
```

```

!! component 2: two doubles
blocklengths(2) = 2; types(2) = MPI_DOUBLE
call MPI_Get_address(myobject%x,next_displacement)
displacements(2) = next_displacement-base_displacement

!! component 3: one int
blocklengths(3) = 1; types(3) = MPI_INT
call MPI_Get_address(myobject%i,next_displacement)
displacements(3) = next_displacement-base_displacement

if (procno==sender) then
  call MPI_Send(myobject,1,newstructuretype,receiver,0,comm)
else if (procno==receiver) then
  call MPI_Recv(myobject,1,newstructuretype,receiver,0,comm,MPI_STATUS_IGNORE)
end if
call MPI_Type_free(newstructuretype)

```

It would have been incorrect to write

```

displacement[0] = 0;
displacement[1] = displacement[0] + sizeof(char);

```

since you do not know the way the *compiler* lays out the structure in memory¹.

If you want to send more than one structure, you have to worry more about padding in the structure. You can solve this by adding an extra type MPI_UB for the ‘upper bound’ on the structure:

```

displacements[3] = sizeof(myobject); types[3] = MPI_UB;
MPI_Type_create_struct(struclen+1,...);

```

MPL note 56: Struct type scalar. One could describe the MPI struct type as a collection of displacements, to be applied to any set of items that conforms to the specifications. An MPL heterogeneous_layout on the other hand, incorporates the actual data. Thus you could write

```

// structscalar.cxx
char c; double x; int i;
if (procno==sender) {
  c = 'x'; x = 2.4; i = 37;
mpl::heterogeneous_layout object( c,x,i );
if (procno==sender)
  comm_world.send( mpl::absolute,object,receiver );
else if (procno==receiver)
  comm_world.recv( mpl::absolute,object,receiver );

```

Here, the absolute indicates the lack of an implicit buffer: the layout is absolute rather than a relative description.

MPL note 57: Struct type general. More complicated data than scalars takes more work:

```

// struct.cxx
char c; vector<double> x(2); int i;

```

1. Homework question: what does the language standard say about this?

```

if (procno==sender) {
    c = 'x'; x[0] = 2.7; x[1] = 1.5; i = 37; }
mpl::heterogeneous_layout object
( c,
  mpl::make_absolute(x.data(),mpl::vector_layout<double>(2)),
  i );
if (procno==sender) {
    comm_world.send( mpl::absolute,object,receiver );
} else if (procno==receiver) {
    comm_world.recv( mpl::absolute,object,receiver );
}

```

Note the `make_absolute` in addition to `absolute` mentioned above.

6.4 Big data types

The `size` parameter in MPI send and receive calls is of type integer, meaning that it's maximally (platform-dependent, but typically:) $2^{31} - 1$. These day computers are big enough that this is a limitation. As of the MPI-4 standard, this has been solved by allowing a larger count parameter of type `MPI_Count`. The implementation of this depends somewhat on the language.

The following material is for the recently released MPI-4 standard and may not be supported yet.

MPL note 58: Large counts.

6.4.1 C

For every routine, such as `MPI_Send` with an integer count, there is a corresponding `MPI_Send_c` with a count of type `MPI_Count`.

```

MPI_Count buffersize = 1000;
double *indata,*outdata;
indata = (double*) malloc( buffersize*sizeof(double) );
outdata = (double*) malloc( buffersize*sizeof(double) );
MPI_Allreduce_c(indata,outdata,buffersize,
                MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);

```

```

Code:
// pingpongbig.c
assert( sizeof(MPI_Count)>4 );
for ( int power=3; power<=10; power++)
{
    MPI_Count length=pow(10,power);
    buffer = (double*)malloc( length*
        sizeof(double) );
    MPI_Ssend_c
        (buffer,length,MPI_DOUBLE,
         processB,0,comm);
    MPI_Recv_c
        (buffer,length,MPI_DOUBLE,
         processB,0,comm,
         MPI_STATUS_IGNORE);
}

```

Output:

```

[examples/mpi/c] pingpongbig:
make[3]: `pingpongbig' is up to
      date.
Ping-pong between ranks 0--1,
      →repeated 10 times
MPI Count has 8 bytes
Size: 10^3, (repeats=10000)
Time 1.399211e-05 for size 10^3:
      →1.1435 Gb/sec
Size: 10^4, (repeats=10000)
Time 4.077882e-05 for size 10^4:
      →3.9236 Gb/sec
Size: 10^5, (repeats=1000)
Time 1.532863e-04 for size 10^5:
      →10.4380 Gb/sec
Size: 10^6, (repeats=1000)
Time 1.418844e-03 for size 10^6:
      →11.2768 Gb/sec
Size: 10^7, (repeats=100)
Time 1.443470e-02 for size 10^7:
      →11.0844 Gb/sec
Size: 10^8, (repeats=100)
Time 1.540918e-01 for size 10^8:
      →10.3834 Gb/sec
Size: 10^9, (repeats=10)
Time 1.813220e+00 for size 10^9:
      →8.8241 Gb/sec
Size: 10^10, (repeats=10)
Time 1.846741e+01 for size 10^10:
      →8.6639 Gb/sec

```

6.4.2 Fortran

The count parameter can be declared to be

```

use mpi_f08
Integer(kind=MPI_COUNT_KIND) :: count

```

Since Fortran has polymorphism, the same routine names can be used.

The legit way of coding:

```

!! typecheckkind.F90
integer(8) :: source,n=1
call MPI_Init()
call MPI_Send(source,n,MPI_INTEGER8, &
             1,0,MPI_COMM_WORLD)

```

```

!! typecheck8.F90
integer(8) :: source,n=1
call MPI_Init()
call MPI_Send(source,n,MPI_INTEGER8, &
             1,0,MPI_COMM_WORLD)

```

... but you can see what's under the hood:

6. MPI topic: Data types

Routines using this type are not available unless using the `mpi_f08` module.

End of MPI-4 material

```
!! pingpongbig.F90
integer :: power,countbytes
Integer(KIND=MPI_COUNT_KIND) :: length
call MPI_Sizeof(length,countbytes,ierr)
if (procno==0) &
    print *, "Bytes in count:",countbytes
    length = 10**power
    allocate( senddata(length),recvdata(length) )
    call MPI_Send(senddata,length,MPI_DOUBLE_PRECISION, &
                  processB,0, comm)
    call MPI_Recv(recvdata,length,MPI_DOUBLE_PRECISION, &
                  processB,0, comm,MPI_STATUS_IGNORE)
```

6.4.3 Count datatype

The `MPI_Count` datatype is defined as being large enough to accomodate values of

- the ordinary 4-byte integer type;
- the `MPI_Aint` type, sections [6.2.4](#) and [6.2.4](#);
- the `MPI_Offset` type, section [10.2.2](#).

The `size_t` type in C/C++ is defined as big enough to contain the output of `sizeof`, that is, being big enough to measure any object.

6.4.4 MPI 3 temporary solution

Large messages were already possible by using derived types: to send a *big data type* of 10^{40} elements you would

- create a contiguous type with 10^{20} elements, and
- send 10^{20} elements of that type.

This often works, but it's not perfect. For instance, the routine `MPI_Get_elements` returns the total number of basic elements sent (as opposed to `MPI_Get_count` which would return the number of elements of the derived type). Since its output argument is of integer type, it can't store the right value.

The MPI-3 standard has addressed this through the introduction of an `MPI_Count` datatype, and new routines with an `_x` extension, that return that type of count.

The following material is for the recently released MPI-4 standard and may not be supported yet.

In view of the ‘embiggened’ routines, this solution is no longer needed, and is deprecated as of MPI-4.1.
End of MPI-4 material

Let us consider an example.

Allocating a buffer of more than 4Gbyte is not hard:

```
// vectorx.c
float *source=NULL,*target=NULL;
int mediumsize = 1<<30;
int nblocks = 8;
size_t datasize = (size_t)mediumsize * nblocks * sizeof(float);
if (procno==sender) {
    source = (float*) malloc(datasize);
```

We use the trick with sending elements of a derived type:

```
MPI_Datatype blocktype;
MPI_Type_contiguous(mediumsize,MPI_FLOAT,&blocktype);
MPI_Type_commit(&blocktype);
if (procno==sender) {
    MPI_Send(source,nblocks,blocktype,receiver,0,comm);
```

We use the same trick for the receive call, but now we catch the status parameter which will later tell us how many elements of the basic type were sent:

```
} else if (procno==receiver) {
    MPI_Status recv_status;
    MPI_Recv(target,nblocks,blocktype, sender,0,comm,
    &recv_status);
```

When we query how many of the basic elements are in the buffer (remember that in the receive call the buffer length is an upper bound on the number of elements received) do we need a counter that is larger than an integer. MPI has introduced a type `MPI_Count` for this, and new routines such as `MPI_Get_elements_x` (figure 4.14) that return a count of this type:

```
MPI_Count recv_count;
MPI_Get_elements_x(&recv_status,MPI_FLOAT,&recv_count);
```

Remark Computing a big number to allocate is not entirely simple.

```
// getx.c
int gig = 1<<30;
int nblocks = 8;
size_t big1 = gig * nblocks * sizeof(double);
size_t big2 = (size_t)1 * gig * nblocks * sizeof(double);
size_t big3 = (size_t) gig * nblocks * sizeof(double);
size_t big4 = gig * nblocks * (size_t) ( sizeof(double) );
size_t big5 = sizeof(double) * gig * nblocks;
;
```

gives as output:

```
size of size_t = 8
0 68719476736 68719476736 0 68719476736
```

Clearly, not only do operations go left-to-right, but casting is done that way too: the computed subexpressions are only cast to `size_t` if one operand is.

Figure 6.16 MPI_Type_get_extent

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Type_get_extent</code>					
	<code>MPI_Type_get_extent_c</code>				
	<code>datatype</code>	datatype to get information on	<code>MPI_Datatype</code>	<code>TYPE</code> <code>(MPI_Datatype)</code>	IN
<code>lb</code>		lower bound of datatype	<code>[MPI_Aint*</code> <code> MPI_Count*</code>	<code>INTEGER</code> <code>(KIND=MPI_ADDRESS_KIND)</code>	OUT
<code>extent</code>		extent of datatype	<code>[MPI_Aint*</code> <code> MPI_Count*</code>	<code>INTEGER</code> <code>(KIND=MPI_ADDRESS_KIND)</code>	OUT
)					

Above, we did not actually create a datatype that was bigger than 2G, but if you do so, you can query its extent by `MPI_Type_get_extent_x` (figure 6.17) and `MPI_Type_get_true_extent_x` (figure 6.17).

Python note 23: Big data. Since python has unlimited size integers there is no explicit need for the ‘x’ variants of routines. Internally, `MPI.Status.Get_elements` is implemented in terms of `MPI_Get_elements_x`.

6.5 Type maps and type matching

With derived types, you saw that it was not necessary for the type of the sender and receiver to match. However, when the send buffer is constructed, and the receive buffer unpacked, it is necessary for the successive types in that buffer to match.

The types in the send and receive buffers also need to match the datatypes of the underlying architecture, with two exceptions. The `MPI_PACKED` and `MPI_BYTE` types can match any underlying type. However, this still does not mean that it is a good idea to use these types on only sender or receiver, and a specific type on the other.

6.6 Type extent

See section 6.2.5 about the related issue of type sizes.

6.6.1 Extent and true extent

The *datatype extent*, measured with `MPI_Type_get_extent` (figure 6.16), is strictly the distance from the first to the last data item of the type, that is, with counting the gaps in the type. It is measured in bytes so the output parameters are of type `MPI_Aint`.

In the following example (see also figure 6.8) we measure the extent of a vector type. Note that the extent is not the stride times the number of blocks, because that would count a ‘trailing gap’.

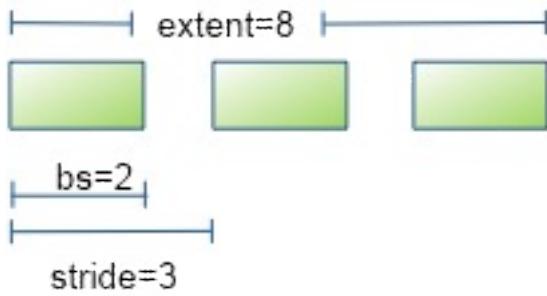


Figure 6.8: Extent of a vector datatype

```

MPI_Aint lb,asize;
MPI_Type_vector(count,bs,stride,MPI_DOUBLE,&newtype);
MPI_Type_commit(&newtype);
MPI_Type_get_extent(newtype,&lb,&asize);
ASSERT( lb==0 );
ASSERT( asize==((count-1)*stride+bs)*sizeof(double) );
MPI_Type_free(&newtype);

```

Similarly, using `MPI_Type_get_extent` counts the gaps in a `struct` induced by *alignment* issues.

```

size_t size_of_struct = sizeof(struct object);
MPI_Aint typesize,typelb;
MPI_Type_get_extent(newstructuretype,&typelb,&typesize);
assert( typesize==size_of_struct );

```

See section 6.3.7 for the code defining the structure type.

Remark Routine `MPI_Type_get_extent` replaces deprecated functions `MPI_Type_extent`, `MPI_Type_lb`, `MPI_Type_ub`.

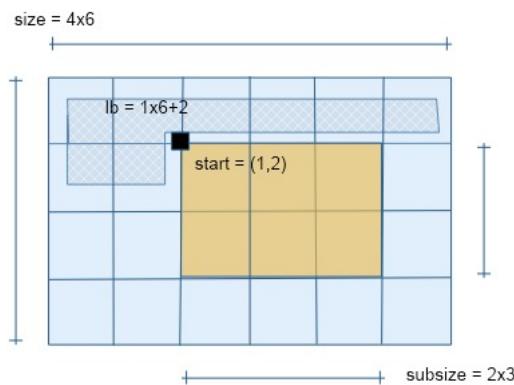


Figure 6.9: True lower bound and extent of a subarray data type

The *subarray datatype* need not start at the first element of the buffer, so the extent is an overstatement

6. MPI topic: Data types

Figure 6.17 MPI_Type_get_true_extent

Name	Param name	Explanation	C type	F type	inout
MPI_Type_get_true_extent					
MPI_Type_get_true_extent_c					
datatype	datatype	datatype to get information on	MPI_Datatype	TYPE (MPI_Datatype)	IN
true_lb		true lower bound of datatype	[MPI_Aint* MPI_Count*]	INTEGER (KIND=MPI_ADDRESS_KIND)	OUT
true_extent		true extent of datatype	[MPI_Aint* MPI_Count*]	INTEGER (KIND=MPI_ADDRESS_KIND)	OUT
)					

of how much data is involved. In fact, the lower bound is zero, and the extent equals the size of the block from which the subarray is taken. The routine `MPI_Type_get_true_extent` (figure 6.17) returns the lower bound, indicating where the data starts, and the extent from that point. This is illustrated in figure 6.9.

```

Code:
// trueextent.c
int sender = 0, receiver = 1, the_other
    = 1-procno;
int sizes[2] = {4,6},subsizes[2] =
{2,3},starts[2] = {1,2};
MPI_Datatype subarraytype;
MPI_Type_create_subarray
(2,sizes,subsizes,starts,
 MPI_ORDER_C,MPI_DOUBLE,&
 subarraytype);
MPI_Type_commit(&subarraytype);

MPI_Aint true_lb,true_extent,extent;
MPI_Type_get_true_extent
(subarraytype,&true_lb,&true_extent
 );
MPI_Aint
comp_lb = sizeof(double) *
( starts[0]*sizes[1]+starts[1]
),
comp_extent = sizeof(double) *
( sizes[1]-starts[1] // first
row
+ starts[1]+subsizes[1] //
last row
+ ( subsizes[0]>1 ? subsizes
[0]-2 : 0 )*sizes[1] );
ASSERT(true_lb==comp_lb);
ASSERT(true_extent==comp_extent);
MPI_Send(source,1,subarraytype,
the_other,0,comm);
MPI_Type_free(&subarraytype);

```

Output:
[examples/mpi/c] trueextent:
In basic array of 192 bytes
find sub array of 48 bytes
Found lb=64, extent=72
Computing lb=64 extent=72
Non-true lb=0, extent=192,
→computed=192
Finished
received: 8.500 9.500 10.500
→14.500 15.500 16.500
1,2
1,3
1,4
2,2
2,3
2,4

There are also ‘big data’ routines `MPI_Type_get_extent_x` `MPI_Type_get_true_extent_x` that has an `MPI_Count` as output.

The following material is for the recently released MPI-4 standard and may not be supported yet.

The C routines `MPI_Type_get_extent_c` `MPI_Type_get_true_extent_c` also output an `MPI_Count`.

End of MPI-4 material

6.6.2 Extent resizing

A type is partly characterized by its lower bound and extent, or equivalently lower bound and upperbound. Somewhat miraculously, you can actually change these to achieve special effects. This is needed for:

- Some cases of gather/scatter operations; see the example in section 6.6.2.2.
- When the count of derived items in a buffer is more than one. See the example in section 6.6.2.1.

6. MPI topic: Data types

Figure 6.18 MPI_Type_create_resized

Name	Param name	Explanation	C type	F type	inout
MPI_Type_create_resized					
	oldtype	input datatype	MPI_Datatype	TYPE (MPI_Datatype)	IN
	lb	new lower bound of datatype	[MPI_Aint MPI_Count]	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
	extent	new extent of datatype	[MPI_Aint MPI_Count]	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
	newtype	output datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)				

MPL:

```
template<typename T>
class layout {

void resize(ssize_t lb, ssize_t extent);
void byte_resize(ssize_t lb, ssize_t extent);
```

The technicality on which the solution hinges is that you can ‘resize’ a type with `MPI_Type_create_resized` (figure 6.18) to give it a different extent, while not affecting how much data there actually is in it.

We can describe the space taken by a data type (the ‘true extent’) and the ‘extent’ as follows. If the send count is more than 1, or if you scatter some data type:

1. A pointer is set at the first data item; then
2. For each instance of the datatype to be sent:
 - (a) data is sent as described by the data type; and
 - (b) the pointer is advanced by the extent of the data type

(Receiving and gathering behave similarly, but with data going in the opposite direction.)

6.6.2.1 Example 1

In the examples of derived types so far we always used a send count of 1. What happens if you use a larger count?

Consider a vector type, with a send count of 2.

```
MPI_Type_vector( count,bs,stride,oldtype,&one_n_type );
MPI_Type_contiguous( 2,&one_n_type,&two_n_type );
```

Contrast this with a twice-as-large vector type: It is clear that

```
MPI_Type_vector( 2*count,bs,stride,oldtype,&two_n_type );
```

The difference is pictured in figure 6.10, where the two illustrates the result of using a send count of 2, and the bottom the desired effect.

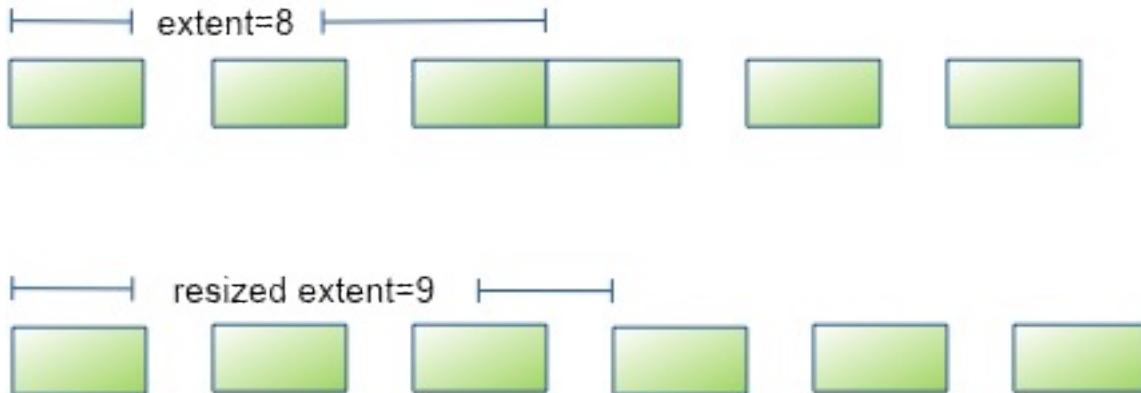


Figure 6.10: Contiguous type of two vectors, before and after resizing the extent.

To show how this problem can be solved by resizing the extent, let's look at a specific example, and consider sending more than one derived type, from a buffer containing consecutive integers:

```
// vectorpadsend.c
for (int i=0; i<max_elements; i++) sendbuffer[i] = i;
MPI_Type_vector(count,blocklength,stride,MPI_INT,&stridetype);
MPI_Type_commit(&stridetype);
MPI_Send( sendbuffer,ntypes,stridetype, receiver,0, comm );
```

We receive into a contiguous buffer:

```
MPI_Recv( recvbuffer,max_elements,MPI_INT, sender,0, comm,&status );
int count; MPI_Get_count(&status,MPI_INT,&count);
printf("Receive %d elements:",count);
for (int i=0; i<count; i++) printf(" %d",recvbuffer[i]);
printf("\n");
```

giving an output of:

Receive 6 elements: 0 2 4 5 7 9

Next, we resize the type to add the gap at the end. This is illustrated in figure 6.10.

Resizing the type looks like:

```
MPI_Type_get_extent(stridetype,&l,&e);
printf("Stride type l=%ld e=%ld\n",l,e);
e += ( stride-blocklength ) * sizeof(int);
MPI_Type_create_resized(stridetype,l,e,&paddedtype);
MPI_Type_get_extent(paddedtype,&l,&e);
printf("Padded type l=%ld e=%ld\n",l,e);
MPI_Type_commit(&paddedtype);
MPI_Send( sendbuffer,ntypes,paddedtype, receiver,0, comm );
```

and the corresponding output, including querying the extents, is:

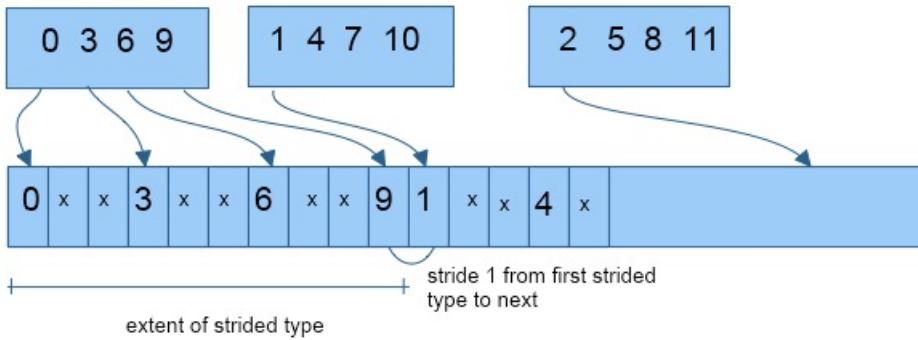


Figure 6.11: Placement of gathered strided types.

```
Strided type l=0 e=20
Padded type l=0 e=24
Receive 6 elements: 0 2 4 6 8 10
```

About the extent routines we make two observations:

1. the lower bound and extent parameters are of type `MPI_Aint`, meaning that they measure in bytes; and
2. the lower bound is typically left unchanged: we give no examples in this book where it is changed.

6.6.2.2 Example 2

For another example, let's revisit exercise 6.4 (and figure 6.5) where each process makes a buffer of integers that will be interleaved in a gather call: Strided data was sent in individual transactions. Would it be possible to address all these interleaved packets in one gather or scatter call?

The problem here is that MPI uses the extent of the send type in a scatter, or the receive type in a gather: if that type is 20 bytes big from its first to its last element, then data will be read out 20 bytes apart in a scatter, or written 20 bytes apart in a gather. This ignores the ‘gaps’ in the type! (See exercise 6.4.)

```
int *mydata = (int*) malloc( localsize*sizeof(int) );
for (int i=0; i<localsize; i++)
    mydata[i] = i*nprocs+procno;
MPI_Gather( mydata,localsize,MPI_INT,
    /* rest to be determined */ );
```

An ordinary gather call will of course not interleave, but put the data end-to-end:

```
MPI_Gather( mydata,localsize,MPI_INT,
    gathered,localsize,MPI_INT, // abutting
    root,comm );
```

```
gather 4 elements from 3 procs:
0 3 6 9 1 4 7 10 2 5 8 11
```

Using a strided type still puts data end-to-end, but now there are unwritten gaps in the gather buffer:

```
MPI_Gather( mydata,localsize,MPI_INT,
            gathered,1,stridetype, // abut with gaps
            root,comm );
```

This is illustrated in figure 6.11. A sample printout of the result would be:

```
0 1879048192 1100361260 3 3 0 6 0 0 9 1 198654
```

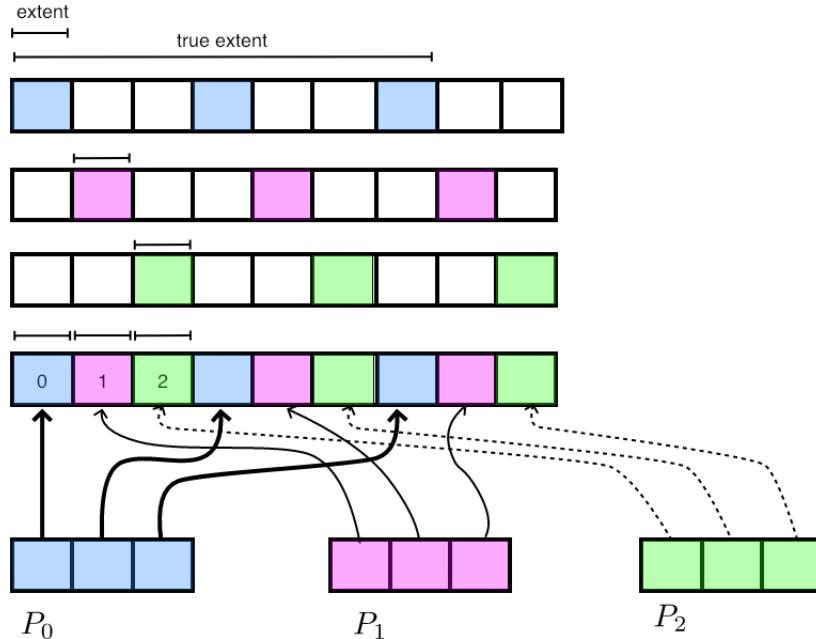


Figure 6.12: Interleaved gather from data with resized extent

The trick is to use **MPI_Type_create_resized** to make the extent of the type only one int long:

```
// interleavegather.c
MPI_Datatype interleavetype;
MPI_Type_create_resized(stridetype,0,sizeof(int),&interleavetype);
MPI_Type_commit(&interleavetype);
MPI_Gather( mydata,localsize,MPI_INT,
            gathered,1,interleavetype, // shrunk extent
            root,comm );
```

Now data is written with the same stride, but at starting points equal to the shrunk extent:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

This is illustrated in figure 6.12.

Fortran note 13: Extent as Aint. The lowerbound and extent parameters are of type

```
Integer(kind=MPI_Address_kind):
```

```
!! stridescatter.F90
```

6. MPI topic: Data types

```
integer(kind=MPI_Address_kind) :: l,e
call MPI_Type_get_extent(scattertype,l,e)
e = c_sizeof(i)
call MPI_Type_create_resized(scattertype,l,e,interleavetype)
call MPI_Type_commit(interleavetype)
```

Exercise 6.7. Rewrite exercise 6.4 to use a gather, rather than individual messages.

MPL note 59: Extent resizing. Resizing a datatype does not give a new type, but does the resize ‘in place’:

```
void layout::resize(ssize_t lb, ssize_t extent);
```

6.6.2.3 Example: dynamic vectors

Does it bother you (a little) that in the vector type you have to specify explicitly how many blocks there are? It would be nice if you could create a ‘block with padding’ and then send however many of those.

Well, you can introduce that padding by resizing a type, making it a little larger.

```
// stridestretch.c
MPI_Datatype oneblock;
MPI_Type_vector(1,1,stride,MPI_DOUBLE,&oneblock);
MPI_Type_commit(&oneblock);
MPI_Aint block_lb,block_x;
MPI_Type_get_extent(oneblock,&block_lb,&block_x);
printf("One block has extent: %ld\n",block_x);

MPI_Datatype paddedblock;
MPI_Type_create_resized(oneblock,0,stride*sizeof(double),&paddedblock);
MPI_Type_commit(&paddedblock);
MPI_Type_get_extent(paddedblock,&block_lb,&block_x);
printf("Padded block has extent: %ld\n",block_x);

// now send a bunch of these padded blocks
MPI_Send(source,count,paddedblock,the_other,0,comm);
```

There is a second solution to this problem, using a structure type. This does not use resizing, but rather indicates a displacement that reaches to the end of the structure. We do this by putting a type `MPI_UB` at this displacement:

```
int blens[2]; MPI_Aint displs[2];
MPI_Datatype types[2], paddedblock;
blens[0] = 1; blens[1] = 1;
displs[0] = 0; displs[1] = 2 * sizeof(double);
types[0] = MPI_DOUBLE; types[1] = MPI_UB;
MPI_Type_struct(2, blens, displs, types, &paddedblock);
MPI_Type_commit(&paddedblock);
MPI_Status recv_status;
MPI_Recv(target,count,paddedblock,the_other,0,comm,&recv_status);
```

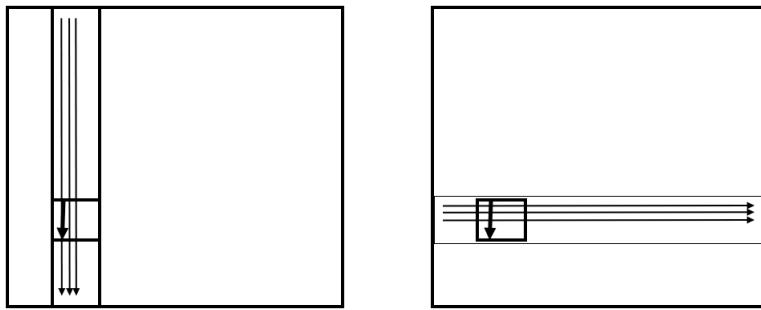


Figure 6.13: Transposing a 1D partitioned array

6.6.2.4 Example: transpose

Transposing data is an important part of such operations as the FFT. We develop this in steps. Refer to figure 6.13.

The source data can be described as a vector type defined as:

- there are b blocks,
- of blocksize b ,
- spaced apart by the global i -size of the array.

```
## transposeblock.py
MPI_Datatype sourceblock;
MPI_Type_vector( blocksize_j, blocksize_i, isize, MPI_INT, &sourceblock );
MPI_Type_commit( &sourceblock );
```

The target type is harder to describe. First we note that each contiguous block from the source type can be described as a vector type with:

- b blocks,
- of size 1 each,
- stided by the global j -size of the matrix.

```
MPI_Datatype targetcolumn;
MPI_Type_vector( blocksize_i, 1, jsize, MPI_INT, &targetcolumn );
MPI_Type_commit( &targetcolumn );
```

For the full type at the receiving process we now need to pack b of these lines together.

Exercise 6.8. Finish the code.

- What is the extent of the `targetcolumn` type?
- What is the spacing of the first elements of the blocks? How do you therefore resize the `targetcolumn` type?

6.7 Reconstructing types

It is possible to find from a datatype how it was constructed. This uses the routines `MPI_Type_get_envelope` and `MPI_Type_get_contents`. The first routine returns the *combiner* (with values such as

`MPI_COMBINER_VECTOR`) and the number of parameters; the second routine is then used to retrieve the actual parameters.

6.8 Packing

One of the reasons for derived datatypes is dealing with noncontiguous data. In older communication libraries this could only be done by *packing* data from its original containers into a buffer, and likewise unpacking it at the receiver into its destination data structures.

MPI offers this packing facility, partly for compatibility with such libraries, but also for reasons of flexibility. Unlike with derived datatypes, which transfers data atomically, packing routines add data sequentially to the buffer and unpacking takes them sequentially.

This means that one could pack an integer describing how many floating point numbers are in the rest of the packed message. Correspondingly, the unpack routine could then investigate the first integer and based on it unpack the right number of floating point numbers.

MPI offers the following:

- The `MPI_Pack` command adds data to a send buffer;
- the `MPI_Unpack` command retrieves data from a receive buffer;
- the buffer is sent with a datatype of `MPI_PACKED`.

With `MPI_Pack` (figure 6.19) data elements can be added to a buffer one at a time. The `position` parameter is updated each time by the packing routine.

Conversely, `MPI_Unpack` (figure 6.20) retrieves one element from the buffer at a time. You need to specify the MPI datatype.

A packed buffer is sent or received with a datatype of `MPI_PACKED`. The sending routine uses the `position` parameter to specify how much data is sent, but the receiving routine does not know this value a priori, so has to specify an upper bound.

Figure 6.19 MPI_Pack

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Pack (</code>					
<code> MPI_Pack_c (</code>					
<code> inbuf</code>	<code> input buffer start</code>		<code>const void*</code>	<code>TYPE(*), DIMENSION(..)</code>	<code>IN</code>
<code> incount</code>	<code> number of input data items</code>		<code> [int MPI_Count</code>	<code> INTEGER</code>	<code>IN</code>
<code> datatype</code>	<code> datatype of each input data item</code>		<code> MPI_Datatype</code>	<code> TYPE (MPI_Datatype)</code>	<code>IN</code>
<code> outbuf</code>	<code> output buffer start</code>		<code> void*</code>	<code> TYPE(*), DIMENSION(..)</code>	<code>OUT</code>
<code> outsize</code>	<code> output buffer size, in bytes</code>		<code> [int MPI_Count</code>	<code> INTEGER</code>	<code>IN</code>
<code> position</code>	<code> current position in buffer, in bytes</code>		<code> [int* MPI_Count*</code>	<code> INTEGER</code>	<code>INOUT</code>
<code> comm</code>	<code> communicator for packed message</code>		<code> MPI_Comm</code>	<code> TYPE (MPI_Comm)</code>	<code>IN</code>
<code>)</code>					

MPL:

Not available in MPL 0.3

Figure 6.20 MPI_Unpack

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Unpack (</code>					
<code> MPI_Unpack_c (</code>					
<code> inbuf</code>	<code> input buffer start</code>		<code>const void*</code>	<code>TYPE(*), DIMENSION(..)</code>	<code>IN</code>
<code> insize</code>	<code> size of input buffer, in bytes</code>		<code> [int MPI_Count</code>	<code> INTEGER</code>	<code>IN</code>
<code> position</code>	<code> current position in bytes</code>		<code> [int* MPI_Count*</code>	<code> INTEGER</code>	<code>INOUT</code>
<code> outbuf</code>	<code> output buffer start</code>		<code> void*</code>	<code> TYPE(*), DIMENSION(..)</code>	<code>OUT</code>
<code> outcount</code>	<code> number of items to be unpacked</code>		<code> [int MPI_Count</code>	<code> INTEGER</code>	<code>IN</code>
<code> datatype</code>	<code> datatype of each output data item</code>		<code> MPI_Datatype</code>	<code> TYPE (MPI_Datatype)</code>	<code>IN</code>
<code> comm</code>	<code> communicator for packed message</code>		<code> MPI_Comm</code>	<code> TYPE (MPI_Comm)</code>	<code>IN</code>
<code>)</code>					

MPL:

Not available in MPL 0.3

6. MPI topic: Data types

Code:

```
if (procno==sender) {
    position = 0;
    MPI_Pack(&nsends,1,MPI_INT,
              buffer,buflen,&position,comm
            );
    for (int i=0; i<nsends; i++) {
        double value = rand()/(double)
                     RAND_MAX;
        printf("[%d] pack %e\n",procno,
               value);
        MPI_Pack(&value,1,MPI_DOUBLE,
                  buffer,buflen,&position,
                  comm);
    }
    MPI_Pack(&nsends,1,MPI_INT,
              buffer,buflen,&position,comm
            );
    MPI_Send(buffer,position,MPI_PACKED,
              other,0,comm);
} else if (procno==receiver) {
    int irecv_value;
    double xrecv_value;
    MPI_Recv(buffer,buflen,MPI_PACKED,
              other,0,
              comm,MPI_STATUS_IGNORE);
    position = 0;
    MPI_Unpack(buffer,buflen,&position,
                &nsends,1,MPI_INT,comm);
    for (int i=0; i<nsends; i++) {
        MPI_Unpack(buffer,buflen,
                   &position,&xrecv_value
                   ,1,MPI_DOUBLE,comm);
        printf("[%d] unpack %e\n",procno,
               xrecv_value);
    }
    MPI_Unpack(buffer,buflen,&position,
                &irecv_value,1,MPI_INT,
                comm);
    ASSERT(irecv_value==nsends);
}
```

Output:

```
[examples/mpi/c] pack:
[0] pack 8.401877e-01
[0] pack 3.943829e-01
[0] pack 7.830992e-01
[0] pack 7.984400e-01
[0] pack 9.116474e-01
[0] pack 1.975514e-01
```

You can precompute the size of the required buffer with `MPI_Pack_size` (figure 6.21).

Figure 6.21 MPI_Pack_size

Name	Param name	Explanation	C type	F type	inout
MPI_Pack_size (
MPI_Pack_size_c (
incount	count argument to packing call		[int MPI_Count	INTEGER	IN
datatype	datatype argument to packing call		MPI_Datatype	TYPE (MPI_Datatype)	IN
comm	communicator argument to packing call		MPI_Comm	TYPE (MPI_Comm)	IN
size	upper bound on size of packed message, in bytes		[int* MPI_Count*	INTEGER	OUT
)					

Code:

```
// pack.c
for (int i=1; i<=4; i++) {
    MPI_Pack_size(i,MPI_CHAR,comm,&s);
    printf("%d chars: %d\n",i,s);
}
for (int i=1; i<=4; i++) {
    MPI_Pack_size(i,MPI_UNSIGNED_SHORT,
                  comm,&s);
    printf("%d unsigned shorts: %d\n",i,s
          );
}
for (int i=1; i<=4; i++) {
    MPI_Pack_size(i,MPI_INT,comm,&s);
    printf("%d ints: %d\n",i,s);
}
```

Output:

```
[examples/mpi/c] packsize:
1 chars: 1
2 chars: 2
3 chars: 3
4 chars: 4
1 unsigned shorts: 2
2 unsigned shorts: 4
3 unsigned shorts: 6
4 unsigned shorts: 8
1 ints: 4
2 ints: 8
3 ints: 12
4 ints: 16
```

Exercise 6.9. Suppose you have a ‘structure of arrays’

```
struct aos {
    int length;
    double *reals;
    double *imags;
};
```

with dynamically created arrays. Write code to send and receive this structure.

6.9 Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

1. Give two examples of MPI derived datatypes. What parameters are used to describe them?
2. Give a practical example where the sender uses a different type to send than the receiver uses in the corresponding receive call. Name the types involved.
3. Fortran only. True or false?
 - (a) Array indices can be different between the send and receive buffer arrays.
 - (b) It is allowed to send an array section.
 - (c) You need to *Reshape* a multi-dimensional array to linear shape before you can send it.
 - (d) An allocatable array, when dimensioned and allocated, is treated by MPI as if it were a normal static array, when used as send buffer.
 - (e) An allocatable array is allocated if you use it as the receive buffer: it is filled with the incoming data.
4. Fortran only: how do you handle the case where you want to use an allocatable array as receive buffer, but it has not been allocated yet, and you do not know the size of the incoming data?

Chapter 7

MPI topic: Communicators

A communicator is an object describing a group of processes. In many applications all processes work together closely coupled, and the only communicator you need is `MPI_COMM_WORLD`, the group describing all processes that your job starts with.

In this chapter you will see ways to make new groups of MPI processes: subgroups of the original world communicator. Chapter 8 discusses dynamic process management, which, while not extending `MPI_COMM_WORLD` does extend the set of available processes. That chapter also discusses the ‘sessions model’, which is another way to constructing communicators.

7.1 Basic communicators

There are three predefined communicators:

- `MPI_COMM_WORLD` comprises all processes that were started together by `mpiexec` (or some related program).
- `MPI_COMM_SELF` is the communicator that contains only the current process.
- `MPI_COMM_NULL` is the invalid communicator. This values results
 - when a communicator is freed; see section 7.3;
 - as error return value from routines that construct communicators;
 - for processes outside a created Cartesian communicator (section 11.1.1);
 - on non-spawned processes when querying their parent (section 7.6.3).

These values are constants, though not necessarily compile-time constants. Thus, they can not be used in switch statements, array declarations, or `constexpr` evaluations.

If you don’t want to write `MPI_COMM_WORLD` repeatedly, you can assign that value to a variable of type `MPI_Comm`.

Examples:

```
// C:  
#include <mpi.h>  
MPI_Comm comm = MPI_COMM_WORLD;
```

7. MPI topic: Communicators

Figure 7.1 MPI_Comm_dup

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_dup (
comm	communicator		MPI_Comm	TYPE (MPI_Comm)	IN
newcomm	copy of comm		MPI_Comm*	TYPE (MPI_Comm)	OUT
)					

MPL:

Done as part of the copy assignment operator.

```
!! Fortran 2008 interface
use mpi_f08
Type(MPI_Comm) :: comm = MPI_COMM_WORLD

!! Fortran legacy interface
#include <mpif.h>
Integer :: comm = MPI_COMM_WORLD
```

Python note 24: Communicator types.

```
comm = MPI.COMM_WORLD
```

MPL note 60: Predefined communicators. The environment namespace has the equivalents of MPI_COMM_WORLD and MPI_COMM_SELF:

```
const communicator& mpl::environment::comm_world();
const communicator& mpl::environment::comm_self();
```

Uses of MPI_COMM_NULL are handled differently.

MPL note 61: Raw communicator handles. Should you need the MPI_Comm object contained in an MPL communicator, there is an access function native_handle.

You can name your communicators with MPI_Comm_set_name, which could improve the quality of error messages when they arise.

7.2 Duplicating communicators

With MPI_Comm_dup (figure 7.1) you can make an exact duplicate of a communicator (see section 7.2.2 for an application). There is a nonblocking variant MPI_Comm_idup (figure 7.2).

These calls do not propagate info hints (sections 15.1.1 and 15.1.1.2); to achieve this, use MPI_Comm_dup_with_info and MPI_Comm_idup_with_info; section 15.1.1.2.

Python note 25: Communicator duplication. Duplicate communicators are created as output of the duplication routine:

Figure 7.2 MPI_Comm_idup

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_idup (
comm	communicator		MPI_Comm	TYPE (MPI_Comm)	IN
newcomm	copy of comm		MPI_Comm*	TYPE (MPI_Comm)	OUT
request	communication request		MPI_Request*	TYPE (MPI_Request)	OUT
)					

```
newcomm = comm.Dup()
```

MPL note 62: Communicator duplication. Communicators can be duplicated but only during initialization. Copy assignment has been deleted. Thus:

```
// LEGAL:  
mpl::communicator init = comm;  
// WRONG:  
mpl::communicator init;  
init = comm;
```

7.2.1 Communicator comparing

You may wonder what ‘an exact copy’ means precisely. For this, think of a communicator as a context label that you can attach to, among others, operations such as sends and receives. And it’s that label that counts, not what processes are in the communicator. A send and a receive ‘belong together’ if they have the same communicator context. Conversely, a send in one communicator can not be matched to a receive in a duplicate communicator, made by [MPI_Comm_dup](#).

Testing whether two communicators are really the same is then more than testing if they comprise the same processes. The call [MPI_Comm_compare](#) returns [MPI_IDENT](#) if two communicator values are the same, and not if one is derived from the other by duplication:

7. MPI topic: Communicators

```
Code:
// commcompare.c
int result;
MPI_Comm copy = comm;
MPI_Comm_compare(comm,copy,&result);
printf("assign:    comm==copy: %d \n",
      result==MPI_IDENT);
printf("            congruent: %d \n",
      result==MPI_CONGRUENT);
printf("            not equal: %d \n",
      result==MPI_UNEQUAL);

MPI_Comm_dup(comm,&copy);
MPI_Comm_compare(comm,copy,&result);
printf("duplicate: comm==copy: %d \n",
      result==MPI_IDENT);
printf("            congruent: %d \n",
      result==MPI_CONGRUENT);
printf("            not equal: %d \n",
      result==MPI_UNEQUAL);
```

```
Output:
[examples/mpi/c] commcompare:

assign:    comm==copy: 1
            congruent: 0
            not equal: 0
duplicate: comm==copy: 0
            congruent: 1
            not equal: 0
```

Communicators that are not actually the same can be

- consisting of the same processes, in the same order, giving `MPI_CONGRUENT`;
- merely consisting of the same processes, but not in the same order, giving `MPI_SIMILAR`;
- different, giving `MPI_UNEQUAL`.

Comparing against `MPI_COMM_NULL` is not allowed.

MPL note 63: Communicator comparing.

```
Code:
const mpl::communicator &comm =
    mpl::environment::comm_world();
MPI_Comm
world_extract = comm.native_handle(),
world_given = MPI_COMM_WORLD;
int result;
MPI_Comm_compare(world_extract,
                 world_given,&result);
cout << "Compare raw comms: " << "\n"
<< "identical: "
<< (result==MPI_IDENT) << "\n"
<< "congruent: "
<< (result==MPI_CONGRUENT) << "\n"
<< "unequal : "
<< (result==MPI_UNEQUAL) << "\n";
```

```
Output:
[examples/mpi/mpl] rawcompare:

Compare raw comms:
identical: true
congruent: false
unequal : false
```

7.2.2 Communicator duplication for library use

Duplicating a communicator may seem pointless, but it is actually very useful for the design of software libraries. Imagine that you have a code

```
MPI_Isend(...); MPI_Irecv(...);
// library call
MPI_Waitall(...);
```

and suppose that the library has receive calls. Now it is possible that the receive in the library inadvertently catches the message that was sent in the outer environment.

Let us consider an example. First of all, here is code where the library stores the communicator of the calling program:

```
// commdupwrong.cxx
class library {
private:
    MPI_Comm comm;
    int procno, nprocs, other;
    MPI_Request request[2];
public:
    library(MPI_Comm incomm) {
        comm = incomm;
        MPI_Comm_rank(comm, &procno);
        other = 1 - procno;
    };
    int communication_start();
    int communication_end();
};
```

This models a main program that does a simple message exchange, and it makes two calls to library routines. Unbeknown to the user, the library also issues send and receive calls, and they turn out to interfere.

Here

- The main program does a send,
- the library call `function_start` does a send and a receive; because the receive can match either send, it is paired with the first one;
- the main program does a receive, which will be paired with the send of the library call;
- both the main program and the library do a wait call, and in both cases all requests are successfully fulfilled, just not the way you intended.

To prevent this confusion, the library should duplicate the outer communicator with `MPI_Comm_dup` and send all messages with respect to its duplicate. Now messages from the user code can never reach the library software, since they are on different communicators.

```
// commdupright.cxx
class library {
private:
    MPI_Comm comm;
```

7. MPI topic: Communicators

```
int procno,nprocs,other;
MPI_Request request[2];
public:
library(MPI_Comm incomm) {
    MPI_Comm_dup(incomm,&comm);
    MPI_Comm_rank(comm,&procno);
    other = 1-procno;
};
~library() {
    MPI_Comm_free(&comm);
}
int communication_start();
int communication_end();
};
```

Note how the preceding example performs the `MPI_Comm_free` call in a C++ *destructor*.

```
## commdup.py
class Library():
    def __init__(self,comm):
        # wrong: self.comm = comm
        self.comm = comm.Dup()
        self.other = self.comm.Get_size()-self.comm.Get_rank()-1
        self.requests = [ None ] * 2
    def __del__(self):
        if self.comm.Get_rank()==0: print(.. freeing communicator")
        self.comm.Free()
    def communication_start(self):
        sendbuf = np.empty(1,dtype=int); sendbuf[0] = 37
        recvbuf = np.empty(1,dtype=int)
        self.requests[0] = self.comm.Isend( sendbuf, dest=other,tag=2 )
        self.requests[1] = self.comm.Irecv( recvbuf, source=other )
    def communication_end(self):
        MPI.Request.Waitall(self.requests)

mylibrary = Library(comm)
my_requests[0] = comm.Isend( sendbuffer,dest=other,tag=1 )
mylibrary.communication_start()
my_requests[1] = comm.Irecv( recvbuffer,source=other )
MPI.Request.Waitall(my_requests,my_status)
mylibrary.communication_end()
```

7.3 Sub-communicators

In many scenarios you divide a large job over all the available processors. However, your job may have two or more parts that can be considered as jobs by themselves. In that case it makes sense to divide your processors into subgroups accordingly.

Suppose that you are running a simulation where inputs are generated, a computation is performed on them, and the results of this computation are analyzed or rendered graphically. You could then consider

dividing your processors in three groups corresponding to generation, computation, rendering. As long as you only do sends and receives, this division works fine. However, if one group of processes needs to perform a collective operation, you don't want the other groups involved in this. Thus, you really want the three groups to be distinct from each other: you want them to be in separate communicators.

In order to make such subsets of processes, MPI has the mechanism of taking a subset of `MPI_COMM_WORLD` (or other communicator) and turning that subset into a new communicator.

Now you understand why the MPI collective calls had an argument for the communicator. A collective involves all processes of that communicator. If only the world communicator existed, no such argument would be needed, but by making a communicator that contains a subset of all available processes, you can do a collective on that subset.

The usage is as follows:

- You create a new communicator with routines such as `MPI_Comm_dup` (section 7.2), `MPI_Comm_split` (section 7.4), `MPI_Comm_create` (section 7.5), `MPI_Intercomm_create` (section 7.6), `MPI_Comm_spawn` (section 8.1);
- you use that communicator for a while;
- and you call `MPI_Comm_free` when you are done with it; this also sets the communicator variable to `MPI_COMM_NULL`. A similar routine, `MPI_Comm_disconnect` waits for all pending communication to finish. Both are collective.

7.3.1 Scenario: distributed linear algebra

For *scalability* reasons (see HPC book, section ??), matrices should often be distributed in a 2D manner, that is, each process receives a subblock that is not a block of full columns or rows. This means that the processors themselves are, at least logically, organized in a 2D grid. Operations then involve reductions or broadcasts inside rows or columns. For this, a row or column of processors needs to be in a subcommunicator.

7.3.2 Scenario: climate model

A climate simulation code has several components, for instance corresponding to land, air, ocean, and ice. You can imagine that each needs a different set of equations and algorithms to simulate. You can then divide your processes, where each subset simulates one component of the climate, occasionally communicating with the other components.

7.3.3 Scenario: quicksort

The popular quicksort algorithm works by splitting the data into two subsets that each can be sorted individually. If you want to sort in parallel, you could implement this by making two subcommunicators, and sorting the data on these, creating recursively more subcommunicators.

7.3.4 Shared memory

There is an important application of communicator splitting in the context of one-sided communication, grouping processes by whether they access the same shared memory area; see section 7.4.1.

Figure 7.3 MPI_Comm_split

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Comm_split (</code>					
<code>comm</code>	<code>communicator</code>		<code>MPI_Comm</code>	<code>TYPE (MPI_Comm)</code>	<code>IN</code>
<code>color</code>	<code>control of subset assignment</code>		<code>int</code>	<code>INTEGER</code>	<code>IN</code>
<code>key</code>	<code>control of rank assignment</code>		<code>int</code>	<code>INTEGER</code>	<code>IN</code>
<code>newcomm</code>	<code>new communicator</code>		<code>MPI_Comm*</code>	<code>TYPE (MPI_Comm)</code>	<code>OUT</code>
<code>)</code>					

MPL:

```
template<typename color_type, typename key_type = int>
mpl::communicator(
    mpl::communicator::split_tag split,
    const mpl::communicator &other,
    color_type color,
    key_type key = 0);
```

7.3.5 Process spawning

Finally, newly created communicators do not always need to be subset of the initial `MPI_COMM_WORLD`. MPI can dynamically spawn new processes (see chapter 8) which start in a `MPI_COMM_WORLD` of their own. Additionally, another communicator will be created that spawns the old and new worlds so that you can communicate with the new processes.

7.4 Splitting a communicator

Above we saw several scenarios where it makes sense to divide `MPI_COMM_WORLD` into disjoint subcommunicators. The command `MPI_Comm_split` (figure 7.3) uses a ‘color’ to define these subcommunicators: all processes in the old communicator with the same color wind up in a new communicator together. The old communicator still exists, so processes now have two different contexts in which to communicate.

The ranking of processes in the new communicator is determined by a ‘key’ value: in a subcommunicator the process with lowest key is given the lowest rank, et cetera. Most of the time, there is no reason to use a relative ranking that is different from the global ranking, so the `MPI_Comm_rank` value of the global communicator is a good choice. Any ties between identical key values are broken by using the rank from the original communicator. Thus, specifying zero as the key will also retain the original process ordering.

Here is one example of communicator splitting. Suppose your processors are in a two-dimensional grid:

```
MPI_Comm_rank( MPI_COMM_WORLD, &mytid );
proc_i = mytid % proc_column_length;
proc_j = mytid / proc_column_length;
```

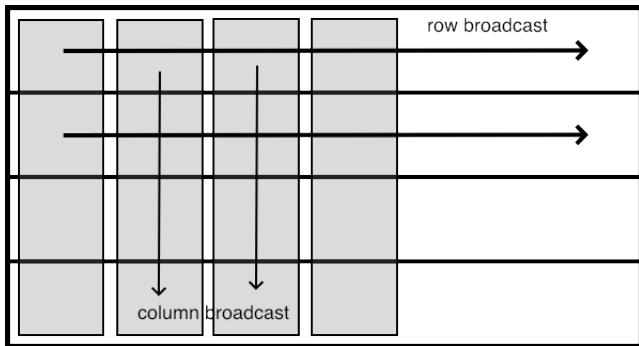


Figure 7.1: Row and column broadcasts in subcommunicators

You can now create a communicator per column:

```
MPI_Comm column_comm;
MPI_Comm_split( MPI_COMM_WORLD, proc_j, mytid, &column_comm );
```

and do a broadcast in that column:

```
MPI_Bcast( data, /* stuff */, column_comm );
```

Because of the SPMD nature of the program, you are now doing in parallel a broadcast in every processor column. Such operations often appear in *dense linear algebra*.

Exercise 7.1. Organize your processes in a grid, and make subcommunicators for the rows and columns. For this compute the row and column number of each process.

In the row and column communicator, compute the rank. For instance, on a 2×3 processor grid you should find:

Global ranks:	Ranks in row:	Ranks in column:
0 1 2	0 1 2	0 0 0
3 4 5	0 1 2	1 1 1

Check that the rank in the row communicator is the column number, and the other way around.

Run your code on different number of processes, for instance a number of rows and columns that is a power of 2, or that is a prime number.

(There is a skeleton for this exercise under the name `procgrid`.)

Remark A process that sets the `color` parameter to `MPI_UNDEFINED`, receives a communicator value of `MPI_COMM_NULL`, that is, it will not be part of any created subcommunicator.

Python note 26: Comm split key is optional. In Python, the ‘key’ argument is optional:

7. MPI topic: Communicators

```
Code:
## commssplit.py
mydata = procid

# communicator modulo 2
color = procid%2
mod2comm = comm.Split(color)
procid2 = mod2comm.Get_rank()

# communicator modulo 4 recursively
color = procid2 % 2
mod4comm = mod2comm.Split(color)
procid4 = mod4comm.Get_rank()
```

```
Output:
[examples/mpi/p] dup:
Proc 0 -> 0 -> 0
Proc 2 -> 1 -> 0
Proc 6 -> 3 -> 1
Proc 4 -> 2 -> 1
Proc 3 -> 1 -> 0
Proc 7 -> 3 -> 1
Proc 1 -> 0 -> 0
Proc 5 -> 2 -> 1
```

MPL note 64: Communicator splitting. In MPL, splitting a communicator is done as one of the overloads of the communicator constructor;

```
// commssplit.cxx
// create sub communicator modulo 2
int color2 = procno % 2;
mpl::communicator comm2
( mpl::communicator::split, comm_world, color2 );
auto procno2 = comm2.rank();

// create sub communicator modulo 4 recursively
int color4 = procno2 % 2;
mpl::communicator
comm4( mpl::communicator::split, comm2, color4 );
auto procno4 = comm4.rank();
```

Implementation note: The `communicator::split` identifier is an object of class `communicator::split_tag`, itself is an otherwise empty subclass of `communicator`:

```
class split_tag {};
static constexpr split_tag split{};
```

As another example of communicator splitting, consider the recursive algorithm for *matrix transposition*. Processors are organized in a square grid. The matrix is divided on 2×2 block form.

Exercise 7.2. Implement a recursive algorithm for matrix transposition:

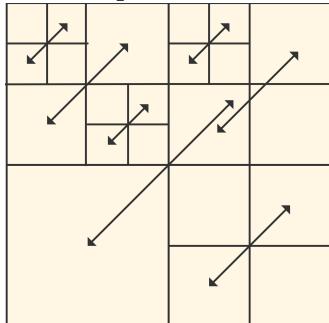


Figure 7.4 MPI_Comm_split_type

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_split_type		(
comm	communicator		MPI_Comm	TYPE (MPI_Comm)	IN
split_type	type of processes to be grouped together		int	INTEGER	IN
key	control of rank assignment		int	INTEGER	IN
info	info argument		MPI_Info	TYPE (MPI_Info)	INOUT
newcomm	new communicator		MPI_Comm*	TYPE (MPI_Comm)	OUT
)					

Python:

```
MPI.Comm.Split_type(  
    self, int split_type, int key=0, Info info=INFO_NULL)
```

- Swap blocks (1, 2) and (2, 1); then
- Divide the processors into four subcommunicators, and apply this algorithm recursively on each;
- If the communicator has only one process, transpose the matrix in place.
(assume one element per process)

7.4.1 Splitting by type

There is also a routine `MPI_Comm_split_type` (figure 7.4) which uses a type rather than a key to split the communicator.

Here the `split_type` parameter has to be from the following (short) list:

- `MPI_COMM_TYPE_SHARED`: split the communicator into subcommunicators of processes sharing a memory area. We will see this in action in section 12.1.

The following material is for the recently released MPI-4 standard and may not be supported yet.

- `MPI_COMM_TYPE_HW_GUIDED` (MPI-4): split using an `info` value from `MPI_Get_hw_resource_types`. The function `MPI_Get_hw_resource_info` (as of MPI-4.1) returns an `MPI_Info` object containing key/value pairs of available hardware resources. (See section 15.1.1 for how to unpack info objects.)
- `MPI_COMM_TYPE_HW_UNGUIDED` (MPI-4): similar to `MPI_COMM_TYPE_HW_GUIDED`, but the resulting communicators should be a strict subset of the original communicator. On processes where this condition can not be fulfilled, `MPI_COMM_NULL` will be returned.
- `MPI_COMM_TYPE_RESOURCE_GUIDED` (MPI-4.1): this splits a communicator by
 - Hardware properties. For this case there is the info key `mpi_hw_resource_type`. One possible key value `mpi_shared_memory` effects the same split as using the split type `MPI_COMM_TYPE_SHARED`.

7. MPI topic: Communicators

Figure 7.5 MPI_Comm_group

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_group (
comm	communicator		MPI_Comm	TYPE (MPI_Comm)	IN
group	group corresponding to comm		MPI_Group*	TYPE (MPI_Group)	OUT
)					

Figure 7.6 MPI_Comm_create

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_create (
comm	communicator		MPI_Comm	TYPE (MPI_Comm)	IN
group	group, which is a subset of the group of comm		MPI_Group	TYPE (MPI_Group)	IN
newcomm	new communicator		MPI_Comm*	TYPE (MPI_Comm)	OUT
)					

- pset names. For this case there is the info key `mpi_pset_name`. If a communicator is not derived from a session, the split communicator will be `MPI_COMM_NULL`.

End of MPI-4 material

Remark The OpenMPI implementation of MPI has a number of non-standard split types, such as `OMPI_COMM_TYPE_SOCKET`; see https://www.open-mpi.org/doc/v4.1/man3/MPI_Comm_split_type.3.php

7.5 Communicators and groups

You saw in section 7.4 that it is possible derive communicators that have a subset of the processes of another communicator. There is a more general mechanism, using `MPI_Group` objects.

Using groups, it takes the following steps to create a new communicator:

1. Access the `MPI_Group` of a communicator object using `MPI_Comm_group` (figure 7.5).
2. Use various routines, discussed next, to form a new group.
Note: you would form that group even on the processes that will not be come part of the new communicator.
3. Make a new communicator object from the group with using `MPI_Comm_create` (figure 7.6), collective on the old communicator.
4. On the ranks that were not in the subgroup, the resulting communicator value will be `MPI_COMM_NULL`.

Figure 7.7 MPI_Group_incl

Name	Param name	Explanation	C type	F type	inout
MPI_Group_incl					
	group	group	MPI_Group	TYPE (MPI_Group)	IN
	n	number of elements in array ranks (and size of newgroup)	int	INTEGER	IN
	ranks	ranks of processes in group to appear in newgroup	const int[]	INTEGER(n)	IN
	newgroup	new group derived from above, in the order defined by ranks	MPI_Group*	TYPE (MPI_Group)	OUT
)				

Figure 7.8 MPI_Group_excl

Name	Param name	Explanation	C type	F type	inout
MPI_Group_excl					
	group	group	MPI_Group	TYPE (MPI_Group)	IN
	n	number of elements in array ranks	int	INTEGER	IN
	ranks	array of integer ranks of processes in group not to appear in newgroup	const int[]	INTEGER(n)	IN
	newgroup	new group derived from above, preserving the order defined by group	MPI_Group*	TYPE (MPI_Group)	OUT
)				

There is also a routine `MPI_Comm_create_group` that only needs to be called on the group that constitutes the new communicator.

7.5.1 Process groups

Groups are manipulated with `MPI_Group_incl` (figure 7.7), `MPI_Group_excl` (figure 7.8), `MPI_Group_difference` and a few more.

```
MPI_Comm_group (comm, group)
MPI_Comm_create (MPI_Comm comm, MPI_Group group, MPI_Comm newcomm)
```

```
MPI_Group_union(group1, group2, newgroup)
MPI_Group_intersection(group1, group2, newgroup)
```

```
MPI_Group_difference(group1, group2, newgroup)
MPI_Group_size(group, size)
MPI_Group_rank(group, rank)
```

Certain MPI types, `MPI_Win` and `MPI_File`, are created on a communicator. While you can not directly extract that communicator from the object, you can get the group with `MPI_Win_get_group` and `MPI_File_get_group`.

There is a pre-defined empty group `MPI_GROUP_EMPTY`, which can be used as an input to group construction routines, or appear as the result of such operations as a zero intersection. This is not the same as `MPI_GROUP_NULL`, which is the output of invalid operations on groups, or the result of `MPI_Group_free`.

MPL note 65: Raw group handles. Should you need the `MPI_Datatype` object contained in an MPI group, there is an access function `native_handle`.

7.5.2 Examples

Suppose you want to split the world communicator into one manager process, with the remaining processes workers.

```
// portapp.c
MPI_Comm comm_work;
{
    MPI_Group world_group,work_group;
    MPI_Comm_group( comm_world,&world_group );
    int manager[] = {0};
    MPI_Group_excl( world_group,1,manager,&work_group );
    MPI_Comm_create( comm_world,work_group,&comm_work );
    MPI_Group_free( &world_group ); MPI_Group_free( &work_group );
}
```

Exercise 7.3. Write a code that does a scaling study: your code needs to contain a loop over increasingly sized subsets of `MPI_COMM_WORLD`.

```
for (int subsize=1; subsize<=worldsize; subsize++) {
    MPI_Comm subcomm;
    // form `subcomm' to be of size `subsize'
    MPI_Allreduce( /* stuff */ subcomm );
}
```

Carefully address which process do the various communicator and group calls; in particular do `MPI_Comm_free` and `MPI_Group_free` on the right processes.

7.6 Intercommunicators

In several scenarios it may be desirable to have a way to communicate between communicators. For instance, an application can have clearly functionally separated modules (preprocessor, simulation, post-processor) that need to stream data pairwise. In another example, dynamically spawned processes (section 8.1) get their own value of `MPI_COMM_WORLD`, but still need to communicate with the process(es) that

spawned them. In this section we will discuss the *inter-communicator* mechanism that serves such use cases.

Communicating between disjoint communicators can of course be done by having a communicator that overlaps them, but this would be complicated: since the ‘inter’ communication happens in the overlap communicator, you have to translate its ordering into those of the two worker communicators. It would be easier to express messages directly in terms of those communicators, and this is what happens in an *inter-communicator*.

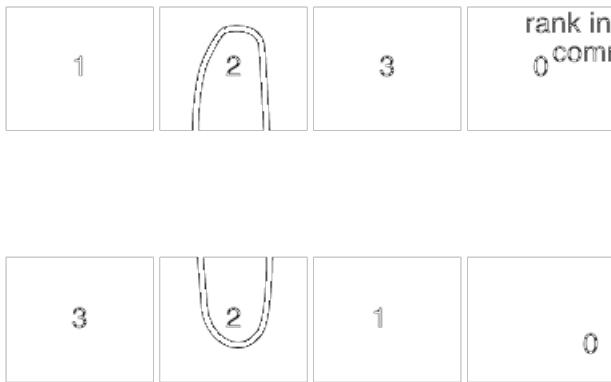


Figure 7.2: Illustration of ranks in an intercommunicator setup

A call to `MPI_Intercomm_create` (figure 7.9) involves the following communicators:

- Two local communicators, which in this context are known as *intra-communicators*: one process in each will act as the local leader, connected to the remote leader;
- The *peer communicator*, often `MPI_COMM_WORLD`, that contains the local communicators;
- An *inter-communicator* that allows the leaders of the subcommunicators to communicate with the other subcommunicator.

Even though the intercommunicator connects only two processes, it is collective on the peer communicator.

7.6.1 Intercommunicator point-to-point

The local leaders can now communicate with each other.

- The sender specifies as target the local number of the other leader in the other sub-communicator;
- Likewise, the receiver specifies as source the local number of the sender in its sub-communicator.

In one way, this design makes sense: processors are referred to in their natural, local, numbering. On the other hand, it means that each group needs to know how the local ordering of the other group is arranged. Using a complicated `key` value makes this difficult.

```
if (i_am_local_leader) {
    if (color==0) {
```

7. MPI topic: Communicators

Figure 7.9 MPI_Intercomm_create

Name	Param name	Explanation	C type	F type	inout
		MPI_Intercomm_create (
	local_comm	local intra-communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	local_leader	rank of local group leader in local_comm	int	INTEGER	IN
	peer_comm	``peer'' communicator; significant only at the local_leader	MPI_Comm	TYPE (MPI_Comm)	IN
	remote_leader	rank of remote group leader in peer_comm; significant only at the local_leader	int	INTEGER	IN
tag	tag		int	INTEGER	IN
newintercomm	new	inter-communicator	MPI_Comm*	TYPE (MPI_Comm)	OUT
)					

```

interdata = 1.2;
int inter_target = local_number_of_other_leader;
printf("%d sending interdata %e to %d\n",
       procno,interdata,inter_target);
MPI_Send(&interdata,1,MPI_DOUBLE,inter_target,0,intercomm);
} else {
    MPI_Status status;
    MPI_Recv(&interdata,1,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,intercomm,&status);
    int inter_source = status.MPI_SOURCE;
    printf("%d received interdata %e from %d\n",
           procno,interdata,inter_source);
    if (inter_source!=local_number_of_other_leader)
        fprintf(stderr,
                "Got inter communication from unexpected %d; s/b %d\n",
                inter_source,local_number_of_other_leader);
}

```

7.6.2 Intercommunicator collectives

The intercommunicator can be used in collectives such as a broadcast.

- In the sending group, the root process passes `MPI_ROOT` as ‘root’ value; all others use `MPI_PROC_NULL`.
 - In the receiving group, all processes use a ‘root’ value that is the rank of the root process in the root group. Note: this is not the global rank!

Figure 7.10 MPI_Comm_get_parent

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_get_parent	parent	the parent communicator	MPI_Comm*	TYPE (MPI_Comm)	OUT

Figure 7.11 MPI_Comm_test_inter

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_test_inter	comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	flag	true if comm is an inter-communicator	int*	LOGICAL	OUT

Gather and scatter behave similarly; the allgather is different: all send buffers of group A are concatenated in rank order, and places on all processes of group B.

Intercommunicators can be used if two groups of process work asynchronously with respect to each other; another application is fault tolerance (section 15.5).

```
if (color==0) { // sending group: the local leader sends
    if (i_am_local_leader)
        root = MPI_ROOT;
    else
        root = MPI_PROC_NULL;
} else { // receiving group: everyone indicates leader of other group
    root = local_number_of_other_leader;
}
if (DEBUG) fprintf(stderr, "[%d] using root value %d\n", procno, root);
MPI_Bcast(&bcast_data, 1, MPI_INT, root, intercomm);
```

7.6.3 Intercommunicator querying

Some of the operations you have seen before for *intra-communicators* behave differently with intercommunicator:

- **MPI_Comm_size** returns the size of the local group, not the size of the intercommunicator.
- **MPI_Comm_rank** returns the rank in the local group.
- **MPI_Comm_group** returns the local group.

Spawned processes can find their parent communicator with **MPI_Comm_get_parent** (figure 7.10) (see examples in section 8.1). On other processes this returns **MPI_COMM_NULL**.

Test whether a communicator is intra or inter: **MPI_Comm_test_inter** (figure 7.11).

7. MPI topic: Communicators

Figure 7.12 MPI_Comm_remote_size

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_remote_size		(comm inter-communicator size number of processes in the remote group of comm)	MPI_Comm	TYPE (MPI_Comm)	IN

Python:

```
Intercomm.Get_remote_size(self)
```

Figure 7.13 MPI_Comm_remote_group

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_remote_group		(comm inter-communicator group remote group corresponding to comm)	MPI_Comm	TYPE (MPI_Comm)	IN

Python:

```
Intercomm.Get_remote_group(self)
```

MPI_Comm_compare works for intercommunicators.

Processes connected through an intercommunicator can query the size of the ‘other’ communicator with **MPI_Comm_remote_size** (figure 7.12). The actual group can be obtained with **MPI_Comm_remote_group** (figure 7.13).

Virtual topologies (chapter 11) cannot be created with an intercommunicator. To set up virtual topologies, first transform the intercommunicator to an intracomunicator with the function **MPI_Intercomm_merge** (figure 7.14).

Figure 7.14 MPI_Intercomm_merge

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Intercomm_merge (</code>					
	<code>intercomm</code>	<code>inter-communicator</code>	<code>MPI_Comm</code>	<code>TYPE (MPI_Comm)</code>	<code>IN</code>
	<code>high</code>	<code>ordering of the local and remote groups in the new intra-communicator</code>	<code>int</code>	<code>LOGICAL</code>	<code>IN</code>
	<code>newintracomm</code>	<code>new intra-communicator</code>	<code>MPI_Comm*</code>	<code>TYPE (MPI_Comm)</code>	<code>OUT</code>
<code>)</code>					

7.7 Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

1. True or false: in each communicator, processes are numbered consecutively from zero.
2. If a process is in two communicators, it has the same rank in both.
3. Any communicator that is not `MPI_COMM_WORLD` is a strict subset of it.
4. The subcommunicators derived by `MPI_Comm_split` are disjoint.
5. If two processes have ranks $p < q$ in some communicator, and they are in the same subcommunicator, then their ranks p', q' in the subcommunicator also obey $p' < q'$.

Chapter 8

MPI topic: Process management

In this course we have up to now only considered the SPMD model of running MPI programs. In some rare cases you may want to run in an MPMD mode, rather than SPMD. This can be achieved either on the OS level, using options of the `mpiexec` mechanism, or you can use MPI's built-in process management. Read on if you're interested in the latter.

8.1 Process spawning

The first version of MPI did not contain any process management routines, even though the earlier PVM project did have that functionality. Process management was later added with MPI-2.

Unlike what you might think, newly added processes do not become part of `MPI_COMM_WORLD`; rather, they get their own communicator, and an *inter-communicator* (section 7.6) is established between this new group and the existing one. The first routine is `MPI_Comm_spawn` (figure 8.1), which tries to fire up multiple copies of a single named executable. Errors in starting up these codes are returned in an array of integers, or if you're feeling sure of yourself, specify `MPI_ERRCODES_IGNORE`.

It is not immediately clear whether there is opportunity for spawning new executables; after all, `MPI_COMM_WORLD` contains all your available processors. You can probably tell your job starter to reserve space for a few extra processes, but that is installation-dependent (see below). However, there is a standard mechanism for querying whether such space has been reserved. The attribute `MPI_UNIVERSE_SIZE`, retrieved with `MPI_Comm_get_attr` (section 15.1.2), will tell you to the total number of hosts available.

If this option is not supported, you can determine yourself how many processes you want to spawn. However, if you exceed the hardware resources, your multi-tasking operating system (which is some variant of Unix for almost everyone) will use *time-slicing* to start the spawned processes, but you will not gain any performance.

8.1.1 Commandline arguments

The `argv` argument contains the *commandline arguments* passed to the spawned process.

- This array needs to be *null-terminated*, so that its length can be determined.

Figure 8.1 MPI_Comm_spawn

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_spawn (
command		name of program to be spawned	const char*	CHARACTER	IN
argv		arguments to command	char* []	CHARACTER(*)	IN
maxprocs		maximum number of processes to start	int	INTEGER	IN
info		a set of key-value pairs telling the runtime system where and how to start the processes	MPI_Info	TYPE (MPI_Info)	IN
root		rank of process in which previous arguments are examined	int	INTEGER	IN
comm		intra-communicator containing group of spawning processes	MPI_Comm	TYPE (MPI_Comm)	IN
intercomm		inter-communicator between original group and the newly spawned group	MPI_Comm*	TYPE (MPI_Comm)	OUT
array_of_errcodes		one code per process	int []	INTEGER(*)	OUT
)					

Python:

```
MPI.Intracomm.Spawn(self,  
    command, args=None, int maxprocs=1, Info info=INFO_NULL,  
    int root=0, errcodes=None)  
returns an intracommunicator
```

- If the spawned process takes no commandline arguments, a value of `MPI_ARGV_NULL` can be used, in both C and Fortran. In C this is the same as `NULL`.
- Unlike the `argv` argument of a main program, the `argv` argument passed in the spawn call does not contain the name of the executable.

8.1.2 Example: work manager

Here is an example of a work manager. First we query how much space we have for new processes, using the flag to see if this option is supported:

```
int universe_size, *universe_size_attr, uflag;  
MPI_Comm_get_attr  
(comm_world,MPI_UNIVERSE_SIZE,  
&universe_size_attr,&uflag);
```

8. MPI topic: Process management

```
if (uflag) {
    universe_size = *universe_size_attr;
} else {
    printf("This MPI does not support UNIVERSE_SIZE.\nUsing world size");
    universe_size = world_n;
}
int work_n = universe_size - world_n;
if (world_p==0) {
    printf("A universe of size %d leaves room for %d workers\n",
           universe_size,work_n);
    printf(.. spawning from %s\n",procname);
}
```

(See section 15.1.2 for that dereference behavior.)

Then we actually spawn the processes:

```
const char *workerprogram = "./spawnapp";
MPI_Comm_spawn(workerprogram,MPI_ARGV_NULL,
               work_n,MPI_INFO_NULL,
               0,comm_world,&comm_inter,NULL);

## spawnmanager.py
try :
    universe_size = comm.Get_attr(MPI.UNIVERSE_SIZE)
    if universe_size is None:
        print("Universe query returned None")
        universe_size = nprocs + 4
    else:
        print("World has {} ranks in a universe of {}"\ \
              .format(nprocs,universe_size))
except :
    print("Exception querying universe size")
    universe_size = nprocs + 4
nworkers = universe_size - nprocs

intercomm = comm.Spawn("./spawn_worker.py", maxprocs=nworkers)
```

A process can detect whether it was a spawning or a spawned process by using `MPI_Comm_get_parent`: the resulting intercommunicator is `MPI_COMM_NULL` on the parent processes.

```
// spawnapp.c
MPI_Comm comm_parent;
MPI_Comm_get_parent(&comm_parent);
int is_child = (comm_parent!=MPI_COMM_NULL);
if (is_child) {
    int nworkers,workerno;
    MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
    MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
    printf("I detect I am worker %d/%d running on %s\n",
           workerno,nworkers,procname);
```

The spawned program looks very much like a regular MPI program, with its own initialization and finalize calls.

```
// spawnworker.c
MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
MPI_Comm_get_parent(&parent);

## spawnworker.py
parentcomm = comm.Get_parent()
nparents = parentcomm.Get_remote_size()
```

Spawned processes wind up with a value of `MPI_COMM_WORLD` of their own, but managers and workers can find each other regardless. The spawn routine returns the intercommunicator to the parent; the children can find it through `MPI_Comm_get_parent` (section 7.6.3). The number of spawning processes can be found through `MPI_Comm_remote_size` on the parent communicator.

```
Running spawnapp with usize=12, wsize=4
%%
%% manager output
%%
A universe of size 12 leaves room for 8 workers
.. spawning from c209-026.frontera.tacc.utexas.edu
%%
%% worker output
%%
Worker deduces 8 workers and 4 parents
I detect I am worker 0/8 running on c209-027.frontera.tacc.utexas.edu
I detect I am worker 1/8 running on c209-027.frontera.tacc.utexas.edu
I detect I am worker 2/8 running on c209-027.frontera.tacc.utexas.edu
I detect I am worker 3/8 running on c209-027.frontera.tacc.utexas.edu
I detect I am worker 4/8 running on c209-028.frontera.tacc.utexas.edu
I detect I am worker 5/8 running on c209-028.frontera.tacc.utexas.edu
I detect I am worker 6/8 running on c209-028.frontera.tacc.utexas.edu
I detect I am worker 7/8 running on c209-028.frontera.tacc.utexas.edu
```

8.1.3 MPI startup with universe

You could start up a single copy of this program with

```
mpiexec -n 1 spawnmanager
```

but with a hostfile that has more than one host.

TACC note. Intel MPI requires you to pass an option `-usize` to `mpiexec` indicating the size of the comm universe. With the TACC jobs starter `ibrun` do the following:

Figure 8.2 MPI_Open_port

Name	Param name	Explanation	C type	F type	inout
MPI_Open_port (info	implementation-specific information on how to establish an address	MPI_Info	TYPE (MPI_Info)	IN
	port_name	newly established port	char*	CHARACTER	OUT
)					

```
export FI_MLX_ENABLE_SPAWN=yes
# specific
MY_MPIRUN_OPTIONS="-usize 8" ibrun -np 4 spawnmanager
# more generic
MY_MPIRUN_OPTIONS="-usize ${SLURM_NPROCS}" ibrun -np 4 spawnmanager
# using mpiexec:
mpiexec -np 2 -usize ${SLURM_NPROCS} spawnmanager
```

8.1.4 MPMD

Instead of spawning a single executable, you can spawn multiple with [MPI_Comm_spawn_multiple](#). In that case a process can retrieve with the attribute [MPI_APPNUM](#) which of the executables it is; section 15.1.2.

Commandline arguments are handled similarly to [MPI_Comm_spawn](#) (section 8.1.1), except that there is now an array of arrays of strings. If no executables take *commandline arguments* of multiple spawns, the value [MPI_ARGVS_NULL](#) can be passed. If only certain executables take no arguments, for them an array of length 1 needs to be passed containing only the *null-terminator*.

8.2 Socket-style communications

It is possible to establish connections with running MPI programs that have their own world communicator.

- The *server* process establishes a port with [MPI_Open_port](#), and calls [MPI_Comm_accept](#) to accept connections to its port.
- The *client* process specifies that port in an [MPI_Comm_connect](#) call. This establishes the connection.

8.2.1 Server calls

The server calls [MPI_Open_port](#) (figure 8.2), yielding a port name. Port names are generated by the system and copied into a character buffer of length at most [MPI_MAX_PORT_NAME](#).

The server then needs to call [MPI_Comm_accept](#) (figure 8.3) prior to the client doing a connect call. This

Figure 8.3 MPI_Comm_accept

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_accept (
port_name	port_name	const char*	CHARACTER	IN	
info	implementation-dependent	MPI_Info	TYPE	IN	
	information		(MPI_Info)		
root	rank in comm of root node	int	INTEGER	IN	
comm	intra-communicator over which call is collective	MPI_Comm	TYPE	IN	
			(MPI_Comm)		
newcomm	inter-communicator with client as remote group	MPI_Comm*	TYPE	OUT	
)			(MPI_Comm)		

is collective over the calling communicator. It returns an intercommunicator (section 7.6) that allows communication with the client.

```

MPI_Comm intercomm;
char myport[MPI_MAX_PORT_NAME];
MPI_Open_port( MPI_INFO_NULL,myport );
int portlen = strlen(myport);
MPI_Send( myport,portlen+1,MPI_CHAR,1,0,comm_world );
printf("Host sent port <<%s>>\n",myport);
MPI_Comm_accept( myport,MPI_INFO_NULL,0,comm_self,&intercomm );
printf("host accepted connection\n");

```

The port can be closed with `MPI_Close_port`.

8.2.2 Client calls

After the server has generated a port name, the client needs to connect to it with `MPI_Comm_connect` (figure 8.4), again specifying the port through a character buffer. The connect call is collective over its communicator.

```

char myport[MPI_MAX_PORT_NAME];
if (work_p==0) {
    MPI_Recv( myport,MPI_MAX_PORT_NAME,MPI_CHAR,
              MPI_ANY_SOURCE,0, comm_world,MPI_STATUS_IGNORE );
    printf("Worker received port <<%s>>\n",myport);
}
MPI_Bcast( myport,MPI_MAX_PORT_NAME,MPI_CHAR,0,comm_work );

/*
 * The workers collective connect over the inter communicator
 */
MPI_Comm intercomm;
MPI_Comm_connect( myport,MPI_INFO_NULL,0,comm_work,&intercomm );

```

8. MPI topic: Process management

Figure 8.4 MPI_Comm_connect

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_connect					
	port_name	network address	const char*	CHARACTER	IN
	info	implementation-dependent information	MPI_Info	TYPE (MPI_Info)	IN
	root	rank in comm of root node	int	INTEGER	IN
	comm	intra-communicator over which call is collective	MPI_Comm	TYPE (MPI_Comm)	IN
	newcomm	inter-communicator with server as remote group	MPI_Comm*	TYPE (MPI_Comm)	OUT
)				

```
if (work_p==0) {
    int manage_n;
    MPI_Comm_remote_size(intercomm,&manage_n);
    printf("%d workers connected to %d managers\n",work_n,manage_n);
}
```

If the named port does not exist (or has been closed), `MPI_Comm_connect` raises an error of class `MPI_ERR_PORT`.

The client can sever the connection with `MPI_Comm_disconnect`.

Running the above code on 5 processes gives:

```
# exchange port name:
Host sent port <<tag#0$0FA#000010e1:0001cde9:0001cdee$rdma_port#1024$rdma_host#10:16:225:0:1:205:199:25
Worker received port <<tag#0$0FA#000010e1:0001cde9:0001cdee$rdma_port#1024$rdma_host#10:16:225:0:1:205:

# Comm accept/connect
host accepted connection
4 workers connected to 1 managers

# Send/recv over the intercommunicator
Manager sent 4 items over intercomm
Worker zero received data
```

8.2.3 Published service names

More elegantly than the port mechanism above, it is possible to publish a named service, with `MPI_Publish_name` (figure 8.5), which can then be discovered by other processes.

```
// publishapp.c
MPI_Comm intercomm;
char myport[MPI_MAX_PORT_NAME];
MPI_Open_port( MPI_INFO_NULL,myport );
```

Figure 8.5 MPI_Publish_name

Name	Param name	Explanation	C type	F type	inout
MPI_Publish_name (
service_name	a service name to associate with the port	const char*	CHARACTER		IN
info	implementation-specific information	MPI_Info	TYPE (MPI_Info)		IN
port_name	a port name	const char*	CHARACTER		IN
)					

Figure 8.6 MPI_Unpublish_name

Name	Param name	Explanation	C type	F type	inout
MPI_Unpublish_name (
service_name	a service name	const char*	CHARACTER		IN
info	implementation-specific information	MPI_Info	TYPE (MPI_Info)		IN
port_name	a port name	const char*	CHARACTER		IN
)					

```
MPI_Publish_name( service_name, MPI_INFO_NULL, myport );
MPI_Comm_accept( myport,MPI_INFO_NULL,0,comm_self,&intercomm );
```

Worker processes connect to the intercommunicator by

```
char myport[MPI_MAX_PORT_NAME];
MPI_Lookup_name( service_name,MPI_INFO_NULL,myport );
MPI_Comm intercomm;
MPI_Comm_connect( myport,MPI_INFO_NULL,0,comm_work,&intercomm );
```

For this it is necessary to have a *name server* running.

Intel note. Start the *hydra* name server and use the corresponding mpi starter:

```
hydra_nameserver &
MPIEXEC=mpiexec.hydra
```

There is an environment variable, but that doesn't seem to be needed.

```
export I_MPI_HYDRA_NAMESERVER=`hostname`:8008
```

It is also possible to specify the name server as an argument to the job starter.

At the end of a run, the service should be unpublished with MPI_Unpublish_name (figure 8.6). Unpublishing a nonexisting or already unpublished service gives an error code of MPI_ERR_SERVICE.

MPI provides no guarantee of fairness in servicing connection attempts. That is, connection attempts are not necessarily satisfied in the order in which they were initiated, and competition from other connection attempts may prevent a particular connection attempt from being satisfied.

Figure 8.7 MPI_Comm_join

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_join (
fd	socket file descriptor		int	INTEGER	IN
intercomm	new inter-communicator		MPI_Comm*	TYPE (MPI_Comm)	OUT
)					

8.2.4 Unix sockets

It is also possible to create an *intercommunicator* from a Unix *socket* with `MPI_Comm_join` (figure 8.7).

8.3 Sessions

The most common way of initializing MPI, with `MPI_Init` (or `MPI_Init_thread`) and `MPI_Finalize`, is known as the *world model* which can be described as:

1. There is a single call to `MPI_Init` or `MPI_Init_thread`;
2. There is a single call to `MPI_Finalize`;
3. With very few exceptions, all MPI calls appear in between the initialize and finalize calls.

This model suffers from some disadvantages:

1. There is no error handling during `MPI_Init`.
2. MPI can not be finalized and restarted;
3. If multiple libraries are active, they can not initialize or finalize MPI, but have to base themselves on subcommunicators; section 7.2.2.
4. There is no threadsafe way of initializing MPI: a library can't safely do

```
MPI_Initialized(&flag);
if (!flag) MPI_Init(0,0);
```

if it is running in a multi-threaded environment.

The following material is for the recently released MPI-4 standard and may not be supported yet.

In addition to the world, where all MPI is bracketed by `MPI_Init` (or `MPI_Init_thread`) and `MPI_Finalize`, there is the *session model*, where entities such as libraries can start/end their MPI session independently.

The two models can be used in the same program, but there are limitations on how they can mix.

8.3.1 Short description of the session model

In the *session model*, each session starts and finalizes MPI independently, giving each a separate `MPI_COMM_WORLD`. The world model then becomes a separate way of starting MPI. You can create a communicator using the world model in addition to starting multiple sessions, each on their own set of processes, possibly identical or overlapping. You can also create sessions without have an `MPI_COMM_WORLD` created by the world model.

Figure 8.8 MPI_Session_init

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Session_init (</code>					
<code>info</code>		info object to specify thread support level and MPI implementation specific resources	<code>MPI_Info</code>	<code>TYPE (MPI_Info)</code>	<code>IN</code>
<code>errhandler</code>		error handler to invoke in the event that an error is encountered during this function call	<code>MPI_Errhandler</code>	<code>TYPE (MPI_Errhandler)</code>	<code>IN</code>
<code>session</code>		new session	<code>MPI_Session*</code>	<code>TYPE (MPI_Session)</code>	<code>OUT</code>
<code>)</code>					

You can not mix in a single call objects from different sessions, from a session and from the world model, or from a session and from `MPI_Comm_get_parent` or `MPI_Comm_join`.

8.3.2 Session creation

An MPI session is initialized and finalized with `MPI_Session_init` (figure 8.8) and `MPI_Session_finalize`, somewhat similar to `MPI_Init` and `MPI_Finalize`.

```
MPI_Session the_session;
MPI_Session_init
( session_request_info,MPI_ERRORS_ARE_FATAL,
&the_session );
MPI_Session_finalize( &the_session );
```

This call is thread-safe, in view of the above reasoning.

8.3.2.1 Session info

The `MPI_Info` object that is passed to `MPI_Session_init` can be null, or it can be used to request a threading level:

```
// session.c
MPI_Info session_request_info = MPI_INFO_NULL;
MPI_Info_create(&session_request_info);
char thread_key[] = "mpi_thread_support_level";
MPI_Info_set(session_request_info,
thread_key,"MPI_THREAD_MULTIPLE");
```

Other info keys can be implementation-dependent, but the key `thread_support` is pre-defined.

Info keys can be retrieved again with `MPI_Session_get_info`:

8. MPI topic: Process management

```
MPI_Info session_actual_info;
MPI_Session_get_info( the_session,&session_actual_info );
char thread_level[100]; int info_len = 100, flag;
MPI_Info_get_string( session_actual_info,
                     thread_key,&info_len,thread_level,&flag );
```

8.3.2.2 Session error handler

The error handler argument accepts a pre-defined error handler (section 15.2.2) or one created by `MPI_Session_create_errhandler`.

8.3.3 Process sets and communicators

A session has a number of *process sets*. Process sets are indicated with a Uniform Resource Identifier (URI), where the URIs `mpi://WORLD` and `mpi://SELF` are always defined.

You query the ‘psets’ with `MPI_Session_get_num_psets` and `MPI_Session_get_nth_pset`:

Code:

```
int npsets;
MPI_Session_get_num_psets
( the_session,MPI_INFO_NULL,&npsets )
;
if (mainproc)
printf("Number of process sets: %d\n",
,npsets);
for (int ipset=0; ipset<npsets; ipset
++)
{
int len_pset; char name_pset[
MPI_MAX_PSET_NAME_LEN];
MPI_Session_get_nth_pset
( the_session,MPI_INFO_NULL,
ipset,&len_pset,name_pset );
if (mainproc)
printf("Process set %2d: <<%s>>\n",
ipset,name_pset);
```

Output:

```
[examples/mpi/c] session:
mpieexec -n 2 ./session
Could not obtain thread
    ↳level,flag=0
Number of process sets: 2
Process set 0: <<mpi://WORLD>>
Process set 1: <<mpi://SELF>>
Found WORLD as pset 0
World has 2 processes
```

The following partial code creates a communicator equivalent to `MPI_COMM_WORLD` in the session model:

```
MPI_Group world_group = MPI_GROUP_NULL;
MPI_Comm world_comm = MPI_COMM_NULL;
MPI_Group_from_session_pset
( the_session,world_name,&world_group );
MPI_Comm_create_from_group
( world_group,"victor-code-session.c",
MPI_INFO_NULL,MPI_ERRORS_ARE_FATAL,
&world_comm );
MPI_Group_free( &world_group );
```

```

int procid = -1, nprocs = 0;
MPI_Comm_size(world_comm,&nprocs);
MPI_Comm_rank(world_comm,&procid);

```

However, comparing communicators (with `MPI_Comm_compare`) from the session and world model, or from different sessions, is undefined behavior.

Get the info object (section 15.1.1) from a process set: `MPI_Session_get_pset_info`. This info object always has the key `mpi_size`.

8.3.4 Example

As an example of the use of sessions, we declare a library class, where each library object starts and ends its own session:

```

// sessionlib.cxx
class Library {
private:
    MPI_Comm world_comm; MPI_Session session;
public:
    Library() {
        MPI_Info info = MPI_INFO_NULL;
        MPI_Session_init(
            ( MPI_INFO_NULL,MPI_ERRORS_ARE_FATAL,&session ) );
        char world_name[] = "mpi://WORLD";
        MPI_Group world_group;
        MPI_Group_from_session_pset
            ( session,world_name,&world_group );
        MPI_Comm_create_from_group
            ( world_group,"world-session",
              MPI_INFO_NULL,MPI_ERRORS_ARE_FATAL,
              &world_comm );
        MPI_Group_free( &world_group );
    };
    ~Library() { MPI_Session_finalize(&session); };
}

```

Now we create a main program, using the world model, which activates two libraries, passing data to them by parameter:

```

int main(int argc,char **argv) {

    Library lib1,lib2;
    MPI_Init(0,0);
    MPI_Comm world = MPI_COMM_WORLD;
    int procno,nprocs;
    MPI_Comm_rank(world,&procno);
    MPI_Comm_size(world,&nprocs);
    auto sum1 = lib1.compute(procno);
    auto sum2 = lib2.compute(procno+1);
}

```

Note that no mpi calls will go between main program and either of the libraries, or between the two libraries, but this seems to make sense in this scenario.

End of MPI-4 material

8.4 Functionality available outside init/finalize

```
MPI_Initialized MPI_Finalized MPI_Get_version MPI_Get_library_version MPI_Info_create  
MPI_Info_create_env MPI_Info_set MPI_Info_delete MPI_Info_get MPI_Info_get_valuelen  
MPI_Info_get_nkeys MPI_Info_get_nthkey MPI_Info_dup MPI_Info_free MPI_Info_f2c MPI_Info_c2f  
MPI_Session_create_errhandler MPI_Session_call_errhandler MPI_Errhandler_free MPI_Errhandler_f2c  
MPI_Errhandler_c2f MPI_Error_string MPI_Error_class
```

Also all routines starting with `MPI_Txxx`.

Chapter 9

MPI topic: One-sided communication

Above, you saw point-to-point operations of the two-sided type: they require the co-operation of a sender and receiver. This co-operation could be loose: you can post a receive with `MPI_ANY_SOURCE` as sender, but there had to be both a send and receive call. This two-sidedness can be limiting. Consider code where the receiving process is a dynamic function of the data:

```
x = f();
p = hash(x);
MPI_Send( x, /* to: */ p );
```

The problem is now: how does `p` know to post a receive, and how does everyone else know not to?

In this section, you will see one-sided communication routines where a process can do a ‘put’ or ‘get’ operation, writing data to or reading it from another processor, without that other processor’s involvement.

In one-sided MPI operations, known as Remote Memory Access (RMA) operations in the standard, or as Remote Direct Memory Access (RDMA) in other literature, there are still two processes involved: the *origin*, which is the process that originates the transfer, whether this is a ‘put’ or a ‘get’, and the *target* whose memory is being accessed. Unlike with two-sided operations, the target does not perform an action that is the counterpart of the action on the origin.

That does not mean that the origin can access arbitrary data on the target at arbitrary times. First of all, one-sided communication in MPI is limited to accessing only a specifically declared memory area on the target: the target declares an area of memory that is accessible to other processes. This is known as a *window*. Windows limit how origin processes can access the target’s memory: you can only ‘get’ data from a window or ‘put’ it into a window; all the other memory is not reachable from other processes. On the origin there is no such limitation; any data can function as the source of a ‘put’ or the recipient of a ‘get’ operation.

The alternative to having windows is to use *distributed shared memory* or *virtual shared memory*: memory is distributed but acts as if it shared. The so-called Partitioned Global Address Space (PGAS) languages such as Unified Parallel C (UPC) use this model.

Within one-sided communication, MPI has two modes: active RMA and passive RMA. In *active RMA*, or *active target synchronization*, the target sets boundaries on the time period (the ‘epoch’) during which its window can be accessed. The main advantage of this mode is that the origin program can perform many

small transfers, which are aggregated behind the scenes. This would be appropriate for applications that are structured in a Bulk Synchronous Parallel (BSP) mode with *supersteps*. Active RMA acts much like asynchronous transfer with a concluding [MPI_Waitall](#).

In *passive RMA*, or *passive target synchronization*, the target process puts no limitation on when its window can be accessed. (PGAS languages such as UPC are based on this model: data is simply read or written at will.) While intuitively it is attractive to be able to write to and read from a target at arbitrary time, there are problems. For instance, it requires a remote agent on the target, which may interfere with execution of the main thread, or conversely it may not be activated at the optimal time. Passive RMA is also very hard to debug and can lead to race conditions.

9.1 Windows

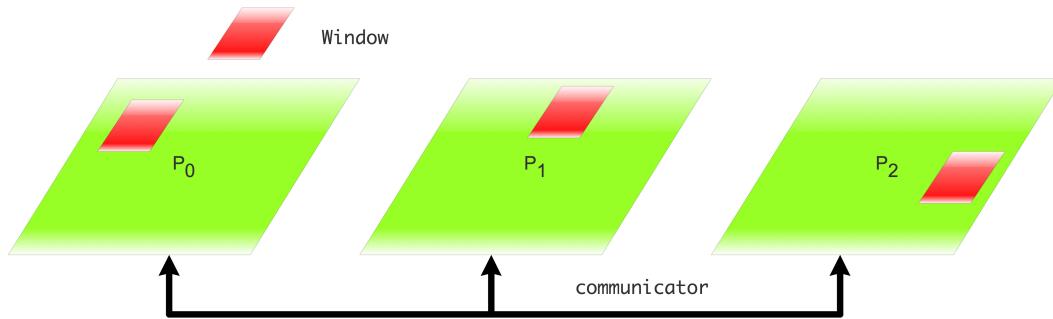


Figure 9.1: Collective definition of a window for one-sided data access

In one-sided communication, each processor can make an area of memory, called a *window*, available to one-sided transfers. This is stored in a variable of type [MPI_Win](#). A process can put an arbitrary item from its own memory (not limited to any window) to the window of another process, or get something from the other process' window in its own memory.

A window can be characterized as follows:

- The window is defined on a communicator, so the create call is collective; see figure 9.1.
- The window size can be set individually on each process. A zero size is allowed, but since window creation is collective, it is not possible to skip the create call.
- You can set a ‘displacement unit’ for the window: this is a number of bytes that will be used as the indexing unit. For example if you use `sizeof(double)` as the displacement unit, an [MPI_Put](#) to location 8 will go to the 8th double. That’s easier than having to specify the 64th byte.
- The window is the target of data in a put operation, or the source of data in a get operation; see figure 9.2.
- There can be memory associated with a window, so it needs to be freed explicitly with [MPI_Win_free](#).

The typical calls involved are:

Figure 9.1 MPI_Win_create

Name	Param name	Explanation	C type	F type	inout
MPI_Win_create (
MPI_Win_create_c (
base	base	initial address of window	void*	TYPE(*), DIMENSION(..)	IN
size	size	size of window in bytes	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
disp_unit	disp_unit	local unit size for displacements, in bytes	[int MPI_Aint	INTEGER	IN
info	info	info argument	MPI_Info	TYPE (MPI_Info)	IN
comm	comm	intra-communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)	win	window object	MPI_Win*	TYPE(MPI_Win)	OUT

Python:

```
MPI.Win.Create
  (memory, int disp_unit=1,
   Info info=INFO_NULL, Intracomm comm=COMM_SELF)
```

```
MPI_Info info;
MPI_Win window;
MPI_Win_allocate( /* size info */, info, comm, &memory, &window );
// do put and get calls
MPI_Win_free( &window );
```

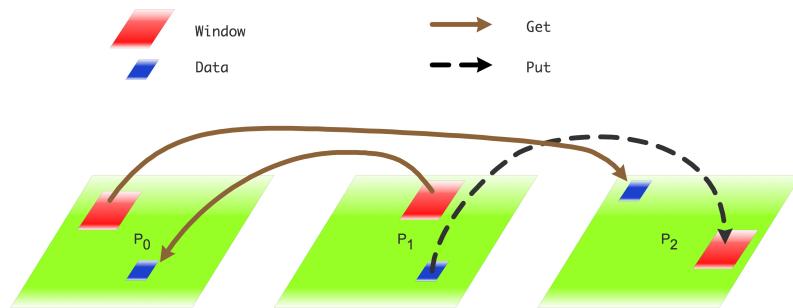


Figure 9.2: Put and get between process memory and windows

9.1.1 Window creation and freeing

The memory for a window is at first sight ordinary data in user space. There are multiple ways you can associate data with a window:

1. You can pass a user buffer to `MPI_Win_create` (figure 9.1). This buffer can be an ordinary array,

Figure 9.2 MPI_Win_allocate

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Win_allocate</code>					
	<code>MPI_Win_allocate_c</code>				
	<code>size</code>	size of window in bytes	<code>MPI_Aint</code>	<code>INTEGER</code> (<code>KIND=MPI_ADDRESS_KIND</code>)	IN
	<code>disp_unit</code>	local unit size for displacements, in bytes	<code>[int MPI_Aint]</code>	<code>INTEGER</code>	IN
	<code>info</code>	info argument	<code>MPI_Info</code>	<code>TYPE</code> (<code>MPI_Info</code>)	IN
	<code>comm</code>	intra-communicator	<code>MPI_Comm</code>	<code>TYPE</code> (<code>MPI_Comm</code>)	IN
	<code>baseptr</code>	initial address of window	<code>void*</code>	<code>TYPE(C_PTR)</code>	OUT
	<code>win</code>	window object returned by call	<code>MPI_Win*</code>	<code>TYPE(MPI_Win)</code>	OUT
)				

or it can be created with `MPI_Alloc_mem`. (In the former case, it may not be possible to lock the window; section 9.4.)

2. You can let MPI do the allocation, so that MPI can perform various optimizations regarding placement of the memory. The user code then receives the pointer to the data from MPI. This can again be done in two ways:
 - Use `MPI_Win_allocate` (figure 9.2) to create the data and the window in one call.
 - If a communicator is on a shared memory (see section 7.4.1) you can create a window in that shared memory with `MPI_Win_allocate_shared`. This will be useful for MPI *shared memory*; see chapter 12.
3. Finally, you can create a window with `MPI_Win_create_dynamic` which postpones the allocation; see section 9.5.3.

First of all, `MPI_Win_create` creates a window from a pointer to memory. The data array must not be `PARAMETER` or `static const`.

The size parameter is measured in bytes. In C this can be done with the `sizeof` operator:

```
// putfencealloc.c
MPI_Win the_window;
int *window_data;
MPI_Win_allocate(2*sizeof(int),sizeof(int),
                MPI_INFO_NULL,comm,
                &window_data,&the_window);
```

for doing this calculation in Fortran, see section 15.3.1.

Python note 27: Displacement byte computations. For computing the displacement in bytes, here is a good way for finding the size of *numpy* datatypes:

```
## putfence.py
```

Figure 9.3 MPI_Alloc_mem

Name	Param name	Explanation	C type	F type	inout
MPI_Alloc_mem (
size		size of memory segment in bytes	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
info		info argument	MPI_Info	TYPE (MPI_Info)	IN
baseptr		pointer to beginning of memory segment allocated	void*	TYPE(C_PTR)	OUT
)					

```
intsize = np.dtype('int').itemsize
window_data = np.zeros(2,dtype=int)
win = MPI.Win.Create(window_data,intsize,comm=comm)
```

Next, one can obtain the memory from MPI by using `MPI_Win_allocate`, which has the data pointer as output. Note the `void*` in the C signature; it is still necessary to pass a pointer to a pointer:

```
double *window_data;
MPI_Win_allocate( ... &window_data ... );
```

The routine `MPI_Alloc_mem` (figure 9.3) performs only the allocation part of `MPI_Win_allocate`, after which you need to `MPI_Win_create`.

- An error of `MPI_ERR_NO_MEM` indicates that no memory could be allocated.

The following material is for the recently released MPI-4 standard and may not be supported yet.

- Allocated memory can be aligned by specifying an `MPI_Info` key of `mpi_minimum_memory_alignment`.
- The type of memory allocated can be controlled by info keys; see section 9.5.2.
- An info key `mpi_accumulate_granularity` can be used to distinguish between accumulate operations for which throughput, or rather latency, is more important. This key has an integer value of the number of bytes in between synchronizations; this can also stand for the number of bytes that is atomically written (MPI-4.1).

End of MPI-4 material

This memory is freed with `MPI_Free_mem`:

```
// getfence.c
int *number_buffer = NULL;
MPI_Alloc_mem
( /* size: */ 2*sizeof(int),
  MPI_INFO_NULL,&number_buffer);
MPI_Win_create
( number_buffer,2*sizeof(int),sizeof(int),
  MPI_INFO_NULL,comm,&the_window);
MPI_Win_free(&the_window);
MPI_Free_mem(number_buffer);
```

Figure 9.4 MPI_Win_free

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Win_free (</code>	<code>win</code>	window object	<code>MPI_Win*</code>	<code>TYPE(MPI_Win)</code>	<code>INOUT</code>

(Note the lack of an ampersand in the free call!)

These calls reduce to `malloc` and `free` if there is no special memory area; SGI is an example where such memory does exist.

A window is freed with a call to the collective `MPI_Win_free` (figure 9.4), which sets the window handle to `MPI_WIN_NULL`. This call must only be done if all RMA operations are concluded, by `MPI_Win_fence`, `MPI_Win_wait`, `MPI_Win_complete`, `MPI_Win_unlock`, depending on the case. If the window memory was allocated internally by MPI through a call to `MPI_Win_allocate` or `MPI_Win_allocate_shared`, it is freed. User memory used for the window can be freed after the `MPI_Win_free` call.

There will be more discussion of window memory in section 9.5.1.

Python note 28: Window buffers. Unlike in C, the python window allocate call does not return a pointer to the buffer memory, but an `MPI.memory` object. Should you need the bare memory, there are the following options:

- Window objects expose the Python buffer interface. So you can do Pythonic things like


```
mview = memoryview(win)
array = numpy.frombuffer(win, dtype='i4')
```
- If you really want the raw base pointer (as an integer), you can do any of these:


```
base, size, disp_unit = win.atts
base = win.Get_attr(MPI.WIN_BASE)
```
- You can use mpi4py's builtin memoryview/buffer-like type, but I do not recommend it, much better to use NumPy as above:


```
mem = win.tomemory() # type(mem) is MPI.memory, similar to memoryview, but quite limited
                           in functionality
base = mem.address
size = mem.nbytes
```

9.1.2 Address arithmetic

Working with windows involves a certain amount of arithmetic on addresses, meaning `MPI_Aint`. See `MPI_Aint_add` and `MPI_Aint_diff` in section 6.2.4.

9.2 Active target synchronization: epochs

One-sided communication has an obvious complication over two-sided: if you do a put call instead of a send, how does the recipient know that the data is there? This process of letting the target know the state

Figure 9.5 MPI_Win_fence

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Win_fence (</code>					

Python:

```
win.Fence(self, int assertion=0)
```

of affairs is called ‘synchronization’, and there are various mechanisms for it. First of all we will consider *active target synchronization*. Here the target knows when the transfer may happen (the *communication epoch*), but does not do any data-related calls.

In this section we look at the first mechanism, which is to use a *fence* operation: `MPI_Win_fence` (figure 9.5). This operation is collective on the communicator of the window. (Another, more sophisticated mechanism for active target synchronization is discussed in section 9.2.2.)

The interval between two fences is known as an *epoch*. Roughly speaking, in an epoch you can make one-sided communication calls, and after the concluding fence all these communications are concluded.

```
MPI_Win_fence(0,win);
MPI_Get( /* operands */, win);
MPI_Win_fence(0, win);
// the `got' data is available
```

In between the two fences the window is exposed, and while it is you should not access it locally. If you absolutely need to access it locally, you can use an RMA operation for that. Also, there can be only one remote process that does a put; multiple accumulate accesses are allowed.

Fences are, together with other window calls, collective operations. That means they imply some amount of synchronization between processes. Consider:

```
MPI_Win_fence( ... win ... ); // start an epoch
if (mytid==0) // do lots of work
MPI_Win_fence( ... win ... ); // end the epoch
```

and assume that all processes execute the first fence more or less at the same time. The zero process does work before it can do the second fence call, but all other processes can call it immediately. However, they can not finish that second fence call until all one-sided communication is finished, which means they wait for the zero process.

As a further restriction, you can not mix `MPI_Get` with `MPI_Put` or `MPI_Accumulate` calls in a single epoch. Hence, we can characterize an epoch as an *access epoch* on the origin, and as an *exposure epoch* on the target.

9.2.1 Fence assertions

You can give various hints to the system about this epoch versus the ones before and after through the `assert` parameter.

9. MPI topic: One-sided communication

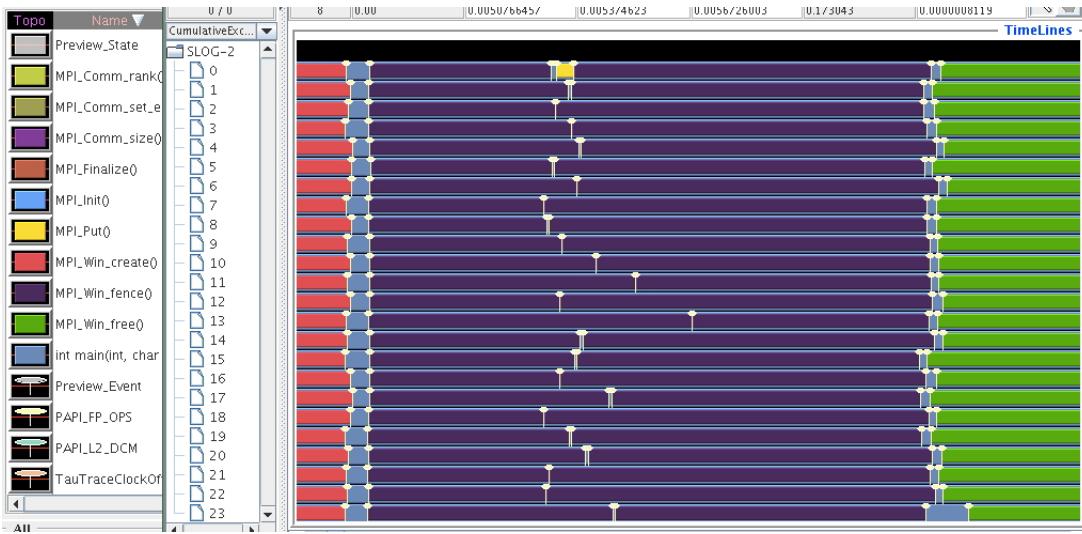


Figure 9.3: A trace of a one-sided communication epoch where process zero only originates a one-sided transfer

- **MPI_MODE_NOSTORE** This value can be specified or not per process.
- **MPI_MODE_NOPUT** This value can be specified or not per process.
- **MPI_MODE_NOPRECEDE** This value has to be specified or not the same on all processes.
- **MPI_MODE_NOSUCCEED** This value has to be specified or not the same on all processes.

Example:

```
MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
MPI_Get( /* operands */, win);
MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
```

Assertions are an integer parameter: you can combine assertions by adding them or using logical-or. The value zero is always correct. For further information, see section 9.6.

9.2.2 Non-global target synchronization

The ‘fence’ mechanism (section 9.2) uses a global synchronization on the communicator of the window, giving a program a BSP like character. As such it is good for applications where the processes are largely synchronized, but it may lead to performance inefficiencies if processors are not in step with each other. Also, global synchronization may have hardware support, making this less restrictive than it may at first seem.

There is a mechanism that is more fine-grained, by using synchronization only on a processor *group*. This takes four different calls, two for starting and two for ending the epoch, separately for target and origin.

You start and complete an *exposure epoch* with **`MPI_Win_post / MPI_Win_wait`**:

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_wait(MPI_Win win)
```

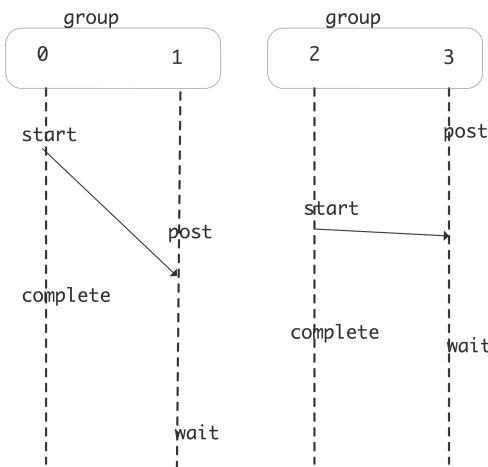


Figure 9.4: Window locking calls in fine-grained active target synchronization

In other words, this turns your window into the *target* for a remote access. There is a non-blocking version `MPI_Win_test` of `MPI_Win_wait`.

You start and complete an *access epoch* with `MPI_Win_start` / `MPI_Win_complete`:

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_complete(MPI_Win win)
```

In other words, these calls border the access to a remote window, with the current processor being the *origin* of the remote access.

In the following snippet a single processor puts data on one other. Note that they both have their own definition of the group, and that the receiving process only does the post and wait calls.

```
// postwaitwin.c
MPI_Comm_group(comm,&all_group);
if (procno==origin) {
    MPI_Group_incl(all_group,1,&target,&two_group);
// access
    MPI_Win_start(two_group,0,the_window);
    MPI_Put( /* data on origin: */ &my_number, 1,MPI_INT,
            /* data on target: */ target,0, 1,MPI_INT,
            the_window);
    MPI_Win_complete(the_window);
}

if (procno==target) {
    MPI_Group_incl(all_group,1,&origin,&two_group);
// exposure
    MPI_Win_post(two_group,0,the_window);
    MPI_Win_wait(the_window);
}
```

Both pairs of operations declare a *group of processors*; see section 7.5.1 for how to get such a group from a communicator. On an origin processor you would specify a group that includes the targets you will interact with, on a target processor you specify a group that includes the possible origins.

9.3 Put, get, accumulate

We will now look at the first three routines for doing one-sided operations: the Put, Get, and Accumulate call. (We will look at so-called ‘atomic’ operations in section 9.3.7.) These calls are somewhat similar to a Send, Receive and Reduce, except that of course only one process makes a call. Since one process does all the work, its calling sequence contains both a description of the data on the origin (the calling process) and the target (the affected other process).

As in the two-sided case, `MPI_PROC_NULL` can be used as a target rank.

The Accumulate routine has an `MPI_Op` argument that can be any of the usual operators, but no user-defined ones (see section 3.10.1).

9.3.1 Put

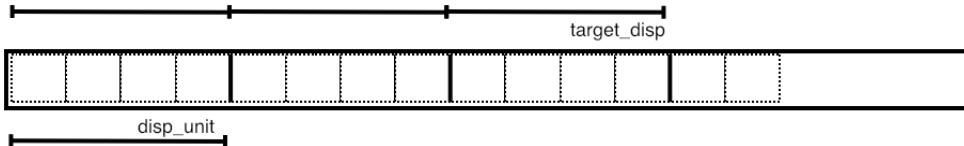
The `MPI_Put` (figure 9.6) call can be considered as a one-sided send. As such, it needs to specify

- the target rank
- the data to be sent from the origin, and
- the location where it is to be written on the target.

The description of the data on the origin is the usual trio of buffer/count/datatype. However, the description of the data on the target is more complicated. It has a count and a datatype, but additionally it has a *displacement* with respect to the start of the window on the target. This displacement can be given in bytes, so its type is `MPI_Aint`, but strictly speaking it is a multiple of the displacement unit that was specified in the window definition.

Specifically, data is written starting at

$$\text{window_base} + \text{target_disp} \times \text{disp_unit}.$$



Here is a single put operation. Note that the window create and window fence calls are collective, so they have to be performed on all processors of the communicator that was used in the create call.

```
// putfence.c
MPI_Win the_window;
MPI_Win_create
(&window_data,2*sizeof(int),sizeof(int),
```

Figure 9.6 MPI_Put

Name	Param name	Explanation	C type	F type	inout
MPI_Put					
	MPI_Put_c				
	origin_addr	initial address of origin buffer	const void*	TYPE(*), DIMENSION(..)	IN
	origin_count	number of entries in origin buffer	[int MPI_Count]	INTEGER	IN
	origin_datatype	datatype of each entry in origin buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
	target_rank	rank of target	int	INTEGER	IN
	target_disp	displacement from start of window to target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
	target_count	number of entries in target buffer	[int MPI_Count]	INTEGER	IN
	target_datatype	datatype of each entry in target buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
	win	window object used for communication	MPI_Win	TYPE(MPI_Win)	IN
)				

Python:

```
win.Put(self, origin, int target_rank, target=None)
```

```

MPI_INFO_NULL,comm,&the_window);
MPI_Win_fence(0,the_window);
if (procno==0) {
    MPI_Put
    ( /* data on origin: */   &my_number, 1,MPI_INT,
     /* data on target: */   other,1,      1,MPI_INT,
     the_window);
}
MPI_Win_fence(0,the_window);
MPI_Win_free(&the_window);

```

Fortran note 14: Displacement unit. The disp_unit variable is declared as an integer of ‘kind’ MPI_ADDRESS_KIND:

```
!! putfence.F90
integer(kind=MPI_ADDRESS_KIND) :: target_displacement
target_displacement = 1
call MPI_Put( my_number, 1,MPI_INTEGER, &
other,target_displacement, &
1,MPI_INTEGER, &
the_window)
```

Figure 9.7 MPI_Get

Name	Param name	Explanation	C type	F type	inout
MPI_Get (
MPI_Get_c (
origin_addr	origin_addr	initial address of origin buffer	void*	TYPE(*), DIMENSION(..)	OUT
origin_count	origin_count	number of entries in origin buffer	[int MPI_Count]	INTEGER	IN
origin_datatype	origin_datatype	datatype of each entry in origin buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
target_rank	target_rank	rank of target	int	INTEGER	IN
target_disp	target_disp	displacement from window start to the beginning of the target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
target_count	target_count	number of entries in target buffer	[int MPI_Count]	INTEGER	IN
target_datatype	target_datatype	datatype of each entry in target buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
win	win	window object used for communication	MPI_Win	TYPE(MPI_Win)	IN
)					

Python:

```
win.Get(self, origin, int target_rank, target=None)
```

Prior to Fortran2008, specifying a literal constant, such as 0, could lead to bizarre runtime errors; the solution was to specify a zero-valued variable of the right type. With the `mpi_f08` module this is no longer allowed. Instead you get an error such as

error #6285: There is no matching specific subroutine for this generic subroutine call. [MPI]

Python note 29: MPI one-sided transfer routines. `MPI_Put` (and Get and Accumulate) accept at minimum the origin buffer and the target rank. The displacement is by default zero.

Exercise 9.1. Revisit exercise 4.3 and solve it using `MPI_Put`.

(There is a skeleton for this exercise under the name `rightput`.)

Exercise 9.2. Write code where:

- process 0 computes a random number r
- if $r < .5$, zero writes in the window on 1;
- if $r \geq .5$, zero writes in the window on 2.

(There is a skeleton for this exercise under the name `randomput`.)

9.3.2 Get

The `MPI_Get` (figure 9.7) call is very similar.

Example:

```
MPI_Win_fence(0,the_window);
if (procno==0) {
    MPI_Get( /* data on origin: */ &my_number, 1,MPI_INT,
             /* data on target: */ other,1,      1,MPI_INT,
             the_window);
}
MPI_Win_fence(0,the_window);
```

We make a null window on processes that do not participate.

```
## getfence.py
if procid==0 or procid==nprocs-1:
    win_mem = np.empty( 1,dtype=np.float64 )
    win = MPI.Win.Create( win_mem,comm=comm )
else:
    win = MPI.Win.Create( None,comm=comm )

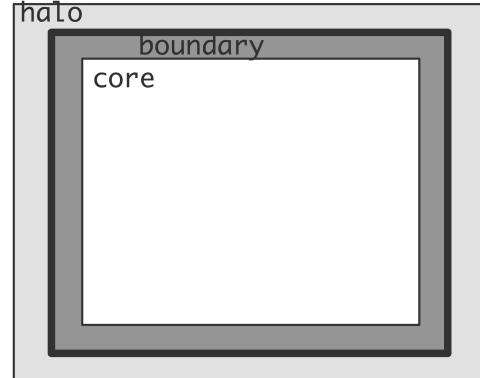
# put data on another process
win.Fence()
if procid==0 or procid==nprocs-1:
    putdata = np.empty( 1,dtype=np.float64 )
    putdata[0] = mydata
    print(" [%d] putting %e" % (procid,mydata))
    win.Put( putdata,other )
win.Fence()
```

9.3.3 Put and get example: halo update

As an example, let's look at *halo update*. The array A is updated using the local values and the halo that comes from bordering processors, either through Put or Get operations.

In a first version we separate computation and communication. Each iteration has two fences. Between the two fences in the loop body we do the `MPI_Put` operation; between the second and and first one of the next iteration there is only computation, so we add the `MPI_MODE_NOPRECEDE` and `MPI_MODE_NOSUCCEED` assertions. The `MPI_MODE_NOSTORE` assertion states that the local window was not updated: the Put operation only works on remote windows.

```
for ( .... ) {
    update(A);
    MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
    for(i=0; i < tneighbors; i++)
        MPI_Put( ... );
```



```
MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
}
```

For much more about assertions, see section 9.6 below.

Next, we split the update in the core part, which can be done purely from local values, and the boundary, which needs local and halo values. Update of the core can overlap the communication of the halo.

```
for ( .... ) {
    update_boundary(A);
    MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
    for(i=0; i < fromneighbors; i++)
        MPI_Get( ... );
    update_core(A);
    MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
}
```

The `MPI_MODE_NOPRECEDE` and `MPI_MODE_NOSUCCEED` assertions still hold, but the `Get` operation implies that instead of `MPI_MODE_NOSTORE` in the second fence, we use `MPI_MODE_NOPUT` in the first.

9.3.4 Accumulate

A third one-sided routine is `MPI_Accumulate` (figure 9.8) which does a reduction operation on the results that are being put.

Accumulate is an atomic reduction with remote result. This means that multiple accumulates to a single target in the same epoch give the correct result. As with `MPI_Reduce`, the order in which the operands are accumulated is undefined.

The same predefined operators are available, but no user-defined ones. There is one extra operator: `MPI_REPLACE`, this has the effect that only the last result to arrive is retained.

Exercise 9.3. Implement an ‘all-gather’ operation using one-sided communication: each processor stores a single number, and you want each processor to build up an array that contains the values from all processors. Note that you do not need a special case for a processor collecting its own value: doing ‘communication’ between a processor and itself is perfectly legal.

For the next exercise, refer to figure 9.5.

Exercise 9.4.

Implement a shared counter:

- One process maintains a counter;
- Iterate: all others at random moments update this counter.
- When the counter is no longer positive, everyone stops iterating.

The problem here is data synchronization: does everyone see the counter the same way?

Figure 9.8 MPI_Accumulate

Name	Param name	Explanation	C type	F type	inout
MPI_Accumulate (
MPI_Accumulate_c (
origin_addr		initial address of buffer	const void*	TYPE(*), DIMENSION(..)	IN
origin_count		number of entries in buffer	[int MPI_Count]	INTEGER	IN
origin_datatype		datatype of each entry	MPI_Datatype	TYPE (MPI_Datatype)	IN
target_rank		rank of target	int	INTEGER	IN
target_disp		displacement from start of window to beginning of target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
target_count		number of entries in target buffer	[int MPI_Count]	INTEGER	IN
target_datatype		datatype of each entry in target buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
op		reduce operation	MPI_Op	TYPE(MPI_Op)	IN
win		window object	MPI_Win	TYPE(MPI_Win)	IN
)					

Python:

```
MPI.Win.Accumulate(self, origin, int target_rank, target=None, Op op=SUM)
```

9.3.5 Ordering and coherence of RMA operations

There are few guarantees about what happens inside one epoch.

- No ordering of Get and Put/Accumulate operations: if you do both, there is no guarantee whether the Get will find the value before or after the update.
- No ordering of multiple Puts. It is safer to do an Accumulate.

The following operations are well-defined inside one epoch:

- Instead of multiple Put operations, use Accumulate with `MPI_REPLACE`.
- `MPI_Get_accumulate` with `MPI_NO_OP` is safe.
- Multiple Accumulate operations from one origin are done in program order by default. To allow reordering, for instance to have all reads happen after all writes, use the info parameter when the window is created; section 9.5.4.

9.3.6 Request-based operations

Analogous to `MPI_Isend` there are request-based one-sided operations: `MPI_Rput` (figure 9.9) and similarly `MPI_Rget` and `MPI_Raccumulate` and `MPI_Rget_accumulate`. These only apply to passive target synchronization. Any `MPI_Win_flush...` call also terminates these transfers.

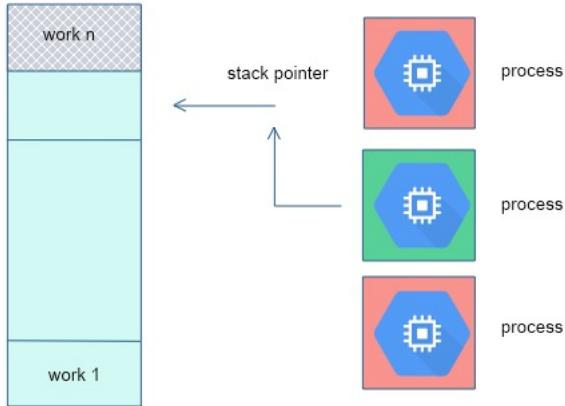


Figure 9.5: Pool of work descriptors with shared stack pointers

9.3.7 Atomic operations

One-sided calls are said to emulate shared memory in MPI, but the put and get calls are not enough for certain scenarios with shared data. Consider the scenario where:

- One process stores a table of work descriptors, and a pointer to the first unprocessed descriptor;
- Each process reads the pointer, reads the corresponding descriptor, and increments the pointer; and
- A process that has read a descriptor then executes the corresponding task.

The problem is that reading and updating the pointer is not an *atomic operation*, so it is possible that multiple processes get hold of the same value; conversely, multiple updates of the pointer may lead to work descriptors being skipped. These different overall behaviors, depending on precise timing of lower level events, are called a *race condition*.

In MPI-3 some atomic routines have been added. Both `MPI_Fetch_and_op` (figure 9.10) and `MPI_Get_accumulate` (figure 9.11) atomically retrieve data from the window indicated, and apply an operator, combining the data on the target with the data on the origin. Unlike Put and Get, it is safe to have multiple atomic operations in the same epoch.

Both routines perform the same operations: return data before the operation, then atomically update data on the target, but `MPI_Get_accumulate` is more flexible in data type handling. The more simple routine, `MPI_Fetch_and_op`, which operates on only a single element, allows for faster implementations, in particular through hardware support.

Use of `MPI_NO_OP` as the `MPI_Op` turns these routines into an atomic Get. Similarly, using `MPI_REPLACE` turns them into an atomic Put.

Exercise 9.5. Redo exercise 9.4 using `MPI_Fetch_and_op`. The problem is again to make sure all processes have the same view of the shared counter.

Does it work to make the fetch-and-op conditional? Is there a way to do it unconditionally? What should the ‘break’ test be, seeing that multiple processes can update the counter at the same time?

Figure 9.9 MPI_Rput

Name	Param name	Explanation	C type	F type	inout
MPI_Rput (
MPI_Rput_c (
origin_addr	origin_addr	initial address of origin buffer	const void*	TYPE(*), DIMENSION(..)	IN
origin_count	origin_count	number of entries in origin buffer	[int MPI_Count	INTEGER	IN
origin_datatype	origin_datatype	datatype of each entry in origin buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
target_rank	target_rank	rank of target	int	INTEGER	IN
target_disp	target_disp	displacement from start of window to target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
target_count	target_count	number of entries in target buffer	[int MPI_Count	INTEGER	IN
target_datatype	target_datatype	datatype of each entry in target buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
win	win	window object used for communication	MPI_Win	TYPE(MPI_Win)	IN
request	request	RMA request	MPI_Request*	TYPE (MPI_Request)	OUT
)					

Figure 9.10 MPI_Fetch_and_op

Name	Param name	Explanation	C type	F type	inout
MPI_Fetch_and_op (
origin_addr	origin_addr	initial address of buffer	const void*	TYPE(*), DIMENSION(..)	IN
result_addr	result_addr	initial address of result buffer	void*	TYPE(*), DIMENSION(..)	OUT
datatype	datatype	datatype of the entry in origin, result, and target buffers	MPI_Datatype	TYPE (MPI_Datatype)	IN
target_rank	target_rank	rank of target	int	INTEGER	IN
target_disp	target_disp	displacement from start of window to beginning of target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
op	op	reduce operation	MPI_Op	TYPE(MPI_Op)	IN
win	win	window object	MPI_Win	TYPE(MPI_Win)	IN
)					

Figure 9.11 MPI_Get_accumulate

Name	Param name	Explanation	C type	F type	inout
MPI_Get_accumulate (
MPI_Get_accumulate_c (
origin_addr	initial address of buffer	const void*	TYPE(*), DIMENSION(..)	IN	
origin_count	number of entries in origin buffer	[int MPI_Count]	INTEGER	IN	
origin_datatype	datatype of each entry in origin buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN	
result_addr	initial address of result buffer	void*	TYPE(*), DIMENSION(..)	OUT	
result_count	number of entries in result buffer	[int MPI_Count]	INTEGER	IN	
result_datatype	datatype of each entry in result buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN	
target_rank	rank of target	int	INTEGER	IN	
target_disp	displacement from start of window to beginning of target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN	
target_count	number of entries in target buffer	[int MPI_Count]	INTEGER	IN	
target_datatype	datatype of each entry in target buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN	
op	reduce operation	MPI_Op	TYPE(MPI_Op)	IN	
win	window object	MPI_Win	TYPE(MPI_Win)	IN	
)					

Example. A root process has a table of data; the other processes do atomic gets and update of that data using *passive target synchronization* through [MPI_Win_lock](#).

```
// passive.cxx
if (procno==repository) {
// Repository processor creates a table of inputs
// and associates that with the window
}
if (procno!=repository) {
float contribution=(float)procno,table_element;
int loc=0;
MPI_Win_lock(MPI_LOCK_EXCLUSIVE,repository,0,the_window);
// read the table element by getting the result from adding zero
MPI_Fetch_and_op
(&contribution,&table_element,MPI_FLOAT,
repository,loc,MPI_SUM,the_window);
MPI_Win_unlock(repository,the_window);
```

Figure 9.12 MPI_Compare_and_swap

Name	Param name	Explanation	C type	F type	inout
MPI_Compare_and_swap	(
	origin_addr	initial address of buffer	const void*	TYPE(*), DIMENSION(..)	IN
	compare_addr	initial address of compare buffer	const void*	TYPE(*), DIMENSION(..)	IN
	result_addr	initial address of result buffer	void*	TYPE(*), DIMENSION(..)	OUT
	datatype	datatype of the element in all buffers	MPI_Datatype	TYPE (MPI_Datatype)	IN
	target_rank	rank of target	int	INTEGER	IN
	target_disp	displacement from start of window to beginning of target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
	win	window object	MPI_Win	TYPE(MPI_Win)	IN
)				

}

```
## passive.py
if procid==repository:
    # repository process creates a table of inputs
    # and associates it with the window
    win_mem = np.empty( ninputs,dtype=np.float32 )
    win = MPI.Win.Create( win_mem,comm=comm )
else:
    # everyone else has an empty window
    win = MPI.Win.Create( None,comm=comm )
if procid!=repository:
    contribution = np.empty( 1,dtype=np.float32 )
    contribution[0] = 1.*procid
    table_element = np.empty( 1,dtype=np.float32 )
    win.Lock( repository,lock_type=MPI.LOCK_EXCLUSIVE )
    win.Fetch_and_op( contribution,table_element,repository,0,MPI.SUM)
    win.Unlock( repository )
```

Finally, `MPI_Compare_and_swap` (figure 9.12) swaps the origin and target data if the target data equals some comparison value.

9.3.7.1 A case study in atomic operations

Let us consider an example where a process, identified by `counter_process`, has a table of work descriptors, and all processes, including the counter process, take items from it to work on. To avoid duplicate work, the counter process has as counter that indicates the highest numbered available item. The part of this application that we simulate is this:

1. a process reads the counter, to find an available work item; and
2. subsequently decrements the counter by one.

We initialize the window content, under the separate memory model:

```
// countdownnop.c
MPI_Win_fence(0,the_window);
if (procno==counter_process)
    MPI_Put(&counter_init,1,MPI_INT,
            counter_process,0,1,MPI_INT,
            the_window);
MPI_Win_fence(0,the_window);
```

We start by considering the naive approach, where we execute the above scheme literally with `MPI_Get` and `MPI_Put`:

```
// countdownput.c
MPI_Win_fence(0,the_window);
int counter_value;
MPI_Get( &counter_value,1,MPI_INT,
        counter_process,0,1,MPI_INT,
        the_window);
MPI_Win_fence(0,the_window);
if (i_am_available) {
    int decrement = -1;
    counter_value += decrement;
    MPI_Put
    ( &counter_value, 1,MPI_INT,
      counter_process,0,1,MPI_INT,
      the_window);
}
MPI_Win_fence(0,the_window);
```

This scheme is correct if only process has a true value for `i_am_available`: that processes ‘owns’ the current counter values, and it correctly updates the counter through the `MPI_Put` operation. However, if more than one process is available, they get duplicate counter values, and the update is also incorrect. If we run this program, we see that the counter did not get decremented by the total number of ‘put’ calls.

Exercise 9.6. Supposing only one process is available, what is the function of the middle of the three fences? Can it be omitted?

We can fix the decrement of the counter by using `MPI_Accumulate` for the counter update, since it is atomic: multiple updates in the same epoch all get processed.

```
// countdownacc.c
MPI_Win_fence(0,the_window);
int counter_value;
MPI_Get( &counter_value,1,MPI_INT,
        counter_process,0,1,MPI_INT,
        the_window);
MPI_Win_fence(0,the_window);
if (i_am_available) {
    int decrement = -1;
```

```

MPI_Accumulate
( &decrement,      1,MPI_INT,
  counter_process,0,1,MPI_INT,
  MPI_SUM,
  the_window);
}

MPI_Win_fence(0,the_window);

```

This scheme still suffers from the problem that processes will obtain duplicate counter values. The true solution is to combine the ‘get’ and ‘put’ operations into one atomic action; in this case **MPI_Fetch_and_op**:

```

MPI_Win_fence(0,the_window);
int
counter_value;
if (i_am_available) {
    int
    decrement = -1;
    total_decrement++;
    MPI_Fetch_and_op
    ( /* operate with data from origin: */ &decrement,
      /* retrieve data from target: */ &counter_value,
      MPI_INT, counter_process, 0, MPI_SUM,
      the_window);
}
MPI_Win_fence(0,the_window);
if (i_am_available) {
    my_counter_values[n_my_counter_values++] = counter_value;
}

```

Now, if there are multiple accesses, each retrieves the counter value and updates it in one atomic, that is, indivisible, action.

9.4 Passive target synchronization

In *passive target synchronization* only the origin is actively involved: the target makes no synchronization calls. This means that the origin process remotely locks the window on the target, performs a one-sided transfer, and releases the window by unlocking it again.

During an access epoch, also called an *passive target epoch* in this case (the concept of ‘exposure epoch’ makes no sense with passive target synchronization), a process can initiate and finish a one-sided transfer. Typically it will lock the window with **MPI_Win_lock** (figure 9.13):

```

if (rank == 0) {
    MPI_Win_lock (MPI_LOCK_EXCLUSIVE, 1, 0, win);
    MPI_Put (outbuf, n, MPI_INT, 1, 0, n, MPI_INT, win);
    MPI_Win_unlock (1, win);
}

```

Remark The possibility to lock a window is not guaranteed for windows that are not created (possibly internally) by **MPI_Alloc_mem**, that is, all but **MPI_Win_create**.

Figure 9.13 MPI_Win_lock

Name	Param name	Explanation	C type	F type	inout
MPI_Win_lock (
lock_type	either MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED		int	INTEGER	IN
rank	rank of locked window		int	INTEGER	IN
assert	program assertion	int	INTEGER	IN	
win	window object	MPI_Win	TYPE(MPI_Win)	IN	
)					

Python:

```
MPI.Win.Lock(self,  
            int rank, int lock_type=LOCK_EXCLUSIVE, int assertion=0)
```

Figure 9.14 MPI_Win_unlock

Name	Param name	Explanation	C type	F type	inout
MPI_Win_unlock (
rank	rank of window	int	INTEGER	IN	
win	window object	MPI_Win	TYPE(MPI_Win)	IN	
)					

9.4.1 Lock types

A lock is needed to start an *access epoch*, that is, for an origin to acquire the capability to access a target. You can either acquire a lock on a specific process with [MPI_Win_lock](#), or on all processes (in a communicator) with [MPI_Win_lock_all](#). Unlike [MPI_Win_fence](#), this is not a collective call. Also, it is possible to have multiple access epochs through [MPI_Win_lock](#) active simultaneously.

The two lock types are:

- [MPI_LOCK_SHARED](#): multiple processes can access the window on the same rank. If multiple processes perform a [MPI_Get](#) call there is no problem; with [MPI_Put](#) and similar calls there is a consistency problem; see below.
- [MPI_LOCK_EXCLUSIVE](#): an origin gets exclusive access to the window on a certain target. Unlike the shared lock, this has no consistency problems.

You can only specify a lock type in [MPI_Win_lock](#); [MPI_Win_lock_all](#) is always shared.

To unlock a window, use [MPI_Win_unlock](#) (figure 9.14), respectively [MPI_Win_unlock_all](#).

Exercise 9.7. Investigate atomic updates using passive target synchronization. Use [MPI_Win_lock](#) with an exclusive lock, which means that each process only acquires the lock when it absolutely has to.

- All processes but one update a window:

```
int one=1;  
MPI_Fetch_and_op(&one, &readout,
```

Figure 9.15 MPI_Win_lock_all

Name	Param name	Explanation	C type	F type	inout
MPI_Win_lock_all (

```
    assert      program assertion      int      INTEGER      IN
    win        window object        MPI_Win    TYPE(MPI_Win)  IN
)
```

```
    MPI_INT, repo, zero_disp, MPI_SUM,
    the_win);
```

- while the remaining process spins until the others have performed their update.

Use an atomic operation for the latter process to read out the shared value.

Can you replace the exclusive lock with a shared one?

(There is a skeleton for this exercise under the name `lockfetch`.)

Exercise 9.8. As exercise 9.7, but now use a shared lock: all processes acquire the lock simultaneously and keep it as long as is needed.

The problem here is that coherence between window buffers and local variables is now not forced by a fence or releasing a lock. Use `MPI_Win_flush_local` to force coherence of a window (on another process) and the local variable from

`MPI_Fetch_and_op`.

(There is a skeleton for this exercise under the name `lockfetchshared`.)

9.4.2 Lock all

To lock the windows of all processes in the group of the windows, use `MPI_Win_lock_all` (figure 9.15). This is not a collective call: the ‘all’ part refers to the fact that one process is locking the window on all processes.

- The assertion value can be zero, or `MPI_MODE_NOCHECK`, which asserts that no other process will acquire a competing lock.
- There is no ‘locktype’ parameter: this is a shared lock.

The corresponding unlock is `MPI_Win_unlock_all`.

The expected use of a ‘lock/unlock all’ is that they surround an extended epoch with get/put and flush calls.

9.4.3 Completion and consistency in passive target synchronization

In one-sided transfer one should keep straight the multiple instances of the data, and the various *completions* that effect their *consistency*.

- The user data. This is the buffer that is passed to an RMA call. For instance, after an `MPI_Put` call, but still in an access epoch, the user buffer is not safe to reuse. Making sure the buffer has been transferred is called *local completion*.

Figure 9.16 MPI_Win_flush_local

Name	Param name	Explanation	C type	F type	inout
MPI_Win_flush_local					
rank	rank	rank of target window	int	INTEGER	IN
win	win	window object	MPI_Win	TYPE(MPI_Win)	IN

- The window data. While this may be publicly accessible, it is not necessarily always consistent with internal copies.
- The remote data. Even a successful `MPI_Put` does not guarantee that the other process has received the data. A successful transfer is a *remote completion*.

As observed, RMA operations are nonblocking, so we need mechanisms to ensure that an operation is completed, and to ensure *consistency* of the user and window data.

Completion of the RMA operations in a passive target epoch is ensured with `MPI_Win_unlock` or `MPI_Win_unlock_all`, similar to the use of `MPI_Win_fence` in active target synchronization.

If the passive target epoch is of greater duration, and no unlock operation is used to ensure completion, the following calls are available.

Remark *Using flush routines with active target synchronization (or generally outside a passive target epoch) you are likely to get a message*

Wrong synchronization of RMA calls

9.4.3.1 Local completion

The call `MPI_Win_flush_local` (figure 9.16) ensure that all operations with a given target is completed at the origin. For instance, for calls to `MPI_Get` or `MPI_Fetch_and_op` the local result is available after the `MPI_Win_flush_local`.

With `MPI_Win_flush_local_all` local operations are concluded for all targets. This will typically be used with `MPI_Win_lock_all` (section 9.4.2).

9.4.3.2 Remote completion

The calls `MPI_Win_flush` (figure 9.17) and `MPI_Win_flush_all` effect completion of all outstanding RMA operations on the target, so that other processes can access its data. This is useful for `MPI_Put` operations, but can also be used for atomic operations such as `MPI_Fetch_and_op`.

9.4.3.3 Window synchronization

Under the *separate memory model*, the user code can hold a buffer that is not coherent with the internal window data. The call `MPI_Win_sync` synchronizes private and public copies of the window.

Figure 9.17 MPI_Win_flush

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Win_flush (</code>					
<code>rank</code>		rank of target window	int	INTEGER	IN
<code>win</code>		window object	<code>MPI_Win</code>	<code>TYPE(MPI_Win)</code>	IN
<code>)</code>					

9.5 More about window memory

9.5.1 Memory models

You may think that the window memory is the same as the buffer you pass to `MPI_Win_create` or that you get from `MPI_Win_allocate` (section 9.1.1). This is not necessarily true, and the actual state of affairs is called the *memory model*. There are two memory models:

- Under the *unified* memory model, the buffer in process space is indeed the window memory, or at least they are kept *coherent*. This means that after *completion* of an epoch you can read the window contents from the buffer. To get this, the window needs to be created with `MPI_Win_allocate_shared`. This memory model is required for MPI shared memory; chapter 12.
- Under the *separate* memory model, the buffer in process space is the *private window* and the target of put/get operations is the *public window* and the two are not the same and are not kept coherent. Under this model, you need to do an explicit get to read the window contents.

You can query the model of a window using the `MPI_Win_get_attr` call with the `MPI_WIN_MODEL` keyword:

```
// window.c
int *modelstar,flag;
MPI_Win_get_attr(the_window,MPI_WIN_MODEL,&modelstar,&flag);
int model = *modelstar;
if (procno==0)
    printf("Window model is unified: %d\n",model==MPI_WIN_UNIFIED);
```

with possible values:

- `MPI_WIN_SEPARATE`,
- `MPI_WIN_UNIFIED`,

For more on attributes, see section 9.5.5.

The following material is for the recently released MPI-4 standard and may not be supported yet.

9.5.2 Allocation kinds

It may be desirable to indicate the type of memory returned by `MPI_Win_allocate` or `MPI_Alloc_mem`, especially if *accelerators* are involved. This functionality was added in MPI-4.1.

Two info keys in MPI-4.1 are used to query/request support, or assert usage, of memory allocation kinds:

- `mpi_memory_alloc_kinds`: the value of this a comma-separated list of memory allocation kinds supported by the MPI implementation. It can be used in the input `MPI_Info` object of

Figure 9.18 MPI_Win_create_dynamic

Name	Param name	Explanation	C type	F type	inout
MPI_Win_create_dynamic (
info	info	argument	MPI_Info	TYPE (MPI_Info)	IN
comm	comm	intra-communicator	MPI_Comm	TYPE (MPI_Comm)	IN
win	win	window object returned by the call	MPI_Win*	TYPE(MPI_Win)	OUT
)					

[MPI_Session_init](#), [MPI_Comm_spawn](#), or [MPI_Comm_spawn_multiple](#), to request support for certain allocation kinds. If used as a query, it reports the allocation kinds supported in the given session, et cetera. This may include kinds not requested by the user.

- [mpi_assert_memory_alloc_kinds](#): by setting this, the user tells MPI that all buffers (on the given communicator, session, et cetera) use only memory of the indicated type.

Some info values for the allocation kinds are predefined:

- [mpi](#): Memory allocated by the MPI library.
- [system](#): Memory returned by standard system allocators.
- [default](#): Memory from one of the supported allocators.

Info values can have restrictors:

kind1:restrict1,kind2:restrict2

Restrict values are:

- [alloc_mem](#)
- [win_allocate](#)
- [win_allocate_shared](#)

End of MPI-4 material

9.5.3 Dynamically attached memory

In section 9.1.1 we looked at simple ways to create a window and its memory.

It is also possible to have windows where the size is dynamically set. Create a dynamic window with [MPI_Win_create_dynamic](#) (figure 9.18) and attach memory to the window with [MPI_Win_attach](#) (figure 9.19).

At first sight, the code looks like splitting up a [MPI_Win_create](#) call into separate creation of the window and declaration of the buffer:

```
// windynamic.c
MPI_Win_create_dynamic(MPI_INFO_NULL,comm,&the_window);
if (procno==data_proc)
    window_buffer = (int*) malloc( 2*sizeof(int) );
MPI_Win_attach(the_window,window_buffer,2*sizeof(int));
```

Figure 9.19 MPI_Win_attach

Name	Param name	Explanation	C type	F type	inout
MPI_Win_attach					
	win	window object	MPI_Win	TYPE(MPI_Win)	IN
	base	initial address of memory to be attached	void*	TYPE(*), DIMENSION(..)	IN
	size	size of memory to be attached in bytes	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	
)				

Figure 9.20 MPI_Win_detach

Name	Param name	Explanation	C type	F type	inout
MPI_Win_detach					
	win	window object	MPI_Win	TYPE(MPI_Win)	IN
	base	initial address of memory to be detached	const void*	TYPE(*), DIMENSION(..)	IN
)				

(where the `window_buffer` represents memory that has been allocated.)

However, there is an important difference in how the window is addressed in RMA operations. With all other window models, the displacement parameter is measured relative in units from the start of the buffer, here the displacement is an absolute address. This means that we need to get the address of the window buffer with `MPI_Get_address` and communicate it to the other processes:

```
MPI_Aint data_address;
if (procno==data_proc) {
    MPI_Get_address(window_buffer,&data_address);
}
MPI_Bcast(&data_address,1,MPI_AINT,data_proc,comm);
```

Location of the data, that is, the displacement parameter, is then given as an absolute location of the start of the buffer plus a count in bytes; in other words, the *displacement unit* is 1. In this example we use `MPI_Get` to find the second integer in a window buffer:

```
MPI_Aint disp = data_address+1*sizeof(int);
MPI_Get( /* data on origin: */           retrieve, 1,MPI_INT,
/* data on target: */ data_proc,disp,      1,MPI_INT,
the_window);
```

Notes.

- The attached memory can be released with `MPI_Win_detach` (figure 9.20).

- The above fragments show that an origin process has the actual address of the window buffer. It is an error to use this if the buffer is not attached to a window.
- In particular, one has to make sure that the attach call is concluded before performing RMA operations on the window.

9.5.4 Window usage hints

The following keys can be passed as info argument:

- `no_locks`: if set to true, passive target synchronization (section 9.4) will not be used on this window.
- `accumulate_ordering`: a comma-separated list of the keywords `rar`, `raw`, `war`, `waw` can be specified. This indicates that reads or writes from `MPI_Accumulate` or `MPI_Get_accumulate` can be reordered, subject to certain constraints.
- `accumulate_ops`: the value `same_op` indicates that *concurrent Accumulate* calls use the same operator; `same_op_no_op` indicates the same operator or `MPI_NO_OP`.

9.5.5 Window information

The `MPI_Info` parameter (see section 15.1.1 for info objects) can be used to pass implementation-dependent information.

A number of attributes are stored with a window when it is created.

- `MPI_WIN_BASE` for obtaining a pointer to the start of the window area:

```
void *base;
MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag)
```

- `MPI_WIN_SIZE` and `MPI_WIN_DISP_UNIT` for obtaining the size and *window displacement unit*:

```
MPI_Aint *size;
MPI_Win_get_attr(win, MPI_WIN_SIZE, &size, &flag),
int *disp_unit;
MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag),
```

- `MPI_WIN_CREATE_FLAVOR` for determining the type of create call used:

```
int *create_kind;
MPI_Win_get_attr(win, MPI_WIN_CREATE_FLAVOR, &create_kind, &flag)
```

with possible values:

- `MPI_WIN_FLAVOR_CREATE` if the window was create with `MPI_Win_create`;
- `MPI_WIN_FLAVOR_ALLOCATE` if the window was create with `MPI_Win_allocate`;
- `MPI_WIN_FLAVOR_DYNAMIC` if the window was create with `MPI_Win_create_dynamic`. In this case the base is `MPI_BOTTOM` and the size is zero;
- `MPI_WIN_FLAVOR_SHARED` if the window was create with `MPI_Win_allocate_shared`;
- `MPI_WIN_MODEL` for querying the window memory model; see section 9.5.1.

Get the group of processes (see section 7.5) associated with a window:

```
int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
```

Window information objects (see section 15.1.1) can be set and retrieved:

```
int MPI_Win_set_info(MPI_Win win, MPI_Info info)

int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)
```

9.6 Assertions

The routines

- (Active target synchronization) `MPI_Win_fence`, `MPI_Win_post`, `MPI_Win_start`;
- (Passive target synchronization) `MPI_Win_lock`, `MPI_Win_lockall`,

take an argument through which assertions can be passed about the activity before, after, and during the epoch. The value zero is always allowed, by you can make your program more efficient by specifying one or more of the following, combined by bitwise OR in C/C++ or IOR in Fortran.

- `MPI_Win_start` Supports the option:
 - `MPI_MODE_NOCHECK` the matching calls to `MPI_Win_post` have already completed on all target processes when the call to `MPI_Win_start` is made. The nocheck option can be specified in a start call if and only if it is specified in each matching post call. This is similar to the optimization of “ready-send” that may save a handshake when the handshake is implicit in the code. (However, ready-send is matched by a regular receive, whereas both start and post must specify the nocheck option.)
- `MPI_Win_post` supports the following options:
 - `MPI_MODE_NOCHECK` the matching calls to `MPI_Win_start` have not yet occurred on any origin processes when the call to `MPI_Win_post` is made. The nocheck option can be specified by a post call if and only if it is specified by each matching start call.
 - `MPI_MODE_NOSTORE` the local window was not updated by local stores (or local get or receive calls) since last synchronization. This may avoid the need for cache synchronization at the post call.
 - `MPI_MODE_NOPUT` the local window will not be updated by put or accumulate calls after the post call, until the ensuing (wait) synchronization. This may avoid the need for cache synchronization at the wait call.
- `MPI_Win_fence` supports the following options:
 - `MPI_MODE_NOSTORE` the local window was not updated by local stores (or local get or receive calls) since last synchronization.
 - `MPI_MODE_NOPUT` the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.
 - `MPI_MODE_NOPRECEDE` the fence does not complete any sequence of locally issued RMA calls. If this assertion is given by any process in the window group, then it must be given by all processes in the group.
 - `MPI_MODE_NOSUCCEED` the fence does not start any sequence of locally issued RMA calls. If the assertion is given by any process in the window group, then it must be given by all processes in the group.

- `MPI_Win_lock` and `MPI_Win_lock_all` support the following option:
 - `MPI_MODE_NOCHECK` no other process holds, or will attempt to acquire a conflicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.

9.7 Implementation

You may wonder how one-sided communication is realized¹. Can a processor somehow get at another processor's data? Unfortunately, no.

Active target synchronization is implemented in terms of two-sided communication. Imagine that the first fence operation does nothing, unless it concludes prior one-sided operations. The Put and Get calls do nothing involving communication, except for marking with what processors they exchange data. The concluding fence is where everything happens: first a global operation determines which targets need to issue send or receive calls, then the actual sends and receive are executed.

Exercise 9.9. Assume that only Get operations are performed during an epoch. Sketch how these are translated to send/receive pairs. The problem here is how the senders find out that they need to send. Show that you can solve this with an `MPI_Reduce_scatter` call.

The previous paragraph noted that a collective operation was necessary to determine the two-sided traffic. Since collective operations induce some amount of synchronization, you may want to limit this.

Exercise 9.10. Argue that the mechanism with window post/wait/start/complete operations still needs a collective, but that this is less burdensome.

Passive target synchronization needs another mechanism entirely. Here the target process needs to have a background task (process, thread, daemon,...) running that listens for requests to lock the window. This can potentially be expensive.

1. For more on this subject, see [27].

9.8 Review questions

Find all the errors in this code.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define MASTER 0

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm comm = MPI_COMM_WORLD;
    int r, p;
    MPI_Comm_rank(comm, &r);
    MPI_Comm_size(comm, &p);
    printf("Hello from %d\n", r);
    int result[1] = {0};
    //int assert = MPI_MODE_NOCHECK;
    int assert = 0;
    int one = 1;
    MPI_Win win_res;
    MPI_Win_allocate(1 * sizeof(MPI_INT), sizeof(MPI_INT), MPI_INFO_NULL, comm, &result[0], &win_res
    );
    MPI_Win_lock_all(assert, win_res);
    if (r == MASTER) {
        result[0] = 0;
        do{
            MPI_Fetch_and_op(&result, &result , MPI_INT, r, 0, MPI_NO_OP, win_res);
            printf("result: %d\n", result[0]);
        } while(result[0] != 4);
        printf("Master is done!\n");
    } else {
        MPI_Fetch_and_op(&one, &result, MPI_INT, 0, 0, MPI_SUM, win_res);
    }
    MPI_Win_unlock_all(win_res);
    MPI_Win_free(&win_res);
    MPI_Finalize();
    return 0;
}
```

Chapter 10

MPI topic: File I/O

This chapter discusses the I/O support of MPI, which is intended to alleviate the problems inherent in parallel file access. Let us first explore the issues. This story partly depends on what sort of parallel computer are you running on. Here are some of the hardware scenarios you may encounter:

- On networks of workstations each node will have a separate drive with its own file system.
- On many clusters there will be a *shared file system* that acts as if every process can access every file.
- Cluster nodes may or may not have a private file system.

Based on this, the following strategies are possible, even before we start talking about MPI I/O.

- One process can collect all data with `MPI_Gather` and write it out. There are at least three things wrong with this: it uses network bandwidth for the gather, it may require a large amount of memory on the root process, and centralized writing is a bottleneck.
- Absent a shared file system, writing can be parallelized by letting every process create a unique file and merge these after the run. This makes the I/O symmetric, but collecting all the files is a bottleneck.
- Even with a with a shared file system this approach is possible, but it can put a lot of strain on the file system, and the post-processing can be a significant task.
- Using a shared file system, there is nothing against every process opening the same existing file for reading, and using an individual file pointer to get its unique data.
- ... but having every process open the same file for output is probably not a good idea. For instance, if two processes try to write at the end of the file, you may need to synchronize them, and synchronize the file system flushes.

For these reasons, MPI has a number of routines that make it possible to read and write a single file from a large number of processes, giving each process its own well-defined location where to access the data. These locations can use MPI *derived datatypes* for both the source data (that is, in memory) and target data (that is, on disk). Thus, in one call that is collective on a communicator each process can address data that is not contiguous in memory, and place it in locations that are not contiguous on disc.

There are dedicated libraries for file I/O, such as *hdf5*, *netcdf*, or *silo*. However, these often add header information to a file that may not be understandable to post-processing applications. With MPI I/O you are in complete control of what goes to the file. (A useful tool for viewing your file is the unix utility *od*.)

Figure 10.1 MPI_File_open

Name	Param name	Explanation	C type	F type	inout
<code>MPI_File_open (</code>					
<code>comm</code>	<code>communicator</code>		<code>MPI_Comm</code>	<code>TYPE (MPI_Comm)</code>	<code>IN</code>
<code>filename</code>	<code>name of file to open</code>		<code>const char*</code>	<code>CHARACTER</code>	<code>IN</code>
<code>amode</code>	<code>file access mode</code>		<code>int</code>	<code>INTEGER</code>	<code>IN</code>
<code>info</code>	<code>info object</code>		<code>MPI_Info</code>	<code>TYPE (MPI_Info)</code>	<code>IN</code>
<code>fh</code>	<code>new file handle</code>		<code>MPI_File*</code>	<code>TYPE (MPI_File)</code>	<code>OUT</code>
<code>)</code>					

Python:

```
Open(type cls, IntraComm comm, filename,
     int amode=MODE_RDONLY, Info info=INFO_NULL)
```

TACC note. Each node has a private /tmp file system (typically flash storage), to which you can write files. Considerations:

- Since these drives are separate from the shared file system, you don't have to worry about stress on the file servers.
- These temporary file systems are wiped after your job finishes, so you have to do the post-processing in your job script.
- The capacity of these local drives are fairly limited; see the userguide for exact numbers.

10.1 File handling

MPI has a datatype for files: `MPI_File`. This acts a little like a traditional file handle, in that there are open, close, read/write, and seek operations on it. However, unlike traditional file handling, which in parallel would mean having one handle per process, this handle is collective: MPI processes act as if they share one file handle.

You open a file with `MPI_File_open` (figure 10.1). This routine is collective, even if only certain processes will access the file with a read or write call. Similarly, `MPI_File_close` is collective.

MPL note 66: File opening. Files are objects of type `mpl::file`. You can use the default constructor, and the `open` method, or open the file in the constructor:

```
// filewrite.cxx
mpl::file mpifile
  (comm_world,"filewrite.dat",
   mpl::file::access_mode::create | mpl::file::access_mode::write_only
  );
mpifile.close();
```

- Failure to open throws an `mpl::io_failure` exception; the `what` method of this exception gives an error string. The copy and copy-assignment constructors have been deleted.

Python note 30: File open is class method. Note the slightly unusual syntax for opening a file:

```
mpifile = MPI.File.Open(comm, filename, mode)
```

Even though the file is opened on a communicator, it is a class method for the `MPI.File` class, rather than for the communicator object. The latter is passed in as an argument.

File access modes:

- `MPI_MODE_RDONLY`: read only,
- `MPI_MODE_RDWR`: reading and writing,
- `MPI_MODE_WRONLY`: write only,
- `MPI_MODE_CREATE`: create the file if it does not exist,
- `MPI_MODE_EXCL`: error if creating file that already exists,
- `MPI_MODE_DELETE_ON_CLOSE`: delete file on close,
- `MPI_MODE_UNIQUE_OPEN`: file will not be *concurrently* opened elsewhere,
- `MPI_MODE_SEQUENTIAL`: file will only be accessed sequentially,
- `MPI_MODE_APPEND`: set initial position of all file pointers to end of file.

These modes can be added or bitwise-or'ed.

As a small illustration:

```
Code:  
  
// filewrite.c  
MPI_File mpifile;  
MPI_File_open  
(comm,"filewrite.dat",  
 MPI_MODE_CREATE | MPI_MODE_WRONLY,  
 MPI_INFO_NULL,  
 &mpifile);  
MPI_File_write_at  
(mpifile,/* offset: */ procno*sizeof(  
 int),  
 &procno,1, MPI_INT,MPI_STATUS_IGNORE  
 );  
MPI_File_close(&mpifile);
```

```
Output:  
[examples/mpi/c] write:  
  
Finished: all 4 correct  
octal dump:  
0000000 000000 000000 000001  
    ↳000000 000002 000000  
    ↳000003 000000  
0000020
```

You can delete a file with `MPI_File_delete`.

Buffers can be flushed with `MPI_File_sync`, which is a collective call.

10.2 File reading and writing

The basic file operations, in between the open and close calls, are the POSIX-like, noncollective, calls

Figure 10.2 MPI_File_seek

Name	Param name	Explanation	C type	F type	inout
MPI_File_seek (
fh	file handle		MPI_File	TYPE (MPI_File)	INOUT
offset	file offset		MPI_Offset	INTEGER (KIND=MPI_OFFSET_KIND)	IN
whence	update mode		int	INTEGER	IN
)					

Figure 10.3 MPI_File_write

Name	Param name	Explanation	C type	F type	inout
MPI_File_write (
MPI_File_write_c (
fh	file handle		MPI_File	TYPE (MPI_File)	INOUT
buf	initial address of buffer		const void*	TYPE(*), DIMENSION(..)	IN
count	number of elements in buffer		[int MPI_Count	INTEGER	IN
datatype	datatype of each buffer element		MPI_Datatype	TYPE (MPI_Datatype)	IN
status	status object		MPI_Status*	TYPE (MPI_Status)	OUT
)					

- **MPI_File_seek** (figure 10.2). The *whence* parameter can be:
 - **MPI_SEEK_SET** The pointer is set to offset.
 - **MPI_SEEK_CUR** The pointer is set to the current pointer position plus offset.
 - **MPI_SEEK_END** The pointer is set to the end of the file plus offset.
- **MPI_File_write** (figure 10.3). This routine writes the specified data in the locations specified with the current file view. The number of items written is returned in the **MPI_Status** argument; all other fields of this argument are undefined. It can not be used if the file was opened with **MPI_MODE_SEQUENTIAL**.
 - If all processes execute a write at the same logical time, it is better to use the collective call **MPI_File_write_all**.
 - MPI_File_read** (figure 10.4) This routine attempts to read the specified data from the locations specified in the current file view. The number of items read is returned in the **MPI_Status** argument; all other fields of this argument are undefined. It can not be used if the file was opened with **MPI_MODE_SEQUENTIAL**.
 - If all processes execute a read at the same logical time, it is better to use the collective call **MPI_File_read_all** (figure 10.5).

For thread safety it is good to combine seek and read/write operations:

- **MPI_File_read_at**: combine read and seek. The collective variant is **MPI_File_read_at_all**.

Figure 10.4 MPI_File_read

Name	Param name	Explanation	C type	F type	inout
MPI_File_read (
MPI_File_read_c (
fh	file handle		MPI_File	TYPE (MPI_File)	INOUT
buf	initial address of buffer		void*	TYPE(*), DIMENSION(..)	OUT
count	number of elements in buffer		[int MPI_Count]	INTEGER	IN
datatype	datatype of each buffer element		MPI_Datatype	TYPE (MPI_Datatype)	IN
status	status object		MPI_Status*	TYPE (MPI_Status)	OUT
)					

Figure 10.5 MPI_File_read_all

Name	Param name	Explanation	C type	F type	inout
MPI_File_read_all (
MPI_File_read_all_c (
fh	file handle		MPI_File	TYPE (MPI_File)	INOUT
buf	initial address of buffer		void*	TYPE(*), DIMENSION(..)	OUT
count	number of elements in buffer		[int MPI_Count]	INTEGER	IN
datatype	datatype of each buffer element		MPI_Datatype	TYPE (MPI_Datatype)	IN
status	status object		MPI_Status*	TYPE (MPI_Status)	OUT
)					

- **MPI_File_write_at**: combine write and seek. The collective variant is **MPI_File_write_at_all**; section 10.2.2.

Writing to and reading from a parallel file is rather similar to sending a receiving:

- The process uses an predefined data type or a derived datatype to describe what elements in an array go to file, or are read from file.
- In the simplest case, your read or write that data to the file using an offset, or first having done a seek operation.
- But you can also set a ‘file view’ to describe explicitly what elements in the file will be involved.

MPL note 67: File writing. Routines with the obvious names exist:

```
mpifile.write_at
( /* offset: */ procno*sizeof(int),
/* data: */ procno );
```

Figure 10.6 MPI_File_iwrite

Name	Param name	Explanation	C type	F type	inout
MPI_File_iwrite (
MPI_File_iwrite_c (
fh	file handle		MPI_File	TYPE (MPI_File)	INOUT
buf	initial address of buffer		const void*	TYPE(*), DIMENSION(..)	IN
count	number of elements in buffer	[int MPI_Count		INTEGER	IN
datatype	datatype of each buffer element		MPI_Datatype	TYPE (MPI_Datatype)	IN
request	request object		MPI_Request*	TYPE (MPI_Request)	OUT
)					

Also `read`, `write`, `iread`, `iwrite`, `read_at`, `write_at`, `iread_at`, `iwrite_at`, `read_all`, `write_all`, `iread_all`, `iwrite_all`, `read_ordered`, `write_ordered`, `iread_ordered`, `iwrite_ordered`, `read_shared`, `write_shared`, `iread_shared`, `iwrite_shared`, `read_at_all`, `write_at_all`, `iread_at_all`, `iwrite_at_all`, `read_all_begin`, `write_all_begin`, `read_all_end`, `write_all_end`, `read_at_all_begin`, `write_at_all_begin`, `read_at_all_end`, `write_at_all_end`, `read_ordered_begin`, `write_ordered_begin`, `read_ordered_end`, `write_ordered_end`,

10.2.1 Nonblocking read/write

Just like there are blocking and nonblocking sends, there are also nonblocking writes and reads: `MPI_File_iwrite` (figure 10.6), `MPI_File_iread` operations, and their collective versions `MPI_File_iwrite_all`, `MPI_File_iread_all`.

Also `MPI_File_iwrite_at`, `MPI_File_iwrite_at_all`, `MPI_File_iread_at`, `MPI_File_iread_at_all`.

These routines output an `MPI_Request` object, which can then be tested with `MPI_Wait` or `MPI_Test`.

Nonblocking collective I/O functions much like other nonblocking collectives (section 3.11): the request is satisfied if all processes finish the collective.

There are also *split collectives* that function like nonblocking collective I/O, but with the request/wait mechanism: `MPI_File_write_all_begin` / `MPI_File_write_all_end` (and similarly `MPI_File_read_all_begin` / `MPI_File_read_all_end`) where the second routine blocks until the collective write/read has been concluded.

Also `MPI_File_iread_shared`, `MPI_File_iwrite_shared`.

10.2.2 Individual file pointers, contiguous writes

After the collective open call, each process holds an *individual file pointer* that it can individually position somewhere in the shared file. Let's explore this modality.

Figure 10.7 MPI_File_write_at

Name	Param name	Explanation	C type	F type	inout
MPI_File_write_at (
MPI_File_write_at_c (
fh	file handle	file handle	MPI_File	TYPE (MPI_File)	INOUT
offset	file offset	file offset	MPI_Offset	INTEGER (KIND=MPI_OFFSET_KIND)	IN
buf	initial address of buffer	initial address of buffer	const void*	TYPE(*), DIMENSION(..)	IN
count	number of elements in buffer	number of elements in buffer	[int MPI_Count	INTEGER	IN
datatype	datatype of each buffer element	datatype of each buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
status	status object	status object	MPI_Status*	TYPE (MPI_Status)	OUT
)					

Python:

```
MPI.File.Write_at(self, Offset offset, buf, Status status=None)
```

The simplest way of writing a data to file is much like a send call: a buffer is specified with the usual count/datatype specification, and a target location in the file is given. The routine `MPI_File_write_at` (figure 10.7) gives this location in absolute terms with a parameter of type `MPI_Offset`, which counts bytes.

Figure 10.1: Writing at an offset



Exercise 10.1. Create a buffer of length `nwords=3` on each process, and write these buffers as a sequence to one file with `MPI_File_write_at`.
(There is a skeleton for this exercise under the name `blockwrite`.)

Instead of giving the position in the file explicitly, you can also use a `MPI_File_seek` call to position the file pointer, and write with `MPI_File_write` at the pointer location. The write call itself also *advances the file pointer* so separate calls for writing contiguous elements need no seek calls with `MPI_SEEK_CUR`.

Exercise 10.2. Rewrite the code of exercise 10.1 to use a loop where each iteration writes only one item to file. Note that no explicit advance of the file pointer is needed.

Exercise 10.3. Construct a file with the consecutive integers $0, \dots, WP$ where W some integer, and P the number of processes. Each process p writes the numbers

Figure 10.8 MPI_File_set_view

Name	Param name	Explanation	C type	F type	inout
<code>MPI_File_set_view (</code>					
<code>fh</code>	<code>file handle</code>		<code>MPI_File</code>	<code>TYPE (MPI_File)</code>	<code>INOUT</code>
<code>disp</code>	<code>displacement</code>		<code>MPI_Offset</code>	<code>INTEGER (KIND=MPI_OFFSET_KIND)</code>	<code>IN</code>
<code>etype</code>	<code>elementary datatype</code>		<code>MPI_Datatype</code>	<code>TYPE (MPI_Datatype)</code>	<code>IN</code>
<code>filetype</code>	<code>filetype</code>		<code>MPI_Datatype</code>	<code>TYPE (MPI_Datatype)</code>	<code>IN</code>
<code>datarep</code>	<code>data representation</code>		<code>const char*</code>	<code>CHARACTER</code>	<code>IN</code>
<code>info</code>	<code>info object</code>		<code>MPI_Info</code>	<code>TYPE (MPI_Info)</code>	<code>IN</code>
<code>)</code>					

Python:

```
mpifile = MPI.File.Open( .... )
mpifile.Set_view
    (self,
     Offset disp=0, Datatype etype=None, Datatype filetype=None,
     datarep=None, Info info=INFO_NULL)
```

$p, p + W, p + 2W, \dots$ Use a loop where each iteration

1. writes a single number with `MPI_File_write`, and
2. advanced the file pointer with `MPI_File_seek` with a `whence` parameter of `MPI_SEEK_CUR`.

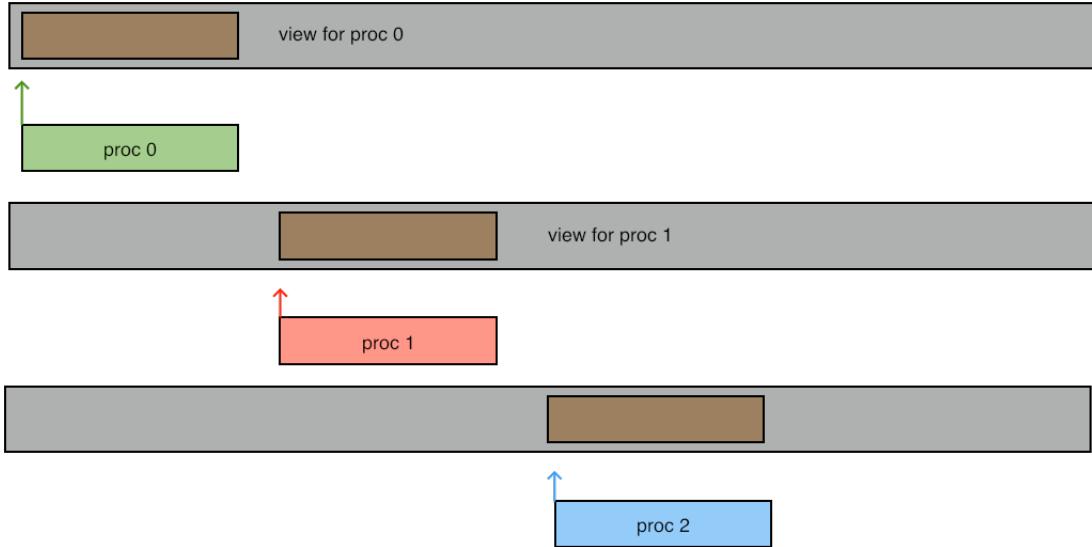
10.2.3 File views

The previous mode of writing is enough for writing simple contiguous blocks in the file. However, you can also access noncontiguous areas in the file. For this you use `MPI_File_set_view` (figure 10.8). This call is collective, even if not all processes access the file.

- The `disp` displacement parameters is measured in bytes. It can differ between processes. On sequential files such as tapes or network streams it does not make sense to set a displacement; for those the `MPI_DISPLACEMENT_CURRENT` value can be used.
- The `etype` describes the data type of the file, it needs to be the same on all processes.
- The `filetype` describes how this process sees the file, so it can differ between processes.
- The `datarep` string can have the following values:
 - `native`: data on disk is represented in exactly the same format as in memory;
 - `internal`: data on disk is represented in whatever internal format is used by the MPI implementation;
 - `external`: data on disk is represented using XDR portable data formats.
- The `info` parameter is an `MPI_Info` object, or `MPI_INFO_NULL`. See section 15.1.1.3 for more on file info. (See `T3PIO` [21] for a tool that assists in setting this object.)

```
// scatterwrite.c
MPI_File_set_view
(mpifile,
 offset,MPI_INT,scattertype,
 "native",MPI_INFO_NULL);
```

Figure 10.2: Writing at a view

**Exercise 10.4.**

(There is a skeleton for this exercise under the name `viewwrite`.) Write a file in the same way as in exercise 10.1, but now use `MPI_File_write` and use `MPI_File_set_view` to set a view that determines where the data is written.

You can get very creative effects by setting the view to a derived datatype.

Fortran note 15: Offset literals. In Fortran you have to assure that the displacement parameter is of ‘kind’ `MPI_OFFSET_KIND`. In particular, you can not specify a literal zero ‘0’ as the displacement; use `0_MPI_OFFSET_KIND` instead.

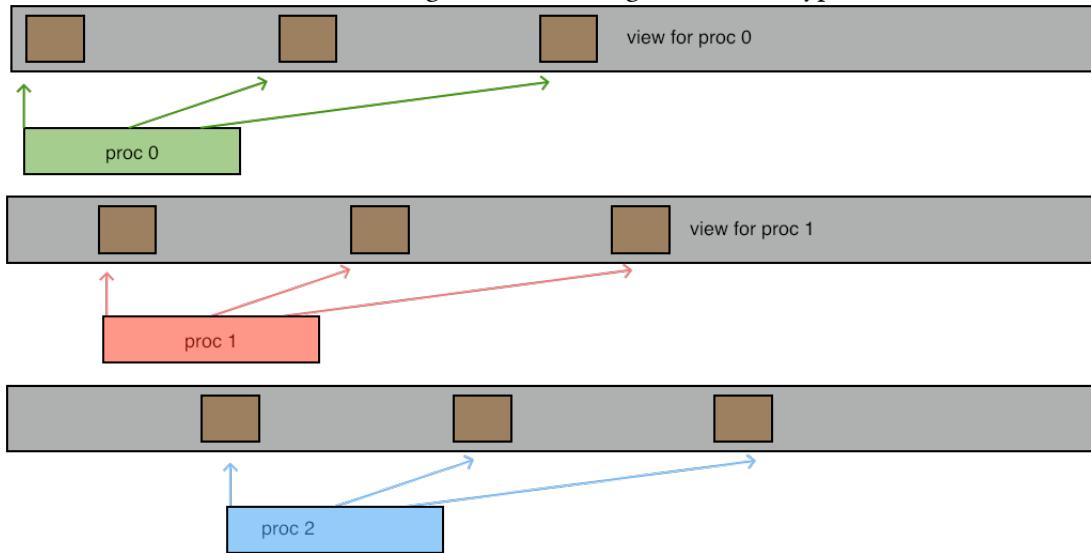
More: `MPI_File_set_size`, `MPI_File_get_size` `MPI_File_pallocate`, `MPI_File_get_view`.

10.2.4 Shared file pointers

It is possible to have a file pointer that is shared (and therefore identical) between all processes of the communicator that was used to open the file. This file pointer is set with `MPI_File_seek_shared`. For reading and writing there are then two sets of routines:

- Individual accesses are done with `MPI_File_read_shared` and `MPI_File_write_shared`. Nonblocking variants are `MPI_File_iread_shared` and `MPI_File_iwrite_shared`.
- Collective accesses are done with `MPI_File_read_ordered` and `MPI_File_write_ordered`, which execute the operations in order ascending by rank.

Figure 10.3: Writing at a derived type



Shared file pointers require that the same view is used on all processes. Also, these operations are less efficient because of the need to maintain the shared pointer.

10.3 Consistency

It is possible for one process to read data previously written by another process. For this, it is of course necessary to impose a temporal order, for instance by using `MPI_Barrier`, or using a zero-byte send from the writing to the reading process.

However, the file also needs to be declared *atomic*: `MPI_File_set_atomicity`.

10.4 Constants

`MPI_SEEK_SET` used to be called `SEEK_SET` which gave conflicts with the C++ library. This had to be circumvented with

```
make CPPFLAGS="-DMPICH_IGNORE_CXX_SEEK -DMPICH_SKIP_MPICXX"
and such.
```

10.5 Error handling

By default, MPI uses `MPI_ERRORS_ARE_FATAL` since parallel errors are almost impossible to recover from. File handling errors, on the other hand, are less serious: if a file is not found, the operation can be abandoned. For this reason, the default error handler for file operations is `MPI_ERRORS_RETURN`.

The default I/O error handler can be queried and set with `MPI_File_get_errhandler` and `MPI_File_set_errhandler` respectively, passing `MPI_FILE_NULL` as argument.

10.6 Review questions

Exercise 10.5. T/F? After your *SLURM* job ends, you can copy from the login node the files you've written to \tmp.

Exercise 10.6. T/F? File views (`MPI_File_set_view`) are intended to

- write MPI derived types to file; without them you can only write contiguous buffers;
- prevent collisions in collective writes; they are not needed for individual writes.

Exercise 10.7. The sequence `MPI_File_seek_shared`, `MPI_File_read_shared` can be replaced by `MPI_File_seek`, `MPI_File_read` if you make what changes?

Chapter 11

MPI topic: Topologies

A communicator describes a group of processes, but the structure of your computation may not be such that every process will communicate with every other process. For instance, in a computation that is mathematically defined on a Cartesian 2D grid, the processes themselves act as if they are two-dimensionally ordered and communicate with N/S/E/W neighbors. If MPI had this knowledge about your application, it could conceivably optimize for it, for instance by renumbering the ranks so that communicating processes are closer together physically in your cluster.

The mechanism to declare this structure of a computation to MPI is known as a *virtual topology*. The following types of topology are defined:

- `MPI_UNDEFINED`: this value holds for communicators where no topology has explicitly been specified.
- `MPI_CART`: this value holds for Cartesian topologies, where processes act as if they are ordered in a multi-dimensional ‘brick’; see section 11.1.
- `MPI_GRAPH`: this value describes the graph topology that was defined in MPI-1; section 11.2.4. It is unnecessarily burdensome, since each process needs to know the total graph, and should therefore be considered obsolete; the type `MPI_DIST_GRAPH` should be used instead.
- `MPI_DIST_GRAPH`: this value describes the distributed graph topology where each process only describes the edges in the process graph that touch itself; see section 11.2.

These values can be discovered with the routine `MPI_Topo_test`.

11.1 Cartesian grid topology

A *Cartesian grid* is a structure, typically in 2 or 3 dimensions, of points that have two neighbors in each of the dimensions. Thus, if a Cartesian grid has sizes $K \times M \times N$, its points have coordinates (k, m, n) with $0 \leq k < K$ et cetera. Most points have six neighbors $(k \pm 1, m, n)$, $(k, m \pm 1, n)$, $(k, m, n \pm 1)$; the exception are the edge points. A grid where edge points are connected through *wraparound connections* is called a *periodic grid*.

MPI has a ‘Cartesian communicator’ construct (that is, a communicator with type `MPI_CART`; see above) for processes to be organized not just linearly through their ranks, but also as if they are organized in a Cartesian grid.

MPL note 68: Cartesian communicator. There is a separate class `cartesian_communicator`.

Additionally, there is a class `dimensions` that describes the shape of the Cartesian grid and its periodicity. The `size(int)` method retrieves the size in the given dimension.

The auxiliary routine `MPI_Dims_create` assists in finding a grid of a given dimension, attempting to minimize the diameter.

```
Code:
// cartdims.c
int *dimensions = (int*) malloc(dim *
    sizeof(int));
for (int idim=0; idim<dim; idim++)
    dimensions[idim] = 0;
MPI_Dims_create(nprocs, dim, dimensions);
```

```
Output:
[examples/mpi/c] cartdims:
mpicc -o cartdims cartdims.o
Cartesian grid size: 3 dim: 1
3
Cartesian grid size: 3 dim: 2
3 x 1
Cartesian grid size: 4 dim: 1
4
Cartesian grid size: 4 dim: 2
2 x 2
Cartesian grid size: 4 dim: 3
2 x 2 x 1
Cartesian grid size: 12 dim: 1
12
Cartesian grid size: 12 dim: 2
4 x 3
Cartesian grid size: 12 dim: 3
3 x 2 x 2
Cartesian grid size: 12 dim: 4
3 x 2 x 2 x 1
```

If the dimensions array is nonzero in a component, that one is not touched. Of course, the product of the specified dimensions has to divide in the input number of nodes.

MPL note 69: Dims create. The `dims_create` routine takes a `dimensions` object with only the dimensionality specified, and creates one with the sizes filled in.

```
mpl::cartesian_communicator::dimensions brick(3);
brick = mpl::dims_create(nprocs, brick);
```

To have certain dimensions be periodic, the initial `dimensions` object needs to be created with periodicity values `periodic` or `non_periodic`.

```
mpl::cartesian_communicator::dimensions pbrick
( { mpl::cartesian_communicator::non_periodic,
    mpl::cartesian_communicator::periodic,
    mpl::cartesian_communicator::non_periodic } );
pbrick = mpl::dims_create(nprocs, pbrick);
```

11.1.1 Cartesian topology communicator

The cartesian topology is specified by giving `MPI_Cart_create` (figure 11.1) the sizes of the processor grid

11. MPI topic: Topologies

Figure 11.1 MPI_Cart_create

Name	Param name	Explanation	C type	F type	inout
MPI_Cart_create (
comm_old	input communicator		MPI_Comm	TYPE (MPI_Comm)	IN
ndims	number of dimensions of Cartesian grid		int	INTEGER	IN
dims	integer array of size ndims specifying the number of processes in each dimension		const int[]	INTEGER (ndims)	IN
periods	logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension		const int[]	LOGICAL (ndims)	IN
reorder	ranking may be reordered (true) or not (false)		int	LOGICAL	IN
comm_cart	communicator with new Cartesian topology		MPI_Comm*	TYPE (MPI_Comm)	OUT
)					

along each axis, and whether the grid is periodic along that axis.

```
MPI_Comm cart_comm;
int *periods = (int*) malloc(dim*sizeof(int));
for ( int id=0; id<dim; id++ ) periods[id] = 0;
MPI_Cart_create
( comm,dim,dimensions,periods,
0,&cart_comm );
```

(The Cartesian grid can have fewer processes than the input communicator: any processes not included get `MPI_COMM_NULL` as output.)

MPL note 70: Cartesian communicator create. The actual Cartesian communicator has a constructor that takes a `dimensions` object as input.

```
mpl::cartesian_communicator cart_comm( comm_world,brick );
```

For a given communicator, you can test what type it is with `MPI_Topo_test` (figure 11.2):

```
int world_type, cart_type;
MPI_Topo_test( comm,&world_type );
MPI_Topo_test( cart_comm,&cart_type );
```

Figure 11.2 MPI_Topo_test

Name	Param name	Explanation	C type	F type	inout
MPI_Topo_test (
comm	communicator		MPI_Comm	TYPE (MPI_Comm)	IN
status	topology type of communicator comm		int*	INTEGER	OUT
)					

```

if (procno==0) {
    printf("World comm type=%d, Cart comm type=%d\n",
           world_type, cart_type);
    printf("no topo      =%d, cart top      =%d\n",
           MPI_UNDEFINED, MPI_CART);
}

```

For a Cartesian communicator, you can retrieve its information with `MPI_Cartdim_get` and `MPI_Cart_get`:

```

int dim;
MPI_Cartdim_get( cart_comm,&dim );
int *dimensions = (int*) malloc(dim*sizeof(int));
int *periods   = (int*) malloc(dim*sizeof(int));
int *coords    = (int*) malloc(dim*sizeof(int));
MPI_Cart_get( cart_comm,dim,dimensions,periods,coords );

```

MPL note 71: Get the dimensions object. The `dimensions` object can be extracted from the communicator

```

mpl::cartesian_communicator::dimensions
dimensions = cart_comm.get_dimensions();

```

after which dimensions and periodicities can be extracted:

```

// cartcoord.cxx
int dsize = dimensions.size(idim);
auto p = dimensions.periodicity(idim);

```

11.1.2 Cartesian vs world rank

Each point in a Cartesian communicator has a coordinate and a rank. The translation from rank to Cartesian coordinate is done by `MPI_Cart_coords` (figure 11.3), and translation from coordinates to a rank is done by `MPI_Cart_rank` (figure 11.4). In both cases, this translation can be done on any process; for the latter routine note that coordinates outside the Cartesian grid are erroneous, if the grid is not periodic in the offending coordinate.

```

// cart.c
MPI_Comm comm2d;
int periodic[ndim]; periodic[0] = periodic[1] = 0;

```

11. MPI topic: Topologies

Figure 11.3 MPI_Cart_coords

Name	Param name	Explanation	C type	F type	inout
MPI_Cart_coords					
	comm	communicator with Cartesian structure	MPI_Comm	TYPE (MPI_Comm)	IN
	rank	rank of a process within group of comm	int	INTEGER	IN
	maxdims	length of vector coords in the calling program	int	INTEGER	IN
	coords	integer array (of size maxdims) containing the Cartesian coordinates of specified process	int []	INTEGER (maxdims)	OUT
)					

Figure 11.4 MPI_Cart_rank

Name	Param name	Explanation	C type	F type	inout
MPI_Cart_rank					
	comm	communicator with Cartesian structure	MPI_Comm	TYPE (MPI_Comm)	IN
	coords	integer array (of size ndims) specifying the Cartesian coordinates of a process	const int []	INTEGER(*)	IN
	rank	rank of specified process	int*	INTEGER	OUT
)					

```

MPI_Cart_create(comm,ndim,dimensions,periodic,1,&comm2d);
if (comm2d==MPI_COMM_NULL) {
    printf("Process %d not included\n",procno);
} else {
    MPI_Cart_coords(comm2d,procno,ndim,coord_2d);
    MPI_Cart_rank(comm2d,coord_2d,&rank_2d);
    printf("I am %d: (%d,%d); originally %d\n",
           rank_2d,coord_2d[0],coord_2d[1],procno);
}

```

The `reorder` parameter to `MPI_Cart_create` indicates whether processes can have a rank in the new communicator that is different from in the old one.

MPL note 72: Rank to coord translation. The `coordinates` method of the cartesian communicator returns a vector-like object describing the process coordinate:

```

for ( int ip=0; ip<nprocs; ip++ ) {
    mpl::cartesian_communicator::vector
        coord = cart_comm.coordinates(ip);
    print("[{:2}] coord: [",ip);
    for ( int id=0; id<dim; id++ )
        print("{}",coord[id]);
    print("]\n");
}

```

11.1.3 Cartesian communication

A common communication pattern in Cartesian grids is to do an `MPI_Sendrecv` with processes that are adjacent along one coordinate axis.

By way of example, consider a 3D grid that is periodic in the first dimension:

```

// cartcoord.c
for ( int id=0; id<dim; id++ )
    periods[id] = id==0 ? 1 : 0;
MPI_Cart_create
( comm,dim,dimensions,periods,
  0,&period_comm );

```

We shift process 0 in dimensions 0 and 1. In dimension 0 we get a wrapped-around source, and a target that is the next process in row-major ordering; in dimension 1 we get `MPI_PROC_NULL` as source, and a legitimate target.

<pre> Code: int pred,succ; MPI_Cart_shift (period_comm,/* dim: */ 0,/* up: */ 1, &pred,&succ); printf ("periodic dimension 0:\n src=%d, tgt=%d\n", pred,succ); MPI_Cart_shift (period_comm,/* dim: */ 1,/* up: */ 1, &pred,&succ); printf ("non-periodic dimension 1:\n src=%d , tgt=%d\n", pred,succ); </pre>	<pre> Output: [examples/mpi/c] cartshift: Grid of size 6 in 3 dimensions: 3 x 2 x 1 Shifting process 0. periodic dimension 0: src=4, tgt=2 non-periodic dimension 1: src=-1, tgt=1 </pre>
---	---

MPL note 73: Cartesian shifting. The routine `cartesian_communicator::shift` takes a dimension and a direction, and gives the source and destination as a `shifted_ranks` object, which is basically a tuple of two integers:

Figure 11.5 MPI_Cart_sub

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Cart_sub (</code>					
	<code>comm</code>	communicator with Cartesian structure	<code>MPI_Comm</code>	<code>TYPE</code> <code>(MPI_Comm)</code>	<code>IN</code>
	<code>remain_dims</code>	the i -th entry of <code>remain_dims</code> specifies whether the i -th dimension is kept in the subgrid (true) or is dropped (false)	<code>const int[]</code>	<code>LOGICAL(*)</code>	<code>IN</code>
	<code>newcomm</code>	communicator containing the subgrid that includes the calling process	<code>MPI_Comm*</code>	<code>TYPE</code> <code>(MPI_Comm)</code>	<code>OUT</code>
	<code>)</code>				

```
int pred,succ;
mpl::shift_ranks shifted = cart_comm.shift
    ( /* dim: */ 1,/* up: */ 1 );
pred = shifted.source; succ = shifted.destination;
```

Exercise 11.1. Use Cartesian topology routines to extend exercise ?? to two dimensions.

11.1.4 Communicators in subgrids

The routine `MPI_Cart_sub` (figure 11.5) is similar to `MPI_Comm_split`, in that it splits a communicator into disjoint subcommunicators. In this case, it splits a Cartesian communicator into disjoint Cartesian communicators, each corresponding to a subset of the dimensions. This subset inherits both sizes and periodicity from the original communicator.

```

Code:
MPI_Cart_sub( period_comm,remain,&
    hyperplane );
if (procno==0) {
    MPI_Topo_test( hyperplane,&topo_type
        );
    MPI_Cartdim_get( hyperplane,&hyperdim
        );
    printf("hyperplane has dimension %d,
        type %d\n",
        hyperdim,topo_type);
    MPI_Cart_get( hyperplane,dim,dims,
        period,coords );
    printf(" periodic: ");
    for (int id=0; id<2; id++)
        printf("%d,",period[id]);
    printf("\n");
}

```

```

Output:
[examples/mpi/c] cartsub:
Grid of size 6 in 3 dimensions:
 3 x 2 x 1
hyperplane has dimension 2, type 2
  periodic: 1,0,

```

11.1.5 Reordering

The `MPI_Cart_create` routine has a possibility of reordering ranks. If this is applied, the routine `MPI_Cart_map` gives the result of this. Given the same parameters as `MPI_Cart_create`, it returns the re-ordered rank for the calling process.

11.2 Distributed graph topology

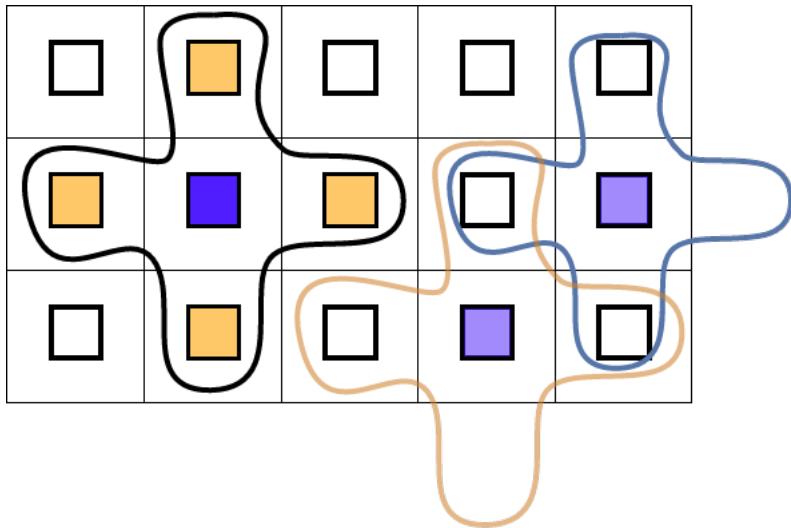


Figure 11.1: Illustration of a distributed graph topology where each node has four neighbors

In many calculations on a grid (using the term in its mathematical, Finite Element Method (FEM), sense), a grid point will collect information from grid points around it. Under a sensible distribution of the grid over processes, this means that each process will collect information from a number of neighbor processes. The number of neighbors is dependent on that process. For instance, in a 2D grid (and assuming a five-point stencil for the computation) most processes communicate with four neighbors; processes on the edge with three, and processes in the corners with two.

Such a topology is illustrated in figure 11.1.

MPI's notion of *graph topology*, and the *neighborhood collectives*, offer an elegant way of expressing such communication structures. There are various reasons for using graph topologies over the older, simpler methods.

- MPI is allowed to reorder the processes, so that network proximity in the cluster corresponds to proximity in the structure of the code.
- Ordinary collectives could not directly be used for graph problems, unless one would adopt a subcommunicator for each graph neighborhood. However, scheduling would then lead to deadlock or serialization.
- The normal way of dealing with graph problems is through nonblocking communications. However, since the user indicates an explicit order in which they are posted, congestion at certain processes may occur.
- Collectives can pipeline data, while send/receive operations need to transfer their data in its entirety.
- Collectives can use spanning trees, while send/receive uses a direct connection.

Thus the minimal description of a process graph contains for each process:

- Degree: the number of neighbor processes; and
- the ranks of the processes to communicate with.

However, this ignores that communication is not always symmetric: maybe the processes you receive from are not the ones you send to. Worse, maybe only one side of this duality is easily described. Therefore, there are two routines:

- `MPI_Dist_graph_create_adjacent` assumes that a process knows both who it is sending to, and who will send to it. This is the most work for the programmer to specify, but it is ultimately the most efficient.
- `MPI_Dist_graph_create` specifies on each process only what it is the source for; that is, who this process will be sending to. Consequently, some amount of processing – including communication – is needed to build the converse information, the ranks that will be sending to a process.

11.2.1 Graph creation

There are two creation routines for process graphs. These routines are fairly general in that they allow any process to specify any part of the topology. In practice, of course, you will mostly let each process describe its own neighbor structure.

The routine `MPI_Dist_graph_create_adjacent` assumes that a process knows both who it is sending to, and who will send to it. This means that every edge in the communication graph is represented twice, so the

memory footprint is double of what is strictly necessary. However, no communication is needed to build the graph.

The second creation routine, `MPI_Dist_graph_create` (figure 11.6), is probably easier to use, especially in cases where the communication structure of your program is symmetric, meaning that a process sends to the same neighbors that it receives from. Now you specify on each process only what it is the source for; that is, who this process will be sending to.¹. Consequently, some amount of processing – including communication – is needed to build the converse information, the ranks that will be sending to a process.

MPL note 74: Distributed graph creation. The class `mpl::dist_graph_communicator` only has a constructor corresponding to `MPI_Dist_graph_create`.

Figure 11.1 describes the common five-point stencil structure. If we let each process only describe itself, we get the following:

- `nsources`= 1 because the calling process describes on node in the graph: itself.
- `sources` is an array of length 1, containing the rank of the calling process.
- `degrees` is an array of length 1, containing the degree (probably: 4) of this process.
- `destinations` is an array of length the degree of this process, probably again 4. The elements of this array are the ranks of the neighbor nodes; strictly speaking the ones that this process will send to.
- `weights` is an array declaring the relative importance of the destinations. For an *unweighted graph* use `MPI_UNWEIGHTED`. In the case the graph is weighted, but the degree of a source is zero, you can pass an empty array as `MPI_WEIGHTS_EMPTY`.
- `reorder` (int in C, LOGICAL in Fortran) indicates whether MPI is allowed to shuffle processes to achieve greater locality.

The resulting communicator has all the processes of the original communicator, with the same ranks. In other words `MPI_Comm_size` and `MPI_Comm_rank` gives the same values on the graph communicator, as on the intra-communicator that it is constructed from. To get information about the grouping, use `MPI_Dist_graph_neighbors` and `MPI_Dist_graph_neighbors_count`; section 11.2.3.

By way of example we build an unsymmetric graph, that is, an edge $v_1 \rightarrow v_2$ between vertices v_1, v_2 does not imply an edge $v_2 \rightarrow v_1$.

1. I disagree with this design decision. Specifying your sources is usually easier than specifying your destinations.

11. MPI topic: Topologies

Figure 11.6 MPI_Dist_graph_create

Name	Param name	Explanation	C type	F type	inout
	MPI_Dist_graph_create (
	comm_old	input communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	n	number of source nodes for which this process specifies edges	int	INTEGER	IN
	sources	array containing the n source nodes for which this process specifies edges	const int []	INTEGER(n)	IN
	degrees	array specifying the number of destinations for each source node in the source node array	const int []	INTEGER(n)	IN
	destinations	destination nodes for the source nodes in the source node array	const int []	INTEGER(*)	IN
	weights	weights for source to destination edges	const int []	INTEGER(*)	IN
	info	hints on optimization and interpretation of weights	MPI_Info	TYPE (MPI_Info)	IN
	reorder	the ranks may be reordered (true) or not (false)	int	LOGICAL	IN
	comm_dist_graph	communicator with distributed graph topology added	MPI_Comm*	TYPE (MPI_Comm)	OUT
)				

MPL:

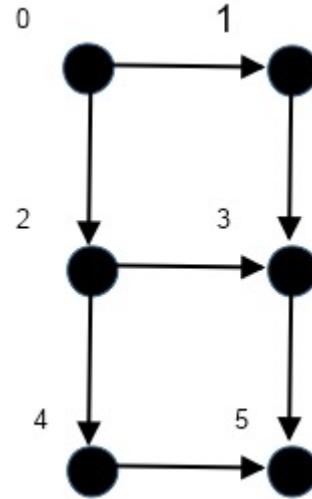
```
dist_graph_communicator
  (const communicator &old_comm,
   const source_set &ss, const dest_set &ds, bool reorder=true)
where:
class dist_graph_communicator::source_set : private set< pair<int,int> >
class dist_graph_communicator::dest_set : private set< pair<int,int> >
```

Python:

```
MPI.Comm.Create_dist_graph
  (self, sources, degrees, destinations, weights=None, Info info=INFO_NULL, bool reorder=False)
returns graph communicator
```

Code:

```
// graph.c
for ( int i=0; i<=1; i++ ) {
    int neighb_i = proc+i;
    if (neighb_i<0 || neighb_i>=idim)
        continue;
    int j = 1-i;
    int neighb_j = proc+j;
    if (neighb_j<0 || neighb_j>=jdim)
        continue;
    destinations[ degree++ ] =
        PROC(neighb_i,neighb_j,idim,jdim);
}
MPI_Dist_graph_create
(comm,
 /* I specify just one proc: me */ 1,
 &procno,&degree,destinations,weights,
 MPI_INFO_NULL,0,
 &comm2d
);
```



Here we gather the coordinates of the source neighbors:

```
Code:
int indegree,outdegree,
weighted;
MPI_Dist_graph_neighbors_count
(comm2d,
 &indegree,&outdegree,
 &weighted);
int
my_ij[2] = {proc,i},procj},
other_ij[4][2];
MPI_Neighbor_allgather
( my_ij,2,MPI_INT,
other_ij,2,MPI_INT,
comm2d );
```

Output:
[examples/mpi/c] graph:
[0 = (0,0)] has 2 outbound: 1, 2,
0 inbound:
[1 = (0,1)] has 1 outbound: 3,
1 inbound: (0,0)=0
[2 = (1,0)] has 2 outbound: 3, 4,
1 inbound: (0,0)=0
[3 = (1,1)] has 1 outbound: 5,
2 inbound: (0,1)=1 (1,0)=2
[4 = (2,0)] has 1 outbound: 5,
1 inbound: (1,0)=2
[5 = (2,1)] has 0 outbound:
2 inbound: (1,1)=3 (2,0)=4

However, we can't rely on the sources being ordered, so the following segment performs an explicit query for the source neighbors:

```
Code:
int indegree=4, sources[indegree],
    inweights[indegree],weighted;
int outdegree=4, targets[outdegree],
    outweights[outdegree];
MPI_Dist_graph_neighbors_count
(comm2d,
 &indegree,&outdegree,
 &weighted);
MPI_Dist_graph_neighbors
(comm2d,
 indegree,sources,inweights,
 outdegree,targets,outweights
);
```

```
Output:
[examples/mpi/c] graphcount:

0 inbound:
1 inbound: 0
1 inbound: 0
2 inbound: 1 2
1 inbound: 2
2 inbound: 4 3
```

Python note 31: Graph communicators. Graph communicator creation is a method of the `Comm` class, and the graph communicator is a function return result:

```
graph_comm = oldcomm.Create_dist_graph(sources, degrees, destinations)
```

The weights, info, and reorder arguments have default values.

MPL note 75: Graph communicators. The constructor `dist_graph_communicator`

```
dist_graph_communicator
(const communicator &old_comm, const source_set &ss,
 const dest_set &ds, bool reorder = true);
```

is a wrapper around `MPI_Dist_graph_create_adjacent`.

MPL note 76: Graph communicator querying. Methods `indegree`, `outdegree` are wrappers around `MPI_Dist_graph_neighbors_count`. Sources and targets can be queried with `inneighbors` and `outneighbors`, which are wrappers around `MPI_Dist_graph_neighbors`.

11.2.2 Neighbor collectives

We can now use the graph topology to perform a gather or allgather `MPI_Neighbor_allgather` (figure 11.7) that combines only the processes directly connected to the calling process.

The neighbor collectives have the same argument list as the regular collectives, but they apply to a graph communicator.

Exercise 11.2. Revisit exercise 4.3 and solve it using `MPI_Dist_graph_create`. Use figure 11.2 for inspiration.

Use a degree value of 1.

(There is a skeleton for this exercise under the name `rightgraph`.)

The previous exercise can be done with a degree value of:

- 1, reflecting that each process communicates with just 1 other; or

Figure 11.7 MPI_Neighbor_allgather

Name	Param name	Explanation	C type	F type	inout
MPI_Neighbor_allgather (
MPI_Neighbor_allgather_c (
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN	
sendcount	number of elements sent to each neighbor	[int MPI_Count]	INTEGER	IN	
sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN	
recvbuf	starting address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT	
recvcount	number of elements received from each neighbor	[int MPI_Count]	INTEGER	IN	
recvtype	datatype of receive buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN	
comm	communicator with topology structure	MPI_Comm	TYPE (MPI_Comm)	IN	
)					

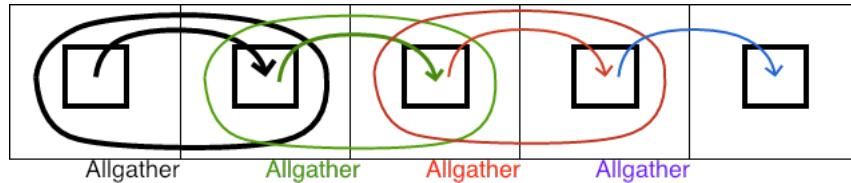


Figure 11.2: Solving the right-send exercise with neighborhood collectives

- 2, reflecting that you really gather from two processes.

In the latter case, results do not wind up in the receive buffer in order of increasing process number as with a traditional gather. Rather, you need to use `MPI_Dist_graph_neighbors` to find their sequencing; see section 11.2.3.

Another neighbor collective is `MPI_Neighbor_alltoall`.

The vector variants are `MPI_Neighbor_allgatherv` and `MPI_Neighbor_alltoallv`.

There is a heterogenous (multiple datatypes) variant: `MPI_Neighbor_alltoallw`.

The list is: `MPI_Neighbor_allgather`, `MPI_Neighbor_allgatherv`, `MPI_Neighbor_alltoall`, `MPI_Neighbor_alltoallv`, `MPI_Neighbor_alltoallw`.

Nonblocking: `MPI_Ineighbor_allgather`, `MPI_Ineighbor_allgatherv`, `MPI_Ineighbor_alltoall`, `MPI_Ineighbor_alltoallv`, `MPI_Ineighbor_alltoallw`.

For unclear reasons there is no `MPI_Neighbor_allreduce`.

Figure 11.8 MPI_Dist_graph_neighbors_count

Name	Param name	Explanation	C type	F type	inout
MPI_Dist_graph_neighbors_count					
comm		communicator with distributed graph topology	MPI_Comm	TYPE (MPI_Comm)	IN
indegree		number of edges into this process	int*	INTEGER	OUT
outdegree		number of edges out of this process	int*	INTEGER	OUT
weighted		false if MPI_UNWEIGHTED was supplied during creation, true otherwise	int*	LOGICAL	OUT
)					

11.2.3 Query

There are two routines for querying the neighbors of a process: `MPI_Dist_graph_neighbors_count` (figure 11.8) and `MPI_Dist_graph_neighbors` (figure 11.9).

While this information seems derivable from the graph construction, that is not entirely true for two reasons.

1. With the nonadjoint version `MPI_Dist_graph_create`, only outdegrees and destinations are specified; this call then supplies the indegrees and sources;
2. As observed above, the order in which data is placed in the receive buffer of a gather call is not determined by the create call, but can only be queried this way.

11.2.4 Graph topology (deprecated)

The original MPI-1 had a graph topology interface `MPI_Graph_create` which required each process to specify the full process graph. Since this is not scalable, it should be considered deprecated. Use the distributed graph topology (section 11.2) instead.

Other legacy routines: `MPI_Graph_neighbors`, `MPI_Graph_neighbors_count`, `MPI_Graph_get`, `MPI_Graphdims_get`.

11.2.5 Re-ordering

Similar to the `MPI_Cart_map` routine (section 11.1.5), the routine `MPI_Graph_map` gives a re-ordered rank for the calling process.

Figure 11.9 MPI_Dist_graph_neighbors

Name	Param name	Explanation	C type	F type	inout
	<code>MPI_Dist_graph_neighbors (</code>				
	<code> comm</code>	communicator with distributed graph topology	<code>MPI_Comm</code>	<code>TYPE (MPI_Comm)</code>	<code>IN</code>
	<code> maxindegree</code>	size of sources and sourceweights arrays	<code>int</code>	<code>INTEGER</code>	<code>IN</code>
	<code> sources</code>	processes for which the calling process is a destination	<code>int []</code>	<code>INTEGER (maxindegree)</code>	<code>OUT</code>
	<code> sourceweights</code>	weights of the edges into the calling process	<code>int []</code>	<code>INTEGER(*)</code>	<code>OUT</code>
	<code> maxoutdegree</code>	size of destinations and destweights arrays	<code>int</code>	<code>INTEGER</code>	<code>IN</code>
	<code> destinations</code>	processes for which the calling process is a source	<code>int []</code>	<code>INTEGER (maxoutdegree)</code>	<code>OUT</code>
	<code> destweights</code>	weights of the edges out of the calling process	<code>int []</code>	<code>INTEGER(*)</code>	<code>OUT</code>
	<code>)</code>				

Chapter 12

MPI topic: Shared memory

Some programmers are under the impression that MPI would not be efficient on shared memory, since all operations are done through what looks like network calls. This is not correct: many MPI implementations have optimizations that detect shared memory and can exploit it, so that data is copied, rather than going through a communication layer. (Conversely, programming systems for shared memory such as *OpenMP* can actually have inefficiencies associated with thread handling.) The main inefficiency associated with using MPI on shared memory is then that processes can not actually share data.

The one-sided MPI calls (chapter 9) can also be used to emulate shared memory, in the sense that an origin process can access data from a target process without the target's active involvement. However, these calls do not distinguish between actually shared memory and one-sided access across the network.

In this chapter we will look at the ways MPI can interact with the presence of actual shared memory. (This functionality was added in the MPI-3 standard.) This relies on the `MPI_Win` windows concept, but otherwise uses direct access of other processes' memory.

12.1 Recognizing shared memory

MPI's one-sided routines take a very symmetric view of processes: each process can access the window of every other process (within a communicator). Of course, in practice there will be a difference in performance depending on whether the origin and target are actually on the same shared memory, or whether they can only communicate through the network. For this reason MPI makes it easy to group processes by shared memory domains using `MPI_Comm_split_type` (see section 7.4.1) with the type `MPI_COMM_TYPE_SHARED`.

Splitting by shared memory:

```
Code:
// commsplittype.c
MPI_Info info;
MPI_Comm_split_type
    (MPI_COMM_WORLD,
     MPI_COMM_TYPE_SHARED,
     procno,info,&sharedcomm);
MPI_Comm_size
    (sharedcomm,&new_nprocs);
MPI_Comm_rank
    (sharedcomm,&new_procno);
```

```
Output:
[examples/mpi/c] commsplittype:
make[3]: `commsplittype' is up to
      date.
TACC: Starting up job 4356245
TACC: Starting parallel tasks...
There are 10 ranks total
[0] is processor 0 in a shared
    ↪group of 5, running on
    ↪c209-010.frontera.tacc.utexas.edu
[5] is processor 0 in a shared
    ↪group of 5, running on
    ↪c209-011.frontera.tacc.utexas.edu
TACC: Shutdown complete. Exiting.
```

Exercise 12.1. Write a program that uses `MPI_Comm_split_type` to analyze for a run

1. How many nodes there are;
2. How many processes there are on each node.

If you run this program on an unequal distribution, say 10 processes on 3 nodes, what distribution do you find?

```
Nodes: 3; processes: 10
TACC: Starting up job 4210429
TACC: Starting parallel tasks...
There are 3 nodes
Node sizes: 4 3 3
TACC: Shutdown complete. Exiting.
```

MPL note 77: Split by shared memory. Similar to ordinary communicator splitting (slide 64): `communicator ::split_shared`.

```
// commsplittype.cxx
mpl::communicator shared_comm
( mpl::communicator::split_shared_memory, world_comm );
int
onnode_procno = shared_comm.rank(),
onnode_nprocs = shared_comm.size();
```

But note: shared memory is currently not available, since windows are not (yet) implemented.

12.2 Shared memory for windows

Processes that exist on the same physical shared memory should be able to move data by copying, rather than through MPI send/receive calls – which of course will do a copy operation under the hood. In order to do such user-level copying:

1. We need to create a shared memory area with `MPI_Win_allocate_shared`. This creates a window with the *unified memory model* (see section 9.5.1); and

Figure 12.1 MPI_Win_allocate_shared

Name	Param name	Explanation	C type	F type	inout
MPI_Win_allocate_shared (
MPI_Win_allocate_shared_c (
size	size	size of local window in bytes	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
disp_unit	local unit size for displacements, in bytes	[int MPI_Aint		INTEGER	IN
info	info argument		MPI_Info	TYPE (MPI_Info)	IN
comm	intra-communicator		MPI_Comm	TYPE (MPI_Comm)	IN
baseptr	address of local allocated window segment		void*	TYPE(C_PTR)	OUT
win	window object returned by the call		MPI_Win*	TYPE(MPI_Win)	OUT
)					

2. We need to get pointers to where a process' area is in this shared space; this is done with `MPI_Win_shared_query`.

Remark As of MPI-4.1, `MPI_Win_shared_query` can be used on memory from `MPI_Win_allocate` and `MPI_Win_create`, as long as this is actually a window on shared memory. Only `MPI_Win_allocate_shared` is guaranteed to yield such shared memory.

12.2.1 Pointers to a shared window

The first step is to create a window (in the sense of one-sided MPI; section 9.1) on the processes on one node. Using the `MPI_Win_allocate_shared` (figure 12.1) call presumably will put the memory close to the socket on which the process runs.

```
// sharedbulk.c
MPI_Win node_window;
MPI_Aint window_size; double *window_data;
if (onnode_procid==0)
    window_size = sizeof(double);
else window_size = 0;
MPI_Win_allocate_shared
( window_size,sizeof(double),MPI_INFO_NULL,
  nodecomm,
  &window_data,&node_window);
```

The memory allocated by `MPI_Win_allocate_shared` is contiguous between the processes. This makes it possible to do address calculation. However, if a cluster node has a Non-Uniform Memory Access (NUMA) structure, for instance if two sockets have memory directly attached to each, this would increase latency

for some processes. To prevent this, the key `alloc_shared_noncontig` can be set to `true` in the `MPI_Info` object.

The following material is for the recently released MPI-4 standard and may not be supported yet.

In the contiguous case, the `mpi_minimum_memory_alignment` info argument (section 9.1.1) applies only to the memory on the first process; in the noncontiguous case it applies to all.

End of MPI-4 material

```
// numa.c
MPI_Info window_info;
MPI_Info_create(&window_info);
MPI_Info_set(window_info,"alloc_shared_noncontig","true");
MPI_Win_allocate_shared( window_size,sizeof(double),window_info,
                        nodecomm,
                        &window_data,&node_window);
MPI_Info_free(&window_info);
```

Let's explore this. We create a shared window where each process stores exactly one double, that is, 8 bytes. The following code fragment queries the window locations, and prints the distance in bytes to the window on process 0.

```
for (int p=1; p<onnode_nprocs; p++) {
    MPI_Aint window_sizep; int windowp_unit; double *winp_addr;
    MPI_Win_shared_query( node_window,p,
                          &window_sizep,&windowp_unit, &winp_addr );
    distp = (size_t)winp_addr-(size_t)win0_addr;
    if (procno==0)
        printf("Distance %d to zero: %ld\n",p,(long)distp);
```

With the default strategy, these windows are contiguous, and so the distances are multiples of 8 bytes. Not so for the the non-contiguous allocation:

Strategy: default behavior of shared window allocation

```
Distance 1 to zero: 8
Distance 2 to zero: 16
Distance 3 to zero: 24
Distance 4 to zero: 32
Distance 5 to zero: 40
Distance 6 to zero: 48
Distance 7 to zero: 56
Distance 8 to zero: 64
Distance 9 to zero: 72
```

Strategy: allow non-contiguous shared window allocation

```
Distance 1 to zero: 4096
Distance 2 to zero: 8192
Distance 3 to zero: 12288
Distance 4 to zero: 16384
Distance 5 to zero: 20480
Distance 6 to zero: 24576
Distance 7 to zero: 28672
Distance 8 to zero: 32768
Distance 9 to zero: 36864
```

The explanation here is that each window is placed on its own *small page*, which on this particular system has a size of 4K.

Remark *The ampersand operator in C is not a physical address, but a virtual address. The translation of where pages are placed in physical memory is determined by the page table.*

Figure 12.2 MPI_Win_shared_query

Name	Param name	Explanation	C type	F type	inout
	MPI_Win_shared_query (
	MPI_Win_shared_query_c (
win	shared memory window object		MPI_Win	TYPE(MPI_Win)	IN
rank	rank in the group of window win or MPI_PROC_NULL		int	INTEGER	IN
size	size of the window segment		MPI_Aint*	INTEGER (KIND=MPI_ADDRESS_KIND)	OUT
disp_unit	local unit size for displacements, in bytes		[int* MPI_Aint*]	INTEGER	OUT
baseptr	address for load/store access to window segment		void*	TYPE(C_PTR)	OUT
)					

12.2.2 Querying the shared structure

Even though the window created above is shared, that doesn't mean it's contiguous. Hence it is necessary to retrieve the pointer to the area of each process that you want to communicate with: `MPI_Win_shared_query` (figure 12.2).

```
MPI_Aint window_size0; int window_unit; double *win0_addr;
MPI_Win_shared_query
( node_window,0,
  &window_size0,&window_unit, &win0_addr );
```

12.2.3 Heat equation example

As an example, which consider the 1D heat equation. On each process we create a local area of three point:

```
// sharedshared.c
MPI_Win_allocate_shared(3,sizeof(int),info,sharedcomm,&shared_baseptr,&shared_window);
```

12.2.4 Shared bulk data

In applications such as *ray tracing*, there is a read-only large data object (the objects in the scene to be rendered) that is needed by all processes. In traditional MPI, this would need to be stored redundantly on each process, which leads to large memory demands. With MPI shared memory we can store the data object once per node. Using as above `MPI_Comm_split_type` to find a communicator per NUMA domain, we store the object on process zero of this node communicator.

Exercise 12.2. Let the ‘shared’ data originate on process zero in `MPI_COMM_WORLD`. Then:

- create a communicator per shared memory domain;
- create a communicator for all the processes with number zero on their node;
- broadcast the shared data to the processes zero on each node.

(There is a skeleton for this exercise under the name `shareddata`.)

Chapter 13

MPI topic: Hybrid computing

While the MPI standard itself makes no mention of threads – process being the primary unit of computation – the use of threads is allowed. Below we will discuss what provisions exist for doing so.

Using threads and other shared memory models in combination with MPI leads of course to the question how *race conditions* are handled. Example of a code with a *data race* that pertains to MPI:

```
#pragma omp sections
#pragma omp section
    MPI_Send( x, /* to process 2 */ )
#pragma omp section
    MPI_Recv( x, /* from process 3 */ )
```

The MPI standard here puts the burden on the user: this code is not legal, and behavior is not defined.

13.1 MPI support for threading

In hybrid execution, the main question is whether all threads are allowed to make MPI calls. To determine this, replace the `MPI_Init` call by `MPI_Init_thread` (figure 13.1) Here the required and provided parameters can take the following (monotonically increasing) values:

- `MPI_THREAD_SINGLE`: Only a single thread will execute.
 - `MPI_THREAD_FUNNELED`: The program may use multiple threads, but only the main thread will make MPI calls.
- The main thread is usually the one selected by the `master` directive, but technically it is the only that executes `MPI_Init_thread`. If you call this routine in a parallel region, the main thread may be different from the master.
- `MPI_THREAD_SERIALIZED`: The program may use multiple threads, all of which may make MPI calls, but there will never be simultaneous MPI calls in more than one thread.
 - `MPI_THREAD_MULTIPLE`: Multiple threads may issue MPI calls, without restrictions.

After the initialization call, you can query the support level with `MPI_Query_thread` (figure 13.2).

In case more than one thread performs communication, `MPI_Is_thread_main` (figure 13.3) can determine whether a thread is the main thread.

Python note 32: Thread level. The thread level can be set through the `mpi4py.rc` object (section 2.2.2):

Figure 13.1 MPI_Init_thread

Name	Param name	Explanation	C type	F type	inout
MPI_Init_thread					
argc		desired level of thread support	int*	INTEGER	IN
argv		provided level of thread support	char***	INTEGER	OUT
)					

Figure 13.2 MPI_Query_thread

Name	Param name	Explanation	C type	F type	inout
MPI_Query_thread					
provided		provided level of thread support	int*	INTEGER	OUT
)					

```
mpi4py.rc.threads # default True
mpi4py.rc.thread_level # default "multiple"
```

Available levels are multiple, serialized, funneled, single.

MPL note 78: Threading support. MPL always calls `MPI_Init_thread` requesting the highest level `MPI_THREAD_MULTIPLE`.

```
enum mpl::threading_modes {
    mpl::threading_modes::single = MPI_THREAD_SINGLE,
    mpl::threading_modes::funneled = MPI_THREAD_FUNNELED,
    mpl::threading_modes::serialized = MPI_THREAD_SERIALIZED,
    mpl::threading_modes::multiple = MPI_THREAD_MULTIPLE
};
threading_modes mpl::environment::threading_mode ();
bool mpl::environment::is_thread_main();
```

The *mvapich* implementation of MPI does have the required threading support, but you need to set this environment variable:

```
export MV2_ENABLE_AFFINITY=0
```

Another solution is to run your code like this:

```
ibrun tacc_affinity <my_multithreaded_mpi_executable
```

Intel MPI uses an environment variable to turn on thread support:

```
I_MPI_LIBRARY_KIND=<value>
where
release : multi-threaded with global lock
release_mt : multi-threaded with per-object lock for thread-split
```

The *mpiexec* program usually propagates *environment variables*, so the value of `OMP_NUM_THREADS` when you call *mpiexec* will be seen by each MPI process.

Figure 13.3 MPI_Is_thread_main

Name	Param name	Explanation	C type	F type	inout
MPI_Is_thread_main (flag	true if calling thread is main thread, false otherwise	int*	LOGICAL	OUT

- It is possible to use blocking sends in threads, and let the threads block. This does away with the need for polling.
- You can not send to a thread number: use the MPI *message tag* to send to a specific thread.

Exercise 13.1. Consider the 2D heat equation and explore the mix of MPI/OpenMP parallelism:

- Give each node one MPI process that is fully multi-threaded.
- Give each core an MPI process and don't use multi-threading.

Discuss theoretically why the former can give higher performance. Implement both schemes as special cases of the general hybrid case, and run tests to find the optimal mix.

```
// thread.c
MPI_Init_thread(&argc,&argv,MPI_THREAD_MULTIPLE,&threading);
comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm,&procno);
MPI_Comm_size(comm,&nprocs);

if (procno==0) {
    switch (threading) {
        case MPI_THREAD_MULTIPLE : printf("Glorious multithreaded MPI\n"); break;
        case MPI_THREAD_SERIALIZED : printf("No simultaneous MPI from threads\n"); break;
        case MPI_THREAD_FUNNELED : printf("MPI from main thread\n"); break;
        case MPI_THREAD_SINGLE : printf("no threading supported\n"); break;
    }
}
MPI_Finalize();
```

Chapter 14

MPI topic: Tools interface

Recent versions of MPI, starting at MPI-3.0 and extended in MPI-3.1 and MPI-4.0, have a standardized way of reading out performance variables: the *tools interface* which improves on the old interface described in section 15.6.2.

14.1 Initializing the tools interface

The tools interface requires a different initialization routine `MPI_T_init_thread`

```
int MPI_T_init_thread( int required,int *provided );
```

Likewise, there is `MPI_T_finalize`

```
int MPI_T_finalize();
```

These matching calls can be made multiple times, after MPI has already been initialized with `MPI_Init` or `MPI_Init_thread`.

Verbosity level is an integer parameter.

```
MPI_T_VERBOSITY_{USER,TUNER,MPIDEV}_{BASIC,DETAIL,ALL}
```

14.2 Control variables

Control variables are implementation-dependent variables that can be used to inspect and/or control the internal workings of MPI. Accessing control variables requires initializing the tools interface; section 14.1.

We query how many *control variables* are available with `MPI_T_cvar_get_num` (figure 14.1). A description of the control variable can be obtained from `MPI_T_cvar_get_info` (figure 14.2).

- An invalid index leads to a function result of `MPI_T_ERR_INVALID_INDEX`.
- Any output parameter can be specified as `NULL` and MPI will not set this.
- The `bind` variable is an object type or `MPI_T_BIND_NO_OBJECT`.
- The `enumtype` variable is `MPI_T_ENUM_NULL` if the variable is not an enum type.

Figure 14.1 MPI_T_cvar_get_num

Name	Param name	Explanation	C type	F type	inout
MPI_T_cvar_get_num	()				

Figure 14.2 MPI_T_cvar_get_info

Name	Param name	Explanation	C type	F type	inout
MPI_T_cvar_get_info	()				

```
// cvar.c
MPI_T_cvar_get_num(&ncvar);
printf("#cvars: %d\n", ncvar);
for (int ivar=0; ivar<ncvar; ivar++) {
    char name[100]; int namelen = 100;
    char desc[256]; int descrlen = 256;
    int verbosity, bind, scope;
    MPI_Datatype datatype;
    MPI_T_enum enumtype;
    MPI_T_cvar_get_info
        (ivar,
         name,&namelen,
         &verbosity,&datatype,&enumtype,desc,&desclen,&bind,&scope
        );
    printf("cvar %3d: %s\n %s\n", ivar, name, desc);
```

Remark There is no constant indicating a maximum buffer length for these variables. However, you can do the following:

1. Call the info routine with NULL values for the buffers, reading out the buffer lengths;
2. allocate the buffers with sufficient length, that is, including an extra position for the null terminator; and
3. calling the info routine a second time, filling in the string buffers.

Conversely, given a variable name, its index can be retrieved with `MPI_T_cvar_get_index`:

```
int MPI_T_cvar_get_index(const char *name, int *cvar_index)
```

If the name can not be matched, the index is `MPI_T_ERR_INVALID_NAME`.

Accessing a control variable is done through a *control variable handle*.

```
int MPI_T_cvar_handle_alloc
    (int cvar_index, void *obj_handle,
     MPI_T_cvar_handle *handle, int *count)
```

The handle is freed with `MPI_T_cvar_handle_free`:

```
int MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)
```

Control variable access is done through `MPI_T_cvar_read` and `MPI_T_cvar_write`:

```
int MPI_T_cvar_read(MPI_T_cvar_handle handle, void* buf);
int MPI_T_cvar_write(MPI_T_cvar_handle handle, const void* buf);
```

14.2.1 Callback interface

The following material is for the recently released MPI-4 standard and may not be supported yet.

`MPI_T_Source_.... MPI_T_Event_.... MPI_T_Category_get_num_events_.... MPI_T_Category_get_events_....`
End of MPI-4 material

14.3 Performance variables

The realization of the tools interface is installation-dependent, you first need to query how much of the tools interface is provided.

```
// mpitpvar.c
MPI_Init_thread(&argc,&argv,MPI_THREAD_SINGLE,&tlevel);
MPI_T_init_thread(MPI_THREAD_SINGLE,&tlevel);
int npvar;
MPI_T_pvar_get_num(&npvar);

int name_len=256,desc_len=256,
    verbosity,var_class,binding,isreadonly,iscontiguous,isatomic;
char var_name[256],description[256];
MPI_Datatype datatype; MPI_T_enum enumtype;
for (int pvar=0; pvar<npvar; pvar++) {
    name_len = 256; desc_len=256;
    MPI_T_pvar_get_info(pvar,var_name,&name_len,
                        &verbosity,&var_class,
                        &datatype,&enumtype,
                        description,&desc_len,
                        &binding,&isreadonly,&iscontiguous,&isatomic);
    if (procid==0)
        printf("pvar %d: %d/%s = %s\n",pvar,var_class,var_name,description);
}
```

Performance variables come in classes: `MPI_T_PVAR_CLASS_STATE` `MPI_T_PVAR_CLASS_LEVEL`
`MPI_T_PVAR_CLASS_SIZE` `MPI_T_PVAR_CLASS_PERCENTAGE` `MPI_T_PVAR_CLASS_HIGHWATERMARK`
`MPI_T_PVAR_CLASS_LOWWATERMARK` `MPI_T_PVAR_CLASS_COUNTER` `MPI_T_PVAR_CLASS_AGGREGATE`
`MPI_T_PVAR_CLASS_TIMER` `MPI_T_PVAR_CLASS_GENERIC`

Query the number of performance variables with `MPI_T_pvar_get_num`:

```
int MPI_T_pvar_get_num(int *num_pvar);
```

Get information about each variable, by index, with `MPI_T_pvar_get_info`:

```
int MPI_T_pvar_get_info
  (int pvar_index, char *name, int *name_len,
   int *verbosity, int *var_class, MPI_Datatype *datatype,
   MPI_T_enum *enumtype, char *desc, int *desc_len, int *bind,
   int *readonly, int *continuous, int *atomic)
```

See general remarks about these in section 14.2.

- The *readonly* variable indicates that the variable can not be written.
- The *continuous* variable requires use of `MPI_T_pvar_start` and `MPI_T_pvar_stop`.

Given a name, the index can be retried with `MPI_T_pvar_get_index`:

```
int MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)
```

Again, see section 14.2.

14.3.1 Performance experiment sessions

To prevent measurements from getting mixed up, they need to be done in *performance experiment sessions*, to be called ‘sessions’ in this chapter. However see section 8.3.

Create a session with `MPI_T_pvar_session_create`

```
int MPI_T_pvar_session_create(MPI_T_pvar_session *session)
```

and release it with `MPI_T_pvar_session_free`:

```
int MPI_T_pvar_session_free(MPI_T_pvar_session *session)
```

which sets the session variable to `MPI_T_PVAR_SESSION_NULL`.

We access a variable through a handle, associated with a certain session. The handle is created with `MPI_T_pvar_handle_alloc`:

```
int MPI_T_pvar_handle_alloc
  (MPI_T_pvar_session session, int pvar_index,
   void *obj_handle, MPI_T_pvar_handle *handle, int *count)
```

(If a routine takes both a session and handle argument, and the two are not associated, an error of `MPI_T_ERR_INVALID_HANDLE` is returned.)

Free the handle with `MPI_T_pvar_handle_free`:

```
int MPI_T_pvar_handle_free
  (MPI_T_pvar_session session,
   MPI_T_pvar_handle *handle)
```

which sets the variable to `MPI_T_PVAR_HANDLE_NULL`.

Continuous variables (see `MPI_T_pvar_get_info` above, which outputs this) can be started and stopped with `MPI_T_pvar_start` and `MPI_T_pvar_stop`:

```
int MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle);
int MPI_T_pvar_stop(MPI_T_pvar_session session, MPI_T_pvar_handle handle)
```

Passing `MPI_T_PVAR_ALL_HANDLES` to the stop call attempts to stop all variables within the session. Failure to stop a variable returns `MPI_T_ERR_PVAR_NO_STARTSTOP`.

Variables can be read and written with `MPI_T_pvar_read` and `MPI_T_pvar_write`:

```
int MPI_T_pvar_read
  (MPI_T_pvar_session session, MPI_T_pvar_handle handle,
   void* buf)
int MPI_T_pvar_write
  (MPI_T_pvar_session session, MPI_T_pvar_handle handle,
   const void* buf)
```

If the variable can not be written (see the `readonly` parameter of `MPI_T_pvar_get_info`), `MPI_T_ERR_PVAR_NO_WRITE` is returned.

A special case of writing the variable is to reset it with

```
int MPI_T_pvar_reset(MPI_T_pvar_session session, MPI_T_pvar_handle handle)
```

The handle value of `MPI_T_PVAR_ALL_HANDLES` is allowed.

A call to `MPI_T_pvar_readreset` is an atomic combination of the read and reset calls:

```
int MPI_T_pvar_readreset
  (MPI_T_pvar_session session, MPI_T_pvar_handle handle,
   void* buf)
```

14.4 Categories of variables

Variables, both the control and performance kind, can be grouped into categories by the MPI implementation.

The number of categories is queried with `MPI_T_category_get_num`:

```
int MPI_T_category_get_num(int *num_cat)
```

and for each category the information is retrieved with `MPI_T_category_get_info`:

```
int MPI_T_category_get_info
  (int cat_index,
   char *name, int *name_len, char *desc, int *desc_len,
   int *num_cvars, int *num_pvars, int *num_categories)
```

For a given category name the index can be found with `MPI_T_category_get_index`:

```
int MPI_T_category_get_index(const char *name, int *cat_index)
```

The contents of a category are retrieved with `MPI_T_category_get_cvars`, `MPI_T_category_get_pvars`, `MPI_T_category_get_categories`:

```
int MPI_T_category_get_cvars(int cat_index, int len, int indices[])
int MPI_T_category_get_pvars(int cat_index, int len, int indices[])
int MPI_T_category_get_categories(int cat_index, int len, int indices[])
```

These indices can subsequently be used in the calls `MPI_T_cvar_get_info`, `MPI_T_pvar_get_info`, `MPI_T_category_get_info`.

If categories change dynamically, this can be detected with `MPI_T_category_changed`

```
int MPI_T_category_changed(int *stamp)
```

14.5 Events

```
// mpitevent.c
int nsources;
MPI_T_source_get_num(&nsources);

int name_len=256,desc_len=256;
char var_name[256],description[256];
MPI_T_source_order ordering;
MPI_Count ticks_per_second,max_ticks;
MPI_Info info;
MPI_Datatype datatype; MPI_T_enum enumtype;
for (int source=0; source<nsources; source++) {
    name_len = 256; desc_len=256;
    MPI_T_source_get_info(source,var_name,&name_len,
                          description,&desc_len,
                          &ordering,&ticks_per_second,&max_ticks,&info);
```

Chapter 15

MPI leftover topics

15.1 Contextual information, attributes, etc.

15.1.1 Info objects

Certain MPI routines can accept `MPI_Info` objects. (For files, see section 15.1.1.3, for windows, see section 9.5.5.) These contain key-value pairs that can offer system or implementation dependent information.

Info objects can be created with `MPI_Info_create` (figure 15.1) and deleted with `MPI_Info_free` (figure 15.2); there is one info object, named `MPI_INFO_ENV`, which is created by `MPI_Init` or `MPI_Init_thread`; see section 15.1.1.1.

Keys are then set with `MPI_Info_set` (figure 15.3), and they can be queried with `MPI_Info_get` (figure 15.4). Note that the output of the ‘get’ routine is not allocated: it is a buffer that is passed. The maximum length of a key is given by the parameter `MPI_MAX_INFO_KEY`. You can delete a key from an info object with `MPI_Info_delete` (figure 15.5).

There is a straightforward duplication of info objects: `MPI_Info_dup` (figure 15.6).

You can also query the number of keys in an info object with `MPI_Info_get_nkeys` (figure 15.7), after which the keys can be queried in succession with `MPI_Info_get_nthkey`.

Info objects that are marked as ‘In’ or ‘Inout’ arguments are parsed before that routine returns. This means that in nonblocking routines they can be freed immediately, unlike, for instance, send buffers.

The following material is for the recently released MPI-4 standard and may not be supported yet.

The routines `MPI_Info_get` and `MPI_Info_get_valuelen` are not robust with respect to the C language *null terminator*. Therefore, they are deprecated, and should be replaced with `MPI_Info_get_string`, which always returns a null-terminated string.

```
int MPI_Info_get_string
  (MPI_Info info, const char *key,
   int *buflen, char *value, int *flag)
```

End of MPI-4 material

MPL note 79: Info objects. There is an `info` object in the `mpl` namespace:

```
mpl::info infoobject; // default constructor
```

15. MPI leftover topics

Figure 15.1 MPI_Info_create

Name	Param name	Explanation	C type	F type	inout
MPI_Info_create					

```

MPI_Info_create (
    info      info object created      MPI_Info*
                                         TYPE
                                         (MPI_Info)
)

```

Figure 15.2 MPI_Info_free

Name	Param name	Explanation	C type	F type	inout
MPI_Info_free					

```

MPI_Info_free (
    info      info object      MPI_Info*
                                         TYPE
                                         (MPI_Info)
)

```

Figure 15.3 MPI_Info_set

Name	Param name	Explanation	C type	F type	inout
MPI_Info_set					

```

MPI_Info_set (
    info      info object      MPI_Info
                                         TYPE
                                         (MPI_Info)
    key       key             const char*
    value     value           const char*
)

```

Figure 15.4 MPI_Info_get

Name	Param name	Explanation	C type	F type	inout
MPI_Info_get					

```

MPI_Info_get (
    info      info object      MPI_Info
                                         TYPE
                                         (MPI_Info)
    key       key             const char*
    valuelen length of value   int
                                         associated with key
    value     value           char*
    flag      true if key     int*
                                         defined, false if
                                         not
)

```

Figure 15.5 MPI_Info_delete

Name	Param name	Explanation	C type	F type	inout
MPI_Info_delete					

```

MPI_Info_delete (
    info      info object      MPI_Info
                                         TYPE
                                         (MPI_Info)
    key       key             const char*
)

```

Figure 15.6 MPI_Info_dup

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Info_dup (</code>					
	<code>info</code>	<code>info object</code>	<code>MPI_Info</code>	<code>TYPE (MPI_Info)</code>	<code>IN</code>
	<code>newinfo</code>	<code>info object created</code>	<code>MPI_Info*</code>	<code>TYPE (MPI_Info)</code>	<code>OUT</code>
<code>)</code>					

Figure 15.7 MPI_Info_get_nkeys

Name	Param name	Explanation	C type	F type	inout
<code>MPI_Info_get_nkeys (</code>					
	<code>info</code>	<code>info object</code>	<code>MPI_Info</code>	<code>TYPE (MPI_Info)</code>	<code>IN</code>
	<code>nkeys</code>	<code>number of defined keys</code>	<code>int*</code>	<code>INTEGER</code>	<code>OUT</code>
<code>)</code>					

Sample methods:

```
void set(const std::string &key, const std::string &value);
[[nodiscard]] std::optional<std::string> value(const std::string &key) const;
```

15.1.1.1 Environment information

The object `MPI_INFO_ENV` is predefined, containing:

- `command` Name of program executed.
- `argv` Space separated arguments to command.
- `maxprocs` Maximum number of MPI processes to start.
- `soft` Allowed values for number of processors.
- `host` Hostname.
- `arch` Architecture name.
- `wdir` Working directory of the MPI process.
- `file` Value is the name of a file in which additional information is specified.
- `thread_level` Requested level of thread support, if requested before the program started execution.

Note that these are the requested values; the running program can for instance have lower thread support.

15.1.1.2 Communicator and window information

MPI has a built-in possibility of attaching information to *communicators* and *windows* using the calls `MPI_Comm_get_info`, `MPI_Comm_set_info`, `MPI_Win_get_info`, `MPI_Win_set_info`.

Copying a communicator with `MPI_Comm_dup` does not cause the info to be copied; to propagate information to the copy there is `MPI_Comm_dup_with_info` (section 7.2).

15.1.1.3 File information

An `MPI_Info` object can be passed to the following file routines:

- `MPI_File_open`
- `MPI_File_set_view`
- `MPI_File_set_info`; collective. The converse routine is `MPI_File_get_info`.

The following keys are defined in the MPI-2 standard:

- `access_style`: A comma separated list of one or more of: `read_once`, `write_once`, `read_mostly`, `write_mostly`, `sequential`, `reverse_sequential`, `random`
- `collective_buffering`: true or false; enables or disables buffering on collective I/O operations
- `cb_block_size`: integer block size for collective buffering, in bytes
- `cb_buffer_size`: integer buffer size for collective buffering, in bytes
- `cb_nodes`: integer number of MPI processes used in collective buffering
- `chunked`: a comma separated list of integers describing the dimensions of a multidimensional array to be accessed using subarrays, starting with the most significant dimension (1st in C, last in Fortran)
- `chunked_item`: a comma separated list specifying the size of each array entry, in bytes
- `chunked_size`: a comma separated list specifying the size of the subarrays used in chunking
- `file_perm`: UNIX file permissions at time of creation, in octal
- `io_node_list`: a comma separated list of I/O nodes to use

The following material is for the recently released MPI-4 standard and may not be supported yet.

- `mpi_minimum_memory_alignment`: alignment of allocated memory.

End of MPI-4 material

- `nb_proc`: integer number of processes expected to access a file simultaneously
- `num_io_nodes`: integer number of I/O nodes to use
- `striping_factor`: integer number of I/O nodes/devices a file should be striped across
- `striping_unit`: integer stripe size, in bytes

Additionally, file system-specific keys can exist.

15.1.2 Attributes

Some runtime (or installation dependent) values are available as attributes through `MPI_Comm_set_attr` (figure 15.8) and `MPI_Comm_get_attr` (figure 15.9) for communicators, or `MPI_Win_get_attr`, `MPI_Type_get_attr`. (The MPI-2 routine `MPI_Attr_get` is deprecated). The flag parameter has two functions:

- it returns whether the attributed was found;
- if on entry it was set to false, the value parameter is ignored and the routines only tests whether the key is present.

The return value parameter is subtle: while it is declared `void*`, it is actually the address of a `void*` pointer.

```
// tags.c
int tag_upperbound;
void *v; int flag=1;
ierr = MPI_Comm_get_attr(comm,MPI_TAG_UB,&v,&flag);
tag_upperbound = *(int*)v;
```

Figure 15.8 MPI_Comm_set_attr

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_set_attr (
comm		communicator to which attribute will be attached	MPI_Comm	TYPE (MPI_Comm)	INOUT
comm_keyval		key value	int	INTEGER	IN
attribute_val		attribute value	void*	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
)					

Figure 15.9 MPI_Comm_get_attr

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_get_attr (
comm		communicator to which the attribute is attached	MPI_Comm	TYPE (MPI_Comm)	IN
comm_keyval		key value	int	INTEGER	IN
attribute_val		attribute value, unless flag = false	void*	INTEGER (KIND=MPI_ADDRESS_KIND)	OUT
flag		false if no attribute is associated with the key	int*	LOGICAL	OUT
)					

Python:

```
MPI.Comm.Get_attr(self, int keyval)
```

```
## tags.py
tag_upperbound = comm.Get_attr(MPI.TAG_UB)
if procid==0:
    print("Determined tag upperbound: {}".format(tag_upperbound))
```

Attributes are:

- **MPI_TAG_UB** Upper bound for *tag value*. (The lower bound is zero.) Note that **MPI_TAG_UB** is the key, not the actual upper bound! This value has to be at least 32767.
- **MPI_HOST** Host process rank, if such exists, **MPI_PROC_NULL**, otherwise. The standard does not define what it means to be a host, or even whether there should be one to begin with. This is deprecated as of MPI-4.1.
- **MPI_IO** rank of a process that has regular I/O facilities. Processes in the same communicator may return different values for this parameter. If this return **MPI_ANY_SOURCE**, all ranks can perform I/O.
- **MPI_WTIME_IS_GLOBAL** Boolean variable that indicates whether clocks are synchronized.

Also:

Figure 15.10 MPI_Comm_create_keyval

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_create_keyval (
comm_copy_attr_fn	copy callback function for comm_keyval		MPI_Comm_copy_attr_fn*	PROCEDURE IN (MPI_Comm_copy_attr_function)	
comm_delete_attr_fn	delete callback function for comm_keyval		MPI_Comm_delete_attr_fn*	PROCEDURE IN (MPI_Comm_delete_attr_function)	
comm_keyval	key value for future access	int*	INTEGER	OUT	
extra_state	extra state for callback function	void*	INTEGER	IN (KIND=MPI_ADDRESS_KIND)	
)					

- **MPI_UNIVERSE_SIZE**: the total number of processes that can be created. This can be more than the size of **MPI_COMM_WORLD** if the host list is larger than the number of initially started processes. See section 8.1.
- **MPI_APPNUM**: if MPI is used in MPMD mode (section 15.9.4), or if **MPI_Comm_spawn_multiple** is used (section 8.1), this attribute reports the how-manyeth program we are in.

Fortran note 16: Attribute querying. Fortran has none of this double indirection stuff. The value of the attribute is returned immediately, as an integer of kind **MPI_ADDRESS_KIND**:

```
!! tags.F90
logical :: flag
integer(KIND=MPI_ADDRESS_KIND) :: attr_v,tag_upperbound
call MPI_Comm_get_attr(comm,MPI_TAG_UB,attr_v,flag,ierr)
tag_upperbound = attr_v
print'("Determined tag upperbound: ",i9)', tag_upperbound
```

Python note 33: Universe size. `mpi4py.MPI.UNIVERSE_SIZE`.

15.1.3 Create new keyval attributes

Create a key value with **MPI_Comm_create_keyval** (figure 15.10), **MPI_Type_create_keyval**, **MPI_Win_create_keyval**. Use this key to set new attributes with **MPI_Comm_set_attr**, **MPI_Type_set_attr**, **MPI_Win_set_attr**. Free the attributed with **MPI_Comm_delete_attr**, **MPI_Type_delete_attr**, **MPI_Win_delete_attr**.

This uses a function type **MPI_Comm_attr_function**. This function is copied when a communicator is duplicated; section 7.2. Free with **MPI_Comm_free_keyval**.

15.1.4 Processor name

You can query the *hostname* of a processor with **MPI_Get_processor_name**. This name need not be unique between different processor ranks.

Figure 15.11 MPI_Get_version

Name	Param name	Explanation	C type	F type	inout
MPI_Get_version (
version	version number		int*	INTEGER	OUT
subversion	subversion number		int*	INTEGER	OUT
)					

You have to pass in the character storage: the character array must be at least `MPI_MAX_PROCESSOR_NAME` characters long. The actual length of the name is returned in the `resultlen` parameter.

15.1.5 Version information

For runtime determination, The *MPI version* is available through two parameters `MPI_VERSION` and `MPI_SUBVERSION` or the function `MPI_Get_version` (figure 15.11).

The library version can be queried with `MPI_Get_library_version`. The result string has to fit in `MPI_MAX_LIBRARY_VERSION_STRING`.

Python note 34: MPI Version. A function is available for version and subversion, as well as explicit parameters:

Code:

```
## version.py
print(MPI.Get_version())
print(MPI.VERSION)
print(MPI.SUBVERSION)
```

Output:

```
[examples/mpi/p] version:
(3, 1)
3
1
```

15.1.6 Python utility functions

Python note 35: Utility functions.

```
## util.py
print(f"Configuration:\n{mpi4py.get_config()}")
print(f"Include dir:\n{mpi4py.get_include()}")
Mac OS X with Python installed through macports:
```

```
Configuration:
{'mpicc': '/opt/local/bin/mpicc-mpich-mp'}
Include dir:
/opt/local/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-packages/mpi4py/include
```

Intel compiler and locally installed Python:

```
Configuration:
{'mpicc': '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpicc',
 'mpicxx': '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpicxx',
 'mpifort': '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpif90',
```

```
'mpif90': '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpif90',
'mpif77': '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpif77'}
Include dir:
/opt/apps/intel19/impi19_0/python3/3.9.2/lib/python3.9/site-packages/mpi4py/include
```

15.2 Error handling

Errors in normal programs can be tricky to deal with; errors in parallel programs can be even harder. This is because in addition to everything that can go wrong with a single executable (floating point errors, memory violation) you now get errors that come from faulty interaction between multiple executables.

A few examples of what can go wrong:

- MPI errors: an MPI routine can exit prematurely for various reasons, such as receiving much more data than its buffer can accommodate. Such errors, as well as the more common type mentioned above, typically cause your whole execution to terminate. That is, if one incarnation of your executable exits, the MPI runtime will kill all others.
- Deadlocks and other hanging executions: there are various scenarios where your processes individually do not exit, but are all waiting for each other. This can happen if two processes are both waiting for a message from each other, and this can be helped by using nonblocking calls. In another scenario, through an error in program logic, one process will be waiting for more messages (including nonblocking ones) than are sent to it.

While it is desirable for an MPI implementation to return an error, this is not always possible. Therefore, some scenarios, whether supplying certain procedure arguments, or doing a certain sequence of procedure calls, are simply marked as ‘erroneous’, and the state of MPI after an erroneous call is undefined.

15.2.1 Error codes

There are a bunch of error codes. These are all positive `int` values, while `MPI_SUCCESS` is zero. The maximum value of any built-in error code is `MPI_ERR_LASTCODE`. User-defined error codes are all larger than this.

- `MPI_ERR_ARG`: an argument was invalid that is not covered by another error code.
- `MPI_ERR_BUFFER`: The buffer pointer is invalid; this typically means that you have supplied a null pointer.
- `MPI_ERR_COMM`: invalid communicator. A common error is to use a null communicator in a call.
- `MPI_ERR_COUNT`: Invalid count argument, usually this is caused by a negative count value; zero is often a valid count.
- `MPI_ERR_INTERN`: An internal error in MPI has been detected.
- `MPI_ERR_IN_STATUS`: A functioning returning an array of statuses has at least one status where the `MPI_ERROR` field is set to other than `MPI_SUCCESS`. See section 4.3.3.
- `MPI_ERR_INFO`: invalid info object.
- `MPI_ERR_NO_MEM`: is returned by `MPI_Alloc_mem` if memory is exhausted.
- `MPI_ERR_OTHER`: an error occurred; use `MPI_Error_string` to retrieve further information about this error; see section 15.2.2.3.
- `MPI_ERR_PORT`: invalid port; this applies to `MPI_Comm_connect` and such.

The following material is for the recently released MPI-4 standard and may not be supported yet.

- `MPI_ERR_PROC_ABORTED`: is returned if a process tries to communicate with a process that has aborted.

End of MPI-4 material

- `MPI_ERR_RANK`: an invalid source or destination rank is specified. Valid ranks are $0 \dots s - 1$ where s is the size of the communicator, or `MPI_PROC_NULL`, or `MPI_ANY_SOURCE` for receive operations.
- `MPI_ERR_SERVICE`: invalid service in `MPI_Unpublish_name`; section 8.2.3.

15.2.2 Error handling

The MPI library has a general mechanism for dealing with errors that it detects: one can specify an error handler, specific to MPI objects.

- Most commonly, an error handler is associated with a communicator: `MPI_Comm_set_errhandler` (and likewise it can be retrieved with `MPI_Comm_get_errhandler`);
- other possibilities are `MPI_File_set_errhandler`, `MPI_File_call_errhandler`,

The following material is for the recently released MPI-4 standard and may not be supported yet.

`MPI_Session_set_errhandler`, `MPI_Session_call_errhandler`,

End of MPI-4 material

`MPI_Win_set_errhandler`, `MPI_Win_call_errhandler`.

Remark The routine `MPI_Errhandler_set` is deprecated, replaced by its MPI-2 variant `MPI_Comm_set_errhandler`.

Some handlers of type `MPI_Errhandler` are predefined (`MPI_ERRORS_ARE_FATAL`, `MPI_ERRORS_ABORT`, `MPI_ERRORS_RETURN`; see below), but you can define your own with `MPI_Errhandler_create`, to be freed later with `MPI_Errhandler_free`.

By default, MPI uses `MPI_ERRORS_ARE_FATAL`, except for file operations; see section 10.5.

Python note 36: Error policy. The policy for dealing with errors can be set through the `mpi4py.rc` object (section 2.2.2):

```
mpi4py.rc.errors # default: "exception"
```

Available levels are `exception`, `default`, `fatal`.

15.2.2.1 Abort

The default behavior, where the full run is aborted, is equivalent to your code having the following call to

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_ARE_FATAL);
```

The handler `MPI_ERRORS_ARE_FATAL`, even though it is associated with a communicator, causes the whole application to abort.

The following material is for the recently released MPI-4 standard and may not be supported yet.

The handler `MPI_ERRORS_ABORT` (MPI-4) aborts on the processes in the communicator for which it is specified.

End of MPI-4 material

15.2.2.2 Return

Another simple possibility is to specify `MPI_ERRORS_RETURN`:

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_RETURN);
```

which causes the error code to be returned to the user. This gives you the opportunity to write code that handles the error return value; see the next section.

15.2.2.3 Error printing

If the `MPI_Errhandler` value `MPI_ERRORS_RETURN` is used, you can compare the return code to `MPI_SUCCESS` and print out debugging information:

15. MPI leftover topics

Figure 15.12 MPI_Comm_create_errhandler

Name	Param name	Explanation	C type	F type	inout
MPI_Comm_create_errhandler	(comm_errhandler_fn errhandler)	user defined error handling procedure MPI error handler	MPI_Comm_errhandler PROCEDURE* IN (MPI_Comm_errhandler_function) MPI_Errhandler* TYPE OUT (MPI_Errhandler)		

```
int ierr;  
ierr = MPI_Something();  
if (ierr!=MPI_SUCCESS) {  
    // print out information about what your programming is doing  
    MPI_Abort();  
}
```

For instance,

```
Fatal error in MPI_Waitall:  
See the MPI_ERROR field in MPI_Status for the error code  
You could then retrieve the MPI_ERROR field of the status, and print out an error string with MPI_Error_string or maximal size MPI_MAX_ERROR_STRING:
```

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_RETURN);  
ierr = MPI_Waitall(2*ntids-2,requests,status);  
if (ierr!=0) {  
    char errtxt[MPI_MAX_ERROR_STRING];  
    for (int i=0; i<2*ntids-2; i++) {  
        int err = status[i].MPI_ERROR;  
        int len=MPI_MAX_ERROR_STRING;  
        MPI_Error_string(err,errtxt,&len);  
        printf("Waitall error: %d %s\n",err,errtxt);  
    }  
    MPI_Abort(MPI_COMM_WORLD,0);  
}
```

One cases where errors can be handled is that of *MPI file I/O*: if an output file has the wrong permissions, code can possibly progress without writing data, or writing to a temporary file.

MPI operators ([MPI_Op](#)) do not return an error code. In case of an error they call [MPI_Abort](#); if [MPI_ERRORS_RETURN](#) is the error handler, error codes may be silently ignored.

You can create your own error handler with [MPI_Comm_create_errhandler](#) (figure 15.12), which is then installed with [MPI_Comm_set_errhandler](#). You can retrieve the error handler with [MPI_Comm_get_errhandler](#).

MPL note 80: Communicator errhandler. MPL does not have a routine for setting the error handler. Instead, use the `native_handle` method to retrieve the embedded communicator.

15.2.3 Defining your own MPI errors

You can define your own errors that behave like MPI errors. As an example, let's write a send routine that refuses to send zero-sized data.

The first step to defining a new error is to define an error class with `MPI_Add_error_class`:

```
int nonzero_class;
MPI_Add_error_class(&nonzero_class);
```

This error number is larger than `MPI_ERR_LASTCODE`, the upper bound on built-in error codes. The attribute `MPI_LASTUSEDCODE` records the last issued value.

Your new error code is then defined in this class with `MPI_Add_error_code`, and an error string can be added with `MPI_Add_error_string`:

```
int nonzero_code;
MPI_Add_error_code(nonzero_class,&nonzero_code);
MPI_Add_error_string(nonzero_code,"Attempting to send zero buffer");
```

You can then call an error handler with this code. For instance to have a wrapped send routine that will not send zero-sized messages:

```
// errorclass.c
int MyPI_Send( void *buffer,int n,MPI_Datatype type, int target,int tag,MPI_Comm comm) {
    if (n==0)
        MPI_Comm_call_errhandler( comm,nonzero_code );
    MPI_Ssend(buffer,n,type,target,tag,comm);
    return MPI_SUCCESS;
};
```

Here we used the default error handler associated with the communicator, but one can set a different one with `MPI_Comm_create_errhandler`.

We test our example:

```
for (int msgsize=1; msgsize>=0; msgsize--) {
    double buffer;
    if (procno==0) {
        printf("Trying to send buffer of length %d\n",msgsize);
        MyPI_Send(&buffer,msgsize,MPI_DOUBLE, 1,0,comm);
        printf(.. success\n");
    } else if (procno==1) {
        MPI_Recv (&buffer,msgsize,MPI_DOUBLE, 0,0,comm,MPI_STATUS_IGNORE);
    }
}
```

which gives:

```
Trying to send buffer of length 1
.. success
Trying to send buffer of length 0
Abort(1073742081) on node 0 (rank 0 in comm 0):
Fatal error in MPI_Comm_call_errhandler: Attempting to send zero buffer
```

15.3 Fortran issues

MPI is typically written in C, what if you program *Fortran*?

See section 6.2.2.1 for MPI types corresponding to *Fortran90 types*.

15.3.1 Assumed-shape arrays

Use of other than contiguous data, for instance `A(1:N:2)`, was a problem in MPI calls, especially nonblocking ones. In that case it was best to copy the data to a contiguous array. This has been fixed in MPI-3.

- Fortran routines have the same signature as C routines except for the addition of an integer error parameter.
- The call for `MPI_Init` in Fortran does not have the commandline arguments; they need to be handled separately.
- The routine `MPI_Sizeof` is only available in Fortran, it provides the functionality of the C/C++ operator `sizeof`.

15.3.2 Prevent compiler optimizations

The Fortran compiler can aggressively optimize by rearranging instructions. This may lead to incorrect behavior in MPI code. In the sequence:

```
call MPI_Isend( buf, ..., request )
call MPI_Wait(request)
print *,buf(1)
```

the wait call does not involve the buffer, so the compiler can translate this into

```
call MPI_Isend( buf, ..., request )
register = buf(1)
call MPI_Wait(request)
print *,register
```

Preventing this is possible with a Fortran2018 mechanism. First of all the buffer should be declared `asynchronous <type>,Asynchronous :: buf`

and introducing

```
IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING) &
    CALL MPI_F_SYNC_REG( buf )
```

The call to `MPI_F_sync_reg` will be removed at compile time if `MPI_ASYNC_PROTECTS_NONBLOCKING` is true.

15.4 Progress

The concept *asynchronous progress* describes that MPI messages continue on their way through the network, while the application is otherwise busy.

The problem here is that, unlike straight `MPI_Send` and `MPI_Recv` calls, communication of this sort can typically not be off-loaded to the network card, so different mechanisms are needed.

This can happen in a number of ways:

- Compute nodes may have a dedicated communications processor. The *Intel Paragon* was of this design; modern multicore processors are a more efficient realization of this idea.
- The MPI library may reserve a core or thread for communications processing. This is implementation dependent; see *Intel MPI* information below.

- Reserving a core, or a thread in a continuous busy-wait *spin loop*, takes away possible performance from the code. For this reason, Ruhela *et al.* [24] propose using a *pthreads* signal to wake up the progress thread.
- Absent such dedicated resources, the application can force MPI to make progress by occasional calls to a *polling* routine such as `MPI_Iprobe`.

Remark The `MPI_Probe` call is somewhat similar, in spirit if not quite in functionality, as `MPI_Test`. However, they behave differently with respect to progress. Quoting the standard:

The MPI implementation of `MPI_Probe` and `MPI_Iprobe` needs to guarantee progress: if a call to `MPI_Probe` has been issued by a process, and a send that matches the probe has been initiated by some process, then the call to `MPI_Probe` will return.

In other words: probing causes MPI to make progress. On the other hand,

A call to `MPI_Test` returns flag = true if the operation identified by request is complete.

In other words, if progress has been made, then testing will report completion, but by itself it does not cause completion.

A similar problem arises with passive target synchronization: it is possible that the origin process may hang until the target process makes an MPI call.

The following commands force progress: `MPI_Win_test`, `MPI_Request_get_status`.

Intel note. Only available with the `release_mt` and `debug_mt` versions of the Intel MPI library. Set `I_MPI_ASYNC_PROGRESS` to 1 to enable asynchronous progress threads, and `I_MPI_ASYNC_PROGRESS_THREADS` to set the number of progress threads.

See <https://software.intel.com/en-us/mpi-developer-guide-linux-asynchronous-progress-control>,
<https://software.intel.com/en-us/mpi-developer-reference-linux-environment-variables-for-asynchronous->

Progress issues play with: `MPI_Test`, `MPI_Request_get_status`, `MPI_Win_test`.

15.5 Fault tolerance

Processors are not completely reliable, so it may happen that one ‘breaks’: for software or hardware reasons it becomes unresponsive. For an MPI program this means that it becomes impossible to send data to it, and any collective operation involving it will hang. Can we deal with this case? Yes, but it involves some programming.

First of all, one of the possible MPI error return codes (section 15.2) is `MPI_ERR_COMM`, which can be returned if a processor in the communicator is unavailable. You may want to catch this error, and add a ‘replacement processor’ to the program. For this, the `MPI_Comm_spawn` can be used (see 8.1 for details). But this requires a change of program design: the communicator containing the new process(es) is not part of the old `MPI_COMM_WORLD`, so it is better to set up your code as a collection of inter-communicators to begin with.

15.6 Performance, tools, and profiling

In most of this book we talk about functionality of the MPI library. There are cases where a problem can be solved in more than one way, and then we wonder which one is the most efficient. In this section we will explicitly address performance. We start with two sections on the mere act of measuring performance.

Figure 15.13 MPI_Wtime

Name	Param name	Explanation	C type	F type	inout
MPI_Wtime ()					

Python:

```
MPI.Wtime()
```

Figure 15.14 MPI_Wtick

Name	Param name	Explanation	C type	F type	inout
MPI_Wtick ()					

Python:

```
MPI.Wtick()
```

15.6.1 Timing

MPI has a *wall clock* timer: **MPI_Wtime** (figure 15.13) which gives the number of seconds from a certain point in the past. (Note the absence of the error parameter in the fortran call.)

```
double t;
t = MPI_Wtime();
for (int n=0; n<NEXPERIMENTS; n++) {
    // do something;
}
t = MPI_Wtime()-t; t /= NEXPERIMENTS;
```

The timer has a resolution of **MPI_Wtick** (figure 15.14).

MPL note 81: Timing. The timing routines **wtime** and **wtick** and **wtime_is_global** are environment methods:

```
double mpl::environment::wtime ();
double mpl::environment::wtick ();
bool mpl::environment::wtime_is_global ();
```

Timing in parallel is a tricky issue. For instance, most clusters do not have a central clock, so you can not relate start and stop times on one process to those on another. You can test for a global clock as follows **MPI_WTIME_IS_GLOBAL**:

```
int *v,flag;
MPI_Attr_get( comm, MPI_WTIME_IS_GLOBAL, &v, &flag );
if (mytid==0) printf("Time synchronized? %d->%d\n",flag,*v);
```

Normally you don't worry about the starting point for this timer: you call it before and after an event and subtract the values.

```
t = MPI_Wtime();
// something happens here
t = MPI_Wtime()-t;
```

If you execute this on a single processor you get fairly reliable timings, except that you would need to subtract the overhead for the timer. This is the usual way to measure timer overhead:

```
t = MPI_Wtime();  
// absolutely nothing here  
t = MPI_Wtime()-t;
```

15.6.1.1 Global timing

However, if you try to time a parallel application you will most likely get different times for each process, so you would have to take the average or maximum. Another solution is to synchronize the processors by using a *barrier* through `MPI_Barrier`:

```
MPI_Barrier(comm)  
t = MPI_Wtime();  
// something happens here  
MPI_Barrier(comm)  
t = MPI_Wtime()-t;
```

Exercise 15.1. This scheme also has some overhead associated with it. How would you measure that?

15.6.1.2 Local timing

Now suppose you want to measure the time for a single send. It is not possible to start a clock on the sender and do the second measurement on the receiver, because the two clocks need not be synchronized. Usually a *ping-pong* is done:

```
if ( proc_source ) {  
    MPI_Send( /* to target */ );  
    MPI_Recv( /* from target */ );  
} else if ( proc_target ) {  
    MPI_Recv( /* from source */ );  
    MPI_Send( /* to source */ );  
}
```

No matter what sort of timing you are doing, it is good to know the accuracy of your timer. The routine `MPI_Wtick` gives the smallest possible timer increment. If you find that your timing result is too close to this ‘tick’, you need to find a better timer (for CPU measurements there are cycle-accurate timers), or you need to increase your running time, for instance by increasing the amount of data.

15.6.2 Simple profiling

Remark This section describes MPI profiling before the introduction of the MPI tools interface. For that, see chapter 14.

MPI allows you to write your own profiling interface. To make this possible, every routine `MPI_Something` calls a routine `PMPI_Something` that does the actual work. You can now write your `MPI_...` routine which calls `PMPI_...`, and inserting your own profiling calls. See figure 15.1.

By default, the MPI routines are defined as *weak linker symbols* as a synonym of the PMPI ones. In the gcc case:

```
#pragma weak MPI_Send = PMPI_Send
```

As you can see in figure 15.2, normally only the PMPI routines show up in the stack trace.

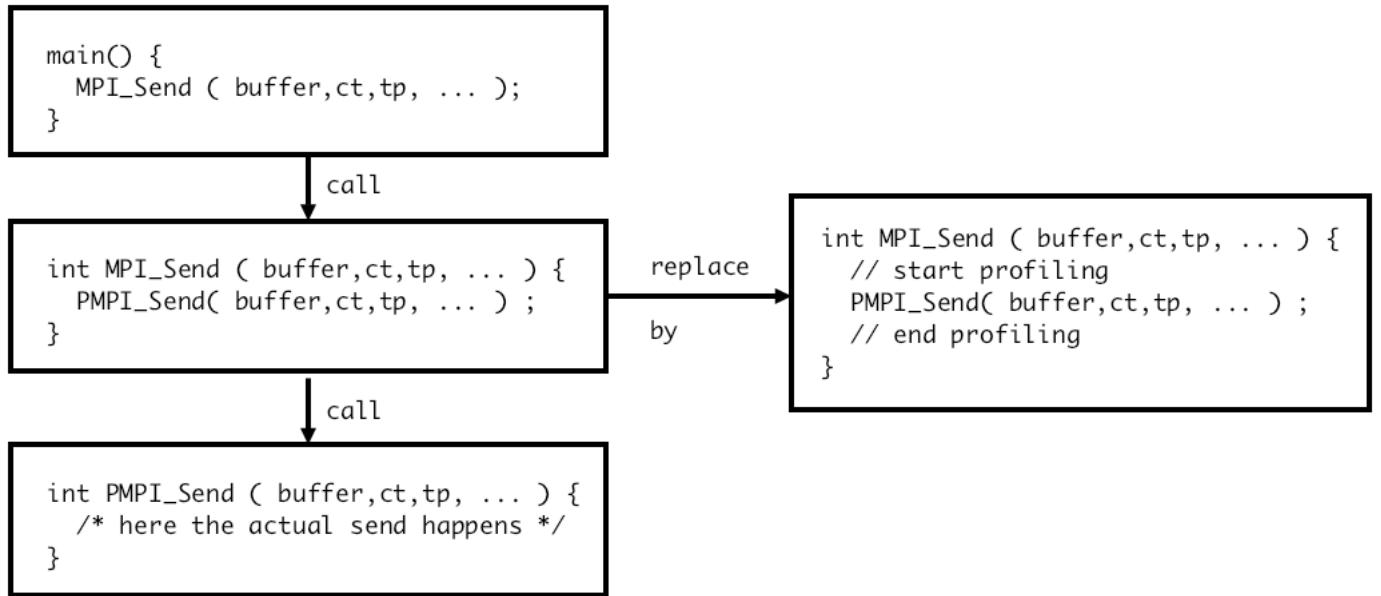


Figure 15.1: Calling hierarchy of MPI and PMPI routines

15.6.3 Programming for performance

We outline some issues pertaining to performance.

Eager limit Short blocking messages are handled by a simpler mechanism than longer. The limit on what is considered ‘short’ is known as the *eager limit* (section 4.1.4.2), and you could tune your code by increasing its value. However, note that a process may likely have a buffer accomodating eager sends for every single other process. This may eat into your available memory.

Blocking versus nonblocking The issue of *blocking versus nonblocking* communication is something of a red herring. While nonblocking communication allows *latency hiding*, we can not consider it an alternative to blocking sends, since replacing nonblocking by blocking calls will usually give *deadlock*.

Still, even if you use nonblocking communication for the mere avoidance of deadlock or serialization (section 4.1.4.3), bear in mind the possibility of overlap of communication and computation. This also brings us to our next point.

Looking at it the other way around, in a code with blocking sends you may get better performance from nonblocking, even if that is not structurally necessary.

Progress MPI is not magically active in the background, especially if the user code is doing scalar work that does not involve MPI. As sketched in section 15.4, there are various ways of ensuring that latency hiding actually happens.

Persistent sends If a communication between the same pair of processes, involving the same buffer, happens regularly, it is possible to set up a *persistent communication*. See section 5.1.

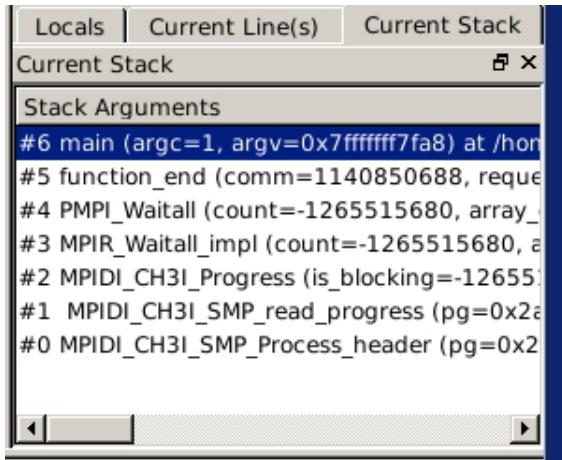


Figure 15.2: A stack trace, showing the MPPI calls.

Buffering MPI uses internal buffers, and the copying from user data to these buffers may affect performance. For instance, derived types (section 6.3) can typically not be streamed straight through the network (this requires special hardware support [19]) so they are first copied. Somewhat surprisingly, we find that *buffered communication* (section 5.5) does not help. Perhaps MPI implementors have not optimized this mode since it is so rarely used.

This issue is extensively investigated in [10].

Graph topology and neighborhood collectives Load balancing and communication minimization are important in irregular applications. There are dedicated programs for this (*ParMetis*, *Zoltan*), and libraries such as *PETSc* may offer convenient access to such capabilities.

In the declaration of a *graph topology* (section 11.2) MPI is allowed to reorder processes, which could be used to support such activities. It can also serve for better message sequencing when *neighborhood collectives* are used.

Network issues In the discussion so far we have assumed that the network is a perfect conduit for data. However, there are issues of port design, in particular caused by *oversubscription* that adversely affect performance. While in an ideal world it may be possible to set up routine to avoid this, in the actual practice of a supercomputer cluster, *network contention* or *message collision* from different user jobs is hard to avoid.

Offloading and onloading There are different philosophies of *network card design*: *Mellanox*, being a network card manufacturer, believes in off-loading network activity to the Network Interface Card (NIC), while *Intel*, being a processor manufacturer, believes in ‘on-loading’ activity to the process. There are arguments either way.

Either way, investigate the capabilities of your network.

15.6.4 MPIR

MPIR is the informally specified debugging interface for processes acquisition and message queue extraction.

15.7 Determinism

MPI processes are only synchronized to a certain extent, so you may wonder what guarantees there are that running a code twice will give the same result. You need to consider two cases: first of all, if the two runs are on different numbers of processors there are already numerical problems; see HPC book, section ??.

Let us then limit ourselves to two runs on the same set of processors. In that case, MPI is deterministic as long as you do not use wildcards such as `MPI_ANY_SOURCE`. Formally, MPI messages are ‘nonovertaking’: two messages between the same sender-receiver pair will arrive in sequence. Actually, they may not arrive in sequence: they are *matched* in sequence in the user program. If the second message is much smaller than the first, it may actually arrive earlier in the lower transport layer.

Another source of non-determinism comes from hybrid computing; see section 45.1.

15.8 Subtleties with processor synchronization

Blocking communication involves a complicated dialog between the two processors involved. Processor one says ‘I have this much data to send; do you have space for that?’, to which processor two replies ‘yes, I do; go ahead and send’, upon which processor one does the actual send. This back-and-forth (technically known as a *handshake*) takes a certain amount of communication overhead. For this reason, network hardware will sometimes forgo the handshake for small messages, and just send them regardless, knowing that the other process has a small buffer for such occasions.

One strange side-effect of this strategy is that a code that should *deadlock* according to the MPI specification does not do so. In effect, you may be shielded from your own programming mistake! Of course, if you then run a larger problem, and the small message becomes larger than the threshold, the deadlock will suddenly occur. So you find yourself in the situation that a bug only manifests itself on large problems, which are usually harder to debug. In this case, replacing every `MPI_Send` with a `MPI_Ssend` will force the handshake, even for small messages.

Conversely, you may sometimes wish to avoid the handshake on large messages. MPI as a solution for this: the `MPI_Rsend` (‘ready send’) routine sends its data immediately, but it needs the receiver to be ready for this. How can you guarantee that the receiving process is ready? You could for instance do the following (this uses nonblocking routines, which are explained below in section 4.2.1):

```
if ( receiving ) {
    MPI_Irecv() // post nonblocking receive
    MPI_Barrier() // synchronize
} else if ( sending ) {
    MPI_Barrier() // synchronize
    MPI_Rsend() // send data fast
```

When the barrier is reached, the receive has been posted, so it is safe to do a ready send. However, global barriers are not a good idea. Instead you would just synchronize the two processes involved.

Exercise 15.2. Give pseudo-code for a scheme where you synchronize the two processes through the exchange of a blocking zero-size message.

15.9 Shell interaction

MPI programs are not run directly from the shell, but are started through an *ssh tunnel*. We briefly discuss ramifications of this.

15.9.1 Standard input

Letting MPI processes interact with the environment is not entirely straightforward. For instance, *shell input redirection* as in

```
mpiexec -n 2 mpiprogram < someinput
```

may not work.

Instead, use a script `programscript` that has one parameter:

```
#!/bin/bash  
mpirunprogram < $1
```

and run this in parallel:

```
mpiexec -n 2 programscript someinput
```

15.9.2 Standard out and error

The `stdout` and `stderr` streams of an MPI process are returned through the ssh tunnel. Thus they can be caught as the `stdout/err` of `mpiexec`.

```
// outerr.c  
fprintf(stdout,"This goes to std out\n");  
fprintf(stderr,"This goes to std err\n");
```

The name of the variable is implementation dependent, for `mpich` and its derivates such as *Intel MPI* it is `PMI_RANK`. (There is a similar `PMI_SIZE`.)

If you are only interested in displaying the rank

- `srun` has an option `--label`.

15.9.3 Process status

The return code of `MPI_Abort` is returned as the *processes status* of `mpiexec`. Running

```
// abort.c  
if (procno==nprocs-1)  
    MPI_Abort(comm,37);
```

as

```
mpiexec -n 4 ./abort ; \  
echo "Return code from ${MPIRUN} is <<$$?>>"
```

gives

```
TACC: Starting up job 3760534  
TACC: Starting parallel tasks...  
application called MPI_Abort(MPI_COMM_WORLD, 37) - process 3  
TACC: MPI job exited with code: 37  
TACC: Shutdown complete. Exiting.  
Return code from ibrun is <<37>>
```

15.9.4 Multiple program start

If the MPI application consists of sub-applications, that is, if we have a true *MPMD* runs, there are usually two ways of starting this up. (Once started, each process can retrieve with `MPI_APPNUM` to which application it belongs.)

The first possibility is that the job starter, `mpiexec` or `mpirun` or a local variant, accepts multiple executables:

```
mpiexec spec0 [ : spec1 [ : spec2 : ... ] ]
```

Absent this mechanism, the sort of script of section 15.9.1 can also be used to implement *MPMD* runs. We let the script start one of a number of programs, and we use the fact that the MPI rank is known in the environment; see section 15.9.2.

Use a script `mpmdscript`:

```
#!/bin/bash
rank=$PMI_RANK
half=$(( ${PMI_SIZE} / 2 ))
if [ $rank -lt $half ] ; then
    ./prog1
else
    ./prog2
fi
```

TACC: Starting up job 4032931
TACC: Starting parallel tasks...
Program 1 has process 1 out of 4
Program 2 has process 2 out of 4
Program 2 has process 3 out of 4
Program 1 has process 0 out of 4
TACC: Shutdown complete. Exiting.

This script is run in parallel:

```
mpiexec -n 25 mpmdscript
```

15.10 Leftover topics

15.10.1 MPI constants

MPI has a number of built-in *constants*. These do not all behave the same.

- Some are *compile-time* constants. Examples are `MPI_VERSION` and `MPI_MAX_PROCESSOR_NAME`. Thus, they can be used in array size declarations, even before `MPI_Init`.
- Some *link-time* constants get their value by MPI initialization, such as `MPI_COMM_WORLD`. Such symbols, which include all predefined handles, can be used in initialization expressions.
- Some link-time symbols can not be used in initialization expressions, such as `MPI_BOTTOM` and `MPI_STATUS_IGNORE`.

For symbols, the binary realization is not defined. For instance, `MPI_COMM_WORLD` is of type `MPI_Comm`, but the implementation of that type is not specified.

See Annex A of the MPI-3.1 standard for full lists.

The following are the compile-time constants:

- `MPI_MAX_PROCESSOR_NAME`
- `MPI_MAX_LIBRARY_VERSION_STRING`
- `MPI_MAX_ERROR_STRING`

- `MPI_MAX_DATAREP_STRING`
- `MPI_MAX_INFO_KEY`
- `MPI_MAX_INFO_VAL`
- `MPI_MAX_OBJECT_NAME`
- `MPI_MAX_PORT_NAME`
- `MPI_VERSION`
- `MPI_SUBVERSION`

Fortran note 17: Fortran-only compile-time constants.

- `MPI_STATUS_SIZE`. No longer needed with Fortran2008 support; see section 9.
- `MPI_ADDRESS_KIND`
- `MPI_COUNT_KIND`
- `MPI_INTEGER_KIND`
- `MPI_OFFSET_KIND`
- `MPI_SUBARRAYS_SUPPORTED`
- `MPI_ASYNC_PROTECTS_NONBLOCKING`

The following are the link-time constants:

- `MPI_BOTTOM`
- `MPI_STATUS_IGNORE`
- `MPI_STATUSES_IGNORE`
- `MPI_ERRCODES_IGNORE`
- `MPI_IN_PLACE`
- `MPI_ARGV_NULL`
- `MPI_ARGVS_NULL`
- `MPI_UNWEIGHTED`
- `MPI_WEIGHTS_EMPTY`

Assorted constants:

- `MPI_PROC_NULL` and other `..._NULL` constants.
- `MPI_ANY_SOURCE`
- `MPI_ANY_TAG`
- `MPI_UNDEFINED`
- `MPI_BSEND_OVERHEAD`
- `MPI_KEYVAL_INVALID`
- `MPI_LOCK_EXCLUSIVE`
- `MPI_LOCK_SHARED`
- `MPI_ROOT`

(This section was inspired by <http://blogs.cisco.com/performance/mpi-outside-of-c-and-fortran>.)

15.10.2 Cancelling messages

In section 4.3.1 we showed a master-worker example where the master accepts in arbitrary order the messages from the workers. Here we will show a slightly more complicated example, where only the result of the first task to complete is needed. Thus, we issue an `MPI_Recv` with `MPI_ANY_SOURCE` as source. When a result comes, we broadcast its source to all processes. All the other workers then use this information to cancel their message with an `MPI_Cancel` operation.

```
// cancel.c
fprintf(stderr,"get set, go!\n");
if (procno==nprocs-1) {
    MPI_Status status;
    MPI_Recv(dummy,0,MPI_INT, MPI_ANY_SOURCE,0,comm,
              &status);
    first_tid = status.MPI_SOURCE;
    MPI_Bcast(&first_tid,1,MPI_INT, nprocs-1,comm);
    fprintf(stderr,"%d first msg came from %d\n",procno,first_tid);
} else {
    float randomfraction = (rand() / (double)RAND_MAX);
    int randomwait = (int) ( nprocs * randomfraction );
    MPI_Request request;
    fprintf(stderr,"%d waits for %e/%d=%d\n",
            procno,randomfraction,nprocs,randomwait);
    sleep(randomwait);
    MPI_Isend(dummy,0,MPI_INT, nprocs-1,0,comm,
              &request);
    MPI_Bcast(&first_tid,1,MPI_INT, nprocs-1,comm
              );
    if (procno!=first_tid) {
        MPI_Cancel(&request);
        fprintf(stderr,"%d canceled\n",procno);
    }
}
```

After the cancelling operation it is still necessary to call `MPI_Request_free`, `MPI_Wait`, or `MPI_Test` in order to free the request object.

The `MPI_Cancel` operation is local, so it can not be used for *nonblocking collectives* or one-sided transfers.

Remark As of MPI-3.2, *cancelling a send is deprecated*.

15.10.3 The origin of one-sided communication in ShMem

The *Cray T3E* had a library called *shmem* which offered a type of shared memory. Rather than having a true global address space it worked by supporting variables that were guaranteed to be identical between processors, and indeed, were guaranteed to occupy the same location in memory. Variables could be declared to be shared a ‘symmetric’ pragma or directive; their values could be retrieved or set by `shmem_get` and `shmem_put` calls.

15.11 Literature

Online resources:

- MPI 1 Complete reference:
<http://www.netlib.org/utk/papers/mpi-book/mpi-book.html>
- Official MPI documents:
<http://www.mpi-forum.org/docs/>
- List of all MPI routines:
<http://www.mcs.anl.gov/research/projects/mpi/www/www3/>

Tutorial books on MPI:

- Using MPI [13] by some of the original authors.

Chapter 16

MPI Examples

16.1 Bandwidth and halfbandwidth

Bandwidth is the quantity that measures the number of bytes per second that can go through a connection. This definition seems straightforward, but comes with many footnotes.

- The size of the message used matters, since there is a *latency* cost to merely starting the message. Often, the bandwidth number quoted is an asymptotic figure, hard to achieve in practice.
- If a certain bandwidth figure is attained between a pair of processes, will two pairs, sending simultaneously, reach the same number?
- Does the bandwidth depend on the choice of processes to measure?
- And combinations of these considerations.

A useful measure comes from asking what bandwidth is achievable if all processes are either sending or receiving. As a further refinement, we ask what the least favorable choice is for the communicating pairs:

Bisection bandwidth is defined as the minimum total bandwidth, over all possible choices of splitting the processes into a sending and receiving half.

See also HPC book, section ??.

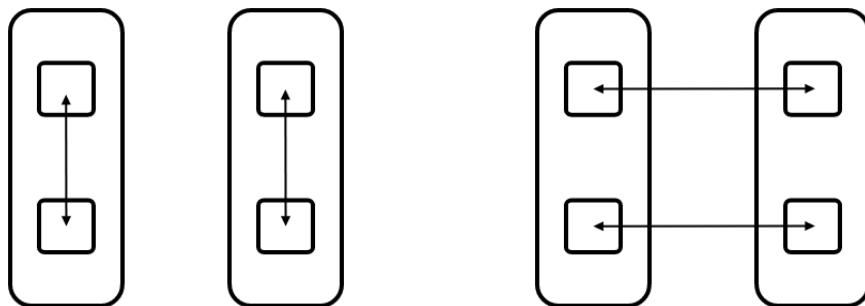


Figure 16.1: Intra and inter schemes for bandwidth

Figure 16.1 illustrates the ‘intra’ (left) and ‘inter’ (right) scheme for letting all processes communicate in pairs. With intra-communication, the messages do not rely on the network so we expect to measure high bandwidth. With inter-communication, all messages go through the network and we expect to measure a lower number.

However, there are more issues to explore, which we will now do.

First of all we need to find pairs of processes. Consecutive pairs:

```
// halfbandwidth.cxx
int sender = procid - procid%2, receiver = sender+1;
```

Pairs that are $P/2$ apart:

```
int sender = procid<halfprocs ? procid : procid-halfprocs,
receiver = sender + halfprocs;
```

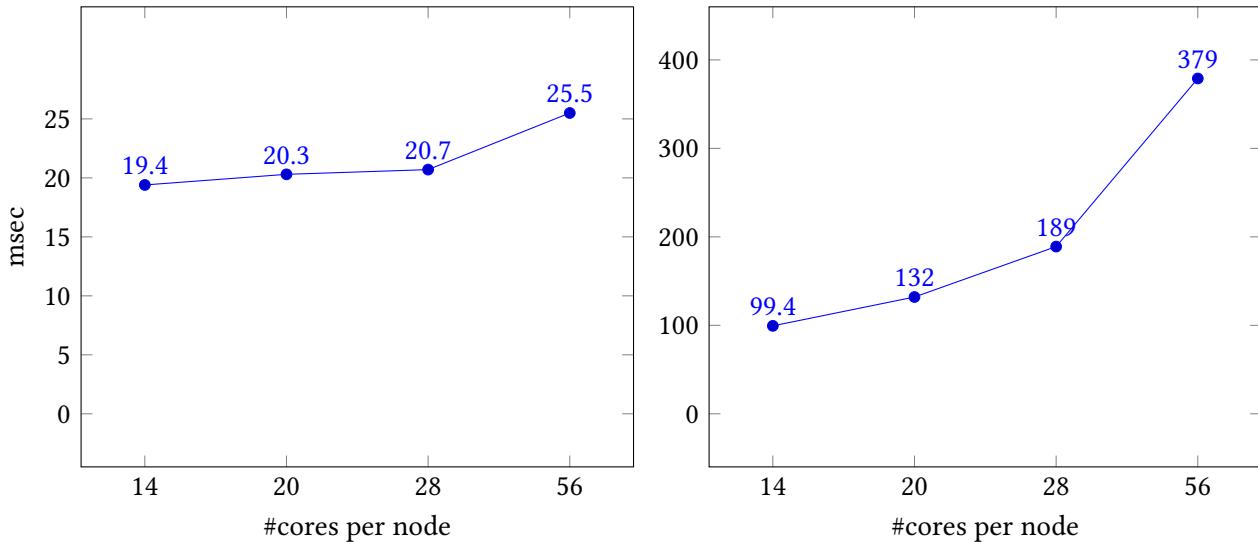


Figure 16.2: Time as a function of core count. Left: on node. Right: between nodes.

The halfbandwidth is measured as the total number of bytes sent divided by the total time. Both numbers are measured outside a repeat loop that does each transaction 100 times.

```
auto duration = myclock::now()-start_time;
auto microsec_duration = std::chrono::duration_cast<std::chrono::microseconds>(duration);
int total_ping_count;
MPI_Allreduce(&pingcount,&total_ping_count,1,MPI_INT,MPI_SUM,comm);
long bytes = buffersize * sizeof(double) * total_ping_count;
float fsec = microsec_duration.count() * 1.e-6,
halfbandwidth = bytes / fsec;
```

In the left graph of figure 16.2 we see that the time for $P/2$ simultaneous pingpongs stays fairly constant. This reflects the fact that, on node, the pingpong operations are data copies, which proceed simultaneously. Thus, the time is independent of the number of cores that are moving data. The exception is the final data point: with all cores active we take up more than the available bandwidth on the node.

In the right graph, each pingpong is inter-node, going through the network. Here we see the runtime go up linearly with the number of pingpongs, or somewhat worse than that. This reflects the fact that network transfers are done sequentially. (Actually, message can be broken up in packets, as long as they satisfy MPI message semantics. This does not alter our argument.)

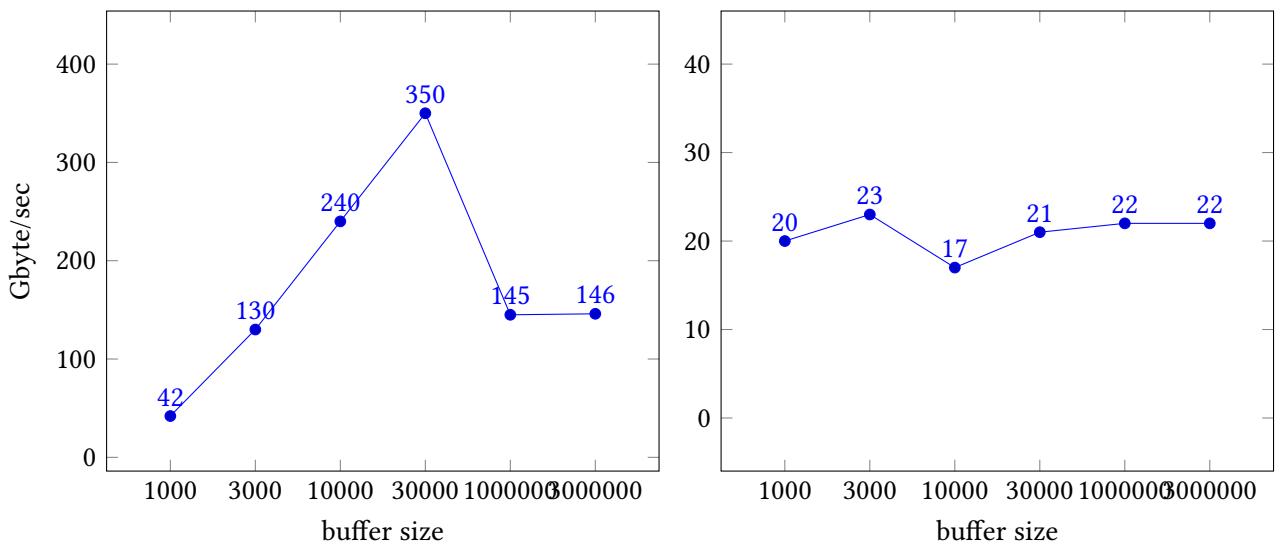


Figure 16.3: Bandwidth as a function of buffer size. Left: on node. Right: between nodes.

Next we explore the influence of the buffer size on performance. The right graph in figure 16.3 show that inter-node bandwidth is almost independent of the buffer size. This means that even our smallest buffer is large enough to overcome any MPI startup cost.

On other hand, the left graph shows a more complicated pattern. Initially, the bandwidth increases, possibly reflecting the decreasing importance of MPI startup. For the final data points, however, performance drops again. This is due to the fact that the data size overflows cache size, and we are dominated by bandwidth from memory, rather than cache.

PART II

OPENMP

This section of the book teaches OpenMP ('Open Multi Processing'), the dominant model for shared memory programming in science and engineering. It will instill the following competencies.

Basic level:

- Threading model: the student will understand the threading model of OpenMP, and the relation between threads and cores (chapter 17); the concept of a parallel region and private versus shared data (chapter 18).
- Loop parallelism: the student will be able to parallelize loops, and understand the impediments to parallelization, and iteration scheduling (chapter 19; reductions (chapter 20).
- The student will understand the concept of worksharing constructs, and its implications for synchronization (chapter 21).

Intermediate level:

- The student will understand the abstract notion of synchronization, its implementations in OpenMP, and implications for performance (chapter 23).
- The student will understand the task model as underlying the thread model, be able to write code that spawns tasks, and be able to distinguish when tasks are needed versus simpler worksharing constructs (chapter 24).
- The student will understand thread/code affinity, how to control it, and possible implications for performance (chapter 25).

Advanced level:

- The student will understand the OpenMP memory model, and sequential consistency (chapter 28.8).
- The student will understand SIMD processing, the extent to which compilers do this outside of OpenMP, and how OpenMP can specify further opportunities for SIMD-ization (chapter 26).
- The student will understand offloading to Graphics Processing Units (GPUs), and the OpenMP directives for effecting this (chapter 27).

Chapter 17

Getting started with OpenMP

This chapter explains the basic concepts of OpenMP, and helps you get started on running your first OpenMP program.

17.1 The OpenMP model

We start by establishing a mental picture of the hardware and software that OpenMP targets.

17.1.1 Target hardware

Modern computers have a multi-layered design. Maybe you have access to a cluster, and maybe you have learned how to use MPI to communicate between cluster nodes. OpenMP, the topic of this chapter, is concerned with a single *cluster node* and getting the most out of the available parallelism available there.



Figure 17.1: A node with two sockets and a co-processor

Figure 17.1 pictures a typical design of a node: within one enclosure you find two *sockets*, single processor chips, plus an *accelerator*. (The picture is of a node of the *TACC Stampede* cluster no longer in service, with two sockets and an *Intel Xeon PHI* co-processor.)

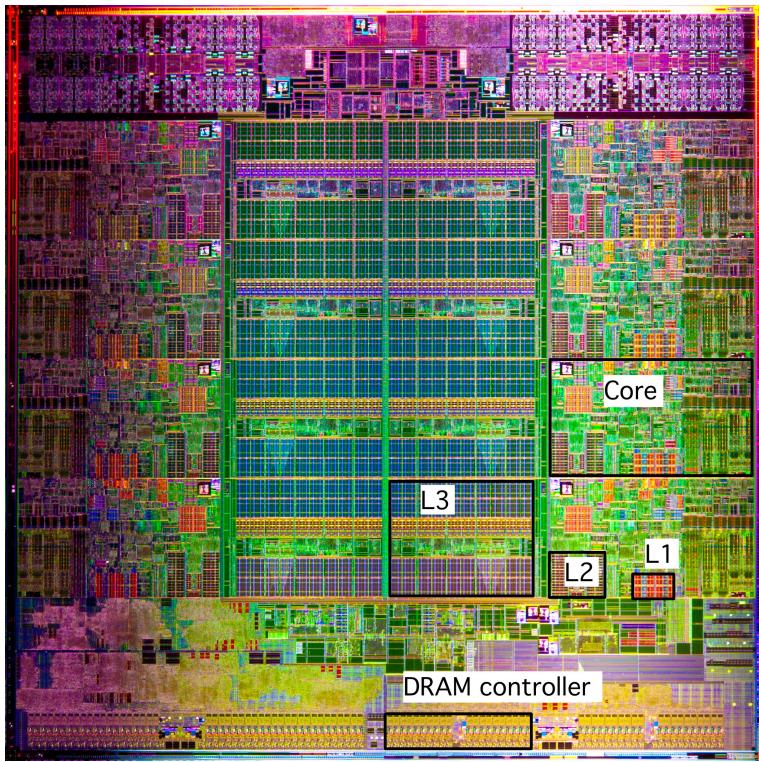


Figure 17.2: Structure of an Intel Sandybridge eight-core socket

Your personal laptop or desktop computer will probably have one socket; while most supercomputers have nodes with two or four sockets. In either case there can be a GPU as co-processor; supercomputer clusters can also have other types of accelerators. OpenMP versions as of OpenMP-4.0 target such offloadable devices.

To see what aspects of this architecture OpenMP addresses we need to dig into the sockets. Figure 17.2 shows a picture of an *Intel Sandybridge* socket. You recognize a structure with eight *cores*: independent processing units, that all have access to the same memory. (In figure 17.1 you saw four memory chips, or DIMMs, attached to each of the two sockets; all of the sixteen cores have access to all that memory.) OpenMP makes it easy to explore all these cores in the same program. The OpenMP-4.0 standard also added the possibility to offload computations to the GPU or other accelerator.

To summarize the structure of the architecture that OpenMP targets:

- A node has a number of sockets, typically 1, 2, or 4;
- each socket has a number of cores, as of 2022 this can be up to 64;
- each core is an independent processing unit, with access to all the memory on the node.
- There can be an accelerator, which can be used to offload computations to.

What OpenMP does not target is the *cluster* structure, where nodes communicate through a library that can access the network, such as Message Passing Interface (MPI)

17.1.2 Target software

OpenMP is based on two concepts: the use of *threads* and the *fork/join model* of parallelism. For now you can think of a thread as a sort of process: the processing unit executes a sequence of instructions. The fork/join model says that a thread can split itself ('fork') into a number of threads that are identical copies. At some point these copies go away and the original thread is left ('join'), but while the *team of threads* created by the fork exists, you have parallelism available to you. The part of the execution between fork and join is known as a *parallel region*.

Figure 17.3 gives a simple picture of this: a thread forks into a team of threads, and these threads themselves can fork again.

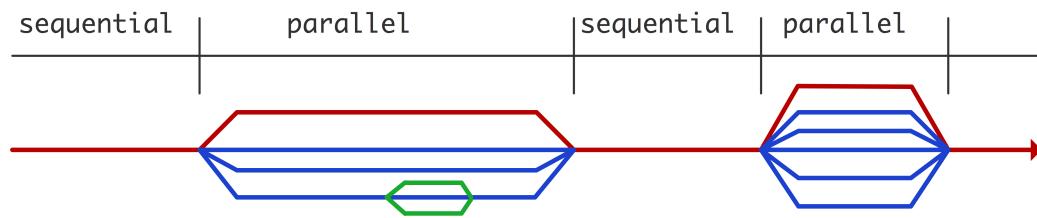


Figure 17.3: Thread creation and deletion during parallel execution

The threads that are forked are all copies of the *master thread*: they have access to all that was computed so far; this is their *shared data*. Of course, if the threads were completely identical the parallelism would be pointless, so they also have private data, and they can identify themselves: they know their thread number. This allows you to do meaningful parallel computations with threads.

This brings us to the third important concept: that of *work sharing* constructs. In a team of threads, initially there will be replicated execution; a work sharing construct divides available work over the threads.

So there you have it: OpenMP uses teams of threads, and inside a parallel region the work is distributed over the threads with a work sharing construct. Threads can access shared data, and they have some private data.

An important difference between OpenMP and MPI is that parallelism in OpenMP is dynamically activated by a thread spawning a team of threads. Furthermore, the number of threads used can differ between parallel regions, and threads can create threads recursively. By contrast, in an MPI program the number of running processes is (mostly) constant throughout the run, and determined by factors external to the program.

17.1.3 About threads and cores

OpenMP programming is typically done to take advantage of *multicore* processors. Thus, to get a good speedup you would typically let your number of threads be equal to the number of cores. However, there is nothing to prevent you from creating more threads if that serves the natural expression of your algorithm: the operating system will use *time slicing* to let them all be executed. You just don't get a speedup beyond the number of actually available cores.

On some modern processors there are *hardware threads*, meaning that a core can actually let more than one thread be executed, with some speedup over the single thread. To use such a processor efficiently you would let the number of OpenMP threads be 2 or 4 times the number of cores, depending on the hardware.

17.2 Logistics of an OpenMP program run

Before we can start looking at OpenMP, we need to get some formalities out of the way about OpenMP programs.

17.2.1 Compiling

A C program needs to contain:

```
#include "omp.h"
```

while a Fortran program needs to contain:

```
use omp_lib
```

or

```
#include "omp_lib.h"
```

OpenMP is handled by extensions to your regular compiler, typically by adding an option to your commandline:

```
# gcc
gcc -o foo foo.c -fopenmp
# Intel compiler
icc -o foo foo.c -qopenmp
```

If you have separate compile and link stages, you need that option in both.

17.2.2 Standards

The OpenMP system has gone through a number of standards, and some features you read about in this course may not be available with your compiler.

When you want to check this, you can query the *OpenMP standard* as follows. When you use the above compiler options, the *OpenMP macro*, (or *cpp macro*) `_OPENMP` will be defined. Thus, you can have conditional compilation by writing

```
#ifdef _OPENMP
...
#else
...
#endif
```

The value of this macro is a decimal value `yyyymm` denoting the OpenMP standard release that this compiler supports; see section [28.7](#).

Fortran note 18: OpenMP version. The parameter `openmp_version` contains the version in `yyyymm` format.

```
!! version.F90
use omp_lib
implicit none
integer :: standard
standard = openmp_version
```

17.2.3 Running an OpenMP program

You run an OpenMP program by invoking it the regular way (for instance `./a.out`), but its behavior is influenced by some *OpenMP environment variables*. The most important one is `OMP_NUM_THREADS`:

```
export OMP_NUM_THREADS=8
```

which sets the number of threads that a program will use. You would typically set this equal to the number of cores in your hardware, and hope for approximately linear speedup.

See section 28.1 for a list of all environment variables.

17.3 Your first OpenMP program

In this section you will see just enough of OpenMP to write a first program and to explore its behavior. For this we need to introduce a couple of OpenMP language constructs. They will all be discussed in much greater detail in later chapters.

17.3.1 Directives

OpenMP is not magic, so you have to tell it when something can be done in parallel. This is mostly done through *directives*; additional specifications can be done through library calls.

In C/C++ the *pragma* mechanism is used: annotations for the benefit of the compiler that are otherwise not part of the language. This looks like:

```
#pragma omp somedirective clause(value,othervalue)
    statement;

#pragma omp somedirective clause(value,othervalue)
{
    statement 1;
    statement 2;
}
```

with

- the `#pragma omp sentinel` to indicate that an OpenMP directive is coming;
- a directive, such as `parallel`;
- and possibly clauses with values.
- After the directive comes either a single statement or a block in *curly braces*.

Directives in C/C++ are case-sensitive. Directives can be broken over multiple lines by escaping the line end.

C++ note 2: Bracket syntax. In keeping with the desire to get rid of the C Preprocessor (CPP) in C++, a new *syntax* for OpenMP *directives* was introduced:

```
// directive.cxx
int nthreads;
[[omp::directive( parallel ) ]]
[[omp::directive( master ) ]]
nthreads = omp_get_num_threads();
```

Fortran note 19: OpenMP sentinel. The sentinel in Fortran looks like a comment:

```
!$omp directive clause(value)
statements
!$omp end directive
```

(For fixed-form sources, c\$omp and *\$omp are also acceptable.)

The difference with the C directive is that Fortran (pre-Fortran2018) does not have code blocks, such as are induced by curly braces in C, so there is often an explicit *end-of directive* line.

If you break a directive over more than one line, all but the last line need to have a continuation character, and each line needs to have the sentinel:

```
!$omp parallel &
!$omp      num_threads(7)
      tp = omp_get_thread_num()
!$omp end parallel
```

The directives are case-insensitive. In *Fortran fixed-form source* files (which is the only possibility in Fortran77), c\$omp and *\$omp are allowed too.

17.3.2 Parallel regions

The simplest way to create parallelism in OpenMP is to use the `parallel` pragma. A block preceded by the `parallel` pragma is called a *parallel region*; it is executed by a newly created team of threads. This is an instance of the *Single Program Multiple Data (SPMD)* model: all threads execute (redundantly) the same segment of code.

```
#pragma omp parallel
{
    // this is executed by a team of threads
}
```

Exercise 17.1. Write a ‘hello world’ program, where the print statement is in a parallel region.

Compile and run.

Run your program with different values of the environment variable `OMP_NUM_THREADS`. If you know how many cores your machine has, can you set the value higher?

Let’s start exploring how OpenMP handles parallelism, using the following functions:

- `omp_get_num_threads` reports how many threads are currently active, and
- `omp_get_thread_num` reports the number of the thread that makes the call.
- `omp_get_num_procs` reports the number of available cores.

Exercise 17.2. Take the hello world program of exercise 17.1 and insert the above functions, before, in, and after the parallel region. What are your observations?

Exercise 17.3. Extend the program from exercise 17.2. Make a complete program based on these lines:

```
Code:
// reduct.c
int tsum=0;
#pragma omp parallel
{
    tsum += // expression
}
printf("Sum is %d\n",tsum);
```

```
Output:
[code/omp/c] sumthread:
With 4 threads, sum s/b 6
Sum is 6
Sum is 5
Sum is 1
Sum is 4
Sum is 6
Sum is 5
Sum is 6
Sum is 5
Sum is 3
Sum is 4
```

Compile and run again. (In fact, run your program a number of times.) Do you see something unexpected? Can you think of an explanation?

If the above puzzles you, read about *race conditions* in HPC book, section ??.

17.3.3 Code and execution structure

Here are a couple of important concepts:

- An OpenMP directive is followed by an *structured block*; in C this is a single statement, a compound statement, or a block in braces; In Fortran it is delimited by the directive and its matching ‘end’ directive. A structured block can not be jumped into, so it can not start with a labeled statement, or contain a jump statement leaving the block.
- An OpenMP *construct* is the section of code starting with a directive and spanning the following structured block, plus in Fortran the end-directive. This is a lexical concept: it contains the statements directly enclosed, and not any subroutines called from them.
- A *region of code* is defined as all statements that are dynamically encountered while executing the code of an OpenMP construct. This is a dynamic concept: unlike a ‘construct’, it does include any subroutines that are called from the code in the structured block.

17.4 Thread data

In most programming languages, visibility of data is governed by rules on the *scope of variables*: a variable is declared in a block, and it is then visible to any statement in that block and blocks with a *lexical scope* contained in it, but not in surrounding blocks:

```
main () {
    // no variable `x' define here
    {
        int x = 5;
        if (somecondition) { x = 6; }
        printf("x=%e\n",x); // prints 5 or 6
    }
    printf("x=%e\n",x); // syntax error: `x' undefined
}
```

Fortran has simpler rules, since it does not have blocks inside blocks.

OpenMP has similar rules concerning data in parallel regions and other OpenMP constructs.

Data is visible in enclosed scopes:

```
main() {
    int x;
#pragma omp parallel
    {
        // you can use and set `x' here
    }
    printf("x=%e\n",x); // value depends on what
                        // happened in the parallel region
}
```

In C, you can redeclare a variable inside a nested scope:

```
{
    int x;
    if (something) {
        double x; // same name, different entity
    }
    x = ... // this refers to the integer again
}
```

Doing so makes the outer variable inaccessible.

OpenMP has a similar mechanism: parallel regions are a scope. There is an important difference with plain C code: each thread in the team gets its own instance of the enclosed variable.

```
{
    int x;
#pragma omp parallel
    {
        double x;
        // do something with x
    }
}
```

- A parallel region is a scope.
- Local variables are per thread.

This is illustrated in figure 17.4.

In addition to such scoped variables, which live on a *stack*, there are variables on the *heap*, typically created by a call to `malloc` (in C) or `new` (in C++). Rules for them are more complicated.

Summarizing the above, there are

- *shared variables*, where each thread refers to the same data item, and
- *private variables*, where each thread has its own instance.

In addition to using scoping, OpenMP also uses options on the directives to control whether data is private or shared.

Many of the difficulties of parallel programming with OpenMP stem from the use of shared variables. For instance, if two threads update a shared variable, there is no guarantee on the order on the updates.

We will discuss all this in detail in section 22.

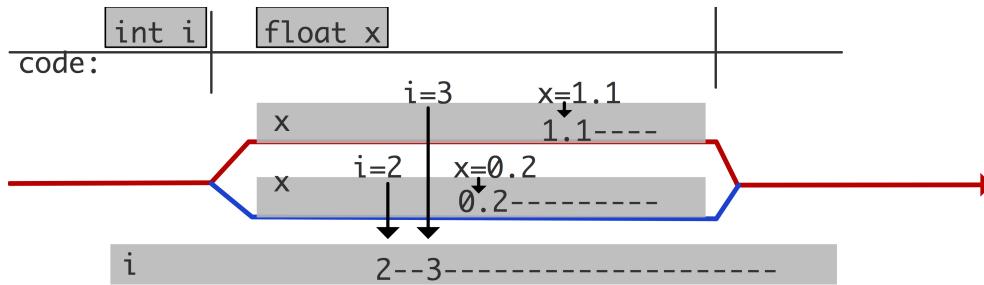


Figure 17.4: Locality of variables in threads

17.5 Creating parallelism

The *fork/join model* of OpenMP means that you need some way of indicating where an activity can be forked for independent execution. There are two ways of doing this:

1. You can declare a parallel region and split one thread into a whole team of threads. We will discuss this next in chapter 18. The division of the work over the threads is controlled by *work sharing construct*; see chapter 21.
2. Alternatively, you can use tasks and specify one parallel activity at a time. You will see this in section 24.

Note that OpenMP only indicates how much parallelism is present; whether independent activities are in fact executed in parallel is a runtime decision.

Declaring a parallel region tells OpenMP that a team of threads can be created. The actual size of the team depends on various factors (see section 28.1 for variables and functions mentioned in this section).

- The *environment variable* `OMP_NUM_THREADS` limits the number of threads that can be created.
- If you don't set this variable, you can also set this limit dynamically with the *library routine* `omp_set_num_threads`. This routine takes precedence over the aforementioned environment variable if both are specified.
- A limit on the number of threads can also be set as a `num_threads` clause on a parallel region:

```
#pragma omp parallel num_threads(ndata)
```

To ask how much parallelism is actually used in your parallel region, use `omp_get_num_threads`. To query these hardware limits, use `omp_get_num_procs`. You can query the maximum number of threads with `omp_get_max_threads`. This equals the value of `OMP_NUM_THREADS`, not the number of actually active threads in a parallel region.

```
// proccount.c
void nested_report() {
#pragma omp parallel
#pragma omp master
printf("Nested      : %2d cores and %2d threads
       out of max %2d\n",
       omp_get_num_procs(),
       omp_get_num_threads(),
       omp_get_max_threads());
}
int env_num_threads;
#pragma omp parallel \
num_threads(2*env_num_threads)
#pragma omp master
{
```

```

printf("Double    : %2d cores and %2d
      threads out of max %2d\n",
      omp_get_num_procs(),
      omp_get_num_threads(),
      omp_get_max_threads());
}

#pragma omp parallel
#pragma omp master
nested_report();

```

```

[c:48] for t in 1 2 4 8 16 ; do OMP_NUM_THREADS=$t ./proccount ; done
----- Parallelism report -----
Sequential: count 4 cores and 1 threads out of max 1
Parallel  : count 4 cores and 1 threads out of max 1
Parallel  : count 4 cores and 1 threads out of max 1
----- Parallelism report -----
Sequential: count 4 cores and 1 threads out of max 2
Parallel  : count 4 cores and 2 threads out of max 2
Parallel  : count 4 cores and 1 threads out of max 2
----- Parallelism report -----
Sequential: count 4 cores and 1 threads out of max 4
Parallel  : count 4 cores and 4 threads out of max 4
Parallel  : count 4 cores and 1 threads out of max 4
----- Parallelism report -----
Sequential: count 4 cores and 1 threads out of max 8
Parallel  : count 4 cores and 8 threads out of max 8
Parallel  : count 4 cores and 1 threads out of max 8
----- Parallelism report -----
Sequential: count 4 cores and 1 threads out of max 16
Parallel  : count 4 cores and 16 threads out of max 16
Parallel  : count 4 cores and 1 threads out of max 16

```

Another limit on the number of threads is imposed when you use nested parallel regions. This can arise if you have a parallel region in a subprogram which is sometimes called sequentially, sometimes in parallel. For details, see section [18.2](#).

Chapter 18

OpenMP topic: Parallel regions

18.1 Creating parallelism with parallel regions

In OpenMP you need to indicate explicitly what passages are parallel. Creating parallelism, which here means: creating a team of threads, is done with the `parallel` pragma. A block preceded by the `omp_parallel` pragma is called a *parallel region*; it is executed by a newly created team of threads. This is an instance of the *SPMD* model: all threads execute the same segment of code.

```
#pragma omp parallel
{
    // this is executed by a team of threads
}
```

It would be pointless to have the block be executed identically by all threads. One way to get a meaningful parallel code is to use the function `omp_get_thread_num` to find out which thread you are, and execute work that is individual to that thread. This function gives a number relative to the current team; recall from figure 17.3 that new teams can be created recursively.

There is also a function `omp_get_num_threads` to find out the total number of threads.

The first thing we want to do is create a team of threads. This is done with a *parallel region*. Here is a very simple example where each thread outputs its number:

```
Code:
// hello.c
#pragma omp parallel
{
    int t = omp_get_thread_num();
    printf("Hello world from %d!\n", t);
}
```

```
Output:
[examples/omp/c] hello:
Hello world from 1!
Hello world from 0!
Hello world from 2!
Hello world from 3!
```

or in Fortran

18. OpenMP topic: Parallel regions

```
Code:  
!! hello.F90  
!$omp parallel  
    print *, "Hello world!"  
!$omp end parallel
```

```
Output:  
[examples/omp/f] hellof:  
Hello world from      1  
Hello world from      2  
Hello world from      3  
Hello world from      0
```

C++ note 3: *Output streams in parallel.* The use of `cout` may give jumbled output: lines can break at each `<<`. Use `stringstream` to form a single stream to output.

```
// hello.cxx  
#pragma omp parallel  
{  
    int t = omp_get_thread_num();  
    stringstream proctext;  
    proctext << "Hello world from "  
        << t << '\n';  
    cerr << proctext.str();  
}
```

C++ note 4: *Parallel regions in lambdas.* OpenMP parallel regions can be in functions, including lambda expressions.

```
const int s = [] () {  
    int s;  
# pragma omp parallel  
# pragma omp master  
    s = 2 * omp_get_num_threads();  
    return s; }();  
  
(‘Immediately Invoked Function Expression’)
```

The following example uses parallelism for an actual calculation:

```
result = f(x)+g(x)+h(x)
```

you could parallelize this as

```
double result,fresult,gresult,hresult;  
#pragma omp parallel  
{ int num = omp_get_thread_num();  
    if (num==0)      fresult = f(x);  
    else if (num==1) gresult = g(x);  
    else if (num==2) hresult = h(x);  
}  
result = fresult + gresult + hresult;
```

This code corresponds to the model we just discussed:

- Immediately preceding the parallel block, one thread will be executing the code. In the main program this is the *initial thread*.
- At the start of the block, a new *team of threads* is created, and the thread that was active before the block becomes the *master thread* of that team.

- After the block only the master thread is active.
- Inside the block there is team of threads: each thread in the team executes the body of the block, and it will have access to all variables of the surrounding environment. How many threads there are can be determined in a number of ways; we will get to that later.

Remark In future versions of OpenMP, the master thread will be called the primary thread. In 5.1 the master construct will be deprecated, and `master` (with added functionality) will take its place. In 6.0 `master` will disappear from the Spec, including `proc_bind master` “variable” and combined master constructs (master taskloop, etc.)

Exercise 18.1. What happens if you call `omp_get_thread_num` and `omp_get_num_threads` outside a parallel region?

18.2 Nested parallelism

What happens if you call a function from inside a parallel region, and that function itself contains a parallel region?

```
int main() {
    ...
#pragma omp parallel
    {
        ...
        func(...)

        ...
    }
} // end of main
```

```
void func(...) {
    #pragma omp parallel
    {
        ...
    }
}
```

Since any thread can create a team, you may expect that every thread, in its call to `func`, will create its own new team. This is called *nested parallelism* and it works as described.

However, by default, the nested parallel region will have only one thread. You need to allow non-trivial nested parallelism explicitly.

To allow nested thread creation, use the environment variable `OMP_MAX_ACTIVE_LEVELS` (default: 1) to set the number of levels of parallel nesting. Equivalently, there are functions `omp_set_max_active_levels` and `omp_get_max_active_levels`:

`OMP_MAX_ACTIVE_LEVELS=3`

or

```
void omp_set_max_active_levels(int);
int omp_get_max_active_levels(void);
```

Remark A deprecated mechanism is to set the environment variable `OMP_NESTED` (default: false) or its corresponding function:

```
OMP_NESTED=true
or
omp_set_nested(1)
```

18. OpenMP topic: Parallel regions

Nested parallelism can happen with nested loops, but it's also possible to have a `sections` construct and a loop nested. Example:

```
Code:  
// sectionnest.c  
#pragma omp parallel sections reduction  
  (+:s)  
{  
#pragma omp section  
{  
    double s1=0;  
    omp_set_num_threads(team);  
    #pragma omp parallel for  
    reduction(+:s1)  
    for (int i=0; i<N; i++) {
```

```
Output:  
[code/omp/c] sectionnest:  
  
Nesting: false  
Threads: 2, speedup: 2.0  
Threads: 4, speedup: 2.0  
Threads: 8, speedup: 2.0  
Threads: 12, speedup: 2.0  
  
Nesting: true  
Threads: 2, speedup: 1.8  
Threads: 4, speedup: 3.7  
Threads: 8, speedup: 6.9  
Threads: 12, speedup: 10.4
```

The amount of nested parallelism can be set:

`OMP_NUM_THREADS=4,2`

means that initially a parallel region will have four threads, and each thread can create two more threads. It is still necessary that set the number of active levels.

The total number of threads active simultaneously (technically: in a *contention group*) can be limited with:

`OMP_THREAD_LIMIT=123`

Its value can be queried with `omp_get_thread_limit`.

More functions: `omp_get_level`, `omp_get_active_level`, `omp_get_ancestor_thread_num`, `omp_get_team_size(level)`.

18.2.1 Subprograms with parallel regions

A common application of nested parallelism is the case where you have a subprogram with a parallel region, which itself gets called from a parallel region.

Exercise 18.2. Test nested parallelism by writing an OpenMP program as follows:

1. Write a subprogram that contains a parallel region.
2. Write a main program with a parallel region; call the subprogram both inside and outside the parallel region.
3. Insert print statements
 - (a) in the main program outside the parallel region,
 - (b) in the parallel region in the main program,
 - (c) in the subprogram outside the parallel region,
 - (d) in the parallel region inside the subprogram.

Run your program and count how many print statements of each type you get.

Writing subprograms that are called in a parallel region illustrates the following point: directives are evaluated with respect to the *dynamic scope* of the parallel region, not just the lexical scope. In the following example:

```
#pragma omp parallel  
{
```

```

    f();
}
void f() {
#pragma omp for
  for ( .... ) {
    ...
  }
}

```

the body of the function `f` falls in the dynamic scope of the parallel region, so the for loop will be parallelized.

If the function may be called both from inside and outside parallel regions, you can test which is the case with `omp_in_parallel`.

C++ note 5: Dynamic scope for class methods. Dynamic scope holds for class methods as for any other function:

Code:

```

// nested.cxx
class withnest {
public:
  void f() {
    stringstream ss;
    ss
      << omp_get_num_threads()
      << '\n';
    cout << ss.str();
  };
};

int main() {
  withnest my_object;
#pragma omp parallel
  my_object.f();
}

```

Output:

```

[examples/omp/cxx] nested:
executing: OMP_MAX_ACTIVE_LEVELS=2
  ↪OMP_PROC_BIND=true
  ↪OMP_NUM_THREADS=2 ./nested
2
2

```

18.3 Cancel parallel construct

It is possible to terminate a parallel construct early with the `cancel` directive:

```
!$omp cancel construct [if (expr)]
```

where construct is `parallel`, `sections`, `do` or `taskgroup`.

See section 29.4 for an example.

Cancelling is disabled by default for performance reasons. To activate it, set the `OMP_CANCELLATION` variable to true.

The state of cancellation can be queried with `omp_get_cancellation`, but there is no function to set it.

Cancellation can happen at most obvious places where OpenMP is active, but additional cancellation points can be set with the `cancellation point` directive:

```
#pragma omp cancellation point <construct>
```

where the construct is `parallel`, `sections`, `for`, `do`, `taskgroup`.

18.4 Review questions

Exercise 18.3. T/F? The function `omp_get_num_threads()` returns a number that is equal to the number of cores.

Exercise 18.4. T/F? The function `omp_set_num_threads()` can not be set to a higher number than the number of cores.

Exercise 18.5. What function can be used to detect the number of cores?

Chapter 19

OpenMP topic: Loop parallelism

Loop parallelism is a very common type of parallelism in scientific codes, so OpenMP has an easy mechanism for it. OpenMP parallel loops are a first example of OpenMP ‘worksharing’ constructs (see section 21.1 for the full list): constructs that take an amount of work and distribute it over the available threads in a parallel region, created with the `parallel` pragma.

The parallel execution of a loop can be handled a number of different ways. For instance, you can create a parallel region around the loop, and adjust the loop bounds:

```
#pragma omp parallel
{
    int threadnum = omp_get_thread_num(),
        numthreads = omp_get_num_threads();
    int low = N*threadnum/numthreads,
        high = N*(threadnum+1)/numthreads;
    for (int i=low; i<high; i++)
        // do something with i
}
```

In effect, this is how you would parallelize a loop in MPI: the `parallel` pragma creates a team of threads, each thread executes the block of code, and based on its thread number finds a unique block of work to do.

Exercise 19.1. What are some important differences between the resulting OpenMP and MPI code?

19.1 Loop parallelism through directives

The natural way to parallelize a loop in OpenMP is to use the `for` pragma where OpenMP does the above chopping of the loop for you:

```
#pragma omp parallel
#pragma omp for
for (int i=0; i<N; i++) {
    // do something with i
}
```

This fragment combines two OpenMP idioms:

1. First the `parallel` directive creates a *team* of threads; after which
2. The `for` directive is a *worksharing construct*: it divides the available work over the available threads.

Remark In this example the loop variable is declared in the loop header, as is the preferred practice, but if you don't do it this way, the loop variable is automatically made private to the threads.

Leaving the work distribution to OpenMP has several advantages. For one, you don't have to calculate the loop segments for the threads yourself, but you can also tell OpenMP to assign the loop iterations according to different schedules (section 19.3).

Fortran note 20: OMP do pragma. The `for` pragma only exists in C; there is a correspondingly named `do` pragma in Fortran.

```
$!omp parallel
$!omp do
  do i=1,N
    ! something with i
  end do
$!omp end do
$!omp end parallel
```

Fortran note 21: Missing sentinel for loops. With the `for` directive, and other loop directives such as `simd`, It is clear what block of code they pertain to. Therefore, the `end` sentinel can be omitted:

```
!! pi.F90
 !$omp parallel do reduction(+:pi4) private(xsample,y)
 do isample=0,N-1
   xsample = isample * h
   y = sqrt(1-xsample*xsample)
   pi4 = pi4 + h*y
 end do
```

19.1.1 About parallelism and worksharing

It is important to realize that the `parallel` directive does not immediately distribute any work: all threads start out executing the same code. As an illustration, figure 19.1 shows the execution on four threads of code that has instructions between the `parallel` and `for` directives:

```
#pragma omp parallel
{
  code1();
  #pragma omp for
  for (int i=1; i<=4*N; i++) {
    code2();
  }
  code3();
}
```

The code before and after the loop is executed identically in each thread; the loop iterations are spread over the four threads.

The `do` and `for` pragmas do not themselves create parallelism: they take the team of threads that is active, and divide the loop iterations over them. This means that the `omp for` or `omp do` directive needs to be inside a parallel region. Outside of a parallel region they would execute sequentially.

As an illustration:

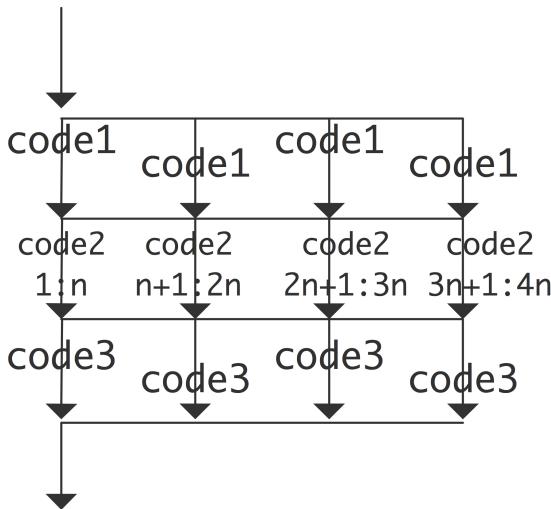


Figure 19.1: Execution of parallel code inside and outside a loop

```
Code:
// parfor.c
#pragma omp parallel
{
    int
    nthreads = omp_get_num_threads(),
    thread_num = omp_get_thread_num();
    printf("Threads entering parallel region:
    %d\n",
    nthreads);
    #pragma omp for
    for (int iter=0; iter<nthreads; iter++)
        printf("thread %d executing iter %d\n",
        thread_num, iter);
}
```

```
Output:
[examples/omp/c] parfor:
%%%% equal thread/core counts %%%
Threads entering parallel region: 4
thread 3 executing iter 3
Threads entering parallel region: 4
thread 0 executing iter 0
Threads entering parallel region: 4
thread 2 executing iter 2
Threads entering parallel region: 4
thread 1 executing iter 1
```

Exercise 19.2. What would happen in the above example if you increase the number of threads to be larger than the number of cores?

It is also possible to have a combined `omp parallel for` or `omp parallel do` directive.

```
#pragma omp parallel for
    for (int i=0; ....
```

C++ note 6: Custom iterators. OpenMP can parallelize any range-based loop with a random-access iterator.

```
// iterator.cxx
template<typename T>
class NewVector {
protected:
    T *storage;
```

```
int s;
public:
// iterator stuff
class iter;
iter begin();
iter end();
};
```

The following methods are needed for the contained `iter` class:

```
NewVector<T>::iter& operator++();
T& operator*();
bool operator==( const NewVector::iter &other ) const;
bool operator!=( const NewVector::iter &other ) const;
// needed for OpenMP
int operator-
( const NewVector::iter& other ) const;
NewVector<T>::iter& operator+=( int add );
```

And then a range-based loop is allowed:

```
NewVector<float> v(s);
#pragma omp parallel for
for ( auto e : v )
    cout << e << " ";
```

19.1.2 Loops are static

There are some restrictions on the loop: basically, OpenMP needs to be able to determine in advance how many iterations there will be.

- The loop can not contain `break`, `return`, `exit` statements, or `goto` to a label outside the loop. However, there is the `cancel` construct; see section 18.3.
- The `continue` (for C/C++) or `cycle` (for Fortran) statement is allowed.
- C++ exceptions need to be caught in the loop body.
- The index update has to be an increment (or decrement) by a fixed amount.
- The loop index variable is automatically private (section 22.2), and no changes to it inside the loop are allowed. The following loop is not parallelizable in OpenMP:

```
for ( int i=0; i<N; ) {
    // something
    if (something)
        i++;
    else
        i += 2;
}
```

Remark *The loop index needs to be an integer value for the loop to be parallelizable. Unsigned values are allowed as of OpenMP-3.*

19.1.3 When is a loop parallel?

OpenMP parallelism is not magic. You can not take a sequential loop, even when it satisfies the restrictions above, put an `omp parallel for` on it, and hope you get the same result, only faster. The speed issue is something we will go into later; for now let's consider the issue of whether the parallel code computes the right result to begin with.

The trivial case of a loop that is executed correctly in parallel, is one where iteration i writes in location i of some array:

```
for (int i=low; i<hi; i++)
    x[i] = // expression
```

The iterations of this loop are independent, and hence can be computed in parallel in any order, if the right-hand-side expression does not contain any references to x , or at best $x[i]$.

Leaving considerations of the right-hand-side expression aside, we can more generally say that a loop is parallelizable if in

```
for (int i=low; i<hi; i++)
    x[ f(i) ] = // expression
```

the function f satisfies:

$$i \neq j \Rightarrow f(i) \neq f(j).$$

Exercise 19.3. Consider the code

```
for (int i=0; i<n; i++)
    x[i/2] += f(i)
```

Argue that this does not satisfy the above condition. Can you rewrite this loop to be parallelizable?

19.2 An example

To illustrate the speedup of perfectly parallel calculations, we consider a simple code that applies the same calculation to each element of an array.

All tests are done on the *TACC Frontera* cluster, which has dual-socket *Intel Cascade Lake* nodes, with a total of 56 cores. We control affinity by setting `OMP_PROC_BIND=true`.

Here is the essential code fragment:

```
// speedup.c
#pragma omp parallel for
    for (int ip=0; ip<N; ip++) {
        for (int jp=0; jp<M; jp++) {
            double f = sin( values[ip] );
            values[ip] = f;
        }
    }
```

Exercise 19.4. Verify that the outer loop is parallel, but the inner one is not.

Exercise 19.5. Compare the time for the sequential code and the single-threaded OpenMP code. Try different optimization levels, and different compilers if you have them.

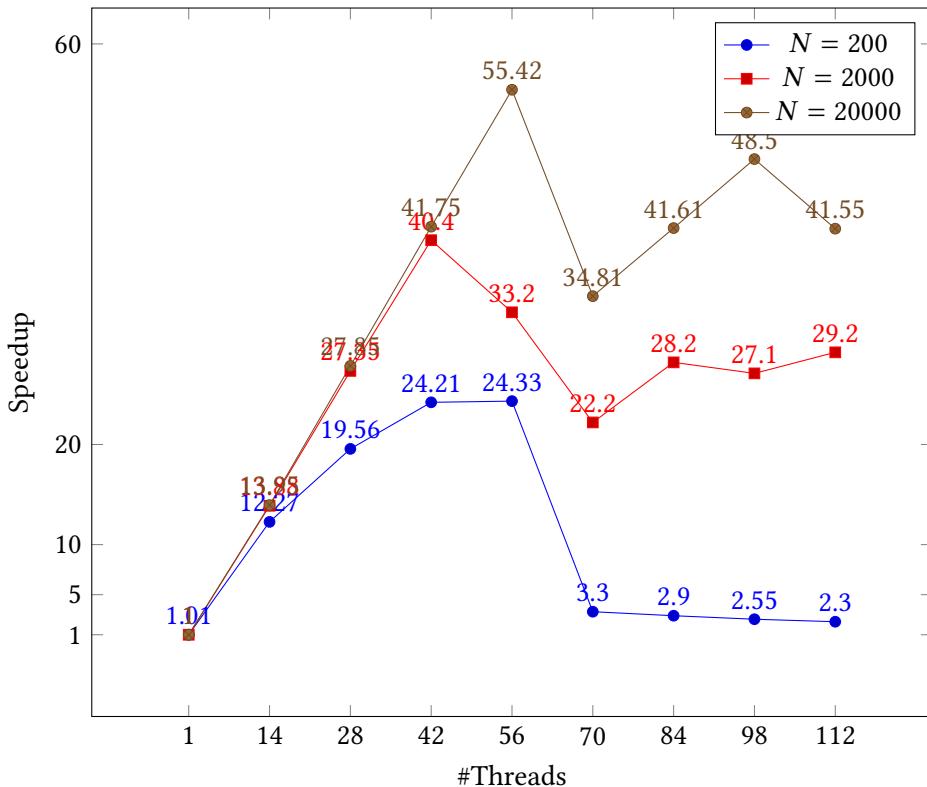


Figure 19.2: Speedup as function of problem size

- Do you sometimes get a significant difference? What would be an explanation?
- Does your compiler have a facility for generating optimization reports? For instance `-qoptreport=5` for the *Intel compiler*.

Now we investigate the influence of two parameters:

1. the OpenMP thread count: while we have 56 cores, values larger than that are allowed; and
2. the size of the problem: the smaller the problem, the larger the relative overhead of creating and synchronizing the team of threads.

We execute the above computation several times to even out effects of cache loading.

The results are in figure 19.2:

- While the problem size is always larger than the number of threads, only for the largest problem, which has at least 400 points per thread, is the speedup essentially linear.
- OpenMP allows for the number of threads to be larger than the core count, but there is no performance improvement in doing so.

The above tests did not use *hyperthreads*, since that is disabled on Frontera. However, the *Intel Knights Landing* nodes of the TACC *Stampede2* cluster have four *hyperthreads* per core. Table 19.3 shows that this will indeed give a modest speedup.

For reference, the commandlines executed were:

```
# frontera
```

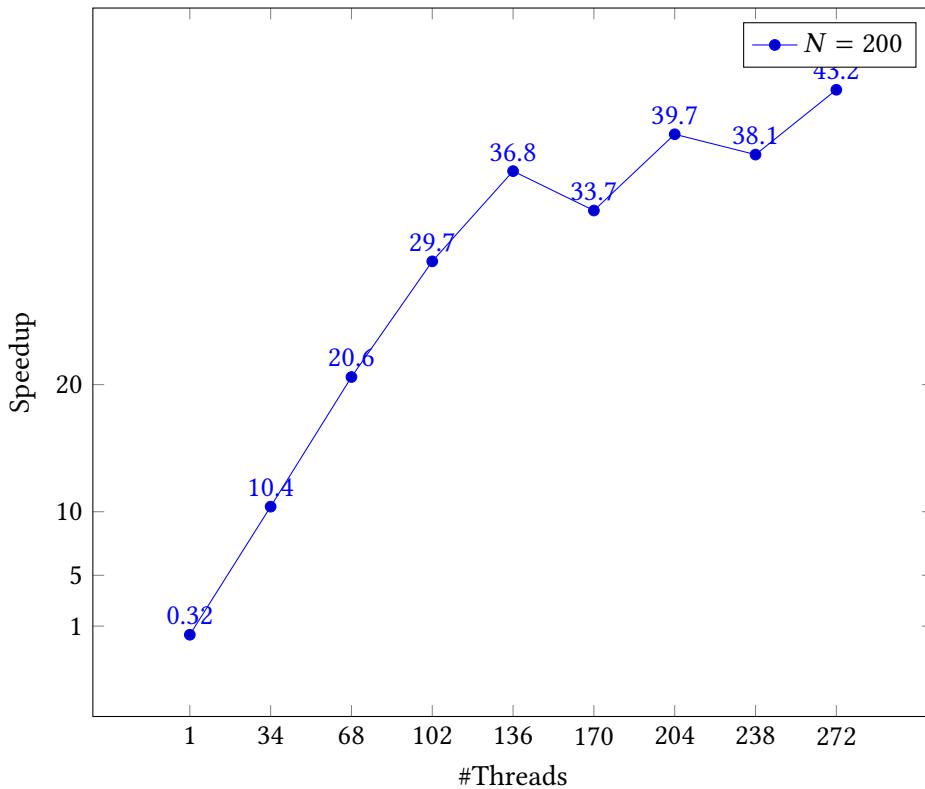


Figure 19.3: Speedup on a hyper-threaded architecture

```

make localclean run_speedup EXTRA_OPTIONS=-DN=200 NDIV=8 NP=112
make localclean run_speedup EXTRA_OPTIONS=-DN=2000 NDIV=8 NP=112
make localclean run_speedup EXTRA_OPTIONS=-DN=20000 NDIV=8 NP=112
# stampede2
make localclean run_speedup NDIV=8 EXTRA_OPTIONS="-DN=200000 -DM=1000" NP=272
C++ note 7: Range syntax. Parallel loops in C++ can use range-based syntax as of OpenMP-5.0:

```

```

// vecdata.cxx
vector<float> values(100);

#pragma omp parallel for
for ( auto& elt : values ) {
    elt = 5.f;
}

float sum{0.f};
#pragma omp parallel for reduction(+:sum)
for ( auto elt : values ) {
    sum += elt;
}

```

Tests show exactly the same speedup as the C code.

19. OpenMP topic: Loop parallelism

C++ note 8: C++20 ranges header. The C++20 `ranges` library is supported:

```
#pragma omp parallel for reduction(+:itcount)
for ( auto e : data
      | std::ranges::views::drop(1) )
    itcount += e;
#pragma omp parallel for reduction(+:itcount)
for ( auto e : data
      | std::ranges::views::transform
      ( []( auto e ) { return 2*e; } ) )
    itcount += e;
```

C++ note 9: C++20 ranges speedup.

```
==== Run range on 1 threads ====
sum of vector: 50000005000000 in 6.148
sum w/ drop 1: 50000004999999 in 6.017
sum times 2 : 100000010000000 in 6.012
==== Run range on 25 threads ====
sum of vector: 50000005000000 in 0.494
sum w/ drop 1: 50000004999999 in 0.477
sum times 2 : 100000010000000 in 0.489
==== Run range on 51 threads ====
sum of vector: 50000005000000 in 0.257
sum w/ drop 1: 50000004999999 in 0.248
sum times 2 : 100000010000000 in 0.245
==== Run range on 76 threads ====
sum of vector: 50000005000000 in 0.182
sum w/ drop 1: 50000004999999 in 0.184
sum times 2 : 100000010000000 in 0.185
==== Run range on 102 threads ====
sum of vector: 50000005000000 in 0.143
sum w/ drop 1: 50000004999999 in 0.139
sum times 2 : 100000010000000 in 0.134
==== Run range on 128 threads ====
sum of vector: 50000005000000 in 0.122
sum w/ drop 1: 50000004999999 in 0.11
sum times 2 : 100000010000000 in 0.106
scaling results in: range-scaling-ls6.out
```

C++ note 10: Ranges and indices. Use `iota_view` to obtain indices:

```
// iota.cxx
vector<long> data(N);
#pragma omp parallel for
for ( auto i : std::ranges::iota_view( 0UZ,data.size() ) )
  data[i] = f(i);
```

Note that this uses C++23 suffix for `unsigned size_t`. For older versions:

```
iota_view( static_cast<size_t>(0),data.size() )
```

19.3 Loop schedules

Usually you will have many more iterations in a loop than there are threads. Thus, there are several ways you can assign your loop iterations to the threads. OpenMP lets you specify this with the `schedule` clause.

```
#pragma omp for schedule(....)
```

The first distinction we now have to make is between static and dynamic schedules. With static schedules, the iterations are assigned purely based on the number of iterations and the number of threads (and the `chunk` parameter; see later). In dynamic schedules, on the other hand, iterations are assigned to threads that are unoccupied. Dynamic schedules are a good idea if iterations take an unpredictable amount of time, so that *load balancing* is needed.

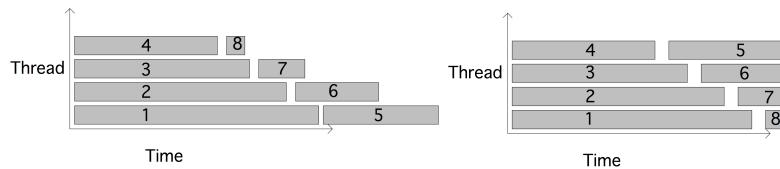


Figure 19.4: Illustration static round-robin scheduling versus dynamic

Figure 19.4 illustrates this: assume that each core gets assigned two (blocks of) iterations and these blocks take gradually less and less time. You see from the left picture that thread 1 gets two fairly long blocks, whereas thread 4 gets two short blocks, thus finishing much earlier. (This phenomenon of threads having unequal amounts of work is known as *load imbalance*.) On the other hand, in the right figure thread 4 gets block 5, since it finishes the first set of blocks early. The effect is a perfect load balancing.

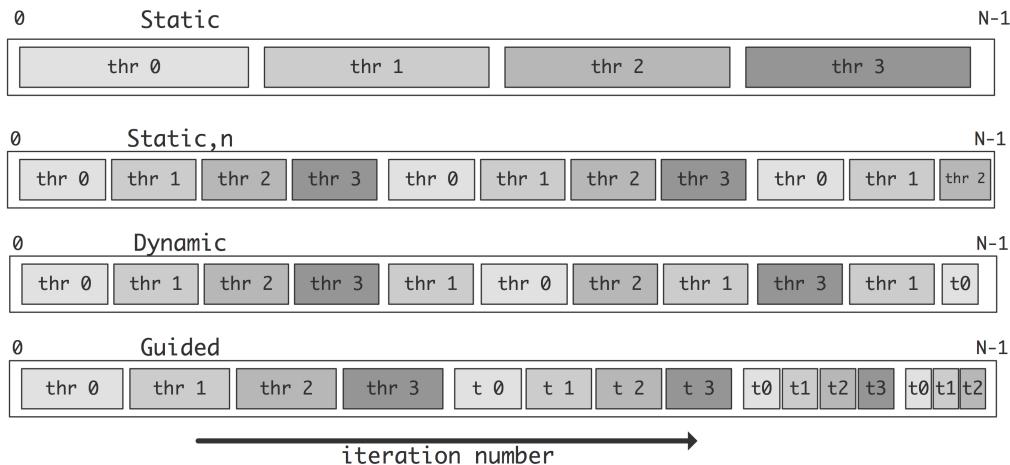


Figure 19.5: Illustration of the scheduling strategies of loop iterations

The default static schedule is to assign one consecutive block of iterations to each thread. If you want different sized blocks you can define a `chunk` size:

```
#pragma omp for schedule(static[,chunk])
```

(where the square brackets indicate an optional argument). With static scheduling, the compiler will determine the assignment of loop iterations to the threads at compile time, so, provided the iterations take roughly the same amount of time, this is the most efficient at runtime.

19. OpenMP topic: Loop parallelism

The choice of a chunk size is often a balance between the low overhead of having only a few chunks, versus the load balancing effect of having smaller chunks.

Exercise 19.6. Why is a chunk size of 1 typically a bad idea? (Hint: think about cache lines, and read HPC book, section ??.)

In dynamic scheduling OpenMP will put blocks of iterations (the default chunk size is 1) in a task queue, and the threads take one of these tasks whenever they are finished with the previous.

```
#pragma omp for schedule(static[,chunk])
```

While this schedule may give good load balancing if the iterations take very differing amounts of time to execute, it does carry runtime overhead for managing the queue of iteration tasks.

Finally, there is the guided schedule, which gradually decreases the chunk size. The thinking here is that large chunks carry the least overhead, but smaller chunks are better for load balancing. The various schedules are illustrated in figure 19.5.

If you don't want to decide on a schedule in your code, you can specify the `runtime` schedule. The actual schedule will then at runtime be read from the `OMP_SCHEDULE` environment variable. You can even just leave it to the runtime library by specifying `auto`

Exercise 19.7. Write a simple prime number tester and a loop that counts primes:

```
// primesched.c
for (int n = 0; n < N; n++) {
    if (is_prime(n))
        j++;
}
```

Do you expect a dynamic schedule to be better than a static one? Finish the program and test with different schedules.

Exercise 19.8. We continue with exercise 20.2. We add ‘adaptive integration’ where needed, the program refines the step size¹. This means that the iterations no longer take a predictable amount of time.

1. Use the `omp parallel for` construct to parallelize the loop. As in the previous lab, you may at first see an incorrect result. Use the `reduction` clause to fix this.
2. Your code should now see a decent speedup, but possibly not for all cores. It is possible to get completely linear speedup by adjusting the schedule.
Start by using `schedule(static,n)`. Experiment with values for `n`. When can you get a better speedup? Explain this.
3. Since this code is somewhat dynamic, try `schedule(dynamic)`. This will actually give a fairly bad result. Why? Use `schedule(dynamic,n)` instead, and experiment with values for `n`.
4. Finally, use `schedule(guided)`, where OpenMP uses a heuristic. What results does that give?

Exercise 19.9. Program the *LU factorization* algorithm without pivoting.

```
for k=1,n:
    A[k,k] = 1./A[k,k]
    for i=k+1,n:
        A[i,k] = A[i,k]/A[k,k]
        for j=k+1,n:
            A[i,j] = A[i,j] - A[i,k]*A[k,j]
```

1. It doesn't actually do this in a mathematically sophisticated way, so this code is more for the sake of the example.

1. Argue that it is not possible to parallelize the outer loop.
2. Argue that it is possible to parallelize both the i and j loops.
3. Parallelize the algorithm by focusing on the i loop. Why is the algorithm as given here best for a matrix on row-storage? What would you do if the matrix was on column storage?
4. Argue that with the default schedule, if a row is updated by one thread in one iteration, it may very well be updated by another thread in another. Can you find a way to schedule loop iterations so that this does not happen? What practical reason is there for doing so?

The schedule can be declared explicitly, set at runtime through the `OMP_SCHEDULE` environment variable, or left up to the runtime system by specifying `auto`. Especially in the last two cases you may want to enquire what schedule is currently being used with `omp_get_schedule` (since OpenMP-5.1):

```
int omp_get_schedule(omp_sched_t * kind, int * modifier );
```

Its mirror call is `omp_set_schedule`, which sets the value that is used when schedule value `runtime` is used. It is in effect equivalent to setting the environment variable `OMP_SCHEDULE`.

```
void omp_set_schedule (omp_sched_t kind, int modifier);
```

Type	environment variable <code>OMP_SCHEDULE=</code>	clause <code>schedule(...)</code>	<code>omp_sched_t</code> name	<code>omp_sched_t</code> value	modifier	default
static	<code>static[,n]</code>	<code>static[,n]</code>	<code>omp_sched_static</code>	1	$N/nthreads$	
dynamic	<code>dynamic[,n]</code>	<code>dynamic[,n]</code>	<code>omp_sched_dynamic</code>	2		1
guided	<code>guided[,n]</code>	<code>guided[,n]</code>	<code>omp_sched_guided</code>	3		
auto	<code>auto</code>	<code>auto</code>	<code>omp_sched_auto</code>	4		

Here are the various schedules you can set with the `schedule` clause:

affinity Set by using value `omp_sched_affinity`

auto The schedule is left up to the implementation. Set by using value `omp_sched_auto`

static value: 1. The modifier parameter is the *chunk* size. Can also be set by using value `omp_sched_static`

dynamic value: 2. The modifier parameter is the *chunk* size; default 1. Can also be set by using value `omp_sched_dynamic`

guided Value: 3. The modifier parameter is the *chunk* size. Set by using value `omp_sched_guided`

runtime Use the value of the `OMP_SCHEDULE` environment variable. Set by using value `omp_sched_runtime`

Some modifiers can be applied to the schedule:

- *monotonic* and *nonmonotonic* dictate whether each thread executes its iterations in order or arbitrarily;
- *simd* rounds the chunk up to a multiple of the *simd width*:

```
schedule( simd:static,7 )
```

will give a chunk size of 8, most likely.

19.4 Timing experiments

19.4.1 Indexing schemes

For two-dimensional loops there are several possible ways of handling the indexing.

1. We can use a nested loop, and translate i, j indices to a linear index;

```
// collapse.cxx
# pragma omp parallel for
    for ( int i=0; i<nsize; i++ )
        for ( int j=0; j<nsize; j++ )
            data[ i*nsize+j ] += sqrt(x*y);
# pragma omp parallel for reduction(+:s)
    for ( int i=0; i<nsize; i++ )
        for ( int j=0; j<nsize; j++ )
            s + sqrt(data[ i*nsize+j ]);
```

2. same, but with a `collapse(2)` directive;

```
# pragma omp parallel for collapse(2)
    for ( int i=0; i<nsize; i++ )
        for ( int j=0; j<nsize; j++ )
            data[ i*nsize+j ] += sqrt(x*y);
# pragma omp parallel for reduction(+:s) collapse(2)
    for ( int i=0; i<nsize; i++ )
        for ( int j=0; j<nsize; j++ )
            s + sqrt(data[ i*nsize+j ]);
```

3. with C++32 `mdspan` we can use true 2D indexing;

```
# pragma omp parallel for
    for ( int i=0; i<nsize; i++ )
        for ( int j=0; j<nsize; j++ )
            mdata[ i,j ] += sqrt(x*y);
# pragma omp parallel for reduction(+:s)
    for ( int i=0; i<nsize; i++ )
        for ( int j=0; j<nsize; j++ )
            s + sqrt(mdata[ i,j ]);
```

4. finally, to deal with Partial Differential Equations (PDEs) with a boundary condition, we use a nested loop that only traverses the interior of the array.

```
# pragma omp parallel for
    for ( int i=1; i<nsize-1; i++ )
        for ( int j=1; j<nsize-1; j++ )
            data[ i*nsize+j ] += sqrt(x*y);
# pragma omp parallel for reduction(+:s)
    for ( int i=1; i<nsize-1; i++ )
        for ( int j=1; j<nsize-1; j++ )
            s + sqrt(data[ i*nsize+j ]);
```

We report the results in figure 19.6. These results were run on the *TACC Frontera* cluster, with 56-core dual-socket *Intel Cascade Lake* processors.

We conclude that all schemes perform approximately equally well. Surprisingly the `collapse(2)` degrades performance. The gaps in the indexing from the ‘inner’ scheme don’t seem to matter. The `mdspan` indexing performs slightly worse than the best scheme.

19.4.2 OpenMP loops vs C++ standard algorithms

C++ note 11: Parallel standard algorithms. The C++17/C++20 standards have introduced the notion of *execution policy* to the standard algorithms, meaning the operations on containers that are in the `algorithm` library.

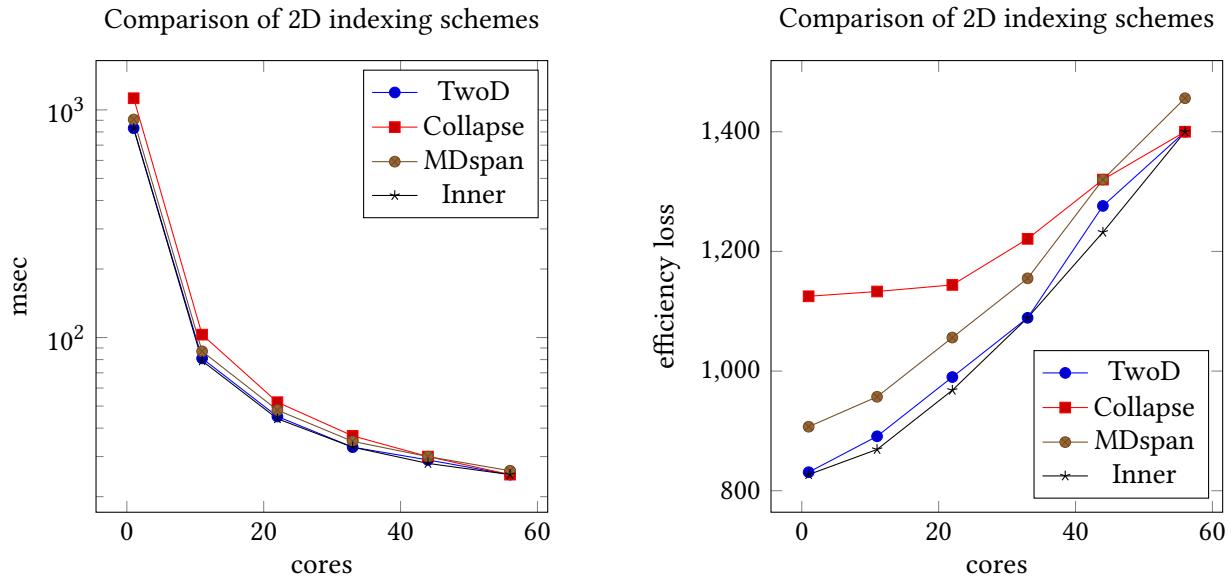


Figure 19.6: Different ways of indexing a 2D parallel loop. Left: absolute times; Right: normalized to t_1 .

This parallelization is often done through Threading Building Blocks (TBB).

As an example, let's consider prime number marking: create an array where $p[i]$ is one if i is prime, zero otherwise.

```
#pragma omp parallel \
    schedule(static)
for ( int i=0; i<nsize; i++ ) {
    results[i] =
        one_if_prime( number(i) );
}

// primepolicy.cpp
transform
( std::execution::par,
  numbers.begin(), numbers.end(),
  results.begin(),
  [] (int n ) -> int {
    return one_if_prime(n); }
);
```

As a result we find (figure 19.8) that the parallel algorithm is competitive with OpenMP loop parallelization for low thread counts, but not for higher.

19.5 Reductions

So far we have focused on loops with independent iterations. Reductions are a common type of loop with dependencies. There is an extended discussion of reductions in chapter 20.

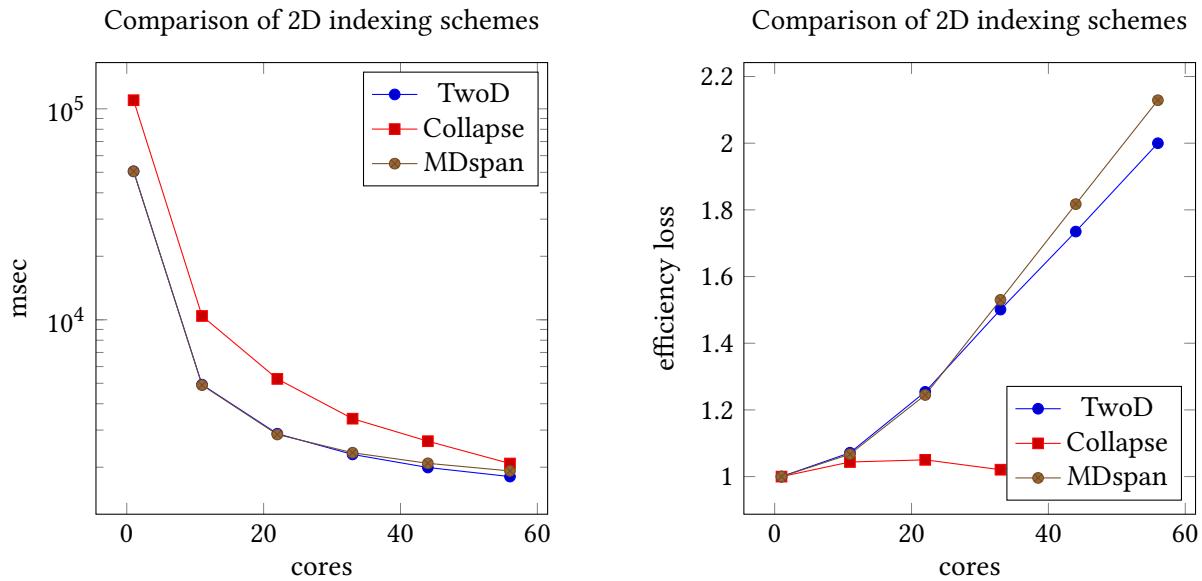


Figure 19.7: Different ways of indexing a five-point Laplace loop. Left: absolute times; Right: normalized to t_1 .

C++ note 12: Performance comparison. Figure 19.8 gives a performance comparison between OpenMP reductions and C++ execution policies.

19.6 Nested loops

19.6.1 Collapsing nested loops

In general, the more work there is to divide over a number of threads, the more efficient the parallelization will be. In the context of parallel loops, it is possible to increase the amount of work by parallelizing all levels of loops instead of just the outer one.

Example: in

```
for ( int i=0; i<N; i++ )
    for ( int j=0; j<N; j++ )
        A[i][j] = B[i][j] + C[i][j]
```

all N^2 iterations are independent, but a regular `omp for` directive will only parallelize one level. The `collapse` clause will parallelize more than one level:

```
#pragma omp for collapse(2)
for ( int i=0; i<N; i++ )
    for ( int j=0; j<N; j++ )
        A[i][j] = B[i][j] + C[i][j]
```

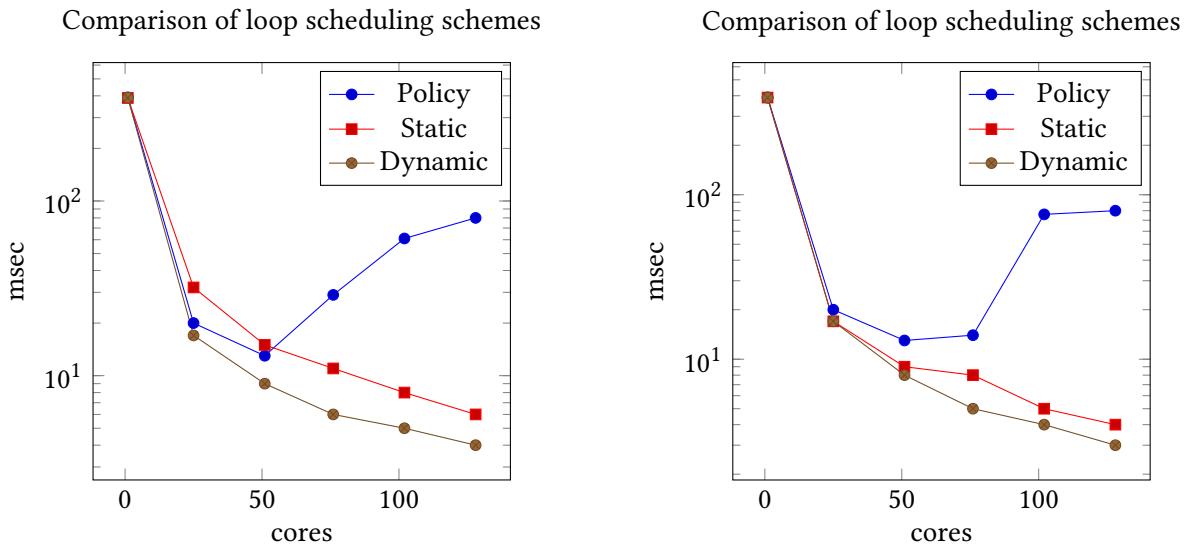


Figure 19.8: Load unbalanced algorithm (left: parallel loop, right: reduction) with 1. execution policy, 2. OpenMP static scheduling, 3. OpenMP dynamic scheduling

It is only possible to collapse perfectly nested loops, that is, the loop body of the outer loop can consist only of the inner loop; there can be no statements before or after the inner loop in the loop body of the outer loop. That is, the two loops in

```
for ( int i=0; i<N; i++ ) {
    y[i] = 0.;
    for ( int j=0; j<N; j++ )
        y[i] += A[i][j] * x[j]
}
```

can not be collapsed.

Exercise 19.10. You could rewrite the above code as

```
for ( int i=0; i<N; i++ )
    y[i] = 0.;
for ( int i=0; i<N; i++ ) {
    for ( int j=0; j<N; j++ )
        y[i] += A[i][j] * x[j]
}
```

Is it now correct to have the `collapse` directive on the nested loop?

Exercise 19.11. Consider this code for matrix transposition:

```
void transposer(int n, int m, double *dst, const double *src) {
    int blocksize;
    for (int i = 0; i < n; i += blocksize) {
        for (int j = 0; j < m; j += blocksize) {
            // transpose the block beginning at [i,j]
            for (int k = i; k < i + blocksize; ++k) {
```

```
        for (int l = j; l < j + blockszie; ++l) {
            dst[k + l*n] = src[l + k*m];
        }
    }
}
```

Assuming that the `src` and `dst` array are disjoint, which loops are parallel, and how many levels can you collapse?

19.6.2 Array traversal

Consider arrays

```
float Amat[N][N];  
float xvec[N],yvec[N];
```

and the operation $s \leftarrow y^t Ax$.

1. Code this as an OpenMP parallel double loop.
 2. Argue that the matrix A can be traversed two ways: by rows and columns, or by columns and rows, both giving the same result (in exact arithmetic).
 3. Argue that the loops can be collapsed with `collapse` directive.
 4. So now you have 4 variants in addition to the sequential code. Time these.

You should find that the row/column (or *row-major*) variant is faster. Can you find reasons for this?

19.7 Ordered iterations

Iterations in a parallel loop that are executed in parallel do not execute in lockstep. That means that in

```
#pragma omp parallel for
for ( ... i ... ) {
    ... f(i) ...
    printf("something with %d\n", i);
}
```

it is not true that all function evaluations happen more or less at the same time, followed by all print statements. The print statements can really happen in any order. The *ordered* clause coupled with the *ordered* directive can force execution in the right order:

```
#pragma omp parallel for ordered
for ( ... i ... ) {
    ... f(i) ...
    #pragma omp ordered
    printf("something with %d\n", i);
}
```

Example code structure:

```
#pragma omp parallel for shared(y) ordered
for ( ... i ... ) {
    int x = f(i)
    #pragma omp ordered
    y[i] += f(x)
    z[i] = g(y[i])
}
```

There is a limitation: each iteration can encounter only one *ordered* directive.

19.8 nowait

An OpenMP loop is a worksharing construct, after which execution in the parallel region goes back to replicated execution. To synchronize this, OpenMP inserts a *barrier*, meaning that threads wait for each other to reach this point. See section 23.1.1 for details.

The implicit barrier at the end of a work sharing construct can be cancelled with a *nowait* clause. This has the effect that threads that are finished can continue with the next code in the parallel region:

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (int i=0; i<N; i++) {
        ...
    }
    // more parallel code
}
```

In the following example, threads that are finished with the first loop can start on the second. Note that this requires both loops to have the same schedule. We specify the static schedule here to have an identical scheduling of iterations over threads:

```
#pragma omp parallel
{
    x = local_computation()
    #pragma omp for schedule(static) nowait
    for (int i=0; i<N; i++) {
        x[i] = ...
    }
    #pragma omp for schedule(static)
    for (int i=0; i<N; i++) {
        y[i] = ... x[i] ...
    }
}
```

19.9 While loops

OpenMP can only handle ‘for’ loops: *while loops* can not be parallelized. So you have to find a way around that. While loops are for instance used to search through data:

19. OpenMP topic: Loop parallelism

```
while ( a[i] !=0 && i<imax ) {  
    i++; }  
// now i is the first index for which a[i] is zero.
```

We replace the while loop by a for loop that examines all locations:

```
result = -1;  
#pragma omp parallel for  
for (int i=0; i<imax; i++) {  
    if (a[i]!=0 && result<0) result = i;  
}
```

Exercise 19.12. Show that this code has a race condition.

You can fix the race condition by making the condition into a critical section; section 23.2.2. In this particular example, with a very small amount of work per iteration, that is likely to be inefficient in this case (why?). A more efficient solution uses the `lastprivate` pragma:

```
result = -1;  
#pragma omp parallel for lastprivate(result)  
for (int i=0; i<imax; i++) {  
    if (a[i]!=0) result = i;  
}
```

You have now solved a slightly different problem: the result variable contains the *last* location where `a[i]` is zero.

19.10 Review questions

Exercise 19.13. The following loop can be parallelized with a `parallel for`. Is it correct to add the directive `collapse(2)`?

```
for (int i=0; i<N; i++) {  
    y[i] = 0.;  
    for (int j=0; j<N; j++)  
        y[i] += A[i][j] * x[j]  
}
```

Exercise 19.14. Same question for the nested loop here:

```
for (int i=0; i<N; i++)  
    y[i] = 0.;  
for (int i=0; i<N; i++) {  
    for (int j=0; j<N; j++)  
        y[i] += A[i][j] * x[j]  
}
```

Exercise 19.15. In this triple loop:

```
for (int i=0; i<n; i++)  
    for (int j=0; j<n; j++)  
        for (int k=0; k<kmax; k++)  
            x[i][j] += f(i,j,k)
```

what OpenMP directives do you use? Can you collapse all levels? Does it matter what the loop bounds are?

Chapter 20

OpenMP topic: Reductions

20.1 Reductions: why, what, how?

Parallel tasks often produce some quantity that needs to be summed or otherwise combined. If you write:

```
int sum=0;
#pragma omp parallel for
for (int i=0; i<N; i++)
    sum += f(i);
```

you will find that the `sum` value depends on the number of threads, and is likely not the same as when you execute the code sequentially. The problem here is the *race condition* involving the `sum` variable, since this variable is shared between all threads.

We will discuss several strategies of dealing with this.

20.1.1 Reduction clause

The easiest way to effect a reduction is of course to use the `reduction` clause. Adding this to an `omp parallel` region has the following effect:

- OpenMP will make a copy of the reduction variable per thread, initialized to the identity of the reduction operator, for instance 1 for multiplication.
- Each thread will then reduce into its local variable;
- At the end of the parallel region, the local results are combined, again using the reduction operator, into the global variable.

The simplest case is a reduction over a parallel loop. Here we compute $\pi/4$ as a *Riemann sum*:

```
// pi.c
#pragma omp parallel for reduction(+:pi4)
for (int isample=0; isample<N; isample++) {
    float xsample = isample * h;
    float y = sqrt(1-xsample*xsample);
    pi4 += h*y;
}
```

You can also reduce over `sections`:

```
// sectionreduct.c
float y=0;
#pragma omp parallel reduction(+:y)
#pragma omp sections
{
#pragma omp section
    y += f();
#pragma omp section
    y += g();
}
```

Another reduction, this time over a parallel region, without any work sharing:

```
// reductpar.c
m = INT_MIN;
#pragma omp parallel reduction(max:m) num_threads(ndata)
{
    int t = omp_get_thread_num();
    int d = data[t];
    m = d>m ? d : m;
};
```

If you want to reduce multiple variables with the same operator, use

`reduction(+:x,y,z)`

For multiple reduction with different operators, use more than one clause.

Remark A reduction is one of those cases where the parallel execution can have a slightly different value from the one that is computed sequentially, because floating point operations are not associative, so roundoff will lead to differing results. See HPC book, section ?? for more explanation.

The OpenMP standard does not even specify that two runs with the same number of threads, and the same scheduling, have to give identical results. Some runtimes may have a setting to enforce this; for instance `KMP_DETERMINISTIC_REDUCTION` for the Intel runtime.

20.1.2 Code your own reduction

While using a `reduction` clause is the preferred way of dealing with codes as in section 20.1 above, it can be instructive to look at other mechanisms.

The most immediate way is to eliminate the race condition by declaring a *critical section*:

```
double result = 0;
#pragma omp parallel
{
    double local_result;
    int num = omp_get_thread_num();
    if (num==0)      local_result = f(x);
    else if (num==1) local_result = g(x);
    else if (num==2) local_result = h(x);
#pragma omp critical
    result += local_result;
}
```

This is a good solution if the amount of serialization in the critical section is small compared to computing the functions f, g, h . On the other hand, you may not want to do that in a loop:

```
double result = 0;
#pragma omp parallel
{
    double local_result;
#pragma omp for
    for (i=0; i<N; i++) {
        local_result = f(x,i);
#pragma omp critical
        result += local_result;
    } // end of for loop
}
```

Exercise 20.1. Can you think of a small modification of this code, that still uses a critical section, that is more efficient? Time both codes.

20.1.2.1 False sharing

If your code can not be easily structured as a reduction, you can realize the above scheme by hand by ‘duplicating’ the global variable and gather the contributions later. This example presumes three threads, and gives each a location of their own to store the result computed on that thread:

```
double result,local_results[3];
#pragma omp parallel
{
    int num = omp_get_thread_num();
    if (num==0)      local_results[num] = f(x)
    else if (num==1) local_results[num] = g(x)
    else if (num==2) local_results[num] = h(x)
}
result = local_results[0]+local_results[1]+local_results[2]
```

While this code is correct, it may be inefficient because of a phenomenon called *false sharing*. Even though the threads write to separate variables, those variables are likely to be on the same *cacheline* (see HPC book, section ?? for an explanation). This means that the cores will be wasting a lot of time and bandwidth updating each other’s copy of this cacheline.

False sharing can be prevent by giving each thread its own cacheline:

```
double result,local_results[3][8];
#pragma omp parallel
{
    int num = omp_get_thread_num();
    if (num==0)      local_results[num][1] = f(x)
// et cetera
}
```

A more elegant solution gives each thread a true local variable, and uses a critical section to sum these, at the very end:

```
double result = 0;
#pragma omp parallel
```

```
{  
    double local_result;  
    local_result = ....  
    #pragma omp critical  
        result += local_result;  
}
```

20.1.3 Exercises

Exercise 20.2. Compute π by *numerical integration*. We use the fact that π is the area of the unit circle, and we approximate this by computing the area of a quarter circle using *Riemann sums*.

- Let $f(x) = \sqrt{1 - x^2}$ be the function that describes the quarter circle for $x = 0 \dots 1$;
- Then we compute

$$\pi/4 \approx \sum_{i=0}^{N-1} \Delta x f(x_i) \quad \text{where } x_i = i\Delta x \text{ and } \Delta x = 1/N$$

Write a program for this, and parallelize it using OpenMP parallel for directives. Measure attained speedup.

1. Put a `parallel` directive around your loop. Does it still compute the right result? Does the time go down with the number of threads? (The answers should be no and no.)
2. Change the `parallel` to `parallel for` (or `parallel do`). Now is the result correct? Does execution speed up? (The answers should now be no and yes.)
3. Put a `critical` directive in front of the update. (Yes and very much no.)
4. Remove the `critical` and add a clause `reduction(+:quarterpi)` to the `for` directive. Now it should be correct and efficient.

Remark In this exercise you may have seen the runtime go up a couple of times where you weren't expecting it. The issue here is false sharing; see HPC book, section ?? for more explanation.

Exercise 20.3. The Jacobi method for solving linear system $Ax = b$ is given by

$$x_i^{(n+1)} = a_{ii} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(n)} \right)$$

Insert OpenMP directives in the code, and check that it converges with the same precision, regardless the number of threads.

Study speedup. Does the problem size play a role?

Exercise 20.4. How much performance improvement do you get from considering

- removing barriers by `nowait` clauses
- affinity
- first-touch

Exercise 20.5. Experiment with scheduling options.

Do you see any effect?

In particular try small chunk sizes.

Exercise 20.6. Can you put the whole iteration loop in a parallel region?

Does this give further performance improvement?

20.2 Built-in reduction

20.2.1 Operators

Arithmetic reductions: `+, *, -, max, min`. The minus operator is deprecated as of OpenMP-5.2.

Logical operator reductions in C: `& && | || ^`

Logical operator reductions in Fortran: `.and. .or. .eqv. .neqv. .iand. .ior. .ieor.`

Reduction can be applied to any type for which the operator is defined. The types to which `max/min` are applicable are limited.

Exercise 20.7. The maximum and minimum reductions were not added to OpenMP until

OpenMP-3.1. Write a parallel loop that computes the maximum and minimum values in an array without using the `reduction` directive. Discuss the various options. Do timings to evaluate the speedup that is attained and to find the best option.

20.2.2 Reduction on arrays

Starting with the OpenMP-4.5 standard, you can reduce on statically and dynamically allocated arrays:

```
// reductarray.c
int data[nthreads];
#pragma omp parallel for schedule(static,1) \
    reduction(+:data[:nthreads])
for (int it=0; it<nthreads; it++) {
    for (int i=0; i<nthreads; i++)
        data[i]++;
}

int *alloced = (int*)malloc( nthreads*sizeof(int) );
for (int i=0; i<nthreads; i++)
    alloced[i] = 0;
#pragma omp parallel for schedule(static,1) \
    reduction(+:alloced[:nthreads])
for (int it=0; it<nthreads; it++) {
    for (int i=0; i<nthreads; i++)
        alloced[i]++;
}
```

C++ note 13: Reductions on vectors. Use the `data` method to extract the array on which to reduce. However, this does not work:

```
vector<float> x;
#pragma omp parallel reduction(+:x.data())
```

because the reduction clause wants a variable, not an expression, for the array, so you need an extra bare pointer:

```
// reductarray.cxx
vector<int> data(nthreads,0);
int *datadata = data.data();
#pragma omp parallel for schedule(static,1) \
    reduction(+:datadata[:nthreads])
```

In the course of the reduction, OpenMP may need to allocate temporary arrays. This may run into limitations of stack size. You may need to increase the value of the `OMP_STACKSIZE` environment variable.

20.3 Initial value for reductions

The treatment of initial values in reductions is slightly involved.

```
x = init_x
#pragma omp parallel for reduction(min:x)
for (int i=0; i<N; i++)
    x = min(x,data[i]);
```

Each thread does a partial reduction, but its initial value is not the user-supplied `init_x` value, but a value dependent on the operator. In the end, the partial results will then be combined with the user initial value. The initialization values are mostly self-evident, such as zero for addition and one for multiplication. For min and max they are respectively the maximal and minimal representable value of the result type.

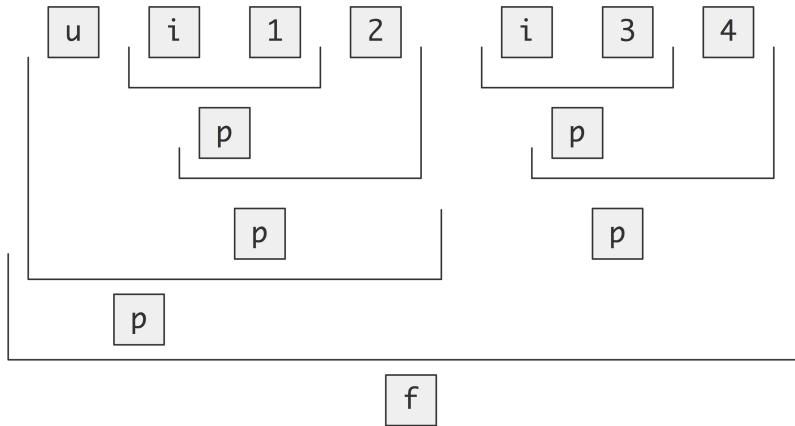


Figure 20.1: Reduction of four items on two threads, taking into account initial values.

Figure 20.1 illustrates this, where `1, 2, 3, 4` are four data items, `i` is the OpenMP initialization, and `u` is the user initialization; each `p` stands for a partial reduction value. The figure is based on execution using two threads.

Exercise 20.8. Write a program to test the fact that the partial results are initialized to the unit of the reduction operator.

20.4 User-defined reductions

In a loop that performs a reduction, most of the element-by-element reduction as done in user code. However, in a parallel version of that loop, OpenMP needs to perform that same reduction on the partial results from the threads. Thus, if you want to perform your own reduction, you need to declare this reduction to OpenMP.

With *user-defined reductions*, the programmer specifies the function that does the elementwise comparison. We discuss two strategies:

1. In non-Object-Oriented (OO) languages you can define a function, and declare that to be a reduction operator with the `declare reduction` construct.

2. In OO languages (C++ and Fortran2003) you can overload ordinary operators for types, including class objects.

20.4.1 Reduction functions

This takes two steps.

1. You need a function of two arguments that returns the result of the comparison. You can do this yourself, but, especially with the C++ standard library, you can use functions such as `std::vector::insert`.
2. Specifying how this function operates on two variables `omp_out` and `omp_in`, corresponding to the partially reduced result and the new operand respectively. The new partial result should be left in `omp_out`.
3. Optionally, you can specify the value to which the reduction should be initialized.

This is the syntax of the definition of the reduction, which can then be used in multiple *reduction* clauses.

```
#pragma omp declare reduction
( identifier : typelist : combiner )
[initializer(initializer-expression)]
```

where:

`identifier` is a name; this can be overloaded for different types, and redefined in inner scopes.

`typelist` is a list of types.

`combiner` is an expression that updates the internal variable `omp_out` as function of itself and `omp_in`.

`initializer` sets `omp_priv` to the identity of the reduction; this can be an expression or a brace initializer. Often: `initializer(omp_priv=omp_orig)` to use the initial value.

Fortran note 22: Reduction declaration. The declaration statement has to be in the declaration section of your subroutine.

Exercise 20.9. Write a custom reduction that finds the coordinate (x, y) with maximal norm

$\sqrt{x^2 + y^2}$. Test on an array of randomly generated coordinates. Decide on how to store this:

```
double coords[N][2];
double coords[2*N];
struct coord { double x,double y };
double *coords = (double*) malloc(2*N*sizeof(double));
```

In C++ consider writing a *Coordinate* class and overloading an operator on it.

20.4.1.1 Explicit expressions

For very simple cases:

```
for (i=0; i<N; i++) {
    if (abs(data[i]) < result) {
        result = abs(data[i]);
    }
}
```

you can declare the reduction through an expression:

```
// reductexpr.c
#pragma omp declare reduction \
(minabs : int : \
    omp_out = abs(omp_in) > omp_out ? omp_out : abs(omp_in) ) \
    initializer (omp_priv=LARGENUM)
```

and use that in the `reduction` clause:

```
#pragma omp parallel for reduction(minabs:result)
```

for the above loop

```
for (i=0; i<N; i++) {
    if (abs(data[i]) < result) {
        result = abs(data[i]);
    }
}
```

C++ note 14: Lambda expressions in declared reductions. You can use lambda expressions in the explicit expression for a declared reduction:

```
// reductexpr.cxx
#pragma omp declare reduction \
(minabs : int : \
omp_out = \
[] (int x,int y) -> int { \
    return abs(x) > abs(y) ? abs(y) : abs(x); } \
(omp_in,omp_out) ) \
initializer (omp_priv=limit::max())
```

You can not assign the lambda expression to a variable and use that, because `omp_in`/`omp_out` are the only variables allowed in the explicit expression.

20.4.1.2 Reduction functions

For instance, recreating the maximum reduction would look like this:

```
// ireduct.c
int mymax(int r,int n) {
// r is the already reduced value
// n is the new value
    int m;
    if (n>r) {
        m = n;
    } else {
        m = r;
    }
    return m;
}
#pragma omp declare reduction \
(rwz:int:omp_out=mymax(omp_out,omp_in)) \
initializer(omp_priv=INT_MIN)
m = INT_MIN;
#pragma omp parallel for reduction(rwz:m)
    for (int idata=0; idata<n; idata++)
        m = mymax(m,data[idata]);
```

Exercise 20.10. Write a reduction routine that operates on an array of nonnegative integers, finding the smallest nonzero one. If the array has size zero, or entirely consists of zeros, return -1.

C++ note 15: Reduction over iterators. Support for C++ iterators

```
#pragma omp declare reduction \
(merge           // identifier
 : std::vector<int> // typelist
 : omp_out.insert(omp_out.end(), omp_in.begin(),
                 omp_in.end()) // combiner
)
```

C++ note 16: Templatized reductions. You can reduce with a templated function if you put both the declaration and the reduction in the same templated function:

```
template<typename T>
T generic_reduction( vector<T> tdata ) {
#pragma omp declare reduction \
(rwzt:T:omp_out=reduce_without_zero<T>(omp_out,omp_in))      \
initializer(omp_priv=-1.f)

    T tmin = -1;
#pragma omp parallel for reduction(rwzt:tmin)
    for (int id=0; id<tdata.size(); id++)
        tmin = reduce_without_zero<T>(tmin,tdata[id]);
    return tmin;
};
```

which is then called with specific data:

```
auto tmin = generic_reduction<float>(fdata);
```

C++ note 17: Example: reduction over a map. Reduction over a `std::map` by merging thread-local maps:

```
// charcount.cxx
template<typename key>
class bincounter : public map<key,int> {
public:
// merge this with other map
    void operator+=
        ( const bincounter<key>& other ) {
            for ( auto [k,v] : other )
                if ( map<key,int>::contains(k) )
                    this->at(k) += v;
                else
                    this->insert( {k,v} );
    };
// insert one char in this map
    void inc(char k) {
        if ( map<key,int>::contains(k) )
            this->at(k) += 1;
        else
            this->insert( {k,1} );
    };
};
```

```
/*
 * Reduction loop in main program
 */
bincounter<char> charcount;
#pragma omp parallel for reduction(+ : charcount)
for ( int i=0; i<text.size(); i++ )
    charcount.inc( text[i] );
```

20.4.2 Overloaded operators

Fortran note 23: Reduction on derived types. Reduction can be applied to any derived type that has the reduction operator defined.

```
!! reducttype.F90
Type inttype
  integer :: value = 0
end type inttype
Interface operator(+)
  module procedure addints
end Interface operator(+)

Type(inttype),dimension(nsize) :: intarray
Type(inttype) :: intsum = inttype(0)
!$OMP declare reduction(+:inttype:omp_out=
  omp_out+omp_in)
!$OMP parallel do reduction(+:intsum)
do i=1,nsize
  intsum = intsum + intarray(i)
end do
```

But note the extra `declare` clause.

C++ note 18: Reduction on class objects. Reduction can be applied to any class for which the reduction operator is defined as `operator+` or whichever operator the case may be.

```
// reductclass.cxx
class Thing {
private:
  float x{0.f};
public:
  Thing() = default;
  Thing( float x ) : x(x) {};
  Thing operator+
  ( const Thing& other ) {
    return Thing( x + other.x );
  };
};
```

```
vector< Thing >
things(500,Thing(1.f));
Thing result(0.f);
#pragma omp parallel for \
reduction( +:result )
for ( const auto& t : things )
  result = result + t;
```

A default constructor is required for the internally used init value; see figure 20.1.

20.5 Scan / prefix operations

A ‘scan’ or *prefix operation* is like a reduction, except that you’re interested in the partial results. For this OpenMP, as of OpenMP-5.0, has the `scan` directive. This needs the following:

- The reduction clause gets a modifier `inscan`:

```
#pragma omp parallel for reduction(inscan,+:sumvar)
```

- In the body of the parallel loop there is a `scan` directive that allows you to store the partial results. For inclusive scans the reduction variable is updated before the `scan` pragma:

```
sumvar // update
#pragma omp scan inclusive(sumvar)
partials[i] = sumvar
```

For exclusive scans the reduction variable is updated after the `scan` pragma:

```
partials[i] = sumvar
#pragma omp scan inclusive(sumvar)
sumvar // update
```

Code:

```
// scanintsum.c
partial_sum=0;
#pragma omp parallel for \
reduction(inscan,+:partial_sum)
for (int i=0; i<nthreads; i++) {
    partial_sum += amounts[i];
    # pragma omp scan inclusive(partial_sum)
    inc_partials[i] = partial_sum;
}
partial_sum=0;
#pragma omp parallel for \
reduction(inscan,+:partial_sum)
for (int i=0; i<nthreads; i++) {
    exc_partials[i] = partial_sum;
    # pragma omp scan exclusive(partial_sum)
    partial_sum += amounts[i];
}
```

Output:
[examples/omp/c] scansum:

<i>Summing</i>	:	1	2	3	4	5	6	7	8
<i>Inclusive</i>	:	1	3	6	10	15	21	28	36
<i>Exclusive</i>	:	0	1	3	6	10	15	21	28

20.6 Reductions and floating-point math

The mechanisms that OpenMP uses to make a reduction parallel go against the strict rules for floating point expression evaluation in C; see HPC book, section ???. OpenMP ignores this issue: it is the programmer's job to ensure proper rounding behavior.

20.7 Reductions in C++ standard algorithms

C++ note 19: Reductions on parallel standard algorithms. In section 19.4.2 you saw how certain loop constructs can be realized in C++ through the *execution policy* argument of the standard algorithms. The same holds for reductions.

```
#pragma omp parallel for \
schedule(guided,8) \
reduction(+:prime_count)
for ( auto n : numbers ) {
    prime_count += one_if_prime( n );
}
```

20. OpenMP topic: Reductions

```
// reducepolicy.cpp
prime_count = transform_reduce
( std::execution::par,
  numbers.begin() ,numbers.end() ,
  0,
  std::plus<>{},
  [] ( int n ) -> int {
    return one_if_prime(n); }
);

Threads: 1
TBB: Time:      391 msec
Stat: Time:     390 msec
Dyn:  Time:     389 msec

Threads: 25
TBB: Time:      20 msec
Stat: Time:     17 msec
Dyn:  Time:     17 msec

Threads: 51
TBB: Time:      13 msec
Stat: Time:      9 msec
Dyn:  Time:      8 msec

Threads: 76
TBB: Time:      14 msec
Stat: Time:      8 msec
Dyn:  Time:      5 msec

Threads: 102
TBB: Time:      76 msec
Stat: Time:      5 msec
Dyn:  Time:      4 msec

Threads: 128
TBB: Time:      80 msec
Stat: Time:      4 msec
Dyn:  Time:      3 msec
```

Chapter 21

OpenMP topic: Work sharing

The declaration of a *parallel region* establishes a team of threads. This offers the possibility of parallelism, but to actually get meaningful parallel activity you need something more. OpenMP uses the concept of a *work sharing construct*: a way of dividing parallelizable work over a team of threads.

You have already seen loop parallelism as a way of distributing parallel work in chapter 19. We will now discuss other work sharing constructs.

21.1 Work sharing constructs

The work sharing constructs are:

- **for** (for C) or **do** (for Fortran): The threads divide up the loop iterations among themselves; see 19.1.
- **sections**: The threads divide a fixed number of sections between themselves; see section 21.2.
- **single**: The section is executed by a single thread; section 21.3.
- **task**: See chapter 24.
- **workshare**: This can parallelize Fortran array syntax; section 21.4.

21.2 Sections

A parallel loop is an example of independent work units that are numbered. If you have a pre-determined number of independent work units, the **sections** is more appropriate. In a **sections** construct can be any number of **section** constructs. These need to be independent, and they can be execute by any available thread in the current team, including having multiple sections done by the same thread.

```
#pragma omp sections
{
    #pragma omp section
    // one calculation
    #pragma omp section
    // another calculation
}
```

This construct can be used to divide large blocks of independent work. Suppose that in the following line, both $f(x)$ and $g(x)$ are big calculations:

$$y = f(x) + g(x) + h(x)$$

You could then write

```
// sections.c
#pragma omp parallel sections
{
    #pragma omp section
    fx = f(1.);
    #pragma omp section
    gx = g(1.);
    #pragma omp section
    hx = h(1.);
}
float s = fx+gx+hx;
```

Instead of using two temporaries, you could also use a critical section; see section 23.2.2. However, the best solution is have a `reduction` clause on the `parallel sections` directive. You could then write

```
float s=0;
#pragma omp parallel sections reduction(+:s)
{
    #pragma omp section
    s += f(1.);
    #pragma omp section
    s += g(1.);
    #pragma omp section
    s += h(1.);
}
```

21.3 Single thread execution

OpenMP has two mechanisms for letting a code section be executed by only a single thread. (Note: that is different from critical section which are executed by a single thread *at a time*.) The `single` directive is to be used for sections that are part of the control flow, since it has an implicit concluding barrier. The `master` and `masked` directives are similar, but assign the execution to the primary thread, and have no concluding barrier.

21.3.1 Single

The `single` pragma limits the execution of a block to a single thread. This can for instance be used to print tracing information or doing I/O operations.

```
#pragma omp parallel
{
    #pragma omp single
    printf("We are starting this section!\n");
    // parallel stuff
}
```

Another use of `single` is to perform initializations in a parallel region:

```
int a;
#pragma omp parallel
{
    #pragma omp single
    a = f(); // some computation
    #pragma omp sections
        // various different computations using a
}
```

The point of the `single` directive in this last example is that the computation needs to be done only once, because of the shared memory. Since it's a work sharing construct there is an *implicit barrier* after it, which guarantees that all threads have the correct value in their local memory (see section 23.4).

Exercise 21.1. What is the difference between this approach and how the same computation would be parallelized in MPI?

21.3.2 Masked/master

The `masked` and `master` directives also enforces execution on a single thread, specifically the primary thread of the team. This is not a work sharing construct, and therefore does not have the synchronization through the implicit barrier.

Remark The `masked` directive is new in OpenMP-5.1. The `master` directive is deprecated as of OpenMP-5.2.

Exercise 21.2. Modify the above code to read:

```
int a;
#pragma omp parallel
{
    #pragma omp master
    a = f(); // some computation
    #pragma omp sections
        // various different computations using a
}
```

This code is no longer correct. Explain.

The `masked` directive has a `filter` clause, that can be used to select other threads than the primary:

```
#pragma omp masked filter(2) // thread #2
```

21.3.3 More

Above we motivated the `single` directive as a way of initializing shared variables. It is also possible to use `single` to initialize private variables. In that case you add the `copyprivate` clause. This is a good solution if setting the variable takes I/O.

Exercise 21.3. Give two other ways to initialize a private variable, with all threads receiving the same value. Can you give scenarios where each of the three strategies would be preferable?

21.4 Fortran array syntax parallelization

The `parallel do` directive is used to parallelize loops, and this applies to both C and Fortran. However, Fortran also has implied loops in its *array syntax*. To parallelize array syntax you can use the `workshare` directive.

21. OpenMP topic: Work sharing

The `workshare` directive exists only in Fortran. It can be used to parallelize the implied loops in *array syntax*, as well as *forall* loops.

We compare two version of $C \leftarrow C + A \times B$ (where all operations are elementwise), running on *TACC Frontera* up to 56 cores.

Workshare based:

```
!! workshare2d.F90
    !$omp parallel workshare
    C = A*B + C
    !$omp end parallel workshare
```

SIMD'ized loop

```
!$omp parallel do simd
do i=1,dim
    do j=1,dim
        C(i,j) = C(i,j) + A(i,j) * B(i,j)
    end do
end do
!$omp end parallel do simd
```

With results:

SIMD times :
0.07115 0.04053 0.02498 0.01609 0.01210 0.01247 0.01765 0.02689

Speedup:

1 1.75549 2.84828 4.422 5.88017 5.70569 4.03116 2.64597

Workshare times:

0.06188 0.03186 0.01625 0.00867 0.00619 0.00379 0.00354 0.00373

Speedup:

1 1.94225 3.808 7.13725 9.99677 16.3272 17.4802 16.5898

Chapter 22

OpenMP topic: Controlling thread data

In a parallel region there are two types of data: private and shared. In this sections we will see the various way you can control what category your data falls under; for private data items we also discuss how their values relate to shared data.

22.1 Shared data

In a parallel region, any data declared outside it will be shared: any thread using a variable `x` will access the same memory location associated with that variable.

Example:

```
int x = 5;
#pragma omp parallel
{
    x = x+1;
    printf("shared: x is %d\n",x);
}
```

All threads increment the same variable, so after the loop it will have a value of five plus the number of threads; or maybe less because of the data races involved. This issue is discussed in HPC book, section ??; see [23.2.2](#) for a solution to data races in OpenMP.

22.2 Private data

In the C/C++ language it is possible to declare variables inside a *lexical scope*; roughly: inside curly braces. This concept extends to OpenMP parallel regions and directives: any variable declared in a block following an OpenMP directive will be local to the executing thread.

In the following example, each thread creates a private variable `x` and sets it to a unique value:

22. OpenMP topic: Controlling thread data

Code:

```
// private.c
int x=5;
#pragma omp parallel num_threads(4)
{
    int t = omp_get_thread_num(),
        x = t+1;
    printf("Thread %d sets x to %d\n",t,x);
}
printf("Outer x is still %d\n",x);
```

Output:

```
[examples/omp/c] private:
Thread 3 sets x to 4
Thread 2 sets x to 3
Thread 0 sets x to 1
Thread 1 sets x to 2
Outer x is still 5
```

After the parallel region the outer variable `x` will still have the value 5: there is no *storage association* between the private variable and global one.

Fortran note 24: Private variables in parallel region. The Fortran language does not have this concept of scope, so you have to use a `private` clause:

Code:

```
!! private.F90
x=5
!$omp parallel private(x,t) num_threads(4)
    t = omp_get_thread_num()
    x = t+1
    print'("Thread ",i2," sets x to ",i2)',t,x
!$omp end parallel
    print'("Outer x is still ",i2)',x
```

Output:

```
[examples/omp/f] private:
Thread 0 sets x to 1
Thread 2 sets x to 3
Thread 3 sets x to 4
Thread 1 sets x to 2
Outer x is still 5
```

C++ note 20: Privatizing class members. Class members can only be privatized from (non-static) class methods.

In this example `f` can not be static:

```
// private.cxx
class foo {
private:
    int x;
public:
    void f() {
#pragma omp parallel private(x)
        somefunction(x);
    };
}
```

```
};
```

You can not privatize just a member:

```
// privateno.cxx
class foo { public: int x; };
int main() {
    foo thing;
#pragma omp parallel private(thing.x) // NOPE
```

The `private` directive declares data to have a separate copy in the memory of each thread. Such private variables are initialized as they would be in a main program. Any computed value goes away at the end of the parallel region. (However, see `lastprivate` below.) Thus, you should not rely on any initial value, or on the value of the outer variable after the region.

```
int x = 5;
#pragma omp parallel private(x)
{
    x = x+1; // dangerous
    printf("private: x is %d\n",x);
}
printf("after: x is %d\n",x);
```

Data that is declared private with the `private` directive is put on a separate *stack per thread*. The OpenMP standard does not dictate the size of these stacks, but beware of *stack overflow*. A typical default is a few megabytes; you can control it with the environment variable `OMP_STACKSIZE`. (You can find the current value by setting `OMP_DISPLAY_ENV`.) Its values can be literal or with suffixes:

```
123 456k 567K 678m 789M 246g 357G
```

Remark The OpenMP stack size also plays a role in reductions on arrays; section 20.2.2.

A normal *Unix process* also has a stack, but this is independent of the OpenMP stacks for private data. You can query or set the Unix stack with `ulimit`:

```
[] ulimit -s
64000
[] ulimit -s 8192
[] ulimit -s
8192
```

The Unix stack can grow dynamically as space is needed. This does not hold for the OpenMP stacks: they are immediately allocated at their requested size. Thus it is important not too make them too large.

22.3 Data in dynamic scope

Functions that are called from a parallel region fall in the *dynamic scope* of that parallel region. The rules for variables in that function are as follows:

- Any variables locally defined to the function are private.
- `static` variables in C and `save` variables in Fortran are shared.
- The function arguments inherit their status from the calling environment.

Fortran note 25: Saved variables. Variables in subprograms are private, as in C, except if they have the *Save* attribute. This attribute is implicitly given to any variable that has value-initialized.

In the following example we have two almost identical routines, except that the first does value-initialization on the local variable, thereby in effect making it shared. The second routine does not have that problem.

Code:

```

subroutine savehello
    use omp_lib
    implicit none
    integer :: thread = -1
    thread = omp_get_thread_num()
    print *, "Hello from", thread
end subroutine savehello
subroutine finehello
    use omp_lib
    implicit none
    integer :: thread
    thread = omp_get_thread_num()
    print *, "World from", thread
end subroutine finehello

```

Output:

[examples/omp/f] save:

Hello from	3
World from	0
World from	1
World from	2
World from	3

22.4 Temporary variables in a loop

It is common to have a variable that is set and used in each loop iteration:

```

#pragma omp parallel for
for ( ... i ... ) {
    x = i*h;
    s = sin(x); c = cos(x);
    a[i] = s+c;
    b[i] = s-c;
}

```

By the above rules, the variables *x*, *s*, *c* are all shared variables. However, the values they receive in one iteration are not used in a next iteration, so they behave in fact like private variables to each iteration.

- In both C and Fortran you can declare these variables private in the parallel for directive.
- In C you can also define the variables locally inside the loop.

Sometimes, even if you forget to declare these temporaries as private, the code may still give the correct output. That is because the compiler can sometimes eliminate them from the loop body, since it detects that their values are not otherwise used.

22.5 Default

There are default rules for whether data in OpenMP constructs is private or shared, and you can control this explicitly.

First the default behavior:

- Variables declared outside a parallel region are shared as described above;
- Loop variables in an `omp for` are private;
- Local variables in the parallel region are private.

You can alter this default behavior with the `default` clause:

```
#pragma omp parallel default(shared) private(x)
{ ... }
#pragma omp parallel default(private) shared(matrix)
{ ... }
```

and if you want to play it safe:

```
#pragma omp parallel default(None) private(x) shared(matrix)
{ ... }
```

- The `shared` clause means that all variables from the outer scope are shared in the parallel region; any private variables need to be declared explicitly. This is the default behavior.
- The `private` clause means that all outer variables become private in the parallel region. They are not initialized; see the next option. Any shared variables in the parallel region need to be declared explicitly. This value is not available in C.
- The `firstprivate` clause means all outer variables are private in the parallel region, and initialized with their outer value. Any shared variables need to be declared explicitly. This value is not available in C.
- The `none` option is good for debugging, because it forces you to specify for each variable in the parallel region whether it's private or shared. Also, if your code behaves differently in parallel from sequential there is probably a data race. Specifying the status of every variable is a good way to debug this.

22.6 First and last private

Above, you saw that private variables are completely separate from any variables by the same name in the surrounding scope. However, there are two cases where you may want some *storage association* between a private variable and a global counterpart.

First of all, private variables are created with an undefined value. You can force their initialization with `firstprivate`

```
int t=2;
#pragma omp parallel firstprivate(t)
{
    t += f( omp_get_thread_num() );
    g(t);
}
```

The variable `t` behaves like a private variable, except that it is initialized to the outside value.

Remark *Variables are `firstprivate` by default in tasks; see chapter 24.*

Secondly, you may want a private value to be preserved to the environment outside the parallel region. This really only makes sense in one case, where you preserve a private variable from the last iteration of a parallel loop, or the last section in an `sections` construct. This is done with `lastprivate`:

```
#pragma omp parallel for \
    lastprivate(tmp)
for (int i=0; i<N; i++) {
    tmp = .....
    x[i] = .... tmp ....
}
.... tmp ....
```

22.7 Array data

The rules for arrays are slightly different from those for scalar data:

1. Statically allocated data, that is with a syntax like

```
int array[100];
integer,dimension(:) :: array(100)
```

can be shared or private, depending on the clause you use.

2. Dynamically allocated data, that is, created with `malloc` or `allocate`, can only be shared.

Example of the first type: each thread gets a private copy of the array, properly initialized.

Code:

```
int array[nthreads];
for (int i=0; i<nthreads; i++)
    array[i] = 0;

#pragma omp parallel firstprivate(array)
{
    int t = omp_get_thread_num();
    array[t] = t+1;
}
```

Output:
[examples/omp/c] privarray:

Executing: OMP_PROC_BIND=true
 ↳OMP_NUM_THREADS=4 ./alloc
Array result:
0:0, 1:0, 2:0, 3:0,

Of course, since only the private copy is altered, the original array is unaffected.

On the other hand, in the following example each thread gets a private pointer, but all pointers point to the same object:

```
Code:
// alloc.c
int *array =
(int*) malloc(nthreads*sizeof(int));
for (int i=0; i<nthreads; i++)
array[i] = 0;

#pragma omp parallel firstprivate(array)
{
    int t = omp_get_thread_num();
// ptr arith: needs private array
    array += t;
    array[0] = t;
}
// ... print the array
```

Output:
[examples/omp/c] pointarray:
Array result:
0:0, 1:1, 2:2, 3:3,

C++ note 21: Vectors are copied, unlike arrays. Compare

```
Code:
// alloc.c
int *array =
(int*) malloc(nthreads*sizeof(int));
for (int i=0; i<nthreads; i++)
array[i] = 0;

#pragma omp parallel firstprivate(array)
{
    int t = omp_get_thread_num();
// ptr arith: needs private array
    array += t;
    array[0] = t;
}
// ... print the array
```

Output:
[examples/omp/c] pointarray:
Array result:
0:0, 1:1, 2:2, 3:3,

and

Code:

```
vector<int> array(nthreads);
#pragma omp parallel firstprivate(array)
{
    int t = omp_get_thread_num();
    array[t] = t+1;
}
// ... print the array
```

Output:

```
[examples/omp/cxx] privvector:
Array result:
0:0, 1:0, 2:0, 3:0,
```

22.8 Persistent data through `threadprivate`

Most data in OpenMP parallel regions is either inherited from the master thread and therefore shared, or temporary within the scope of the region and fully private. There is also a mechanism for *thread-private data*, which is not limited in lifetime to one parallel region. The `threadprivate` pragma is used to declare that each thread is to have a private copy of a variable:

```
#pragma omp threadprivate(var)
```

The variable needs be:

- a file or static variable in C,
- a static class member in C++, or
- a program variable or common block in Fortran.

22.8.1 Thread private initialization

If each thread needs a different value in its `threadprivate` variable, the initialization needs to happen in a parallel region.

In the following example a team of 7 threads is created, all of which set their `thread-private` variable. Later, this variable is read by a larger team: the variables that have not been set are undefined, though often simply zero:

```
// threadprivate.c
static int tp;
#pragma omp threadprivate(tp)

int main(int argc, char **argv) {

#pragma omp parallel num_threads(7)
tp = omp_get_thread_num();

#pragma omp parallel num_threads(9)
printf("Thread %d has %d\n",omp_get_thread_num(),tp);
```

Fortran note 26: Private common blocks. Named common blocks can be made thread-private with the syntax

```
$!OMP threadprivate( /blockname/ )
```

Example:

Code:

```
!! threadprivate.F90
common /threaddata/tp
integer :: tp
!$omp threadprivate(/threaddata/)
```

Output:

```
[examples/omp/f] private:
Thread 0 sets x to 1
Thread 2 sets x to 3
Thread 3 sets x to 4
Thread 1 sets x to 2
Outer x is still 5
```

On the other hand, if the thread private data starts out identical in all threads, the `copyin` clause can be used:

```
#pragma omp threadprivate(private_var)

private_var = 1;
#pragma omp parallel copyin(private_var)
private_var += omp_get_thread_num()
```

If one thread needs to set all thread private data to its value, the `copyprivate` clause can be used:

```
#pragma omp parallel
{
    ...
#pragma omp single copyprivate(private_var)
    private_var = read_data();
    ...
}
```

Threadprivate variables require `OMP_DYNAMIC` to be switched off.

22.8.2 Thread private example

The typical application for thread-private variables is in *random number generators*. A random number generator needs saved state, since it computes each next value from the current one. To have a parallel generator, each thread will create and initialize a private ‘current value’ variable. This will persist even when the execution is not in a parallel region; it gets updated only in a parallel region.

Exercise 22.1. Calculate the area of the *Mandelbrot set* by random sampling. Initialize the random number generator separately for each thread; then use a parallel loop to evaluate the points. Explore performance implications of the different loop scheduling strategies.

C++ note 22: Threadprivate random number generators. The new C++ `random` header has a threadsafe generator, by virtue of the statement in the standard that no STL object can rely on global state. The usual idiom can not be made threadsafe because of the initialization:

```
static random_device rd;
static mt19937 rng(rd);
```

However, the following works:

```
// privaterandom.cxx
static random_device rd;
static mt19937 rng;
#pragma omp threadprivate(rd)
#pragma omp threadprivate(rng)

int main() {

#pragma omp parallel
    rng = mt19937(rd());
```

C++ note 23: Threadprivate random use. Based on the previous note, you can use the generator safely and independently:

```
#pragma omp parallel
{
    stringstream res;
    uniform_int_distribution<int> percent(1, 100);
    res << "Thread " << omp_get_thread_num() << ": " << percent(rng) << "\n";
    cout << res.str();
}
```

22.9 Allocators

OpenMP was initially designed for shared memory. With accelerators (see chapter 27), non-coherent memory was added to this. In the OpenMP-5 standard, the story is further complicated, to account for new memory types such as *high-bandwidth memory* and *non-volatile memory*.

There are several ways of using the OpenMP memory allocators.

- First, in a directive on a static array:

```
float A[N], B[N];
#pragma omp allocate(A) \
    allocator(omp_large_cap_mem_alloc)
```

- As a clause on private variables:

```
#pragma omp task private(B) allocate(omp_const_mem_alloc: B)
```

- With `omp_alloc`, using a (possibly user-defined) allocator.

Next, there are memory spaces. The binding between OpenMP identifiers and hardware is implementation defined.

22.9.1 Pre-defined types

Allocators: `omp_default_mem_alloc`, `omp_large_cap_mem_alloc`, `omp_const_mem_alloc`, `omp_high_bw_mem_alloc`,
`omp_low_lat_mem_alloc`, `omp_cgroup_mem_alloc`, `omp_pteam_mem_alloc`, `omp_thread_mem_alloc`.

Memory spaces: `omp_default_mem_space`, `omp_large_cap_mem_space`, `omp_const_mem_space`,
`omp_high_bw_mem_space`, `omp_low_lat_mem_space`.

Chapter 23

OpenMP topic: Synchronization

In the constructs for declaring parallel regions above, you had little control over in what order threads executed the work they were assigned. This section will discuss *synchronization* constructs: ways of telling threads to bring a certain order to the sequence in which they do things.

- **critical**: a section of code can only be executed by one thread at a time; see [23.2.2](#).
- **atomic** *Atomic update* of a single memory location. Only certain specified syntax patterns are supported. This was added in order to be able to use hardware support for atomic updates.
- **barrier**: section [23.1](#).
- **locks**: section [23.3](#).
- **flush**: section [23.4](#).

Loop-related synchronization constructs were discussed earlier:

- **ordered**: section [19.7](#).
- **nowait**: section [19.8](#).

23.1 Barrier

A *barrier* defines a point in the code where all active threads will stop until all threads have arrived at that point. With this, you can guarantee that certain calculations are finished. For instance, in this code snippet, computation of *y* can not proceed until another thread has computed its value of *x*.

```
#pragma omp parallel
{
    int mytid = omp_get_thread_num();
    x[mytid] = some_calculation();
    y[mytid] = x[mytid]+x[mytid+1];
}
```

This can be guaranteed with a *barrier* pragma:

```
#pragma omp parallel
{
    int mytid = omp_get_thread_num();
    x[mytid] = some_calculation();
    #pragma omp barrier
    y[mytid] = x[mytid]+x[mytid+1];
}
```

23.1.1 Implicit barriers

Apart from the barrier directive, which inserts an explicit barrier, OpenMP has *implicit barriers* after a work sharing construct; see section 21.3. Thus the following code is well defined:

```
#pragma omp parallel
{
    #pragma omp for
    for (int mytid=0; mytid<number_of_threads; mytid++)
        x[mytid] = some_calculation();
    #pragma omp for
    for (int mytid=0; mytid<number_of_threads-1; mytid++)
        y[mytid] = x[mytid]+x[mytid+1];
}
```

You can also put each parallel loop in a parallel region of its own, but there is some overhead associated with creating and deleting the team of threads in between the regions.

At the end of a parallel region the team of threads is dissolved and only the primary thread continues. Therefore, there is an *implicit barrier at the end of a parallel region*. This barrier behavior can be canceled with the `nowait` clause.

You will often see the idiom

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (i=0; i<N; i++)
        a[i] = // some expression
    #pragma omp for
    for (i=0; i<N; i++)
        b[i] = ..... a[i] .....
```

Here the `nowait` clause implies that threads can start on the second loop while other threads are still working on the first. Since the two loops use the same schedule here, an iteration that uses `a[i]` can indeed rely on it that that value has been computed.

23.1.2 A barrier idiom

Replacing implicit barriers by explicit can be used to make larger parallel regions and thereby cutting down on thread creation cost:

Multiple parallel regions:

```
#pragma omp parallel
// some workshare

#pragma omp parallel
// another workshare
```

Merged into one:

```
#pragma omp parallel
{
    // some workshare
    #pragma omp barrier
    // another workshare
}
```

For example, although `while` loops are strictly parallelizable, you can enclose the loop in a parallel region and use barriers to synchronize:

```
#pragma omp parallel
{
    while ( /* something */ ) {
        // single thread code
        #pragma omp barrier
        #pragma omp for
        for ( /* ... */ ) ...
    }
}
```

23.2 Mutual exclusion

Sometimes it is necessary to limit a piece of code so that it can be executed by only one thread at a time. Such a piece of code is called a *critical section*, and OpenMP has several mechanisms for realizing this.

23.2.1 Race conditions

OpenMP, being based on shared memory, has a potential for *race conditions*. These happen when two threads access the same data item, with at least one access a write. The problem with race conditions is that programmer convenience runs counter to efficient execution.

For a simple example:

<p>Code:</p> <pre>// race.c #pragma omp parallel for shared(counter) for (int i=0; i<count; i++) counter += f(counter,i); printf("Counter should be %d, is %d\n", count,counter);</pre>	<p>Output:</p> <pre>[examples/omp/c] race: On 1 threads: Counter should be 100000, is 100000 On 2 threads: Counter should be 100000, is 100000 On 4 threads: Counter should be 100000, is 75000 On 8 threads: Counter should be 100000, is 87500 On 12 threads: Counter should be 100000, is 100000</pre>
--	--

The basic rule about multiple-thread access of a single data item is:

Any memory location that is *written* by one thread, can not be *read* by another thread in the same parallel region, if no synchronization is done.

To start with that last clause: any workshare construct ends with an *implicit barrier*, so data written before that barrier can safely be read after it.

23.2.2 critical and atomic

There are two pragmas for critical sections: *critical* and *atomic*. Both denote *atomic operations* in a technical sense. The first one is general and can contain an arbitrary sequence of instructions; the second one is more limited but has performance advantages.

Beginning programmers are often tempted to use *critical* for updates in a loop:

```
#pragma omp parallel
{
    int mytid = omp_get_thread_num();
    double tmp = some_function(mytid);
    // This Works, but Not Best Solution:
    #pragma omp critical
    sum += tmp;
}
```

but this should really be done with a *reduction* clause, which will be far more efficient.

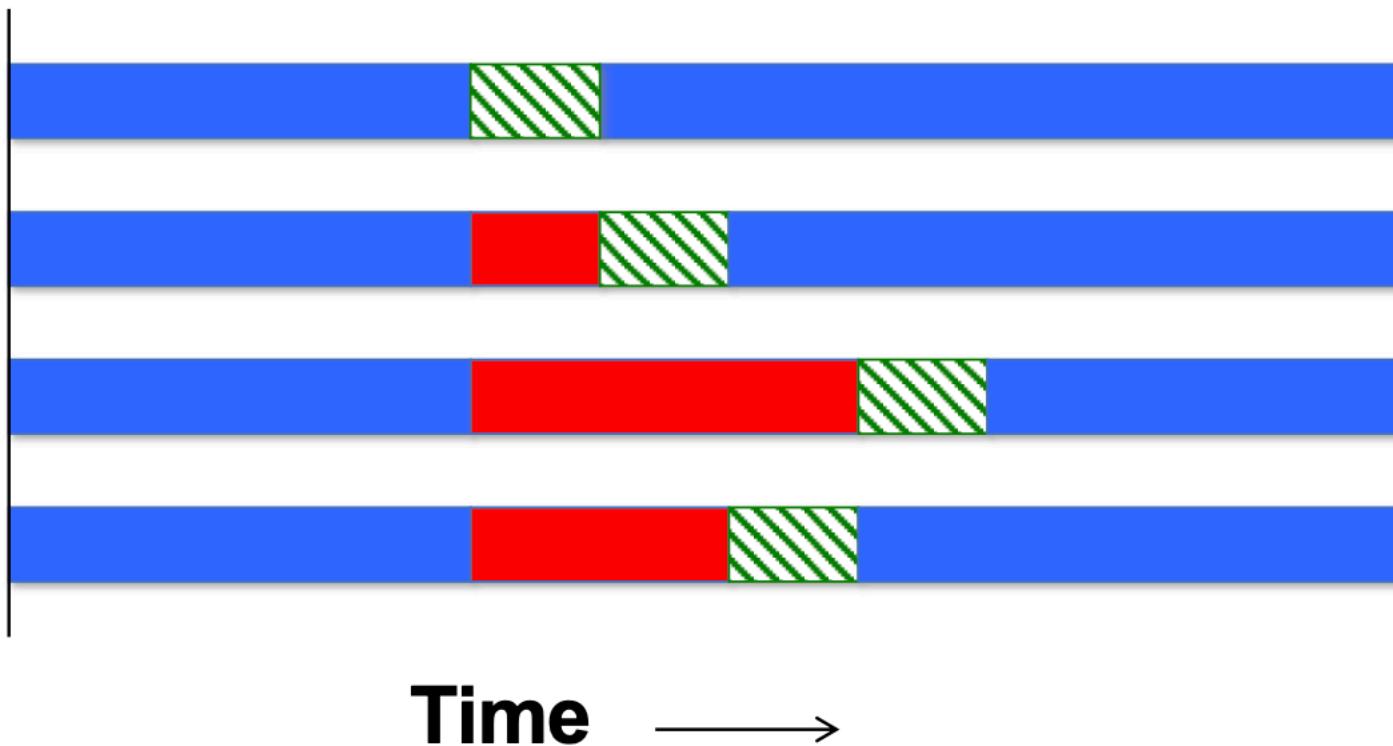


Figure 23.1: Idle time induced by a critical section

Figure fig:omp-idle illustrates how a critical section induces idle time: threads have to wait at the initial barrier until it's their turn to enter the critical section. In effect, the execution becomes sequential!

A good use of critical sections is doing file writes or database updates.

Exercise 23.1. Consider a loop where each iteration updates a variable.

```
#pragma omp parallel for shared(result)
for ( i ) {
    result += some_function_of(i);
}
```

Discuss qualitatively the difference between:

- turning the update statement into a critical section, versus
- letting the threads accumulate into a private variable `tmp` as above, and summing these after the loop.

Do an Ahmdal-style quantitative analysis of the first case, assuming that you do n iterations on p threads, and each iteration has a critical section that takes a fraction f . Assume the number of iterations n is a multiple of the number of threads p . Also assume the default static distribution of loop iterations over the threads.

Critical sections are an easy way to turn an existing code into a correct parallel code. However, there are performance disadvantages to critical sections, and sometimes a more drastic rewrite is called for.

A critical section works by acquiring a lock, which carries a substantial overhead. Furthermore, if your code has multiple critical sections, they are all mutually exclusive: if a thread is in one critical section, the other ones are all blocked.

The problem with `critical` sections being mutually exclusive can be mitigated by naming them:

```
#pragma omp critical (optional_name_in_parens)
```

On the other hand, the syntax for `atomic` sections is limited to the update of a single memory location, but such sections are not exclusive and they can be more efficient, since they assume that there is a hardware mechanism for making them critical. See the next section.

23.2.3 atomic construct

While the `critical` construct can enclose arbitrary blocks of code, the `atomic` clause has one of a limited number of forms, for which hardware support is likely. Those consist of assigning to a variable:

```
x++;
// or:
x += y;
```

possibly combination with reading that variable:

```
v = x; x++;
```

There are various further refinements on the atomic specification:

1. `omp atomic write` is followed by a single assignment statement to a shared variable.
2. `omp atomic read` is followed by a single assignment statement from a shared variable.
3. `omp atomic` is equivalent to `omp atomic update`; it accomodates statements such as
`x++; x += 1.5;`
4. `omp atomic capture` can accommodate a single statement similar to `omp atomic update`, or a block that essentially combines a `read` and `update` form.

23.3 Locks

OpenMP also has the traditional mechanism of a *lock*. A lock is somewhat similar to a critical section: it guarantees that some instructions can only be performed by one process at a time. However, a critical section is indeed about code; a lock is about data. With a lock you make sure that some data elements can only be touched by one process at a time.

23.3.1 Routines

Create/destroy:

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

Set and release:

```
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
```

Since the set call is blocking, there is also

```
int omp_test_lock();
```

which returns true if the lock was successfully set; otherwise it return false and continues execution with the next statement.

Unsetting a lock needs to be done by the thread that set it.

Here is a simple example:

Create and destroy:

```
// lock.c
omp_lock_t the_lock;
omp_init_lock( &the_lock );
omp_destroy_lock( &the_lock );
```

```
#pragma omp parallel
{
    omp_set_lock( &the_lock );
    sum += omp_get_thread_num();
    omp_unset_lock( &the_lock );
}
```

Use:

Lock operations implicitly have a *flush*; see section 23.4.

Exercise 23.2. In the following code, one process sets array A and then uses it to update B; the other process sets array B and then uses it to update A. Argue that this code can deadlock. How could you fix this?

```
#pragma omp parallel shared(a, b, nthreads, locka, lockb)
#pragma omp sections nowait
{
    #pragma omp section
    {
        omp_set_lock(&locka);
        for (i=0; i<N; i++)
            a[i] = ...;

        omp_set_lock(&lockb);
        for (i=0; i<N; i++)
            b[i] = ... a[i] ...
        omp_unset_lock(&lockb);
        omp_unset_lock(&locka);
    }

    #pragma omp section
{
```

```

    omp_set_lock(&lockb);
    for (i=0; i<N; i++)
        b[i] = ...

    omp_set_lock(&locka);
    for (i=0; i<N; i++)
        a[i] = ... b[i] ...
    omp_unset_lock(&locka);
    omp_unset_lock(&lockb);
}
} /* end of sections */
} /* end of parallel region */

```

23.3.2 Example: Mandelbrot set

Section 49.2.2 has an approach to the *Mandelbrot set* based locking a FIFO that has the coordinates to be processed.

23.3.3 Example: object with atomic update

OO languages such as C++ allow for syntactic simplification, for instance building the locking and unlocking actions into the update operator.

C++ note 24: Lock inside overloaded operator.

```

// lockobject.cxx
class atomic_int {
private:
    omp_lock_t the_lock;
    int _value{0};
public:
    atomic_int() {
        omp_init_lock(&the_lock);
    };
    atomic_int( const atomic_int& )
        = delete;
    atomic_int& operator=( const atomic_int& )
        = delete;
    ~atomic_int() {
        omp_destroy_lock(&the_lock);
    };
}

```

Running this:

```

atomic_int my_object;
vector<std::thread> threads;
for (int ithread=0;
     ithread<NTHREADS;
     ithread++) {
    threads.push_back
    ( std::thread(
        [=,&my_object] () {
            for (int iop=0; iop<nops; iop++)

```

```
        my_object += 1; } ) );
}
for ( auto &t : threads )
    t.join();
```

23.3.4 Example: histogram / binning

See section 29.1.

23.3.5 Nested locks

A lock as explained above can not be locked if it is already locked. A *nested lock* can be locked multiple times by the same thread before being unlocked.

- `omp_init_nest_lock`
- `omp_destroy_nest_lock`
- `omp_set_nest_lock`
- `omp_unset_nest_lock`
- `omp_test_nest_lock`

23.4 Relaxed memory model

flush

- There is an implicit flush of all variables at the start and end of a *parallel region*.
- There is a flush at each barrier, whether explicit or implicit, such as at the end of a *work sharing*.
- At entry and exit of a *critical section*
- When a *lock* is set or unset.

23.5 Example: Fibonacci computation

The *Fibonacci sequence* is recursively defined as

$$F(0) = 1, \quad F(1) = 1, \quad F(n) = F(n - 1) + F(n - 2) \text{ for } n \geq 2.$$

We start by sketching the basic single-threaded solution. The naive code looks like:

```
int main() {
    value = new int[nmax+1];
    value[0] = 1;
    value[1] = 1;
    fib(10);
}

int fib(int n) {
    int i, j, result;
    if (n>=2) {
        i=fib(n-1); j=fib(n-2);
        value[n] = i+j;
    }
    return value[n];
}
```

23. OpenMP topic: Synchronization

However, this is inefficient, since most intermediate values will be computed more than once. We solve this by keeping track of which results are known:

```
...
done = new int[nmax+1];
for (i=0; i<=nmax; i++)
    done[i] = 0;
done[0] = 1;
done[1] = 1;
...
int fib(int n) {
    int i, j;
    if (!done[n]) {
        i = fib(n-1); j = fib(n-2);
        value[n] = i+j; done[n] = 1;
    }
    return value[n];
}
```

The OpenMP parallel solution calls for two different ideas. First of all, we parallelize the recursion by using tasks (section 24):

```
int fib(int n) {
    int i, j;
    if (n>=2) {
#pragma omp task shared(i) firstprivate(n)
        i=fib(n-1);
#pragma omp task shared(j) firstprivate(n)
        j=fib(n-2);
#pragma omp taskwait
        value[n] = i+j;
    }
    return value[n];
}
```

This computes the right solution, but, as in the naive single-threaded solution, it recomputes many of the intermediate values.

A naive addition of the done array leads to data races, and probably an incorrect solution:

```
int fib(int n) {
    int i, j, result;
    if (!done[n]) {
#pragma omp task shared(i) firstprivate(n)
        i=fib(n-1);
#pragma omp task shared(i) firstprivate(n)
        j=fib(n-2);
#pragma omp taskwait
        value[n] = i+j;
        done[n] = 1;
    }
    return value[n];
}
```

For instance, there is no guarantee that the `done` array is updated later than the `value` array, so a thread can think that `done[n-1]` is true, but `value[n-1]` does not have the right value yet.

One solution to this problem is to use a lock, and make sure that, for a given index `n`, the values `done[n]` and `value[n]` are never touched by more than one thread at a time:

```
int fib(int n)
{
    int i, j;
    omp_set_lock( &(dolock[n]) );
    if (!done[n]) {
        #pragma omp task shared(i) firstprivate(n)
        i = fib(n-1);
        #pragma omp task shared(j) firstprivate(n)
        j = fib(n-2);
        #pragma omp taskwait
        value[n] = i+j;
        done[n] = 1;
    }
    omp_unset_lock( &(dolock[n]) );
    return value[n];
}
```

This solution is correct, optimally efficient in the sense that it does not recompute anything, and it uses tasks to obtain a parallel execution.

However, the efficiency of this solution is only up to a constant. A lock is still being set, even if a value is already computed and therefore will only be read. This can be solved with a complicated use of critical sections, but we will forego this.

Chapter 24

OpenMP topic: Tasks

Tasks are a mechanism that OpenMP uses behind the scenes: if you specify something as being a task, OpenMP will create a ‘block of work’: a section of code plus the data environment in which it occurred. This block is set aside for execution at some later point. Thus, task-based code usually looks something like this:

```
#pragma omp parallel
{
    // generate a bunch of tasks
    #pragma omp taskwait
    // the result from the tasks is now available
}
```

For instance, a parallel loop was always implicitly translated to something like:

Sequential loop:

```
for (int i=0; i<N; i++)
    f(i);
```

Parallel loop:

```
for (int ib=0; ib<nblocks; ib++) {
    int first=... last=... ;
    #pragma omp task
    for (int i=first; i<last; i++)
        f(i)
}
#pragma omp taskwait
// the results from the loop are available
```

24.1 Task generation

If we stick with this example of implementing a parallel loop through tasks, the next question is: precisely who generates the tasks? The following code has a serious problem:

```
// WRONG. DO NOT WRITE THIS
#pragma omp parallel
for (int ib=0; ib<nblocks; ib++) {
    int first=... last=... ;
    #pragma omp task
    for (int i=first; i<last; i++)
        f(i)
}
```

because the parallel region creates a team, and each thread in the team executes the task-generating code. Instead, we use the following idiom:

```
#pragma omp parallel
#pragma omp single
for (int ib=0; ib<nblocks; ib++) {
    // setup stuff
    #pragma omp task
    // task stuff
}
```

1. A parallel region creates a team of threads;
2. a single thread then creates the tasks, adding them to a queue that belongs to the team,
3. and all the threads in that team (possibly including the one that generated the tasks)

Btw, the actual task queue is not visible to the programmer. Another aspect that is out of the programmer's control is the exact timing of the execution of the task: this is up to a *task scheduler*, which operates invisible to the programmer.

The task mechanism allows you to do things that are hard or impossible with the loop and section constructs. For instance, a *while loop* traversing a *linked list* can be implemented with tasks:

Code	Execution
p = head_of_list();	one thread traverses the list
while (!end_of_list(p)) {	
#pragma omp task	a task is created,
process(p);	one for each element
p = next_element(p);	the generating thread goes on without waiting
}	the tasks are executed while more are being generated.

Another concept that was hard to parallelize earlier is the ‘while loop’. This does not fit the requirement for OpenMP parallel loops that the loop bound needs to be known before the loop executes.

Exercise 24.1. Use tasks to find the smallest factor of a large number (using $2999 \cdot 3001$ as test case):

generate a task for each trial factor.

- Turn the factor finding block into a task.
- Run your program a number of times:
`for i in `seq 1 1000` ; do ./taskfactor ; done | grep -v 2999`
 Does it find the wrong factor? Why? Try to fix this.
- Once a factor has been found, you should stop generating tasks. Let tasks that should not have been generated, meaning that they test a candidate larger than the factor found, print out a message.

24.2 Task data

Treatment of data in a task is somewhat subtle. The basic problem is that a task gets created at one time, and executed at some later time. Thus, if shared data is accessed, does the task see the value at creation time or at execution time? In fact, both possibilities make sense depending on the application, so we need to discuss the rules which possibility applies when.

The first rule is that shared data is shared in the task, but private data becomes *firstprivate*. To see the distinction, consider two code fragments.

```

int count = 100;
#pragma omp parallel
#pragma omp single
{
    while (count>0) {
        #pragma omp task
        {
            int countcopy = count;
            if (count==50) {
                sleep(1);
                printf("%d,%d\n",
                       count,countcopy);
            } // end if
        } // end task
        count--;
    } // end while
} // end single

```



```

#pragma omp parallel
#pragma omp single
{
    int count = 100;
    while (count>0) {
        #pragma omp task
        {
            int countcopy = count;
            if (count==50) {
                sleep(1);
                printf("%d,%d\n",
                       count,countcopy);
            } // end if
        } // end task
        count--;
    } // end while
} // end single

```

In the first example, the variable `count` is declared outside the parallel region and is therefore shared. When the `printf` statement is executed, all tasks will have been generated, and so `count` will be zero. Thus, the output will likely be `0,50`.

In the second example, the `count` variable is private to the thread creating the tasks, and so it will be `firstprivate` in the task, preserving the value that was current when the task was created.

24.3 Task synchronization

Even though the above segment looks like a linear set of statements, it is impossible to say when the code after the `task` directive will be executed. This means that the following code is incorrect:

```

x = f();
#pragma omp task
{ y = g(x); }
z = h(y);

```

Explanation: when the statement computing `z` is executed, the task computing `y` has only been scheduled; it has not necessarily been executed yet.

24.3.1 Deferred vs undeferred tasks

Tasks are unusually ‘deferred’: meaning that they are executed at some undetermined point in the future. Tasks can also be undeferred, meaning that they are executed synchronously, and the creating thread does not progress beyond them until they are fully executed.

Prime example of undeferred tasks: with `if` clause

```

#pragma omp task if (level>5)
{
    ...
}

```

Note that, even though the body of the task is executed as if it were inlined, a task is still created, with all its data space implications.

24.3.2 Undefined task waiting

In order to have a guarantee that a deferred task is finished, you can first of all use the `taskwait` directive. The following creates two tasks, which can be executed in parallel, and then waits for the results:

Code	Execution
<code>x = f();</code>	the variable <code>x</code> gets a value
<code>#pragma omp task { y1 = g1(x); }</code>	two tasks are created with the current value of <code>x</code>
<code>#pragma omp task { y2 = g2(x); }</code>	
<code>#pragma omp taskwait</code>	the thread waits until the tasks are finished
<code>z = h(y1)+h(y2);</code>	the variable <code>z</code> is computed using the task results

The `task` pragma is followed by a structured block. Each time the structured block is encountered, a new task is generated. On the other hand `taskwait` is a standalone directive; the code that follows is just code, it is not a structured block belonging to the directive.

You can indicate task dependencies in several ways:

1. Using the ‘task wait’ directive you can explicitly indicate the *join* of the *forked* tasks. The instruction after the wait directive will therefore be dependent on the spawned tasks.
2. The `taskgroup` directive is discussed in section 24.3.3.
3. The `taskloop` directive is discussed in section 24.3.4.
4. Each OpenMP task can have a `depend` clause, indicating what *data dependency* of the task; section 24.4. By indicating what data is produced or absorbed by the tasks, the scheduler can construct the dependency graph for you.

24.3.3 Task groups

The `taskgroup` directive, followed by a structured block, ensures completion of all tasks created in the block, even if recursively created.

A task group is somewhat similar to having a `taskwait` directive after the block. The big difference is that that `taskwait` directive does not wait for tasks that are recursively generated, while a `taskgroup` does.

24.3.4 Task loop

The `taskloop` directive prefaces a for/do loop, just like an `for` pragma. The difference is that now every iteration is turned into a task, rather than groups of iterations as in the `for` case. The end of the loop is a synchronization point: statements after the loop are only executed when all tasks from the loop are finished.

There is a `master` `taskloop` directive that is shorthand for a `master` containing only a `taskloop`.

24.4 Task dependencies

It is possible to put a partial ordering on tasks through use of the `depend` clause. For example, in

```
#pragma omp task
x = f()
#pragma omp task
y = g(x)
```

it is conceivable that the second task is executed before the first, possibly leading to an incorrect result. This is remedied by specifying:

```
#pragma omp task depend(out:x)
x = f()
#pragma omp task depend(in:x)
y = g(x)
```

- These dependencies only hold between sibling tasks.
- The depending data items of the various tasks are either identical or disjoint. In particular, dependencies on different sections of an array are not allowed, though a compiler may not always catch this.

Exercise 24.2. Consider the following code:

```
for i in [1:N]:
    x[0,i] = some_function_of(i)
    x[i,0] = some_function_of(i)

for i in [1:N]:
    for j in [1:N]:
        x[i,j] = x[i-1,j]+x[i,j-1]
```

- Observe that the second loop nest is not amenable to OpenMP loop parallelism.
- Can you think of a way to realize the computation with OpenMP loop parallelism?
Hint: you need to rewrite the code so that the same operations are done in a different order.
- Use tasks with dependencies to make this code parallel without any rewriting: the only change is to add OpenMP directives.

Tasks dependencies are used to indicated how two uses of one data item relate to each other. Since either use can be a read or a write, there are four types of dependencies.

RaW (Read after Write) The second task reads an item that the first task writes. The second task has to be executed after the first:

```
... omp task depend(OUT:x)
foo(x)
... omp task depend( IN:x)
foo(x)
```

WaR (Write after Read) The first task reads an item, and the second task overwrites it. The second task has to be executed second to prevent overwriting the initial value:

```
... omp task depend( IN:x)
foo(x)
... omp task depend(OUT:x)
foo(x)
```

WaW (Write after Write) Both tasks set the same variable. Since the variable can be used by an intermediate task, the two writes have to be executed in this order.

```
... omp task depend(OUT:x)
    foo(x)
... omp task depend(OUT:x)
    foo(x)
```

RaR (Read after Read) Both tasks read a variable. Since neither tasks has an ‘out’ declaration, they can run in either order.

```
... omp task depend(IN:x)
    foo(x)
... omp task depend(IN:x)
    foo(x)
```

In addition to tasks, the `taskwait` can also have a `depend` clause. This can be used to wait for only certain tasks. In this example:

```
#pragma omp task
{ f1(); }
#pragma omp task shared(x) depend(out:x)
{ x = f2(); }
#pragma omp taskwait depend(in:x)
```

the `taskwait` waits only for the second task.

24.5 Task reduction

The `reduction` clause only pertains to ordinary parallel loops, not to `taskgroup` loops of tasks. To do a reduction over computations in tasks you need the `task_reduction` clause (a OpenMP-5.0 feature):

```
#pragma omp taskgroup task_reduction(+:sum)
```

The task group can contain both task that contribute to the reduction, and ones that don’t. The former type needs a clause `in_reduction`:

```
#pragma omp task in_reduction(+:sum)
```

As an example, here the sum $\sum_{i=1}^{100} i$ is computed with tasks:

```
// taskreduce.c
#pragma omp parallel
#pragma omp single
{
#pragma omp taskgroup task_reduction(+:sum)
for (int itask=1; itask<=bound; itask++) {
#pragma omp task in_reduction(+:sum)
    sum += itask;
}
}
```

24.6 More

24.6.1 Scheduling points

Normally, a task stays tied to the thread that first executes it. However, at a *task scheduling point* the thread may switch to the execution of another task created by the same team.

- There is a scheduling point after explicit task creation. This means that, in the above examples, the thread creating the tasks can also participate in executing them.
- There is a scheduling point at `taskwait` and `taskyield`.

On the other hand a task created with them *untied* clause on the task pragma is never tied to one thread. This means that after suspension at a scheduling point any thread can resume execution of the task. If you do this, beware that the value of a thread-id does not stay fixed. Also locks become a problem.

Example: if a thread is waiting for a lock, with a scheduling point it can suspend the task and work on another task.

```
while (!omp_test_lock(lock))
#pragma omp taskyield
;
```

24.6.2 Hints for performance improvement

If a task involves only a small amount of work, the scheduling overhead may negate any performance gain. There are two ways of executing the task code directly:

- The `if` clause will only create a task if the test is true:

```
#pragma omp task if (n>100)
f(n)
```

Note that this will still create a task, with all the implications for dataspace it implies.

- The `if` clause may still lead to recursively generated tasks. On the other hand, `final` will execute the code, and will also skip any recursively created tasks:

```
#pragma omp task final(level<3)
```

If you want to indicate that certain tasks are more important than others, use the `priority` clause:

```
#pragma omp task priority(5)
```

where the priority is any non-negative scalar less than the value of environment variable `OMP_MAX_TASK_PRIORITY`. The default value is zero, meaning that priority clauses are effectively ignored.

24.6.3 Task canceling

It is possible (in OpenMP-4.0) to cancel tasks. This is useful when tasks are used to perform a search: the task that finds the result first can cancel any outstanding search tasks. See section 18.3 for details.

Exercise 24.3. Modify the prime finding example to use `cancel`.

24.7 Examples

24.7.1 Recursive matrix-matrix multiplication

Large matrices can be multiplied recursively using the formula

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

with

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$

et cetera. You can implement this by creating four tasks, each of which can create another four.

C++ note 25: Use `mspan` for submatrices. For the data structure, use `mspan`.

24.7.2 Fibonacci

As an example of the use of tasks, consider computing an array of Fibonacci values:

```
// taskgroup0.c
for (int i=2; i<N; i++)
{
    fibo_values[i] = fibo_values[i-1]+fibo_values[i-2];
}
```

If you simply turn each calculation into a task, results will be unpredictable (confirm this!) since tasks can be executed in any sequence. To solve this, we put dependencies on the tasks:

```
// taskgroup2.c
for (int i=2; i<N; i++)
#pragma omp task \
depend(out:fibo_values[i]) \
depend(in:fibo_values[i-1],fibo_values[i-2])
{
    fibo_values[i] = fibo_values[i-1]+fibo_values[i-2];
}
```

24.7.3 Binomial coefficients

Exercise 24.4. An array of binomial coefficients can be computed as follows:

```
// binomial1.c
for (int row=1; row<=n; row++)
    for (int col=1; col<=row; col++)
        if (row==1 || col==1 || col==row)
            array[row][col] = 1;
        else
            array[row][col] = array[row-1][col-1] + array[row-1][col];
```

Putting a single task group around the double loop, and use `depend` clauses to make the execution satisfy the proper dependencies.

Chapter 25

OpenMP topic: Affinity

25.1 OpenMP thread affinity control

The matter of thread affinity becomes important on *multi-socket nodes*; see the example in section 25.2.

Thread placement can be controlled with two environment variables:

- the environment variable `OMP_PROC_BIND` describes how threads are bound to *OpenMP places*; while
- the variable `OMP_PLACES` describes these places in terms of the available hardware.
- When you're experimenting with these variables it is a good idea to set `OMP_DISPLAY_ENV` to true, so that OpenMP will print out at runtime how it has interpreted your specification. The examples in the following sections will display this output.

25.1.1 Thread binding

The variable `OMP_PLACES` defines a series of places to which the threads are assigned, and `OMP_PROC_BIND` describes how threads are tied to those places.

Typical values for `OMP_PLACES` are

- `socket`: threads are bound to a *socket*, but can be moved between cores in the socket;
- `core`: threads are bound to a *core*, but can be moved between hyperthreads in the core;
- `thread`: threads are bound to a specific *hyper-thread*.

Values for `OMP_PROC_BIND` are implementation-defined, but typically:

- `master`: threads are bound to the same place as the master thread;
- `close`: subsequent thread numbers are placed close together in the defined places;
- `spread`: subsequent thread numbers are maximally spread over places;
- `true`: threads are bound to their initial placement;
- `false`: threads are not bound to their initial placement;

where the values `master`, `close`, `spread` are ordained by the standard, and the others depend on the implementation.

There is no runtime function for setting the binding, but the Internal Control Variable (ICV) `bind-var` can be retrieved with `omp_get_proc_bind`. The binding can also be set with the `proc_bind` clause on the `parallel` directive, with values `master`, `close`, `spread`.

Example: if you have two sockets and you define

`OMP_PLACES=sockets`

then

- thread 0 goes to socket 0,
- thread 1 goes to socket 1,
- thread 2 goes to socket 0 again,
- and so on.

On the other hand, if the two sockets have a total of sixteen cores and you define

```
OMP_PLACES=cores  
OMP_PROC_BIND=close
```

then

- thread 0 goes to core 0, which is on socket 0,
- thread 1 goes to core 1, which is on socket 0,
- thread 2 goes to core 2, which is on socket 0,
- and so on, until thread 7 goes to core 7 on socket 0, and
- thread 8 goes to core 8, which is on socket 1,
- et cetera.

The value `OMP_PROC_BIND=close` means that the assignment goes successively through the available places. The variable `OMP_PROC_BIND` can also be set to `spread`, which spreads the threads over the places. With

```
OMP_PLACES=cores  
OMP_PROC_BIND=spread
```

you find that

- thread 0 goes to core 0, which is on socket 0,
- thread 1 goes to core 8, which is on socket 1,
- thread 2 goes to core 1, which is on socket 0,
- thread 3 goes to core 9, which is on socket 1,
- and so on, until thread 14 goes to core 7 on socket 0, and
- thread 15 goes to core 15, which is on socket 1.

So you see that `OMP_PLACES=cores` and `OMP_PROC_BIND=spread` very similar to `OMP_PLACES=sockets`. The difference is that the latter choice does not bind a thread to a specific core, so the operating system can move threads about, and it can put more than one thread on the same core, even if there is another core still unused.

The value `OMP_PROC_BIND=master` puts the threads in the same place as the master of the team. This is convenient if you create teams recursively. In that case you would use the `dproc` clause rather than the environment variable, set to `spread` for the initial team, and to `master` for the recursively created team.

25.1.2 Effects of thread binding

Let's consider two example program. First we consider the program for computing π , which is purely compute-bound.

#threads	close/cores	spread/sockets	spread/cores
1	0.359	0.354	0.353
2	0.177	0.177	0.177
4	0.088	0.088	0.088
6	0.059	0.059	0.059
8	0.044	0.044	0.044
12	0.029	0.045	0.029
16	0.022	0.050	0.022

25. OpenMP topic: Affinity

We see pretty much perfect speedup for the `OMP_PLACES=cores` strategy; with `OMP_PLACES=sockets` we probably get occasional collisions where two threads wind up on the same core.

Next we take a program for computing the time evolution of the *heat equation*:

$$t = 0, 1, 2, \dots : \forall_i : x_i^{(t+1)} = 2x_i^{(t)} - x_{i-1}^{(t)} - x_{i+1}^{(t)}$$

This is a bandwidth-bound operation because the amount of computation per data item is low.

#threads	close/cores	spread/sockets	spread/cores
1	2.88	2.89	2.88
2	1.71	1.41	1.42
4	1.11	0.74	0.74
6	1.09	0.57	0.57
8	1.12	0.57	0.53
12	0.72	0.53	0.52
16	0.52	0.61	0.53

Again we see that `OMP_PLACES=sockets` gives worse performance for high core counts, probably because of threads winding up on the same core. The thing to observe in this example is that with 6 or 8 cores the `OMP_PROC_BIND=spread` strategy gives twice the performance of `OMP_PROC_BIND=close`.

The reason for this is that a single socket does not have enough bandwidth for all eight cores on the socket. Therefore, dividing the eight threads over two sockets gives each thread a higher available bandwidth than putting all threads on one socket.

25.1.3 Place definition

There are three predefined values for the `OMP_PLACES` variable: `sockets`, `cores`, `threads`. You have already seen the first two; the `threads` value becomes relevant on processors that have hardware threads. In that case, `OMP_PLACES=cores` does not tie a thread to a specific hardware thread, leading again to possible collisions as in the above example. Setting `OMP_PLACES=threads` ties each OpenMP thread to a specific hardware thread.

There is also a very general syntax for defining places that uses a

`location:number:stride`

syntax. Examples:

- `OMP_PLACES="{}0:8:1},{}8:8:1"`
is equivalent to `sockets` on a two-socket design with eight cores per socket: it defines two places, each having eight consecutive cores. The threads are then places alternating between the two places, but not further specified inside the place.
- The setting `cores` is equivalent to
`OMP_PLACES="{}0},{}1},{}2},\dots,{}15"`
- On a four-socket design, the specification
`OMP_PLACES="{}0:4:8}:4:1"`
states that the place `0,8,16,24` needs to be repeated four times, with a stride of one. In other words, thread 0 winds up on core 0 of some socket, the thread 1 winds up on core 1 of some socket, et cetera.

25.1.4 Binding possibilities

Values for `OMP_PROC_BIND` are: `false`, `true`, `master`, `close`, `spread`.

- false: set no binding
- true: lock threads to a core
- master: collocate threads with the master thread
- close: place threads close to the master in the places list
- spread: spread out threads as much as possible

This effect can be made local by giving the `dproc` clause in the `parallel` directive.

A safe default setting is

```
export OMP_PROC_BIND=true
```

which prevents the operating system from *migrating a thread*. This prevents many scaling problems.

Good examples of *thread placement* on the *Intel Knight's Landing*: <https://software.intel.com/en-us/articles/process-and-thread-affinity-for-intel-xeon-phi-processors-x200>

As an example, consider a code where two threads write to a shared location.

```
// sharing.c
#pragma omp parallel
{ // not a parallel for: just a bunch of reps
    for (int j = 0; j < reps; j++) {
#pragma omp for schedule(static,1)
        for (int i = 0; i < N; i++){
#pragma omp atomic
        a++;
    }
}
}
```

There is now a big difference in runtime depending on how close the threads are. We test this on a processor with both cores and hyperthreads. First we bind the OpenMP threads to the cores:

```
OMP_NUM_THREADS=2 OMP_PLACES=cores OMP_PROC_BIND=close ./sharing
run time = 4752.231836usec
sum = 80000000.0
```

Next we force the OpenMP threads to bind to hyperthreads inside one core:

```
OMP_PLACES=threads OMP_PROC_BIND=close ./sharing
run time = 941.970110usec
sum = 80000000.0
```

Of course in this example the inner loop is pretty much meaningless and parallelism does not speed up anything:

```
OMP_NUM_THREADS=1 OMP_PLACES=cores OMP_PROC_BIND=close ./sharing
run time = 806.669950usec
sum = 80000000.0
```

However, we see that the two-thread result is almost as fast, meaning that there is very little parallelization overhead.

25.2 First-touch

The affinity issue shows up in the *first-touch* phenomenon.

A little background knowledge. Memory is organized in *memory pages* (see HPC book, section ??), and what we think of as ‘addresses’ really are *virtual addresses*, mapped to *physical addresses* through a *page table*.

This means that data in your program can be anywhere in physical memory. In particular, on a *dual socket* node, the memory can be mapped to either of the sockets.

The next thing to know is that memory allocated with `malloc / new` and like routines is not immediately mapped; that only happens when data is written to it. In light of this, consider the following OpenMP code:

```
double *x = (double*) malloc(N*sizeof(double));

for (i=0; i<N; i++)
    x[i] = 0;

#pragma omp parallel for
for (i=0; i<N; i++)
    .... something with x[i] ...
```

Since the initialization loop is not parallel it is executed by the main thread, making all the memory associated with the socket of that thread. Subsequent access by the other socket will then access data from memory not attached to that socket, which induces a considerable delay, and performance degradation.

25.2.1 Example

Let’s consider an example. We make the initialization parallel subject to an option:

```
// heat.c
#pragma omp parallel if (init>0)
{
    #pragma omp for
    for (int i=0; i<N; i++)
        y[i] = x[i] = 0.;
    x[0] = 0; x[N-1] = 1.;
}
```

If the initialization is not parallel, the array will be mapped to the socket of the master thread; if it is parallel, it may be mapped to different sockets, depending on where the threads run.

As a simple application we run a heat equation, which is parallel, though not embarrassingly so:

```
for (int it=0; it<1000; it++) {
    #pragma omp parallel for
    for (int i=1; i<N-1; i++)
        y[i] = ( x[i-1]+x[i]+x[i+1] )/3.;
    #pragma omp parallel for
    for (int i=1; i<N-1; i++)
        x[i] = y[i];
}
```

On the *TACC Frontera* machine, with dual 28-core *Intel Cascade Lake* processors, we use the following settings:

```
export OMP_PLACES=cores
export OMP_PROC_BIND=close
```

```
# no parallel initialization
make heat && OMP_NUM_THREADS=56 ./heat
# yes parallel initialization
make heat && OMP_NUM_THREADS=56 ./heat 1
```

This gives us a remarkable difference in runtime:

- Sequential init: avg=2.089, stddev=0.1083
- Parallel init: avg=1.006, stddev=0.0216

This large difference will be mitigated for algorithms with higher arithmetic intensity.

Exercise 25.1. How do the OpenMP dynamic schedules relate to this issue?

25.2.2 Solution in C++

The problem with realizing first-touch in *C++* is that *std::vector* fills its allocation with default values. This is known as ‘value-initialization’, and it makes

```
vector<double> x(N);
```

equivalent to the non-parallel allocation and initialization above.

Here is a solution.

C++ note 26: Uninitialized containers. Default initialization is a problem. We make a template for uninitialized types:

```
// heatalloc.cxx
template<typename T>
struct uninitialized {
    uninitialized() {};
    T val;
    constexpr operator T() const {return val;};
    T operator=( const T& v ) { val = v; return val; };
};
```

so that we can create vectors that behave normally:

```
vector<uninitialized<double>> x(N),y(N);
```

```
#pragma omp parallel for
for (int i=0; i<N; i++)
    y[i] = x[i] = 0.;
x[0] = 0; x[N-1] = 1.;
```

Running the code with the regular definition of a vector, and the above modification, reproduces the runtimes of the C variant above.

Another option is to wrap memory allocated with *new* in a *unique_ptr*:

```
// heatptr.cxx
unique_ptr<double[]> x( new double[N] );
unique_ptr<double[]> y( new double[N] );

#pragma omp parallel for
for (int i=0; i<N; i++) {
```

```

y[i] = x[i] = 0. ;
}
x[0] = 0; x[N-1] = 1. ;

```

Note that this gives fairly elegant code, since square bracket indexing is overloaded for `unique_ptr`. The only disadvantage is that we can not query the `size` of these arrays, or do bound checking with `at`, but in high performance contexts that is usually not appropriate anyway.

25.2.3 Remarks

You could move pages with `move_pages`.

By regarding affinity, in effect you are adopting an SPMD style of programming. You could make this explicit by having each thread allocate its part of the arrays separately, and storing a private pointer as `threadprivate` [20]. However, this makes it impossible for threads to access each other's parts of the distributed array, so this is only suitable for total *data parallel* or *embarrassingly parallel* applications.

25.3 Affinity control outside OpenMP

There are various utilities to control process and thread placement.

Process placement can be controlled on the Operating system level by `numactl` (the TACC utility `tacc_affinity` is a wrapper around this) on Linux (also `taskset`); Windows `start/affinity`.

Corresponding system calls: `pbng` on Solaris, `sched_setaffinity` on Linux, `SetThreadAffinityMask` on Windows.

Corresponding environment variables: `SUNW_MP_PROCBIND` on Solaris, `KMP_AFFINITY` on Intel.

The *Intel compiler* has an environment variable for affinity control:

```
export KMP_AFFINITY=verbose,scatter
values: none,scatter,compact
```

For *gcc*:

```
export GOMP_CPU_AFFINITY=0,8,1,9
```

For the *Sun compiler*:

```
SUNW_MP_PROCBIND
```

25.4 Tests

We take a simple loop and consider the influence of binding parameters.

```
// speedup.c
#pragma omp parallel for
    for (int ip=0; ip<N; ip++) {
        for (int jp=0; jp<M; jp++) {
            double f = sin( values[ip] );
            values[ip] = f;
        }
    }
```

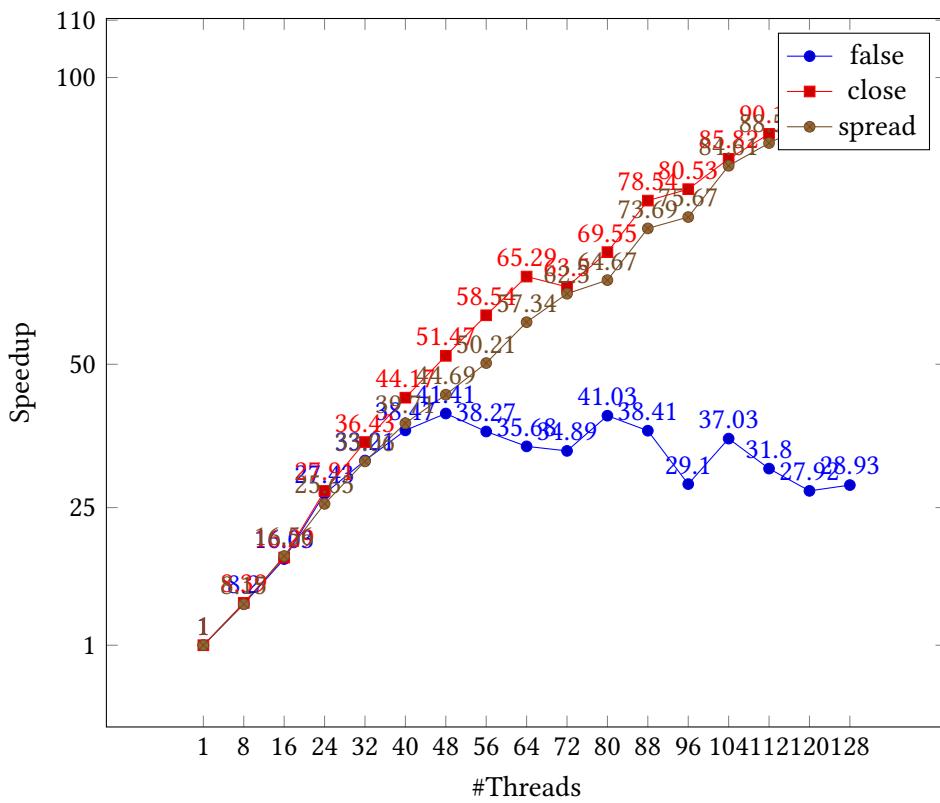


Figure 25.1: Speedup as function of thread count, Lonestar 6 cluster, different binding parameters

25.4.1 Lonestar 6

Lonestar 6, dual socket *AMD Milan*, total 112 cores: figure 25.1.

25.4.2 Frontera

Intel Cascade Lake, dual socket, 56 cores total; figure 25.2.

For all core counts to half the total, performance for all binding strategies seems equal. After that, `close` and `spread` perform equally, but the speedup for the `false` value gives erratic numbers.

25.4.3 Stampede2 skylake

Dual 24-core *Intel Skylake*; figure 25.3.

We see that `close` binding gives worse performance than `spread`. Setting binding to `false` only gives bad performance for large core counts.

25.4.4 Stampede2 Knights Landing

We test on a single socket 68-core processor: the *Intel Knights Landing*.

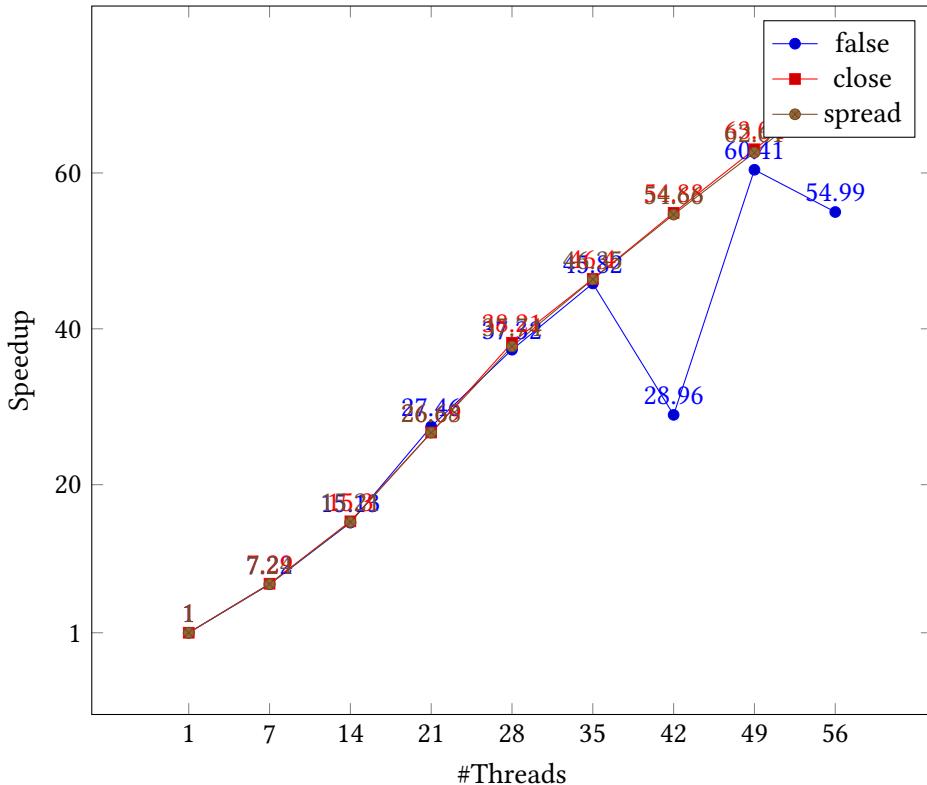


Figure 25.2: Speedup as function of thread count, Frontera cluster, different binding parameters

Since this is a single socket design, we don't distinguish between the `close` and `spread` binding. However, the binding value of `true` shows good speedup – in fact beyond the core count – while `false` gives worse performance than in other architectures.

25.4.5 Longhorn

Dual 20-core *IBM Power9*, 4 hyperthreads; 25.5

Unlike the Intel processors, here we use the hyperthreads. Figure 25.5 shows dip in the speedup at 40 threads. For higher thread counts the speedup increases to well beyond the physical core count of 40.

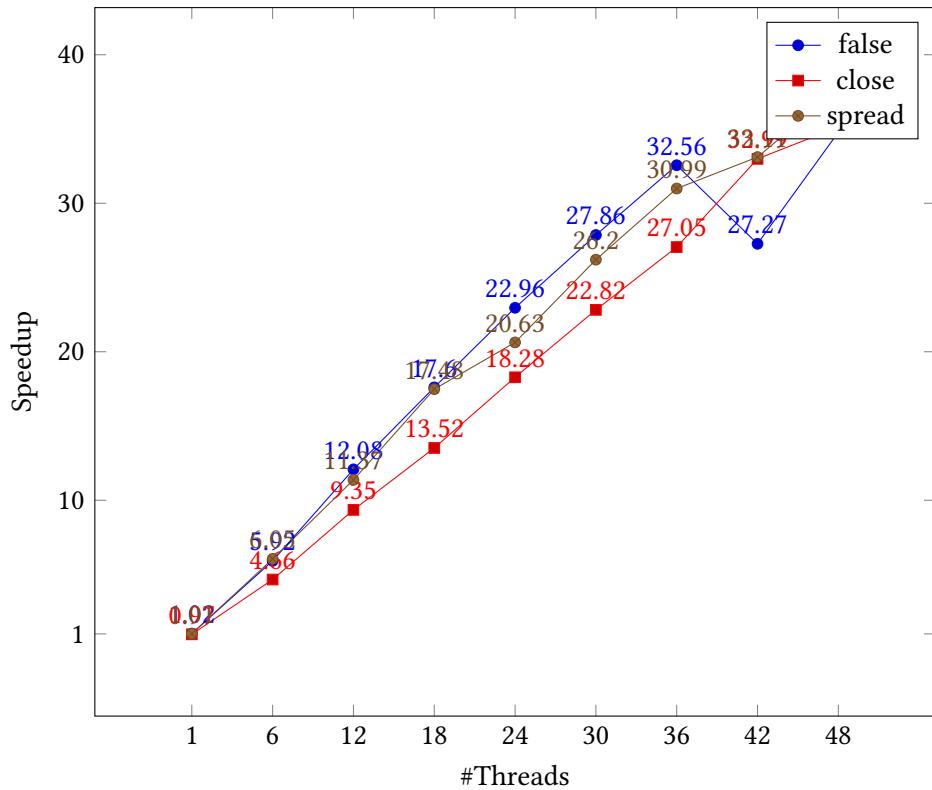


Figure 25.3: Speedup as function of thread count, Stampede2 skylake cluster, different binding parameters

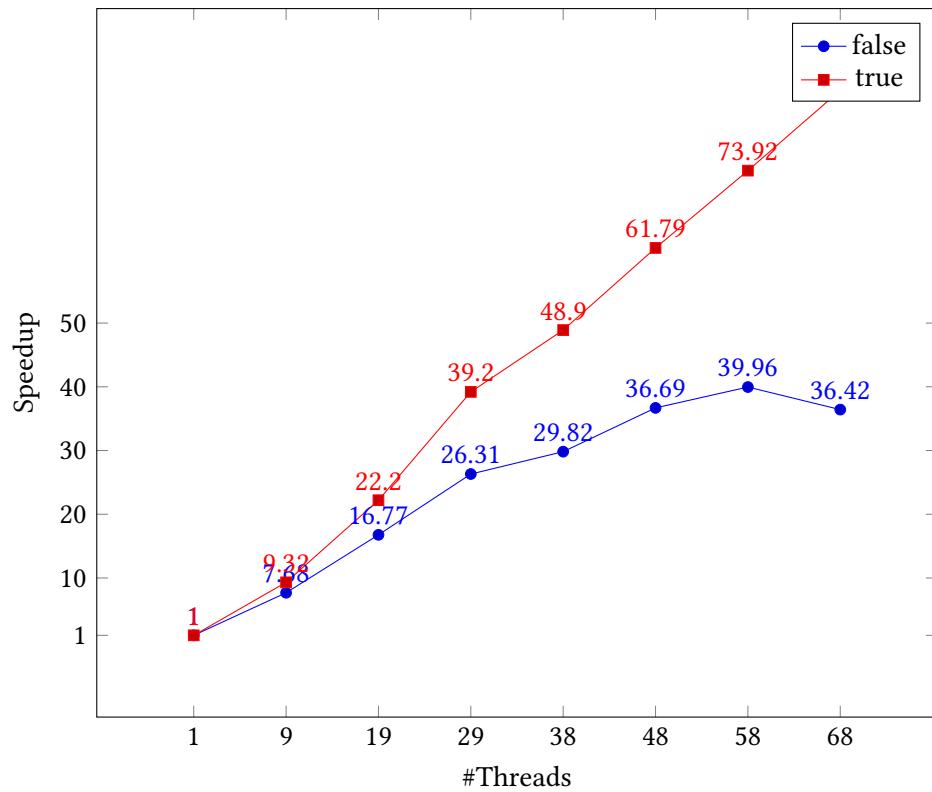


Figure 25.4: Speedup as function of thread count, Stampede2 Knights Landing cluster, different binding parameters

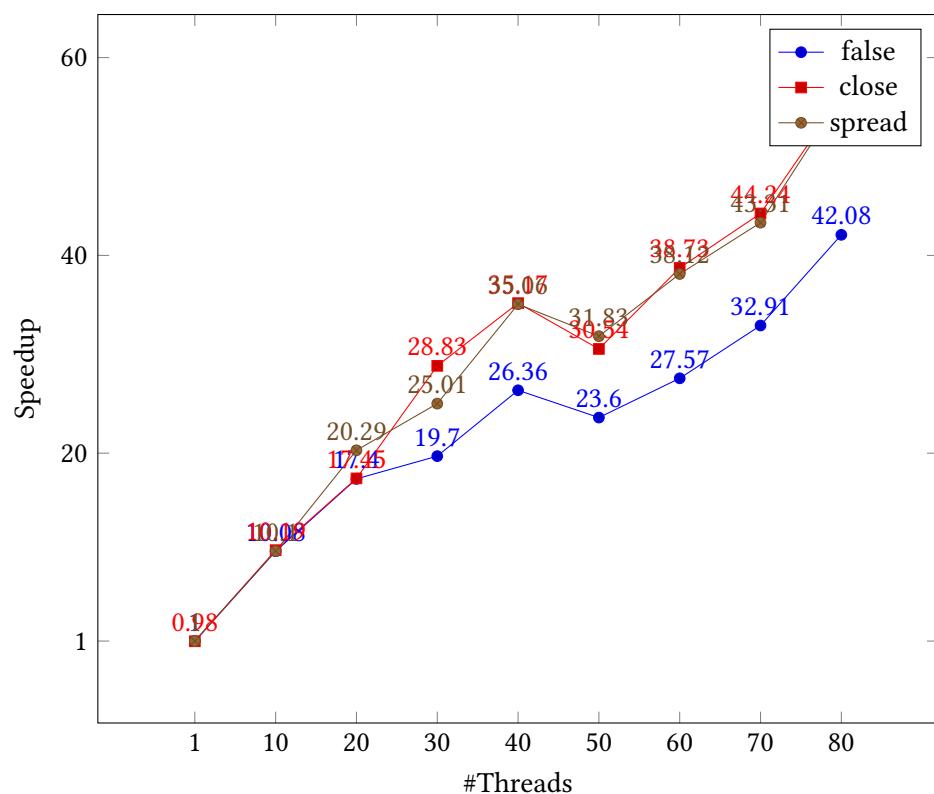


Figure 25.5: Speedup as function of thread count, Longhorn cluster, different binding parameters

Chapter 26

OpenMP topic: SIMD processing

You can declare a loop to be executable with *vector instructions* with *simd*.

Remark Depending on your compiler, it may be necessary to give an extra option enabling SIMD:

- `-fopenmp-simd` for GCC / Clang, and
- `-qopenmp-simd` for ICC.

The *simd* pragma has the following clauses:

- `safelen(n)`: limits the number of iterations in a SIMD chunk. Presumably useful if you combine `parallel for simd`.
- `linear`: lists variables that have a linear relation to the iteration parameter.
- `aligned`: specifies alignment of variables.

26.1 SIMD loops

Adding a `simd` directive to a loop schedule:

```
#pragma omp parallel for schedule(simd:static)
```

causes the chunk size to be increased to a multiple of the vector width.

26.2 SIMD function calls

If your SIMD loop includes a function call, you can declare that the function can be turned into vector instructions with `declare simd`

If a loop is both multi-threadable and vectorizable, you can combine directives as `pragma omp parallel for simd`.

Compilers can be made to report whether a loop was vectorized:

```
LOOP BEGIN at simdf.c(61,15)
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
LOOP END
```

with such options as `-Qvec-report=3` for the Intel compiler.

Performance improvements of these directives need not be immediately obvious. In cases where the operation is bandwidth-limited, using `simd` parallelism may give the same or worse performance as thread parallelism.

The following function can be vectorized:

```
// simdfunctions.c
#pragma omp declare simd
double cs(double x1,double x2,double y1,double y2) {
    double
        inprod = x1*x2+y1*y2,
        xnorm = sqrt(x1*x1 + x2*x2),
        ynorm = sqrt(y1*y1 + y2*y2);
    return inprod / (xnorm*ynorm);
}
#pragma omp declare simd uniform(x1,x2,y1,y2) linear(i)
double csa(double *x1,double *x2,double *y1,double *y2, int i) {
    double
        inprod = x1[i]*x2[i]+y1[i]*y2[i],
        xnorm = sqrt(x1[i]*x1[i] + x2[i]*x2[i]),
        ynorm = sqrt(y1[i]*y1[i] + y2[i]*y2[i]);
    return inprod / (xnorm*ynorm);
}
```

Compiling this the regular way

```
# parameter 1(x1): %xmm0
# parameter 2(x2): %xmm1
# parameter 3(y1): %xmm2
# parameter 4(y2): %xmm3

movaps    %xmm0, %xmm5      5 <- x1
movaps    %xmm2, %xmm4      4 <- y1
mulsd    %xmm1, %xmm5      5 <- 5 * x2 = x1 * x2
mulsd    %xmm3, %xmm4      4 <- 4 * y2 = y1 * y2
mulsd    %xmm0, %xmm0      0 <- 0 * 0 = x1 * x1
mulsd    %xmm1, %xmm1      1 <- 1 * 1 = x2 * x2
addsd    %xmm4, %xmm5      5 <- 5 + 4 = x1*x2 + y1*y2
mulsd    %xmm2, %xmm2      2 <- 2 * 2 = y1 * y1
mulsd    %xmm3, %xmm3      3 <- 3 * 3 = y2 * y2
addsd    %xmm1, %xmm0      0 <- 0 + 1 = x1*x1 + x2*x2
addsd    %xmm3, %xmm2      2 <- 2 + 3 = y1*y1 + y2*y2
sqrtsd   %xmm0, %xmm0      0 <- sqrt(0) = sqrt( x1*x1 + x2*x2 )
sqrtsd   %xmm2, %xmm2      2 <- sqrt(2) = sqrt( y1*y1 + y2*y2 )
```

which uses the scalar instruction `mulsd`: multiply scalar double precision.

With a `declare simd` directive:

```
movaps    %xmm0, %xmm7
movaps    %xmm2, %xmm4
mulpd    %xmm1, %xmm7
mulpd    %xmm3, %xmm4
```

which uses the vector instruction `mulpd`: multiply packed double precision, operating on 128-bit SSE2 registers.

Compiling for the *Intel Knight's Landing* gives more complicated code:

```
# parameter 1(x1): %xmm0
# parameter 2(x2): %xmm1
# parameter 3(y1): %xmm2
# parameter 4(y2): %xmm3

vmulpd    %xmm3, %xmm2, %xmm4
vmulpd    %xmm1, %xmm1, %xmm5
vbroadcastsd .L_2il0floatpacket.0(%rip), %zmm21
movl      $3, %eax
vbroadcastsd .L_2il0floatpacket.5(%rip), %zmm24
kmovw     %eax, %k3
vmulpd    %xmm3, %xmm3, %xmm6
vfmadd231pd %xmm0, %xmm1, %xmm4
vfmadd213pd %xmm5, %xmm0, %xmm0
vmovaps   %zmm21, %zmm18
vmovapd   %zmm0, %zmm3[%k3]{z}
vfmadd213pd %xmm6, %xmm2, %xmm2
vpcmpgtq %zmm0, %zmm21, %k1[%k3]
vscalefpd .L_2il0floatpacket.1(%rip){1to8}, %zmm0, %zmm3[%k1] #25.26 c15
vmovaps   %zmm4, %zmm26
vmovapd   %zmm2, %zmm7[%k3]{z}
vpcmpgtq %zmm2, %zmm21, %k2[%k3]
vscalefpd .L_2il0floatpacket.1(%rip){1to8}, %zmm2, %zmm7[%k2] #25.26 c19
vrsqrt28pd %zmm3, %zmm16[%k3]{z}
vpxorq    %zmm4, %zmm4, %zmm26[%k3]
vrsqrt28pd %zmm7, %zmm20[%k3]{z}
vmulpd    {rn-sae}, %zmm3, %zmm16, %zmm19[%k3]{z} #25.26 c27 stall 2
vscalefpd .L_2il0floatpacket.2(%rip){1to8}, %zmm16, %zmm17[%k3]{z} #25.26 c27
vmulpd    {rn-sae}, %zmm7, %zmm20, %zmm23[%k3]{z} #25.26 c29
vscalefpd .L_2il0floatpacket.2(%rip){1to8}, %zmm20, %zmm22[%k3]{z} #25.26 c29
vfnmadd231pd {rn-sae}, %zmm17, %zmm19, %zmm18[%k3] #25.26 c33 stall 1
vfnmadd231pd {rn-sae}, %zmm22, %zmm23, %zmm21[%k3]
vfmadd231pd {rn-sae}, %zmm19, %zmm18, %zmm19[%k3]
vfmadd231pd {rn-sae}, %zmm23, %zmm21, %zmm23[%k3]
vfmadd213pd {rn-sae}, %zmm17, %zmm17, %zmm18[%k3]
vfnmadd231pd {rn-sae}, %zmm19, %zmm19, %zmm3[%k3]
vfmadd213pd {rn-sae}, %zmm22, %zmm22, %zmm21[%k3]
vfnmadd231pd {rn-sae}, %zmm23, %zmm23, %zmm7[%k3]
vfmadd213pd %zmm19, %zmm18, %zmm3[%k3]
vfmadd213pd %zmm23, %zmm21, %zmm7[%k3] #25.26 c59
vscalefpd .L_2il0floatpacket.3(%rip){1to8}, %zmm3, %zmm3[%k1] #25.26 c63 stall 1
vscalefpd .L_2il0floatpacket.3(%rip){1to8}, %zmm7, %zmm7[%k2] #25.26 c65
vfixupimmpd $112, .L_2il0floatpacket.4(%rip){1to8}, %zmm0, %zmm3[%k3] #25.26 c65
vfixupimmpd $112, .L_2il0floatpacket.4(%rip){1to8}, %zmm2, %zmm7[%k3] #25.26 c67
vmulpd    %xmm7, %xmm3, %xmm0 #25.26 c71
vmovaps   %zmm0, %zmm27 #25.26 c79
```

```

vmovaps    %zmm0, %zmm25          #25.26 c79
vrcp28pd   {sae}, %zmm0, %zmm27{%k3} #25.26 c81
vfnmadd213pd {rn-sae}, %zmm24, %zmm27, %zmm25{%k3} #25.26 c89 stall 3
vfmadd213pd {rn-sae}, %zmm27, %zmm25, %zmm27{%k3} #25.26 c95 stall 2
vcmpdd    $8, %zmm26, %zmm27, %k1{%k3} #25.26 c101 stall 2
vmulpd    %zmm27, %zmm4, %zmm1{%k3}{z} #25.26 c101
kortestw   %k1, %k1              #25.26 c103
je         ..B1.3      # Prob 25% #25.26 c105
vdivpd    %zmm0, %zmm4, %zmm1{%k1} #25.26 c3 stall 1
vmovaps    %xmm1, %xmm0          #25.26 c77
ret

#pragma omp declare simd uniform(op1) linear(k) notinbranch
double SqrtMul(double *op1, double op2, int k) {
    return (sqrt(op1[k]) * sqrt(op2));
}

```

Chapter 27

OpenMP topic: Offloading

This chapter explains the mechanisms for offloading work to a Graphics Processing Unit (GPU), introduced in OpenMP-4.0.

The memory of a processor and that of an attached GPU are not *coherent*: there are separate memory spaces and writing data in one is not automatically reflected in the other.

OpenMP transfers data (or maps it) when you enter an `target` construct.

```
#pragma omp target
{
    // do stuff on the GPU
}
```

You can test whether the target region is indeed executed on a device with `omp_is_initial_device`:

```
#pragma omp target
if (omp_is_initial_device()) printf("Offloading failed\n");
```

27.0.1 Targets and tasks

The `target` clause causes OpenMP to create a *target task*. This is a task running on the host, dedicated to managing the offloaded region.

The `target` region is executed by a new *initial task*. This is distinct from the initial task that executes the main program.

The task that created the target task is called the *generating task*.

By default, the generating task is blocked while the task on the device is running, but adding the `targetnowait` clause makes it asynchronous. This requires a `taskwait` directive to synchronize host and device.

27.1 Data on the device

- Scalars are treated as `firstprivate`, that is, they are copied in but not out.
- Stack arrays `tofrom`.
- Heap arrays are not mapped by default.

For explicit mapping with `map`:

```
#pragma omp target map(...)  
{  
    // do stuff on the GPU  
}
```

The following map options exist:

- `map(to: x,y,z)` copy from host to device when entering the target region.
- `map(from: x,y,z)` copy from device to host when exiting the target region.
- `map(tofrom: x,y,z)` is equivalent to combining the previous two.
- `map(allo: x,y,z)` allocates data on the device.

Fortran note 27: Array sizes in map clause. If the compiler can deduce the array bounds and size, it is not necessary to specify them in the ‘map’ clause.

Data transfer to a device is probably slow, so mapping the data at the start of an offloaded section of code is probably not the best idea. Additionally, in many cases data will stay resident on the device throughout several iterations of, for instance, a time-stepping PDE solver. For such reasons, it is possible to move data onto, and off from, the device explicitly, using the `enter data` and `exit data` directives.

```
#pragma omp target enter data map(to: x,y)  
#pragma omp target  
{  
    // do something  
}  
#pragma omp target enter data map(from: x,y)
```

Also update `to` (synchronize data from host to device), update `from` (synchronize data to host from device).

27.2 Execution on the device

For parallel execution of a loop on the device use the `teams` clause:

```
#pragma omp target teams distribute parallel do
```

On GPU devices and the like, there is a structure to threads:

- threads are grouped in `teams`, and they can be synchronized only within these teams;
- teams are groups in `leagues`, and no synchronization between leagues is possible inside a `target` region.

The combination `teams distribute` splits the iteration space over teams. By default a static schedule is used, but the option `dist_schedule` can be used to specify a different one. However, this combination only gives the chunk of space to the master thread in each team. Next we need `parallel for` or `parallel do` to spread the chunk over the threads in the team.

When creating teams, it's often useful to limit the number of threads in each with `thread_limit`. This can also be set with the `OMP_THREAD_LIMIT` environment variable. The value can be queried with `omp_get_thread_limit`.

Chapter 28

OpenMP remaining topics

28.1 Runtime functions, environment variables, internal control variables

OpenMP has a number of settings that can be set through *environment variables*, and both queried and set through *library routines*. These settings are called *Internal Control Variables (ICVs)*: an OpenMP implementation behaves as if there is an internal variable storing this setting.

The runtime functions are:

- Counting threads and cores: `omp_set_num_threads`, `omp_get_num_threads`, `omp_get_max_threads`, `omp_get_num_procs`; see section 17.5.
- Querying the current thread: `omp_get_thread_num`, `omp_in_parallel`
- `omp_get_cancellation`
- `omp_set_dynamic`
- `omp_get_dynamic`
- `omp_set_nested`
- `omp_get_nested`
- `omp_get_wtime`
- `omp_get_wtick`
- `omp_set_schedule`
- `omp_get_schedule`
- `omp_set_max_active_levels`
- `omp_get_max_active_levels`
- `omp_get_thread_limit`
- `omp_get_level`
- `omp_get_active_level`
- `omp_get_ancestor_thread_num`
- `omp_get_team_size`

Here are the OpenMP *environment variables*:

- `OMP_CANCELLATION` Set whether cancellation is activated; see section 18.3. Can be queried with `omp_get_cancellation` but there is no routine for setting the value.
- `OMP_DISPLAY_ENV` Show OpenMP version (section 28.7) and environment variables.
- `OMP_DEFAULT_DEVICE` Set the device used in target regions; see chapter 27.
- `OMP_DYNAMIC` Dynamic adjustment of threads. Set and query with `omp_set_dynamic` and `omp_get_dynamic` respectively.
- `OMP_MAX_ACTIVE_LEVELS` Set the maximum number of nested parallel regions; section 18.2. Access with `omp_set_max_active_levels` and `omp_get_max_active_levels`.

- **OMP_NESTED** Use of nested parallel regions. Access with `omp_set_nested` and `omp_get_nested`. Deprecated: use ‘active levels’ instead.
- **OMP_MAX_TASK_PRIORITY** Set the maximum task priority value; section 24.6.2.
- **OMP_NUM_THREADS** Specifies the number of threads to use
- **OMP_PROC_BIND** Whether theads may be moved between CPUs; section 25.1.
- **OMP_PLACES** Specifies on which CPUs the theads should be placed; section 25.1.
- **OMP_STACKSIZE** Set default thread stack size; section 22.2.
- **OMP_SCHEDULE** How threads are scheduled; section 19.3.
- **OMP_THREAD_LIMIT** Set the maximum number of threads; see section 27.2.
- **OMP_WAIT_POLICY** How waiting threads are handled; ICV `wait-policy-var`. Values: ACTIVE for keeping threads spinning, PASSIVE for possibly yielding the processor when threads are waiting. There is no runtime function for setting this.

There are 4 ICVs that behave as if each thread has its own copy of them. The default is implementation-defined unless otherwise noted.

- It may be possible to adjust dynamically the number of threads for a parallel region. Variable: `OMP_DYNAMIC`; routines: `omp_set_dynamic`, `omp_get_dynamic`.
- If a code contains *nested parallel regions*, the inner regions may create new teams, or they may be executed by the single thread that encounters them. Variable: `OMP_NESTED`; routines `omp_set_nested`, `omp_get_nested`. Allowed values are TRUE and FALSE; the default is false.
- The number of threads used for an encountered parallel region can be controlled. Variable: `OMP_NUM_THREADS`; routines `omp_set_num_threads`, `omp_get_max_threads`.
- The schedule for a parallel loop can be set. Variable: `OMP_SCHEDULE`; routines `omp_set_schedule`, `omp_get_schedule`.

Nonobvious syntax:

```
export OMP_SCHEDULE="static,100"
```

Other settings:

- `omp_get_num_threads`: query the number of threads active at the current place in the code; this can be lower than what was set with `omp_set_num_threads`. For a meaningful answer, this should be done in a parallel region.
- `omp_get_thread_num`
- `omp_in_parallel`: test if you are in a parallel region.
- `omp_get_num_procs`: query the physical number of cores available.

Other environment variables:

- **OMP_STACKSIZE** controls the amount of space that is allocated as per-thread *stack*. This is used as space for private variables, see section 22.2, or reductions, see section 20.2.2.
- **OMP_WAIT_POLICY** determines the behavior of threads that wait, for instance for *critical section*:
 - ACTIVE puts the thread in a *spin-lock*, where it actively checks whether it can continue;
 - PASSIVE puts the thread to sleep until the Operating System (OS) wakes it up.
- The ‘active’ strategy uses CPU while the thread is waiting; on the other hand, activating it after the wait is instantaneous. With the ‘passive’ strategy, the thread does not use any CPU while waiting, but activating it again is expensive. Thus, the passive strategy only makes sense if threads will be waiting for a (relatively) long time.
- **OMP_PROC_BIND** with values TRUE and FALSE can bind threads to a processor. On the one hand, doing so can minimize data movement; on the other hand, it may increase load imbalance.

28.2 Timing

OpenMP has a wall clock timer routine `omp_get_wtime`

```
double omp_get_wtime(void);
```

The starting point is arbitrary and is different for each program run; however, in one run it is identical for all threads. This timer has a resolution given by `omp_get_wtick`.

Exercise 28.1. Use the timing routines to demonstrate speedup from using multiple threads.

- Write a code segment that takes a measurable amount of time, that is, it should take a multiple of the tick time.
- Write a parallel loop and measure the speedup. You can for instance do this

```
for (int use_threads=1; use_threads<=nthreads; use_threads++) {  
    #pragma omp parallel for num_threads(use_threads)  
    for (int i=0; i<nthreads; i++) {  
        ....  
    }  
    if (use_threads==1)  
        time1 = tend-tstart;  
    else // compute speedup
```

- In order to prevent the compiler from optimizing your loop away, let the body compute a result and use a reduction to preserve these results.

28.3 Thread safety

With OpenMP it is relatively easy to take existing code and make it parallel by introducing parallel sections. If you're careful to declare the appropriate variables shared and private, this may work fine. However, your code may include calls to library routines that include a *race condition*; such code is said not to be *thread-safe*.

For example a routine

```
static int isave;  
int next_one() {  
    int i = isave;  
    isave += 1;  
    return i;  
}  
  
...  
for ( .... ) {  
    int ivalue = next_one();  
}
```

has a clear race condition, as the iterations of the loop may get different `next_one` values, as they are supposed to, or not. This can be solved by using an `critical` pragma for the `next_one` call; another solution is to use an `threadprivate` declaration for `isave`. This is for instance the right solution if the `next_one` routine implements a *random number generator*.

28.4 Performance and tuning

The performance of an OpenMP code can be influenced by the following.

Amdahl effects Your code needs to have enough parts that are parallel (see HPC book, section ??). Sequential parts may be sped up by having them executed redundantly on each thread, since that keeps data locally.

Dynamism Creating a thread team takes time. In practice, a team is not created and deleted for each parallel region, but creating teams of different sizes, or resize thread creation, may introduce overhead.

Load imbalance Even if your program is parallel, you need to worry about load balance. In the case of a parallel loop you can set the *schedule* clause to *dynamic*, which evens out the work, but may cause increased communication.

Communication Cache coherence causes communication. Threads should, as much as possible, refer to their own data.

- Threads are likely to read from each other's data. That is largely unavoidable.
- Threads writing to each other's data should be avoided: it may require synchronization, and it causes coherence traffic.
- If threads can migrate, data that was local at one time is no longer local after migration.
- Reading data from one socket that was allocated on another socket is inefficient; see section 25.2.

Affinity Both data and execution threads can be bound to a specific locale to some extent. Using local data is more efficient than remote data, so you want to use local data, and minimize the extent to which data or execution can move.

- See the above points about phenomena that cause communication.
- Section 25.1.1 describes how you can specify the binding of threads to places. There can, but does not need, to be an effect on affinity. For instance, if an OpenMP thread can migrate between hardware threads, cached data will stay local. Leaving an OpenMP thread completely free to migrate can be advantageous for load balancing, but you should only do that if data affinity is of lesser importance.
- Static loop schedules have a higher chance of using data that has affinity with the place of execution, but they are worse for load balancing. On the other hand, the *nowait* clause can alleviate some of the problems with static loop schedules.

Binding You can choose to put OpenMP threads close together or to spread them apart. Having them close together makes sense if they use lots of shared data. Spreading them apart may increase bandwidth. (See the examples in section 25.1.2.)

Synchronization Barriers are a form of synchronization. They are expensive by themselves, and they expose load imbalance. Implicit barriers happen at the end of worksharing constructs; they can be removed with *nowait*.

Critical sections imply a loss of parallelism, but they are also slow as they are realized through *operating system* functions. These are often quite costly, taking many thousands of cycles. Critical sections should be used only if the parallel work far outweighs it.

28.5 Accelerators

In OpenMP-4.0 there is support for offloading work to an *accelerator* or *co-processor*:

```
#pragma omp target [clauses]
```

with clauses such as

- *data*: place data
- *update*: make data consistent between host and device

28.6 Tools interface

The OpenMP-5.0 defines a tools interface. This means that routines can be defined that get called by the OpenMP runtime. For instance, the following example defines callback that are evaluated when OpenMP is initialized and finalized, thereby giving the runtime for the application.

```
int ompt_initialize(ompt_function_lookup_t lookup, int initial_device_num,
                     ompt_data_t *tool_data) {
    printf("libomp init time: %f\n",
           omp_get_wtime() - *(double*)(tool_data->ptr));
    *(double*)(tool_data->ptr) = omp_get_wtime();
    return 1; // success: activates tool
}

void ompt_finalize(ompt_data_t *tool_data) {
    printf("application runtime: %f\n",
           omp_get_wtime() - *(double*)(tool_data->ptr));
}

ompt_start_tool_result_t *ompt_start_tool(unsigned int omp_version,
                                         const char *runtime_version) {
    static double time = 0; // static defintion needs constant assignment
    time = omp_get_wtime();
    static ompt_start_tool_result_t ompt_start_tool_result = {
        &ompt_initialize, &ompt_finalize, {.ptr = &time}};
    return &ompt_start_tool_result; // success: registers tool
}
```

(Example courtesy of <https://git.rwth-aachen.de/OpenMPTools/OMPT-Examples>.)

28.7 OpenMP standards

Here is the correspondence between the value of OpenMP versions (given by the `_OPENMP` macro) and the *standard versions*:

- OpenMP-3.1
 - proc bind environment variable
 - extensions to tasks
- OpenMP-4.0
 - procbind clause, places environment variable
 - simd directives
 - device directives for GPUs
 - taskgroups
 - depend clause on tasks
 - cancel
 - user-defined reductions
- 201511 OpenMP-4.5, Many extensions of existing constructs.
- 201611 Technical report 4: information about the OpenMP-5.0 but not yet mandated.
- 201811 OpenMP-5.0
 - Better support for C11, C++11/14/18, Fortran2008
 - Non-rectangular loop nests.

- `scan` extended to have in/exclusive versions
- reduction on tasks, taskloops.
- memory spaces.
- 202011 OpenMP-5.1,
- 202111 OpenMP-5.2.

```
// version.c
int standard = _OPENMP;
printf("Supported OpenMP standard: %d\n",
      standard);
switch (standard) {
case 201511: printf("4.5\n");
  break;
case 201611: printf("Technical report 4:
                   information about 5.0 but not yet mandated.\n");
  break;
case 201811: printf("5.0\n");
  break;
case 202011:
  printf("5.1\n");
  break;
case 202111: printf("5.2\n");
  break;
default:
  printf("Unrecognized version\n");
  break;
}
```

The `openmp.org` website maintains a record of which compilers support which standards: <https://www.openmp.org/resources/openmp-compilers-tools/>.

28.8 Memory model

28.8.1 Dekker's algorithm

A standard illustration of the weak memory model is *Dekker's algorithm*. We model that in OpenMP as follows;

```
// weak1.c
int a=0,b=0,r1,r2;
#pragma omp parallel sections shared(a, b, r1, r2)
{
#pragma omp section
{
    a = 1;
    r1 = b;
    tasks++;
}
#pragma omp section
{
    b = 1;
    r2 = a;
    tasks++;
}
}
```

Under any reasonable interpretation of parallel execution, the possible values for `r1, r2` are 1, 1 0, 1 or 1, 0. This is known as *sequential consistency*: the parallel outcome is consistent with a sequential execution that interleaves the parallel computations, respecting their local statement orderings. (See also HPC book, section ??.)

However, running this, we get a small number of cases where $r_1 = r_2 = 0$. There are two possible explanations:

1. The compiler is allowed to interchange the first and second statements, since there is no dependence between them; or
2. The thread is allowed to have a local copy of the variable that is not coherent with the value in memory.

We fix this by flushing both a,b:

```
// weak2.c
int a=0,b=0,r1,r2;
#pragma omp parallel sections shared(a, b, r1, r2)
{
#pragma omp section
{
    a = 1;
#pragma omp flush (a,b)
    r1 = b;
    tasks++;
}
#pragma omp section
{
    b = 1;
#pragma omp flush (a,b)
    r2 = a;
    tasks++;
}
}
```

Chapter 29

OpenMP Exercises and examples

29.1 Histograms

A *histogram* is a way of counting a large number of objects into a small number of bins:

```
for ( i /* lots of cases */ )
    bin[ property(i) ]++;
```

Without a specific application, we let the bin number be random:

MISSING SNIPPET histobasic in codesnippetsdir=snippets

(Here we use `rand_r` as a threadsafe variant of `rand`.)

Exercise 29.1. Realize this code by only putting a parallel region around it. Observe that the sum over the bins is not the total number of cases executed. Why is this?
Experiment with number of threads and bins.

The simplest correct solution to this code uses a reduction. See section 20.2.2 for the syntax.

Exercise 29.2. Use a reduction to get the right result of the histogram.

Investigate runtime as a function of the number of threads and bins.

C++ only: can you make a histogram class with an overloaded plus-operator?

A different solution would be the use of locks; section 23.3. Here we would have a lock for each bin:

MISSING SNIPPET histolockalloc in codesnippetsdir=snippets

and each time we want to update a bit, we first lock it.

Exercise 29.3. Write correct histogramming code using locks.

Investigate runtime as a function of the number of threads and bins.

C++ only: can you make a bin class that incorporates the lock? Overload the increment operator, or use an increment function to set/unset the lock.

Did you see the runtime go down as a function of increasing number of bins? Can you explain?

29.2 N-body problems

So-called *N-body problems* come up with we describe the interactions between a, probably large, number of entities under a force such as gravity. Examples are molecular dynamics and star clusters.

29. OpenMP Exercises and examples

While clever algorithms exist that take into account the decay of the force over distance, we here consider the naive algorithm that explicitly computes all interactions.

A particle has x, y coordinates and a mass c . For two particles $(x_1, y_1, c_1), (x_2, y_2, c_2)$ the force on particle 1 from particle 2 is:

$$\vec{F}_{12} = \frac{c_1 \cdot c_2}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \cdot \vec{r}_{12}$$

where \vec{r}_{12} is the unit vector pointing from particle 2 to 1. With n particles, each particle i feels a force

$$\vec{F}_i = \sum_{j \neq i} \vec{F}_{ij}.$$

Let's start with a couple of building blocks.

```
// molecularstruct.c
struct point{ double x,y; double c; };
struct force{ double x,y; double f; };

/* Force on p1 from p2 */
struct force force_calc( struct point p1,struct point p2 ) {
    double dx = p2.x - p1.x, dy = p2.y - p1.y;
    double f = p1.c * p2.c / sqrt( dx*dx + dy*dy );
    struct force exert = {dx,dy,f};
    return exert;
}
```

Force accumulation:

```
void add_force( struct force *f,struct force g ) {
    f->x += g.x; f->y += g.y; f->f += g.f;
}
void sub_force( struct force *f,struct force g ) {
    f->x -= g.x; f->y -= g.y; f->f += g.f;
}
```

In C++ we can have a class with an addition operator and such:

```
// molecular.cxx
class force {
private:
    double _x{0.},_y{0.}; double _f{0.};
public:
    force() {};
    force(double x,double y,double f)
        : _x(x),_y(y),_f(f) {};

    force operator+( const force& g ) {
        return { _x+g._x, _y+g._y, _f+g._f };
    }
}
```

For reference, this is the sequential code:

```

for (int ip=0; ip<N; ip++) {
    for (int jp=ip+1; jp<N; jp++) {
        struct force f = force_calc(points[ip],points[jp]);
        add_force( forces+ip,f );
        sub_force( forces+jp,f );
    }
}

```

Here \vec{F}_{ij} is only computed for $j > i$, and then added to both \vec{F}_i and \vec{F}_j .

In C++ we use the overloaded operators:

```

for (int ip=0; ip<N; ip++) {
    for (int jp=ip+1; jp<N; jp++) {
        force f = points[ip].force_calc(points[jp]);
        forces[ip] += f;
        forces[jp] -= f;
    }
}

```

Exercise 29.4. Argue that both the outer loop and the inner are not directly parallelizable.

We will now explore a number of different strategies for parallelization. All tests are done on the *TACC Frontera* cluster, which has dual-socket *Intel Cascade Lake* nodes, with a total of 56 cores. Our code uses 10 thousand particles, and each interaction evaluation is repeated 10 times to eliminate cache loading effects.

29.2.1 Solution 1: no conflicting writes

In our first attempt at an efficient parallel code, we compute the full N^2 interactions. One solution would be to compute the \vec{F}_{ij} interactions for all i, j , so that there are no conflicting writes.

```

for (int ip=0; ip<N; ip++) {
    struct force sumforce;
    sumforce.x=0.; sumforce.y=0.; sumforce.f=0.;
#pragma omp parallel for reduction(:sumforce)
    for (int jp=0; jp<N; jp++) {
        if (ip==jp) continue;
        struct force f = force_calc(points[ip],points[jp]);
        sumforce.x += f.x; sumforce.y += f.y; sumforce.f += f.f;
    } // end parallel jp loop
    add_force( forces+ip, sumforce );
} // end ip loop

```

In C++ we use the fact that we can reduce on any class that has an addition operator:

```

for (int ip=0; ip<N; ip++) {
    force sumforce;
#pragma omp parallel for reduction(:sumforce)
    for (int jp=0; jp<N; jp++) {
        if (ip==jp) continue;
        force f = points[ip].force_calc(points[jp]);
        sumforce += f;
    } // end parallel jp loop
    forces[ip] += sumforce;
} // end ip loop

```

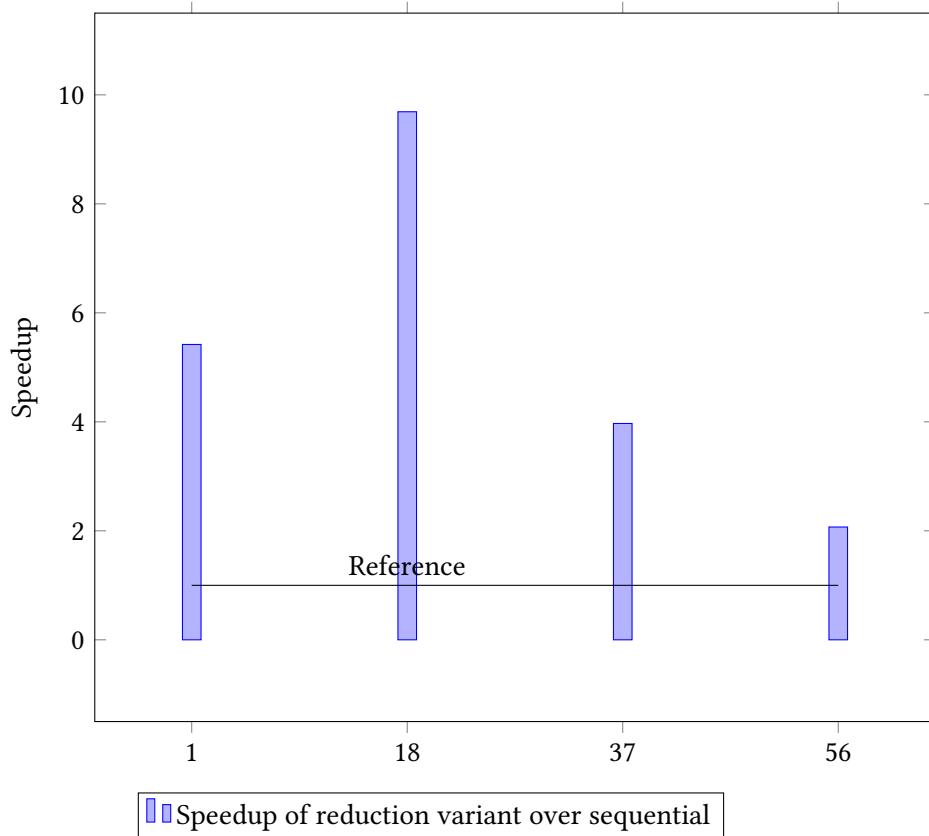


Figure 29.1: Speedup of reduction variant over sequential

This increases the scalar work by a factor of two, but surprisingly, on a single thread the run time improves: we measure a speedup of 6.51 over the supposedly ‘optimal’ code.

Exercise 29.5. What would be an explanation?

However, increasing the number of threads has limited benefits for this strategy. Figure 29.1 shows that the speedup is not only sublinear: it actually decreases with increasing core count.

Exercise 29.6. What would be an explanation?

29.2.2 Solution 2: using atomics

Next we try to parallelize the outer loop.

```
#pragma omp parallel for schedule(guided,4)
for (int ip=0; ip<N; ip++) {
    for (int jp=ip+1; jp<N; jp++) {
        struct force f = force_calc(points[ip],points[jp]);
        add_force( forces+ip,f );
        sub_force( forces+jp,f );
    }
}
```

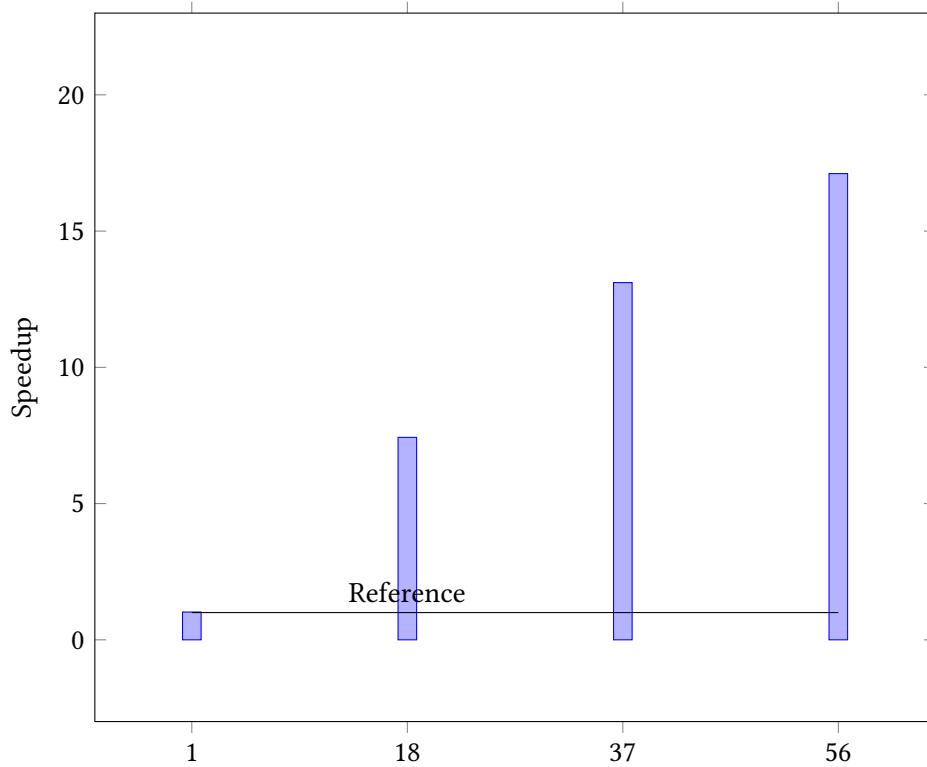


Figure 29.2: Speedup of triangular loop with atomic update

To deal with the conflicting jp writes, we make the writes atomic:

```
void sub_force( struct force *f,struct force g ) {
    #pragma omp atomic
    f->x -= g.x;
    #pragma omp atomic
    f->y -= g.y;
    #pragma omp atomic
    f->f += g.f;
}
```

This works fairly well, as figure 29.2 shows.

29.2.3 Solution 3: all interactions atomic

But if we decide to use atomic updates, we can take the full square loop, collapse the two loops, and make every write atomic.

```
#pragma omp parallel for collapse(2)
    for (int ip=0; ip<N; ip++) {
        for (int jp=0; jp<N; jp++) {
            if (ip==jp) continue;
            struct force f = force_calc(points[ip],points[jp]);
```

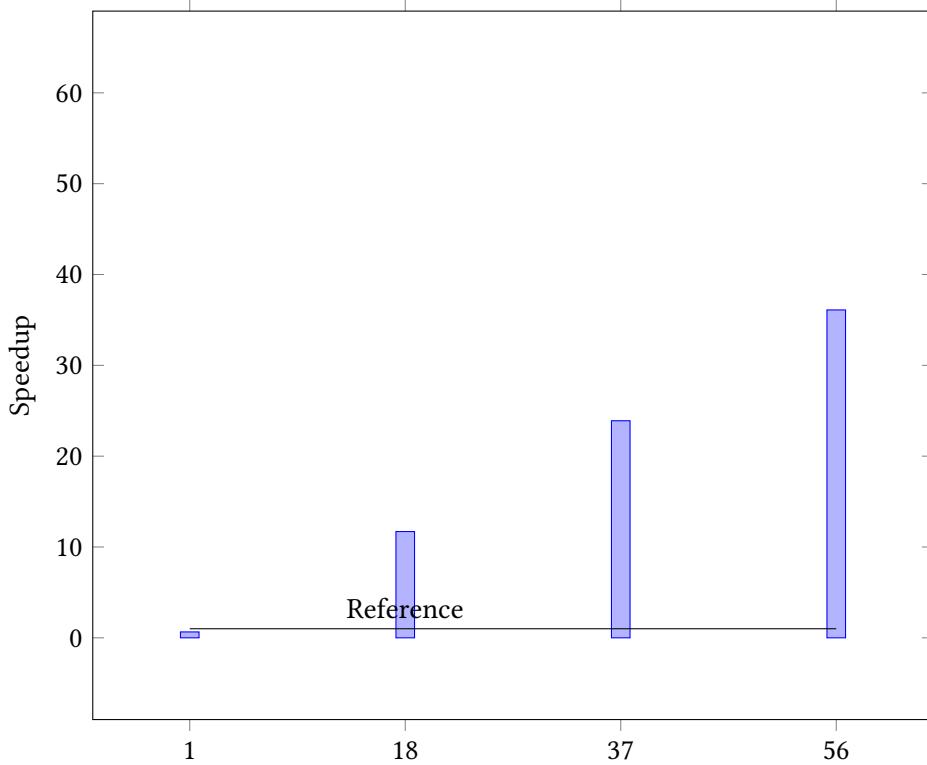


Figure 29.3: Speedup of atomic full interaction calculation

```
    add_force( forces+ip, f );
} // end parallel jp loop
} // end ip loop
```

Figure 29.3 shows that this is pretty close to perfect.

Everything in one plot in figure 29.4.

29.3 Tree traversal

OpenMP tasks are a great way of handling trees.

In *post-order tree traversal* you visit the subtrees before visiting the root. This is the traversal that you use to find summary information about a tree, for instance the sum of all nodes, and the sums of nodes of all subtrees:

```
for all children c do
    compute the sum  $s_c$ 
```

$$s \leftarrow \sum_c s_c$$

Another example is matrix factorization:

$$S = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}$$

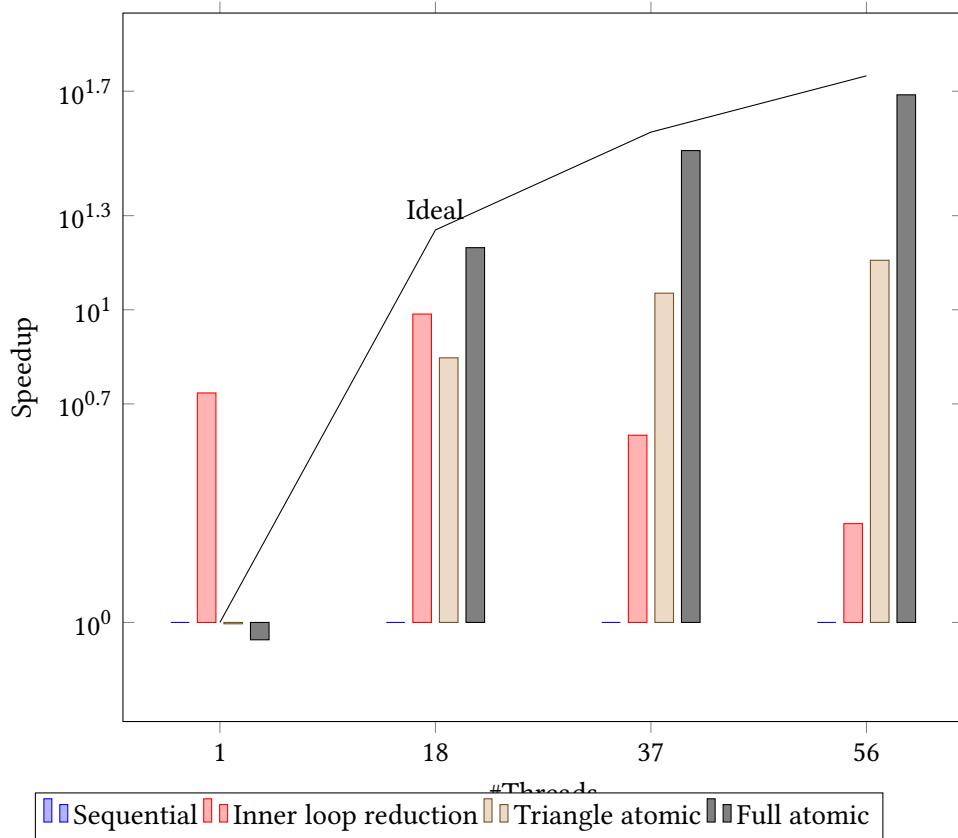


Figure 29.4: All strategies together

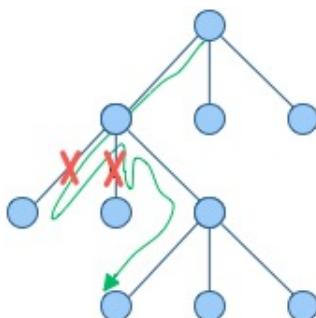
where the two inverses A_{11}^{-1}, A_{22}^{-1} can be computed independently and recursively.

29.4 Depth-first search

In this section we look at the ‘eight queens’ problem, as an example of *Depth First Search (DFS)*: is it possible to put eight queens on a chess board so that none of them threaten each other? With DFS, the search space of possibilities is organized as a tree – each partial solution leads to several possibilities for the next steps – which is traversed in a particular manner: a chain of possibilities is extended as far as feasible, after which the search backtracks to the next chain.

The sequential implementation is easy enough. The main program fires off:

```
placement initial; initial.fill(empty);
auto solution = place_queen(0,initial);
```



29. OpenMP Exercises and examples

where I hope you can take the details on trust.

The recursive call then has this structure:

```
optional<placement> place_queen(int iqueen, const
placement& current) {
    for (int col=0; col<N; col++) {
        placement next = current;
        next.at(iqueen) = col;
        if (feasible(next)) {
            if (iqueen==N-1)
                return next;
            auto attempt = place_queen(iqueen+1,next);
            if (attempt.has_value())
                return attempt;
        } // end if(feasible)
    }
    return {};
};
```

(This uses the C++17 *optional* header.) At each *iqueen* level we

- go through a loop of all column positions;
- filter out positions that are not feasible;
- report success if this was the last level; or
- recursively continue the next level otherwise.

This problem seems a prime candidate for OpenMP tasks, so we start with the usual idiom for the main program:

```
placement initial; initial.fill(empty);
optional<placement> eightqueens;
#pragma omp parallel
#pragma omp single
eightqueens = place_queen(0,initial);
```

We create a task for each column, and since they are in a loop we use **taskgroup** rather than **taskwait**.

```
#pragma omp taskgroup
for (int col=0; col<N; col++) {
    placement next = current;
    next.at(iqueen) = col;
    #pragma omp task firstprivate(next)
    if (feasible(next)) {
        // stuff
    } // end if(feasible)
}
```

However, the sequential program had **return** and **break** statements in the loop, which is not allowed in workshare constructs such as **taskgroup**. Therefore we introduce a return variable, declared as shared:

```
// queens0.cxx
optional<placement> result = {};
#pragma omp taskgroup
for (int col=0; col<N; col++) {
```

```

placement next = current;
next.at(iqueen) = col;
#pragma omp task firstprivate(next) shared(result)
if (feasible(next)) {
    if (iqueen==N-1) {
        result = next;
    } else { // do next level
        auto attempt = place_queen(iqueen+1,next);
        if (attempt.has_value()) {
            result = attempt;
        }
    }
} // end if(iqueen==N-1)
} // end if(feasible)
}
return result;
}

```

So that was easy, this computes the right solution, and it uses OpenMP tasks. Done?

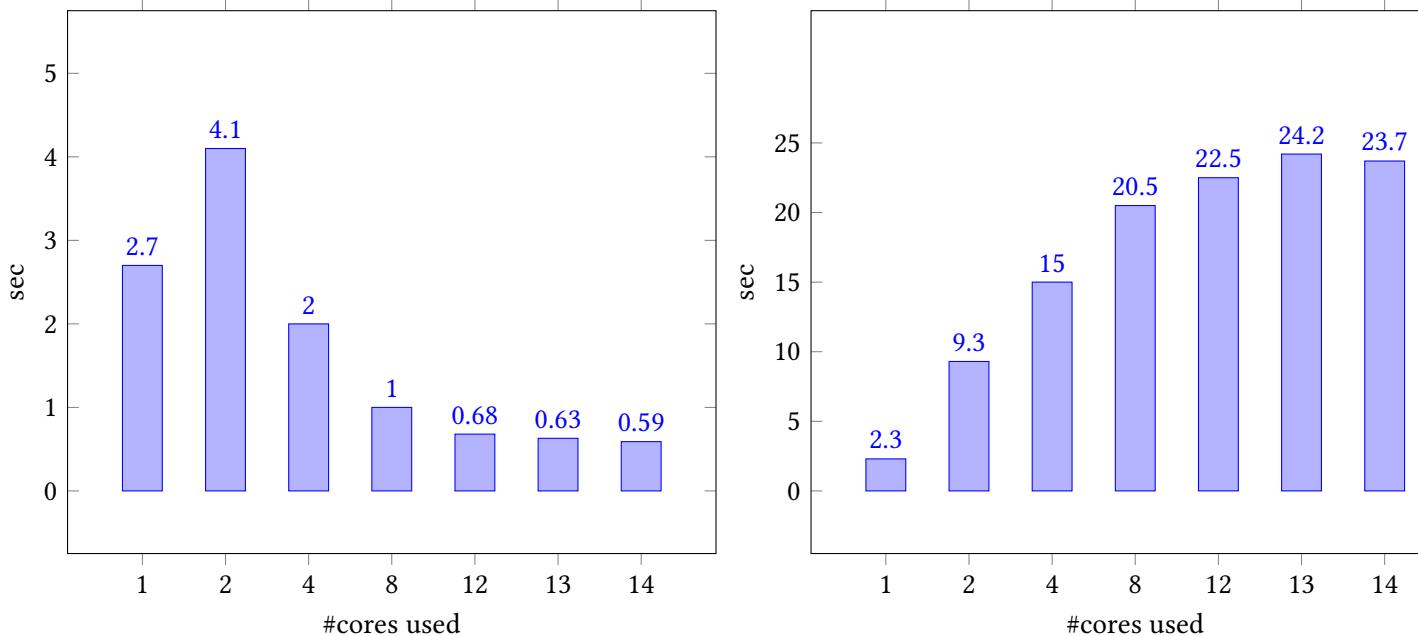


Figure 29.5: Using taskgroups for $N = 12$; left Intel compiler, right GCC

Actually this runs very slowly because, now that we've dispensed with all early breaks from the loop, we in effect traverse the whole search tree. (It's not quite breadth-first, though.) Figure 29.5 shows this for $N = 12$ with the Intel compiler (version 2019) in the left panel, and the GNU compiler (version 9.1) in the middle. In both cases, the blue bars give the result for the code with only the `taskgroup` directive, with time plotted as function of core count.

We see that, for the Intel compiler, running time indeed goes down with core count. So, while we compute too much (the whole search space), at least parallelization helps. With a number of threads greater than the problem size, the benefit of parallelization disappears, which makes some sort of sense.

We also see that the GCC compiler is really bad at OpenMP tasks: the running time actually increases with the number of threads.

Fortunately, with OpenMP-4 we can break out of the loop with a `cancel` of the task group:

```
// queenfinal.cxx
if (feasible(next)) {
    if (iqueen==N-1) {
        result = next;
        #pragma omp cancel taskgroup
    } else { // do next level
        auto attempt = place_queen(iqueen+1,next);
        if (attempt.has_value()) {
            result = attempt;
        #pragma omp cancel taskgroup
        }
    } // end if (iqueen==N-1)
} // end if (feasible)
```

Surprisingly, this does not immediately give a performance improvement. The reason for this is that cancellation is disabled by default, and we have to set the environment variable

`OMP_CANCELLATION=true`

With that, we get very good performance, as figure 29.6 shows, which lists sequential time, and multicore running time on the code with `cancel` directives. Running time is now approximately the same as the sequential time. Some questions are still left:

- Why does the time go up with core count?
- Why is the multicore code slower than the sequential code, and would the parallel code be faster than sequential if the amount of scalar work (for instance in the `feasible` function) would be larger?

One observation not reported here is that the GNU compiler has basically the same running time with and without cancellation. This is again shows that the GNU compiler is really bad at OpenMP tasks.

29.5 Filtering array elements

Let's assume we have an array of elements (integers, for the sake of the argument) and we want to construct a subarray of only those elements that satisfy some test

```
bool f(int);
```

C++ note 27: List filtering example. We will do this example only in C++ because of its ease of handling `std::vector`s.

The sequential code is as follows:

```
vector<int> data(100);
// fill the data
vector<int> filtered;
for ( auto e : data ) {
    if ( f(e) )
        filtered.push_back(e);
}
```

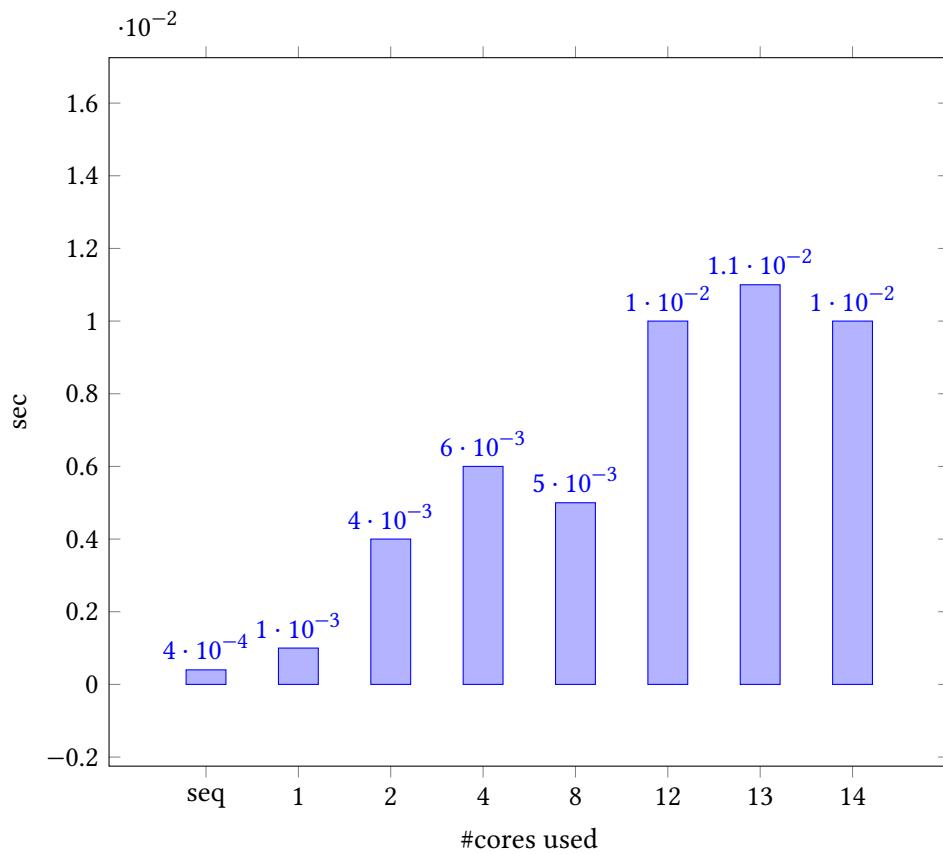


Figure 29.6: Using taskgroup cancelling, Intel compiler

There are two problems here. First is the *race condition* on the filtered array. Even if we fix this with a critical region, there remains the lack of ordering of inserted elements.

The key to the solution is to let each thread have a local array, and then to concatenate these:

```
#pragma omp parallel
{
    vector<int> local;
# pragma omp for
    for ( auto e : data ) {
        if ( f(e) ) {
            local.push_back(e);
        }
    }
    filtered += local;
}
```

where we have used an append operation on vectors:

```
// filterreduct.cxx
template<typename T>
vector<T>& operator+=( vector<T>& me, const vector<T>& other ) {
```

```
me.insert( me.end(), other.begin(), other.end() );
return me;
};
```

29.5.1 Attempt 1: reduction

We could use the plus-is operation to declare a reduction:

```
#pragma omp declare reduction \
(           \
  +:vector<int>:omp_out += omp_in \
) \
initializer( omp_priv = vector<int>{} )
```

The problem here is that OpenMP reductions can not be declared non-commutative, so the contributions from the threads may not appear in order.

Code:

```
#pragma omp parallel \
reduction(+ : filtered)
{
    vector<int> local;
# pragma omp for
    for ( auto e : data )
        if ( f(e) )
            local.push_back(e);
    filtered += local;
}
```

Output:

```
[code/omp/cxx] filterreduct:
Mod 5: 80 85 90 95 100 5 10 15 20
      ↢25 30 35 40 45 50 55 60 65
      ↢70 75
```

29.5.2 Attempt 2: sequential appending

If we don't use a reduction, but do the appending explicitly, we first of all have to make this operation into a critical section. Secondly, we have to impose the correct ordering.

Here is an attempt to do this by keeping a shared counter:

Code:

```
// filteratomic.cxx
# pragma omp critical
    if (threadnum==ithread) {
        filtered += local;
        ithread++;
    }
```

Output:

```
[code/omp/cxx] filteratomic:
Mod 5: 5 10 15 20 25 30 35 40 45 50
```

The problem here is that threads who decide it's not their turn will simply skip the append operation: there is no way to tell them to wait their turn. (You could experiment with a while loop. Try it.)

The best solution is to use a completely different mechanism.

29.5.3 Attempt 3: using tasks

With a task it becomes possible to have a spin-wait loop:

```
Code:
# pragma omp task \
    shared(filtered,ithread)
{
// wait your turn
    while (threadnum>ithread) {
#     pragma omp taskyield
    }
// merge
    filtered += local;
    ithread++;
}
```

```
Output:
[code/omp/cxx] filtertask:
Mod 5: 5 10 15 20 25 30 35 40 45
→50 55 60 65 70 75 80 85 90
→95 100
```

29.6 Thread synchronization

Let's do a *producer-consumer* model¹. This can be implemented with sections, where one section, the producer, sets a flag when data is available, and the other, the consumer, waits until the flag is set.

```
#pragma omp parallel sections
{
    // the producer
    #pragma omp section
    {
        ... do some producing work ...
        flag = 1;
    }
    // the consumer
    #pragma omp section
    {
        while (flag==0) { }
        ... do some consuming work ...
    }
}
```

One reason this doesn't work, is that the compiler will see that the flag is never used in the producing section, and that is never changed in the consuming section, so it may optimize these statements, to the point of optimizing them away.

The producer then needs to do:

```
... do some producing work ...
#pragma omp flush
#pragma atomic write
    flag = 1;
#pragma omp flush(flag)
```

1. This example is from Intel's excellent OMP course by Tim Mattson

and the consumer does:

```
#pragma omp flush(flag)
while (flag==0) {
    #pragma omp flush(flag)
}
#pragma omp flush
```

This code strictly speaking has a *race condition* on the `flag` variable.

The solution is to make this an *atomic operation* and use an `atomic` pragma here: the producer has

```
#pragma atomic write
    flag = 1;
```

and the consumer:

```
while (1) {
    #pragma omp flush(flag)
    #pragma omp atomic read
        flag_read = flag
    if (flag_read==1) break;
}
```

PART III

PETSC

Chapter 30

PETSc basics

30.1 What is PETSc and why?

PETSc is a library with a great many uses, but for now let's say that it's primarily a library for dealing with the sort of linear algebra that comes from discretized PDEs. On a single processor, the basics of such computations can be coded out by a grad student during a semester course in numerical analysis, but on large scale issues get much more complicated and a library becomes indispensable.

PETSc's prime justification is then that it helps you realize scientific computations at large scales, meaning large problem sizes on large numbers of processors.

There are two points to emphasize here:

- Linear algebra with dense matrices is relatively simple to formulate. For sparse matrices the amount of logistics in dealing with nonzero patterns increases greatly. PETSc does most of that for you.
- Linear algebra on a single processor, even a multicore one, is manageable; distributed memory parallelism is much harder, and distributed memory sparse linear algebra operations are doubly so. Using PETSc will save you many, many, Many! hours of coding over developing everything yourself from scratch.

Remark *The PETSc library has hundreds of routines. In this chapter and the next few we will only touch on a basic subset of these. The full list of man pages can be found at <https://petsc.org/release/docs/manualpages/singleindex.html>. Each man page comes with links to related routines, as well as (usually) example codes for that routine.*

30.1.1 What is in PETSc?

The routines in PETSc (of which there are hundreds) can roughly be divided in these classes:

- Basic linear algebra tools: dense and sparse matrices, both sequential and parallel, their construction and simple operations.
- Solvers for linear systems, and to a lesser extent nonlinear systems; also time-stepping methods.
- Profiling and tracing: after a successful run, timing for various routines can be given. In case of failure, there are traceback and memory tracing facilities.

30.1.2 Programming model

PETSc, being based on MPI, uses the SPMD programming model (section 2.1), where all processes execute the same executable. Even more than in regular MPI codes, this makes sense here, since most PETSc objects are collectively created on some communicator, often `MPI_COMM_WORLD`. With the object-oriented design (section 30.1.3) this means that a PETSc program almost looks like a sequential program.

```
MatMult(A,x,y);      // y <- Ax
VecCopy(y,res);     // r <- y
VecAXPY(res,-1.,b); // r <- r - b
```

This is sometimes called *sequential semantics*.

30.1.3 Design philosophy

PETSc has an object-oriented design, even though it is written in C. There are classes of objects, such as **Mat** for matrices and **Vec** for Vectors, but there is also the **KSP** (for "Krylov SSpace solver") class of linear system solvers, and **PetscViewer** for outputting matrices and vectors to screen or file.

Part of the object-oriented design is the polymorphism of objects: after you have created a **Mat** matrix as sparse or dense, all methods such as **MatMult** (for the matrix-vector product) take the same arguments: the matrix, and an input and output vector.

This design where the programmer manipulates a ‘handle’ also means that the internal of the object, the actual storage of the elements, is hidden from the programmer. This hiding goes so far that even filling in elements is not done directly but through function calls:

```
VecSetValue(i,j,v,mode)
MatSetValue(i,j,v,mode)
MatSetValues(ni,is,nj,js,v,mode)
```

30.1.4 Language support

30.1.4.1 C/C++

PETSc is implemented in C, so there is a natural interface to C. There is no separate C++ interface.

30.1.4.2 Fortran

A *Fortran90* interface exists. The *Fortran77* interface is only of interest for historical reasons.

To use Fortran, include both a module and a cpp header file:

```
#include "petsc/finclude/petscXXX.h"
use petscXXX
```

(here XXX stands for one of the PETSc types, but including **petsc.h** and using **use petsc** gives inclusion of the whole library.)

Variables can be declared with their type (**Vec**, **Mat**, **KSP** et cetera), but internally they are Fortran *Type* objects so they can be declared as such.

Example:

```
#include "petsc/finclude/petscvec.h"
use petscvec
Vec b
type(tVec) x
```

The output arguments of many query routines are optional in PETSc. While in C a generic **NULL** can be passed, Fortran has type-specific nulls, such as **PETSC_NULL_INTEGER**, **PETSC_NULL_OBJECT**.

30.1.4.3 Python

A *python* interface was written by *Lisandro Dalcin*. It can be added to PETSc at installation time; section 30.3.

This book discusses the Python interface in short remarks in the appropriate sections.

30.1.5 Documentation

PETSc comes with a manual in pdf form and web pages with the documentation for every routine. The starting point is the web page <https://petsc.org/release/documentation/>.

There is also a mailing list with excellent support for questions and bug reports.

TACC note. For questions specific to using PETSc on TACC resources, submit tickets to the *TACC* or *XSEDE portal*.

30.2 Basics of running a PETSc program

30.2.1 Compilation

A PETSc compilation needs a number of include and library paths, probably too many to specify interactively. The easiest solution is to create a makefile and load the standard variables and compilation rules. (You can use `$PETSC_DIR/share/petsc/Makefile.user` for inspiration.)

Throughout, we will assume that variables `PETSC_DIR` and `PETSC_ARCH` have been set. These depend on your local installation; see section 30.3.

In the easiest setup, you leave the compilation to PETSc and your make rules only do the link step, using *CLINKER* or *FLINKER* for C/Fortran respectively:

```
include ${PETSC_DIR}/lib/petsc/conf/variables
include ${PETSC_DIR}/lib/petsc/conf/rules
program : program.o
    ${CLINKER} -o $@ $^ ${PETSC_LIB}
```

The two include lines provide the compilation rule and the library variable.

You can use these rules:

```
% : %.F90
    $(LINK.F) -o $@ $^ $(LDLIBS)
%.o: %.F90
    $(COMPILE.F) $(OUTPUT_OPTION) $<
% : %.cxx
    $(LINK.cc) -o $@ $^ $(LDLIBS)
%.o: %.cxx
    $(COMPILE.cc) $(OUTPUT_OPTION) $<
```

```
## example link rule:
# app : a.o b.o c.o
#     $(LINK.F) -o $@ $^ $(LDLIBS)
```

(The `PETSC_CC_INCLUDES` variable contains all paths for compilation of C programs; correspondingly there is `PETSC_FC_INCLUDES` for Fortran source.)

If don't want to include those configuration files, you can find out the include options by:

```
cd $PETSC_DIR
make getincludedirs
make getlinklibs
and copying the results into your compilation script.
```

There is an example makefile `$PETSC_DIR/share/petsc/Makefile.user` you can take for inspiration. Invoked without arguments it prints out the relevant variables:

```
[c:246] make -f ! $PETSC_DIR/share/petsc/Makefile.user
CC=/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/bin/mpicc
CXX=/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/bin/mpicxx
FC=/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/bin/mpif90
CFLAGS=-Wall -Wwrite-strings -Wno-strict-aliasing -Wno-unknown-pragmas -fstack-protector -Qunused-arguments
CXXFLAGS=-Wall -Wwrite-strings -Wno-strict-aliasing -Wno-unknown-pragmas -fstack-protector -fvisibility=hidden
FFLAGS=-m64 -g
CPPFLAGS=-I/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/include -I/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/include
LDFLAGS=-L/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/lib -Wl,-rpath,/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/lib
LDLIBS=-lpetsc -lm
```

TACC note. On TACC clusters, a petsc installation is loaded by commands such as

```
module load petsc/3.16
```

Use `module avail petsc` to see what configurations exist. The basic versions are

```
# development
module load petsc/3.11-debug
# production
module load petsc/3.11
```

Other installations are real versus complex, or 64bit integers instead of the default 32. The command

```
module spider petsc
```

tells you all the available petsc versions. The listed modules have a naming convention such as `petsc/3.11-i64debug` where the 3.11 is the PETSc release (minor patches are not included in this version; TACC aims to install only the latest patch, but generally several versions are available), and `i64debug` describes the debug version of the installation with 64bit integers.

30.2.2 Running

PETSc programs use MPI for parallelism, so they are started like any other MPI program:

```
mpiexec -n 5 -machinefile mf \
    your_petsc_program option1 option2 option3
```

TACC note. On TACC clusters, use `ibrun`.

30.2.3 Initialization and finalization

PETSc has an call that initializes both PETSc and MPI, so normally you would replace `MPI_Init` by `PetscInitialize` (figure 30.1). Unlike with MPI, you do not want to use a NULL value for the `argc`, `argv` arguments, since PETSc makes extensive use of commandline options; see section 37.3.

```
// init.c
PetscCall( PetscInitialize
    (&argc,&argv,(char*)0,help) );
```

Figure 30.1 PetscInitialize

C:

```
PetscErrorCode PetscInitialize
    (int *argc,char ***args,const char file[],const char help[])
```

Input Parameters:

argc - count of number of command line arguments

args - the command line arguments

file - [optional] PETSc database file.

help - [optional] Help message to print, use NULL for no message

Fortran:

```
call PetscInitialize(file,ierr)
```

Input parameters:

ierr - error return code

file - [optional] PETSc database file,

use PETSC_NULL_CHARACTER to not check for code specific file.

```
int flag;
MPI_Initialized(&flag);
if (flag)
    printf("MPI was initialized by PETSc\n");
else
    printf("MPI not yet initialized\n");
```

There are two further arguments to **PetscInitialize**:

1. the name of an options database file; and
2. a help string, that is displayed if you run your program with the -h option.

Fortran note 28: Petsc Initialization. The Fortran version has no arguments for commandline options; however, you can pass a file of database options:

```
PetscInitialize(filename,ierr)
```

If none is specified, give **PETSC_NULL_CHARACTER** as argument.

For passing help information there is a variant that takes a help string:

Code:

```
!! mainhelp.F90
Character(len=50) :: help = "This program
demonstrates help info"
help = trim(help) // NEW_LINE('A')
call PetscInitialize(PETSC_NULL_CHARACTER,help,
ierr)
CHKERRA(ierr)
```

Output:

```
[examples/petsc/f] mainhelp:
This program demonstrates help info
```

If your main program is in C, but some of your PETSc calls are in Fortran files, it is necessary to call `PetscInitializeFortran` after `PetscInitialize`.

```
!! init.F90
call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
CHKERRA(ierr)
call MPI_Initialized(flag,ierr)
CHKERRA(ierr)
if (flag) then
  print *,"MPI was initialized by PETSc"
```

Python note 37: Init, and with commandline options. The following works if you don't need commandline options.

```
from petsc4py import PETSc
```

To pass commandline arguments to PETSc, do:

```
import sys
from petsc4py import init
init(sys.argv)
from petsc4py import PETSc
```

After initialization, you can use `MPI_COMM_WORLD` or `PETSC_COMM_WORLD` (which is created by `MPI_Comm_dup` and used internally by PETSc):

```
MPI_Comm comm = PETSC_COMM_WORLD;
MPI_Comm_rank(comm,&mytid);
MPI_Comm_size(comm,&ntids);
```

Python note 38: Communicator object.

```
comm = PETSc.COMM_WORLD
nprocs = comm.getSize(self)
procno = comm.getRank(self)
```

The corresponding call to replace `MPI_Finalize` is `PetscFinalize`. You can elegantly capture and return the error code by the idiom

```
return PetscFinalize();
```

at the end of your main program.

30.3 PETSc installation

PETSc has a large number of installation options. These can roughly be divided into:

1. Options to describe the environment in which PETSc is being installed, such as the names of the compilers or the location of the MPI library;
2. Options to specify the type of PETSc installation: real versus complex, 32 versus 64-bit integers, et cetera;
3. Options to specify additional packages to download.

For an existing installation, you can find the options used, and other aspects of the build history, in the `configure.log` / `make.log` files:

```
$PETSC_DIR/$PETSC_ARCH/lib/petsc/conf/configure.log  
$PETSC_DIR/$PETSC_ARCH/lib/petsc/conf/make.log
```

30.3.1 Versions

PETSc is up to version 3.18.x as of this writing. Older versions may miss certain routines, or display certain bugs. However, older versions may also contain routines and keywords that have subsequently been removed. PETSc version are not backwards compatible!

The version is stored in macros `PETSC_VERSION`, `PETSC_VERSION_MAJOR`, `PETSC_VERSION_MINOR`, `PETSC_VERSION_SUBMINOR`.

For testing, the following macros are defined: `PETSC_VERSION_EQ/LT/LE/GT/GE` Example:

```
// cudainit316.c  
#include <petsc.h>  
#if PETSC_VERSION_LT(3,17,0)  
#else  
#error This program uses APIs abandoned in 3.17  
#endif
```

30.3.2 Debug

For any set of options, you will typically make two installations: one with `-with-debugging=yes` and once no. See section [37.1.1](#) for more detail on the differences between debug and non-debug mode.

30.3.3 Environment options

Compilers, compiler options, MPI.

While it is possible to specify `-download_mpich`, this should only be done on machines that you are certain do not already have an MPI library, such as your personal laptop. Supercomputer clusters are likely to have an optimized MPI library, and letting PETSc download its own will lead to degraded performance.

30.3.4 Variants

- Scalars: the option `-with-scalar-type` has values `real`, `complex`; `-with-precision` has values `single`, `double`, `--float128`, `--fp16`.

30.4 External packages

PETSc can extend its functionality through external packages such as *mumps*, *Hypre*, *fftw*. These can be specified in two ways:

1. Referring to an installation already on your system:
`--with-hdf5-include=${TACC_HDF5_INC}
--with-hf5_lib=${TACC_HDF5_LIB}`
2. By letting petsc download and install them itself:
`--with-parmetis=1 --download-parmetis=1`

Python note 39: petsc4py interface. The Python interface (section 30.1.4.3) can be installed with the option

`--download-petsc4py=<no, yes, filename, url>`

This is easiest if your python already includes *mpi4py*; see section 1.5.4.

Remark There are two packages that PETSc is capable of downloading and install, but that you may want to avoid:

- *fblaslapack*: this gives you BLAS/LAPACK through the Fortran ‘reference implementation’. If you have an optimized version, such as Intel’s mkl available, this will give much higher performance.
- *mpich*: this installs a MPI implementation, which may be required for your laptop. However, supercomputer clusters will already have an MPI implementation that uses the high-speed network. PETSc’s downloaded version does not do that. Again, finding and using the already installed software may greatly improve your performance.

30.4.1 Slep̄c

Most external packages add functionality to the lower layers of Petsc. For instance, the *Hypre* package adds some preconditioners to Petsc’s repertoire (section 34.1.7.3), while *Mumps* (section 34.2) makes it possible to use the LU preconditioner in parallel.

On the other hand, there are packages that use Petsc as a lower level tool. In particular, the eigenvalue solver package *Slep̄c* [28] can be installed through the options

```
--download-slepc=<no, yes, filename, url>
    Download and install slepc  current: no
--download-slepc-commit=commitid
    The commit id from a git repository to use for the build of slepc  current: 0
--download-slepc-configure-arguments=string
    Additional configure arguments for the build of SLEPc
```

The slepc header files wind up in the same directory as the petsc headers, so no change to your compilation rules are needed. However, you need to add `-lslepc` to the link line.

Chapter 31

PETSc objects

31.1 Distributed objects

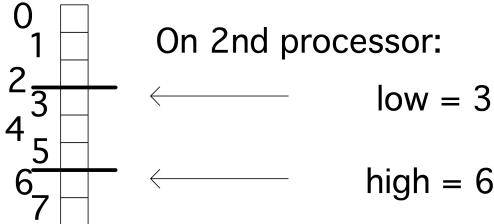
PETSc is based on the SPMD model, and all its objects act like they exist in parallel, spread out over all the processes. Therefore, prior to discussing specific objects in detail, we briefly discuss how PETSc treats distributed objects.

For a matrix or vector you need to specify the size. This can be done two ways:

- you specify the global size and PETSc distributes the object over the processes, or
- you specify on each process the local size

If you specify both the global size and the local sizes, PETSc will check for consistency.

For example, if you have a vector of N components, or a matrix of N rows, and you have P processes, each process will receive N/P components or rows if P divides evenly in N . If P does not divide evenly, the excess is spread over the processes.



The way the distribution is done is by contiguous blocks: with 10 processes and 1000 components in a vector, process 0 gets the range 0 … 99, process 1 gets 1 … 199, et cetera. This simple scheme suffices for many cases, but PETSc has facilities for more sophisticated load balancing.

31.1.1 Support for distributions

Once an object has been created and distributed, you do not need to remember the size or the distribution yourself: you can query these with calls such as `VecGetSize`, `VecGetLocalSize`.

The corresponding matrix routines `MatGetSize`, `MatGetLocalSize` give both information for the distributions in i and j direction, which can be independent. Since a matrix is distributed by rows, `MatGetOwnershipRange` only gives a row range.

Figure 31.1 `PetscSplitOwnership``PetscSplitOwnership`

Synopsis

```
#include "petscsys.h"
PetscErrorCode PetscSplitOwnership
  (MPI_Comm comm,PetscInt *n,PetscInt *N)

Collective (if n or N is PETSC_DECIDE)
```

Input Parameters

comm - MPI communicator that shares the object being divided
 n - local length (or PETSC_DECIDE to have it set)
 N - global length (or PETSC_DECIDE)

```
// split.c
N = 100; n = PETSC_DECIDE;
PetscSplitOwnership(comm,&n,&N);
PetscPrintf(comm,"Global %d, local %d\n",N,n);

N = PETSC_DECIDE; n = 10;
PetscSplitOwnership(comm,&n,&N);
PetscPrintf(comm,"Global %d, local %d\n",N,n);
```

While PETSc objects are implemented using local memory on each process, conceptually they act like global objects, with a global indexing scheme. Thus, each process can query which elements out of the global object are stored locally. For vectors, the relevant routine is `VecGetOwnershipRange`, which returns two parameters, `low` and `high`, respectively the first element index stored, and one-more-than-the-last index stored.

This gives the idiom:

```
VecGetOwnershipRange(myvector,&low,&high);
for (int myidx=low; myidx<high; myidx++)
  // do something at index myidx
```

These conversions between local and global size can also be done explicitly, using the `PetscSplitOwnership` (figure 31.1) routine. This routine takes two parameter, for the local and global size, and whichever one is initialized to `PETSC_DECIDE` gets computed from the other.

31.2 Scalars

Unlike programming languages that explicitly distinguish between single and double precision numbers, PETSc has only a single scalar type: `PetscScalar`. The precision of this is determined at installation time. In fact, a `PetscScalar` can even be a complex number if the installation specified that the scalar type is complex.

Even in applications that use complex numbers there can be quantities that are real: for instance, the norm of a complex vector is a real number. For that reason, PETSc also has the type `PetscReal`. There is also an explicit `PetscComplex`.

Furthermore, there is

```
#define PETSC_BINARY_INT_SIZE      (32/8)
#define PETSC_BINARY_FLOAT_SIZE    (32/8)
#define PETSC_BINARY_CHAR_SIZE     (8/8)
#define PETSC_BINARY_SHORT_SIZE   (16/8)
#define PETSC_BINARY_DOUBLE_SIZE  (64/8)
#define PETSC_BINARY_SCALAR_SIZE sizeof(PetscScalar)
```

31.2.1 Integers

Integers in PETSc are likewise of a size determined at installation time: `PetscInt` can be 32 or 64 bits. The latter possibility is useful for indexing into large vectors and matrices. Furthermore, there is a `PetscErrorCode` type for catching the return code of PETSc routines; see section 37.1.2.

For compatibility with other packages there are two more integer types:

- `PetscBLASInt` is the integer type used by the *Basic Linear Algebra Subprograms (BLAS) / Linear Algebra Package (LAPACK)* library. This is 32-bits if the `-download-blas-lapack` option is used, but it can be 64-bit if *MKL* is used. The routine `PetscBLASIntCast` casts a `PetscInt` to `PetscBLASInt`, or returns `PETSC_ERR_ARG_OUTOFRANGE` if it is too large.
- `PetscMPIInt` is the integer type of the MPI library, which is always 32-bits. The routine `PetscMPIIntCast` casts a `PetscInt` to `PetscMPIInt`, or returns `PETSC_ERR_ARG_OUTOFRANGE` if it is too large.

Many external packages do not support 64-bit integers.

31.2.2 Complex

Numbers of type `PetscComplex` have a precision matching `PetscReal`.

Form a complex number using `PETSC_i`:

```
PetscComplex x = 1.0 + 2.0 * PETSC_i;
```

The real and imaginary part can be extract with the functions `PetscRealPart` and `PetscImaginaryPart` which return a `PetscReal`.

There are also routines `VecRealPart` and `VecImaginaryPart` that replace a vector with its real or imaginary part respectively. Likewise `MatRealPart` and `MatImaginaryPart`.

31.2.3 MPI Scalars

For MPI calls, `MPIU_REAL` is the MPI type corresponding to the current `PetscReal`.

For MPI calls, `MPIU_SCALAR` is the MPI type corresponding to the current `PetscScalar`.

For MPI calls, `MPIU_COMPLEX` is the MPI type corresponding to the current `PetscComplex`.

31.2.4 Booleans

There is a `PetscBool` datatype with values `PETSC_TRUE` and `PETSC_FALSE`.

31.3 Vec: Vectors

Vectors are objects with a linear index. The elements of a vector are floating point numbers or complex numbers (see section 31.2), but not integers: for that see section 31.5.1.

Figure 31.2 `VecCreate`

```
C:
PetscErrorCode VecCreate(MPI_Comm comm,Vec *v);

F:
VecCreate( comm,v,ierr )
MPI_Comm :: comm
Vec      :: v
PetscErrorCode :: ierr

Python:
vec = PETSc.Vec()
vec.create()
# or:
vec = PETSc.Vec().create()
```

Figure 31.3 `VecDestroy`

Synopsis

```
#include "petscvec.h"
PetscErrorCode VecDestroy(Vec *v)
```

Collective on Vec

Input Parameters:

v -the vector

31.3.1 Vector construction

Constructing a vector takes a number of steps. First of all, the vector object needs to be created on a communicator with `VecCreate` (figure 31.2)

Python note 40: Vector creation. In python, `PETSc.Vec()` creates an object with null handle, so a subsequent `create()` call is needed. In C and Fortran, the vector type is a keyword; in Python it is a member of `PETSc.Vec.Type`.

```
## setvalues.py
comm = PETSc.COMM_WORLD
x = PETSc.Vec().create(comm=comm)
x.setType(PETSc.Vec.Type.MPI)
```

The corresponding routine `VecDestroy` (figure 31.3) deallocates data and zeros the pointer. (This and all other Destroy routines are collective because of underlying MPI technicalities.)

The vector type needs to be set with `VecSetType` (figure 31.4).

The most common vector types are:

- `VECSEQ` for sequential vectors, that is, living on a single process; This is typically created on the `MPI_COMM_SELF` or `PETSC_COMM_SELF` communicator.
- `VECMPI` for a vector distributed over the communicator. This is typically created on the `MPI_COMM_WORLD` or `PETSC_COMM_WORLD` communicator, or one derived from it.
- `VECSTANDARD` is `VECSEQ` when used on a single process, or `VECMPI` on multiple.

Figure 31.4 VecSetType

Synopsis:

```
#include "petscvec.h"
PetscErrorCode VecSetType(Vec vec, VecType method)
```

Collective on Vec

Input Parameters:

vec - The vector object
method - The name of the vector type

Options Database Key

```
-vec_type <type> -Sets the vector type; use -help for a list of available types
```

Figure 31.5 VecSetSizes

C:

```
#include "petscvec.h"
PetscErrorCode VecSetSizes(Vec v, PetscInt n, PetscInt N)
Collective on Vec
```

Input Parameters

v : the vector
n : the local size (or PETSC_DECIDE to have it set)
N : the global size (or PETSC_DECIDE)

Python:

```
PETSc.Vec.setSizes(self, size, bsize=None)
size is a tuple of local/global
```

You may wonder why these types exist: you could have just one type, which would be as parallel as possible. The reason is that in a parallel run you may occasionally have a separate linear system on each process, which would require a sequential vector (and matrix) on each process, not part of a larger linear system.

Once you have created one vector, you can make more like it by **VecDuplicate**,

```
VecDuplicate(Vec old,Vec *new);
```

or **VecDuplicateVecs**

```
VecDuplicateVecs(Vec old,PetscInt n,Vec **new);
```

for multiple vectors. For the latter, there is a joint destroy call **VecDestroyVecs**:

```
VecDestroyVecs(PetscInt n,Vec **vecs);
```

(which is different in Fortran).

31.3.2 Vector layout

Next in the creation process the vector size is set with **VecSetSizes** (figure 31.5). Since a vector is typically distributed, this involves the global size and the sizes on the processors. Setting both is redundant, so it is possible to specify one and let the other be computed by the library. This is indicated by setting it to **PETSC_DECIDE**.

Figure 31.6 VecGetSize**VecGetSize / VecGetLocalSize**

C:

```
#include "petscvec.h"
PetscErrorCode VecGetSize(Vec x,PetscInt *gsize)
PetscErrorCode VecGetLocalSize(Vec x,PetscInt *lsize)
```

Input Parameter
x - the vector

Output Parameters
gsize - the global length of the vector
lsize - the local length of the vector

Python:

```
PETSc.Vec.getLocalSize(self)
PETSc.Vec.getSize(self)
PETSc.Vec.getSizes(self)
```

Figure 31.7 VecGetOwnershipRange

```
#include "petscvec.h"
PetscErrorCode VecGetOwnershipRange(Vec x,PetscInt *low,PetscInt *high)
```

Input parameter:
x - the vector

Output parameters:
low - the first local element, pass in NULL if not interested
high - one more than the last local element, pass in NULL if not interested

Fortran note:
use PETSC_NULL_INTEGER for NULL.

Python note 41: Vector size. Use `PETSc.DECIDE` for the parameter not specified:

```
x.setSizes([2,PETSc.DECIDE])
```

The size is queried with `VecGetSize` (figure 31.6) for the global size and `VecGetLocalSize` (figure 31.6) for the local size.

Each processor gets a contiguous part of the vector. Use `VecGetOwnershipRange` (figure 31.7) to query the first index on this process, and the first one of the next process.

In general it is best to let PETSc take care of memory management of matrix and vector objects, including allocating and freeing the memory. However, in cases where PETSc interfaces to other applications it maybe desirable to create a `Vec` object from an already allocated array: `VecCreateSeqWithArray` and `VecCreateMPIWithArray`.

```
VecCreateSeqWithArray
  (MPI_Comm comm,PetscInt bs,
   PetscInt n,PetscScalar *array,Vec *V);
VecCreateMPIWithArray
```

Figure 31.8 **VecAXPY**

Synopsis:
`#include "petscvec.h"`
`PetscErrorCode VecAXPY(Vec y,PetscScalar alpha,Vec x)`

Not collective on Vec

Input Parameters:
`alpha` - the scalar
`x, y` - the vectors

Output Parameter:
`y` - output vector

Figure 31.9 **VecView**

C:
`#include "petscvec.h"`
`PetscErrorCode VecView(Vec vec,PetscViewer viewer)`

for ascii output use:
`PETSC_VIEWER_STDOUT_WORLD`

Python:
`PETSc.Vec.view(self, Viewer viewer=None)`

ascii output is default or use:
`PETSc.Viewer.STDOUT(type cls, comm=None)`

`(MPI_Comm comm,PetscInt bs,`
`PetscInt n,PetscInt N,PetscScalar *array,Vec *vv);`

As you will see in section 31.4.1, you can also create vectors based on the layout of a matrix, using **MatCreateVecs**.

31.3.3 Vector operations

There are many routines operating on vectors that you need to write scientific applications. Examples are: norms, vector addition (including BLAS-type ‘AXPY’ routines: **VecAXPY** (figure 31.8)), pointwise scaling, inner products. A large number of such operations are available in PETSc through single function calls to **VecXYZ** routines.

For debugging purposes, the **VecView** (figure 31.9) routine can be used to display vectors on screen as ascii output,

```
// fftsine.c
PetscCall( VecView(signal,PETSC_VIEWER_STDOUT_WORLD) );
PetscCall( MatMult(transform,signal,frequencies) );
PetscCall( VecScale(frequencies,1./Nglobal) );
PetscCall( VecView(frequencies,PETSC_VIEWER_STDOUT_WORLD) );
```

but the routine call also use more general **PetscViewer** objects, for instance to dump a vector to file.

Here are a couple of representative vector routines:

```
PetscReal lambda;
```

Figure 31.10 VecDot

Synopsis:
`#include "petscvec.h"`
`PetscErrorCode VecDot(Vec x,Vec y,PetscScalar *val)`

Collective on Vec

Input Parameters:
`x, y` - the vectors

Output Parameter:
`val` - the dot product

Figure 31.11 VecScale

Synopsis:
`#include "petscvec.h"`
`PetscErrorCode VecScale(Vec x, PetscScalar alpha)`

Not collective on Vec

Input Parameters:
`x` - the vector
`alpha` - the scalar

Output Parameter:
`x` - the scaled vector

```
ierr = VecNorm(y,NORM_2,&lambda); CHKERRQ(ierr);
ierr = VecScale(y,1./lambda); CHKERRQ(ierr);
```

Exercise 31.1. Create a vector where the values are a single sine wave. using `VecGetSize`, `VecGetLocalSize`, `VecGetOwnershipRange`. Quick visual inspection:

```
iBrun vec -n 12 -vec_view
(There is a skeleton for this exercise under the name vec.)
```

Exercise 31.2. Use the routines `VecDot` (figure 31.10), `VecScale` (figure 31.11) and `VecNorm` (figure 31.12) to compute the inner product of vectors `x, y`, scale the vector `x`, and check its norm:

$$\begin{aligned} p &\leftarrow x^t y \\ x &\leftarrow x/p \\ n &\leftarrow \|x\|_2 \end{aligned}$$

Python note 42: Vector operations. The plus operator is overloaded so that

`x+y`

is defined.

```
x.sum() # max,min,....
x.dot(y)
x.norm(PETSc.NormType.NORM_INFINITY)
```

Figure 31.12 **VecNorm**

```
C:
#include "petscvec.h"
PetscErrorCode VecNorm(Vec x,NormType type,PetscReal *val)
where type is
    NORM_1, NORM_2, NORM_FROBENIUS, NORM_INFINITY

Python:
PETSc.Vec.norm(self, norm_type=None)

where norm is variable in PETSc.NormType:
    NORM_1, NORM_2, NORM_FROBENIUS, NORM_INFINITY or
    N1, N2, FRB, INF
```

Figure 31.13 **VecSetValue**

Synopsis

```
#include <petscvec.h>
PetscErrorCode VecSetValue
  (Vec v,PetscInt row,PetscScalar value,InsertMode mode);
```

Not Collective

Input Parameters

v- the vector
row- the row location of the entry
value- the value to insert
mode- either INSERT_VALUES or ADD_VALUES

31.3.3.1 Split collectives

MPI is capable (in principle) of ‘overlapping computation and communication’, or *latency hiding*. PETSc supports this by splitting norms and inner products into two phases.

- Start inner product / norm with **VecDotBegin** / **VecNormBegin**;
- Conclude inner product / norm with **VecDotEnd** / **VecNormEnd**;

Even if you achieve no overlap, it is possible to use these calls to combine a number of ‘collectives’: do the Begin calls of one inner product and one norm; then do (in the same sequence) the End calls. This means that only a single reduction is performed on a two-word package, rather than two separate reductions on a single word.

31.3.4 Vector elements

Setting elements of a traditional array is simple. Setting elements of a distributed array is harder. First of all, **VecSet** sets the vector to a constant value:

```
ierr = VecSet(x,1.); CHKERRQ(ierr);
```

In the general case, setting elements in a PETSc vector is done through a function **VecSetValue** (figure 31.13) for setting elements that uses global numbering; any process can set any elements in the vector. There is also a routine **VecSetValues** (figure 31.14) for setting multiple elements. This is mostly useful for setting dense subblocks of a block matrix.

We illustrate both routines by setting a single element with **VecSetValue**, and two elements with **VecSetValues**. In the latter case we need an array of length two for both the indices and values. The indices need not be successive.

Figure 31.14 VecSetValues

Synopsis

```
#include "petscvec.h"
PetscErrorCode VecSetValues
  (Vec x,PetscInt ni,const PetscInt
   ix[],const PetscScalar y[],InsertMode iora)

Not Collective

Input Parameters:
x - vector to insert in
ni - number of elements to add
ix - indices where to add
y - array of values
iora - either INSERT_VALUES or ADD_VALUES, where
       ADD_VALUES adds values to any existing entries, and
       INSERT_VALUES replaces existing entries with new values
```

Figure 31.15 VecAssemblyBegin

```
#include "petscvec.h"
PetscErrorCode VecAssemblyBegin(Vec vec)
PetscErrorCode VecAssemblyEnd(Vec vec)

Collective on Vec

Input Parameter
vec -the vector
```

```
i = 1; v = 3.14;
VecSetValue(x,i,v,INSERT_VALUES);
ii[0] = 1; ii[1] = 2; vv[0] = 2.7; vv[1] = 3.1;
VecSetValues(x,2,ii,vv,INSERT_VALUES);
```

Fortran note 29: Setting values. The value/values routines work the same way in Fortran. Note that despite type checking, using the ‘values’ routine and passing scalars, is allowed:

Python note 43: Setting vector values. Single element:

```
x.setValue(0,1.)
```

Multiple elements:

```
x.setValues( [2*procno,2*procno+1], [2.,3.] )
```

Using **VecSetValue** for specifying a local vector element corresponds to simple insertion in the local array. However, an element that belongs to another process needs to be transferred. This done in two calls: **VecAssemblyBegin** (figure 31.15) and **VecAssemblyEnd**.

```
if (myrank==0) then
  do vecidx=0,globalsize-1
    vecelt = vecidx
```

Figure 31.16 `VecGetArray`

```
// vecarray.c
PetscScalar const *in_array;
PetscScalar *out_array;
VecGetArrayRead(x,&in_array);
VecGetArray(y,&out_array);
PetscInt localsize;
VecGetLocalSize(x,&localsize);
for (int i=0; i<localsize; i++)
    out_array[i] = 2*in_array[i];
VecRestoreArrayRead(x,&in_array);
VecRestoreArray(y,&out_array);
```

```
call VecSetValue(vector,vecidx,vecelt,INSERT_VALUES,ierr)
end do
end if
call VecAssemblyBegin(vector,ierr)
call VecAssemblyEnd(vector,ierr)
```

(If you know the MPI library, you'll recognize that the first call corresponds to posting nonblocking send and receive calls; the second then contains the wait calls. Thus, the existence of these separate calls make *latency hiding* possible.)

```
VecAssemblyBegin(myvec);
// do work that does not need the vector myvec
VecAssemblyEnd(myvec);
```

Elements can either be inserted with `INSERT_VALUES`, or added with `ADD_VALUES` in the `VecSetValue / VecSetValues` call. You can not immediately mix these modes; to do so you need to call `VecAssemblyBegin / VecAssemblyEnd` in between add/insert phases.

31.3.4.1 Explicit element access

Since the vector routines cover a large repertoire of operations, you hardly ever need to access the actual elements. Should you still need those elements, you can use `VecGetArray` (figure 31.16) for general access or `VecGetArrayRead` (figure 31.16) for read-only.

PETSc insists that you properly release this pointer again with `VecRestoreArray` (figure 31.17) or `VecRestoreArrayRead` (figure 31.17).

In the following example, a vector is scaled through direct array access. Note the differing calls for the source and target vector, and note the `const` qualifier on the source array:

```
// vecarray.c
PetscScalar const *in_array;
PetscScalar *out_array;
VecGetArrayRead(x,&in_array);
VecGetArray(y,&out_array);
PetscInt localsize;
VecGetLocalSize(x,&localsize);
for (int i=0; i<localsize; i++)
    out_array[i] = 2*in_array[i];
VecRestoreArrayRead(x,&in_array);
VecRestoreArray(y,&out_array);
```

Figure 31.17 `VecRestoreArray`

C:

```
#include "petscvec.h"
PetscErrorCode VecRestoreArray(Vec x,PetscScalar **a)
```

Logically Collective on Vec

Input Parameters:
x- the vector
a- location of pointer to array obtained from `VecGetArray()`

Fortran90:

```
#include <petsc/finclude/petscvec.h>
use petscvec
VecRestoreArrayF90(Vec x,{Scalar, pointer :: xx_v(:)},integer ierr)
```

Input Parameters:
x- vector
xx_v- the Fortran90 pointer to the array

Figure 31.18 `VecPlaceArray`

Replace the storage of a vector by another array

Synopsis

```
#include "petscvec.h"
PetscErrorCode VecPlaceArray(Vec vec,const PetscScalar array[])
PetscErrorCode VecReplaceArray(Vec vec,const PetscScalar array[])
```

Input Parameters
vec - the vector
array - the array

This example also uses `VecGetLocalSize` to determine the size of the data accessed. Even running in a distributed context you can only get the array of local elements. Accessing the elements from another process requires explicit communication; see section 31.5.2.

There are some variants to the `VecGetArray` operation:

- `VecReplaceArray` (figure 31.18) frees the memory of the `Vec` object, and replaces it with a different array. That latter array needs to be allocated with `PetscMalloc`.
- `VecPlaceArray` (figure 31.18) also installs a new array in the vector, but it keeps the original array; this can be restored with `VecResetArray`.

Putting the array of one vector into another has a common application, where you have a distributed vector, but want to apply PETSc operations to its local section as if it were a sequential vector. In that case you would create a sequential vector, and `VecPlaceArray` the contents of the distributed vector into it.

Fortran note 30: F90 array access through pointer. There are routines such as `VecGetArrayF90` (with corresponding `VecRestoreArrayF90`) that return a (Fortran) pointer to a one-dimensional array.

```
!! vecset.F90
Vec           :: vector
PetscScalar,dimension(:),pointer :: elements
```

```
call VecGetArrayF90(vector,elements,ierr)
write (msg,10) myrank,elements(1)
10 format("First element on process",i3,":",f7.4,"\\n")
call PetscSynchronizedPrintf(comm,msg,ierr)
call PetscSynchronizedFlush(comm,PETSC_STDOUT,ierr)
call VecRestoreArrayF90(vector,elements,ierr)

!! vecarray.F90
PetscScalar,dimension(:),Pointer :: &
    in_array,out_array
call VecGetArrayReadF90( x,in_array,ierr )
call VecGetArrayF90( y,out_array,ierr )
call VecGetLocalSize( x,localsize,ierr )
do index=1,localsize
    out_array(index) = 2*in_array(index)
end do
call VecRestoreArrayReadF90( x,in_array,ierr )
call VecRestoreArrayF90( y,out_array,ierr )
```

Python note 44: Vector access.

```
x.getArray()
x.getValues(3)
x.getValues([1, 2])
```

31.3.5 File I/O

As mentioned above, `VecView` can be used for displaying a vector on the terminal screen. However, viewers are actually much more general. As explained in section 37.2.2, they can also be used to export vector data, for instance to file.

The converse operation, to load a vector that was exported in this manner, is `VecLoad`.

Since these operations are each other's inverses, usually you don't need to know the file format. But just in case:

```
PetscInt      VEC_FILE_CLASSID
PetscInt      number of rows
PetscScalar *values of all entries
```

That is, the file starts with a magic number, then the number of vector elements, and subsequently all scalar values.

31.4 Mat: Matrices

PETSc matrices come in a number of types, sparse and dense being the most important ones. Another possibility is to have the matrix in operation form, where only the action $y \leftarrow Ax$ is defined.

Figure 31.19 MatCreate

```
C:
PetscErrorCode MatCreate(MPI_Comm comm,Mat *v);

Python:
mat = PETSc.Mat()
mat.create()
# or:
mat = PETSc.Mat().create()
```

Figure 31.20 MatSetType

```
#include "petscmat.h"
PetscErrorCode MatSetType(Mat mat, MatType matype)

Collective on Mat

Input Parameters:
mat - the matrix object
matype - matrix type

Options Database Key
-mat_type <method> -Sets the type; use -help for a list of available methods (for instance, seqaij)
```

31.4.1 Matrix creation

Creating a matrix also starts by specifying a communicator on which the matrix lives collectively: `MatCreate` (figure 31.19)

Set the matrix type with `MatSetType` (figure 31.20). The main choices are between sequential versus distributed and dense versus sparse, giving types: `MATMPIDENSE`, `MATMPIAIJ`, `MATSEQDENSE`, `MATSEQAIJ`.

Distributed matrices are partitioned by block rows: each process stores a *block row*, that is, a contiguous set of matrix rows. It stores all elements in that block row. In order for a matrix-vector product to be executable, both the input and output vector need to be partitioned conforming to the matrix.

While for dense matrices the block row scheme is not scalable, for matrices from PDEs it makes sense. There, a subdivision by matrix blocks would lead to many empty blocks.

Just as with vectors, there is a local and global size; except that that now applies to rows and columns. Set sizes with `MatSetSizes` (figure 31.21) and subsequently query them with `MatSizes` (figure 31.22). The concept of local column size is tricky: since a process stores a full block row you may expect the local column size to be the full matrix size, but that is not true. The exact definition will be discussed later, but for square matrices it is a safe strategy to let the local row and column size to be equal.

Instead of querying a matrix size and creating vectors accordingly, the routine `MatCreateVecs` (figure 31.23) can be used. (Sometimes this is even required; see section 31.4.9.)

31.4.2 Nonzero structure

In case of a dense matrix, once you have specified the size and the number of MPI processes, it is simple to determine how much space PETSc needs to allocate for the matrix. For a sparse matrix this is more complicated, since the

Figure 31.21 MatSetSizes

C:

```
#include "petscmat.h"
PetscErrorCode MatSetSizes(Mat A,
    PetscInt m, PetscInt n, PetscInt M, PetscInt N)
```

Input Parameters

A : the matrix
m : number of local rows (or PETSC_DECIDE)
n : number of local columns (or PETSC_DECIDE)
M : number of global rows (or PETSC_DETERMINE)
N : number of global columns (or PETSC_DETERMINE)

Python:

```
PETSc.Mat.setSizes(self, size, bsize=None)
where 'size' is a tuple of 2 global sizes
or a tuple of 2 local/global pairs
```

Figure 31.22 MatSizes

C:

```
#include "petscmat.h"
PetscErrorCode MatGetSize(Mat mat,PetscInt *m,PetscInt *n)
PetscErrorCode MatGetLocalSize(Mat mat,PetscInt *m,PetscInt *n)
```

Python:

```
PETSc.Mat.getSize(self) # tuple of global sizes
PETSc.Mat.getLocalSize(self) # tuple of local sizes
PETSc.Mat.getSizes(self) # tuple of local/global size tuples
```

Figure 31.23 MatCreateVecs**Synopsis**

Get vector(s) compatible with the matrix, i.e. with the same parallel layout

```
#include "petscmat.h"
PetscErrorCode MatCreateVecs(Mat mat,Vec *right,Vec *left)
```

Collective on Mat

Input Parameter
mat - the matrix

Output Parameter;
right - (optional) vector that the matrix can be multiplied against
left - (optional) vector that the matrix vector product can be stored in

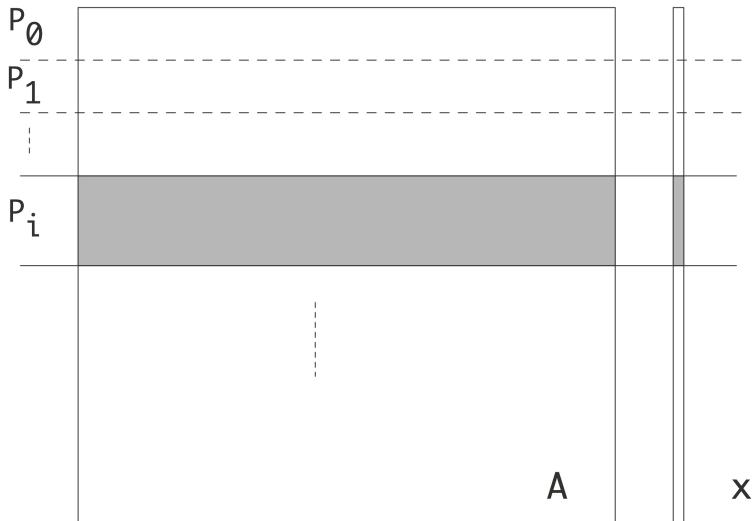


Figure 31.1: Matrix partitioning by block rows

matrix can be anywhere between completely empty and completely filled in. It would be possible to have a dynamic approach where, as elements are specified, the space grows; however, repeated allocations and re-allocations are inefficient. For this reason PETSc puts a small burden on the programmer: you need to specify a bound on how many elements the matrix will contain.

We explain this by looking at some cases. First we consider a matrix that only lives on a single process. You would then use `MatSeqAIJSetPreallocation` (figure 31.24). In the case of a tridiagonal matrix you would specify that each row has three elements:

```
MatSeqAIJSetPreallocation(A,3, PETSC_NULLPTR);
```

If the matrix is less regular you can use the third argument to give an array of explicit row lengths:

```
int *rowlengths;
// allocate, and then:
for (int row=0; row<nrows; row++)
    rowlengths[row] = // calculation of row length
MatSeqAIJSetPreallocation(A,PETSC_NULLPTR,rowlengths);
```

In case of a distributed matrix you need to specify this bound with respect to the block structure of the matrix. As illustrated in figure 31.2, a matrix has a diagonal part and an off-diagonal part. The diagonal part describes the matrix elements that couple elements of the input and output vector that live on this process. The off-diagonal part contains the matrix elements that are multiplied with elements not on this process, in order to compute elements that do live on this process.

The preallocation specification now has separate parameters for these diagonal and off-diagonal parts: with `MatMPIAIJSetPreallocation` (figure 31.24), you specify for both either a global upper bound on the number of nonzeros, or a detailed listing of row lengths. For the matrix of the *Laplace equation*, this specification would seem to be:

```
MatMPIAIJSetPreallocation(A, 3, PETSC_NULLPTR, 2, PETSC_NULLPTR);
```

Figure 31.24 MatSeqAIJSetPreallocation

```
#include "petscmat.h"
PetscErrorCode MatSeqAIJSetPreallocation
  (Mat B,PetscInt nz,const PetscInt nnz[])
PetscErrorCode MatMPIAIJSetPreallocation
  (Mat B,PetscInt d_nz,const PetscInt d_nnz[],
   PetscInt o_nz,const PetscInt o_nnz[])
```

Input Parameters

B - the matrix
nz/d_nz/o_nz - number of nonzeros per row in matrix or
diagonal/off-diagonal portion of local submatrix
nnz/d_nnz/o_nnz - array containing the number of nonzeros in the various rows of
the sequential matrix / diagonal / offdiagonal part of the local submatrix
or NULL (PETSC_NULL_INTEGER in Fortran) if nz/d_nz/o_nz is used.

Python:

```
PETSc.Mat.setPreallocationNNZ(self, [nnz_d,nnz_o] )
PETSc.Mat.setPreallocationCSR(self, csr)
PETSc.Mat.setPreallocationDense(self, array)
```

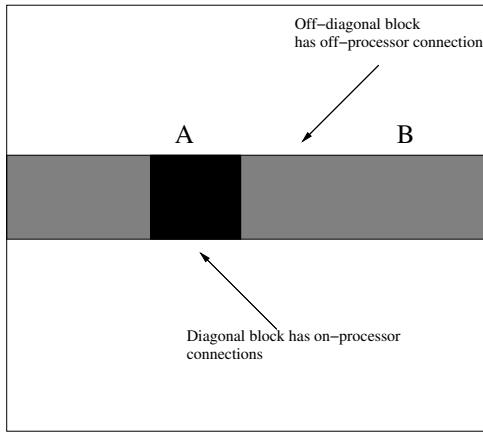


Figure 31.2: The diagonal and off-diagonal parts of a matrix

However, this is only correct if the block structure from the parallel division equals that from the lines in the domain. In general it may be necessary to use values that are an overestimate. It is then possible to contract the storage by copying the matrix.

Specifying bounds on the number of nonzeros is often enough, and not too wasteful. However, if many rows have fewer nonzeros than these bounds, a lot of space is wasted. In that case you can replace the *PETSC_NULLPTR* arguments by an array that lists for each row the number of nonzeros in that row.

31.4.3 Matrix elements

You can set a single matrix element with **MatSetValue** (figure 31.25) or a block of them, where you supply a set of *i* and *j* indices, using **MatSetValues**.

Figure 31.25 MatSetValue

C:

```
#include <petscmat.h>
PetscErrorCode MatSetValue(
    Mat m, PetscInt row, PetscInt col, PetscScalar value, InsertMode mode)
```

Input Parameters

- m : the matrix
- row : the row location of the entry
- col : the column location of the entry
- value : the value to insert
- mode : either INSERT_VALUES or ADD_VALUES

Python:

```
PETSc.Mat.setValue(self, row, col, value, addv=None)
also supported:
A[row,col] = value
```

Figure 31.26 MatAssemblyBegin

C:

```
#include "petscmat.h"
PetscErrorCode MatAssemblyBegin(Mat mat, MatAssemblyType type)
PetscErrorCode MatAssemblyEnd(Mat mat, MatAssemblyType type)
```

Input Parameters

- mat- the matrix
- type- type of assembly, either MAT_FLUSH_ASSEMBLY
or MAT_FINAL_ASSEMBLY

Python:

```
assemble(self, assembly=None)
assemblyBegin(self, assembly=None)
assemblyEnd(self, assembly=None)

there is a class PETSc.Mat.AssemblyType:
FINAL = FINAL_ASSEMBLY = 0
FLUSH = FLUSH_ASSEMBLY = 1
```

After setting matrix elements, the matrix needs to be assembled. This is where PETSc moves matrix elements to the right processor, if they were specified elsewhere. As with vectors this takes two calls: `MatAssemblyBegin` (figure 31.26) and `MatAssemblyEnd` (figure 31.26) which can be used to achieve *latency hiding*.

Elements can either be inserted (`INSERT_VALUES`) or added (`ADD_VALUES`). You can not immediately mix these modes; to do so you need to call `MatAssemblyBegin / MatAssemblyEnd` with a value of `MAT_FLUSH_ASSEMBLY`.

PETSc sparse matrices are very flexible: you can create them empty and then start adding elements. However, this is very inefficient in execution since the OS needs to reallocate the matrix every time it grows a little. Therefore, PETSc has calls for the user to indicate how many elements the matrix will ultimately contain.

```
MatSetOption(A, MAT_NEW_NONZERO_ALLOCATION_ERR, PETSC_FALSE)
```

Figure 31.27 MatGetRow

Synopsis:

```
#include "petscmat.h"
PetscErrorCode MatGetRow
  (Mat mat,PetscInt row,
   PetscInt *ncols,const PetscInt *cols[],const PetscScalar *vals[])
PetscErrorCode MatRestoreRow
  (Mat mat,PetscInt row,
   PetscInt *ncols,const PetscInt *cols[],const PetscScalar *vals[])
```

Input Parameters:

mat - the matrix
row - the row to get

Output Parameters

ncols - if not NULL, the number of nonzeros in the row
cols - if not NULL, the column numbers
vals - if not NULL, the values

31.4.3.1 Element access

If you absolutely need access to the matrix elements, there are routines such as **MatGetRow** (figure 31.27). With this, any process can request, using global row numbering, the contents of a row that it owns. (Requesting elements that are not local requires the different mechanism of taking submatrices; section 31.4.6.)

Since PETSc is geared towards *sparse matrices*, this returns not only the element values, but also the column numbers, as well as the mere number of stored columns. If any of these three return values are not needed, they can be unrequested by setting the parameter passed to *PETSC_NULLPTR*.

PETSc insists that you properly release the row again with **MatRestoreRow** (figure 31.27).

It is also possible to retrieve the full Compressed Row Storage (CRS) contents of the local matrix with **MatDenseGetArray**, **MatDenseRestoreArray**, **MatSeqAIJGetArray**, **MatSeqAIJRestoreArray**. (Routines **MatGetArray** / **MatRestoreArray** are deprecated.)

31.4.4 Matrix viewers

Matrices can be ‘viewed’ (see section 37.2.2 for a discussion of the **PetscViewer** mechanism) in a variety of ways, starting with the **MatView** call. However, often it is more convenient to use online options such as

```
yourprogram -mat_view
yourprogram -mat_view draw
yourprogram -ksp_mat_view draw
```

where **-mat_view** is activated by the assembly routine, while **-ksp_mat_view** shows only the matrix used as operator for a **KSP** object. Without further option refinements this will display the matrix elements inside the sparsity pattern. Using a sub-option **draw** will cause the sparsity pattern to be displayed in an *X11* window.

Figure 31.28 MatMult

Synopsis

```
#include "petscmat.h"
PetscErrorCode MatMult(Mat mat,Vec x,Vec y)
PetscErrorCode MatMultTranspose(Mat mat,Vec x,Vec y)
```

Neighbor-wise Collective on Mat

Input Parameters

mat - the matrix
x - the vector to be multiplied

Output Parameters

y - the result

Figure 31.29 MatMultAdd

Synopsis

```
#include "petscmat.h"
PetscErrorCode MatMultAdd(Mat mat,Vec x,Vec y,Vec z)
```

Neighbor-wise Collective on Mat

Input Parameters

mat - the matrix
x, y - the vectors

Output Parameters

z -the result

Notes

The vectors x and z cannot be the same.

31.4.5 Matrix operations**31.4.5.1 Matrix-vector operations**

In the typical application of PETSc, solving large sparse linear systems of equations with iterative methods, matrix-vector operations are most important. Foremost there is the matrix-vector product **MatMult** (figure 31.28) and the transpose product **MatMultTranspose** (figure 31.28). (In the complex case, the transpose product is not the Hermitian matrix product; for that use **MatMultHermitianTranspose**.)

For the BLAS *gemv* semantics $y \leftarrow \alpha Ax + \beta y$, **MatMultAdd** (figure 31.29) computes $z \leftarrow Ax + y$.

31.4.5.2 Matrix-matrix operations

There is a number of matrix-matrix routines such as **MatMatMult**.

31.4.6 Submatrices

Given a parallel matrix, there are two routines for extracting submatrices:

- **MatCreateSubMatrix** creates a single parallel submatrix.
- **MatCreateSubMatrices** creates a sequential submatrix on each process.

Figure 31.30 `MatCreateShell`

```
#include "petscmat.h"
PetscErrorCode MatCreateShell
  (MPI_Comm comm,
   PetscInt m,PetscInt n,PetscInt M,PetscInt N,
   void *ctx,Mat *A)

Collective

Input Parameters:
comm- MPI communicator
m- number of local rows (must be given)
n- number of local columns (must be given)
M- number of global rows (may be PETSC_DETERMINE)
N- number of global columns (may be PETSC_DETERMINE)
ctx- pointer to data needed by the shell matrix routines

Output Parameter:
A -the matrix
```

Figure 31.31 `MatShellSetOperation`

```
#include "petscmat.h"
PetscErrorCode MatShellSetOperation
  (Mat mat,MatOperation op,void (*g)(void))

Logically Collective on Mat

Input Parameters:
mat- the shell matrix
op- the name of the operation
g- the function that provides the operation.
```

31.4.7 Shell matrices

In many scientific applications, a matrix stands for some operator, and we are not intrinsically interested in the matrix elements, but only in the action of the matrix on a vector. In fact, under certain circumstances it is more convenient to implement a routine that computes the matrix action than to construct the matrix explicitly.

Maybe surprisingly, solving a linear system of equations can be handled this way. The reason is that PETSc's iterative solvers (section 34.1) only need the matrix-times-vector (and perhaps the matrix-transpose-times-vector) product.

PETSc supports this mode of working. The routine `MatCreateShell` (figure 31.30) declares the argument to be a matrix given in operator form.

31.4.7.1 Shell operations

The next step is then to add the custom multiplication routine, which will be invoked by `MatMult`: `MatShellSetOperation` (figure 31.31)

The routine that implements the actual product should have the same signature as `MatMult`, accepting a matrix and two vectors. The key to realizing your own product routine lies in the 'context' argument to the create routine. With `MatShellSetContext` (figure 31.32) you pass a pointer to some structure that contains all contextual information you

Figure 31.32 MatShellSetContext

Synopsis

```
#include "petscmat.h"
PetscErrorCode MatShellSetContext(Mat mat,void *ctx)
```

Input Parameters

mat - the shell matrix
ctx - the context

Figure 31.33 MatShellGetContext

```
#include "petscmat.h"
PetscErrorCode MatShellGetContext(Mat mat,void *ctx)
```

Not Collective

Input Parameter:

mat -the matrix, should have been created with MatCreateShell()

Output Parameter:

ctx -the user provided context

need. In your multiplication routine you then retrieve this with `MatShellGetContext` (figure 31.33).

What operation is specified is determined by a keyword `MATOP_<OP>` where `OP` is the name of the matrix routine, minus the `Mat` part, in all caps.

```
MatCreate(comm,&A);
MatSetSizes(A,localsize,localsize,matrix_size,matrix_size);
MatsetType(A,MATSHELL);
MatSetFromOptions(A);
MatShellSetOperation(A,MATOP_MULT,(void*)&mymatmult);
MatShellSetContext(A,(void*)Diag);
MatSetUp(A);
```

(The call to `MatSetSizes` needs to come before `MatsetType`.)

31.4.7.2 Shell context

Setting the context means passing a pointer (really: an address) to some allocated structure

```
struct matrix_data mystruct;
MatShellSetContext( A, &mystruct );
```

The routine signature has this argument as a `void*` but it's not necessary to cast it to that. Getting the context means that a pointer to your structure needs to be set

```
struct matrix_data *mystruct;
MatShellGetContext( A, &mystruct );
```

Somewhat confusingly, the Get routine also has a `void*` argument, even though it's really a pointer variable.

31.4.8 Multi-component matrices

For multi-component physics problems there are essentially two ways of storing the linear system

1. Grouping the physics equations together, or
2. grouping the domain nodes together.

In both cases this corresponds to a block matrix, but for a problem of N nodes and 3 equations, the respective structures are:

1. 3×3 blocks of size N , versus
2. $N \times N$ blocks of size 3.

The first case can be pictured as

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}$$

and while it looks natural, there is a computational problem with it. Preconditioners for such problems often look like

$$\begin{pmatrix} A_{00} & & \\ & A_{11} & \\ & & A_{22} \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} A_{00} & & \\ A_{10} & A_{11} & \\ A_{20} & A_{21} & A_{22} \end{pmatrix}$$

With the block-row partitioning of PETSc's matrices, this means at most a 50% efficiency for the preconditioner solve.

It is better to use the second scheme, which requires the *MATMPIBIJ* format, and use so-called *field-split preconditioners*; see section 34.1.7.3.5.

31.4.9 Fourier transform

The *Fast Fourier Transform (FFT)* can be considered a matrix-vector multiplication. PETSc supports this by letting you create a matrix with `MatCreateFFT`. This requires that you add an FFT library, such as *fftw*, at configuration time; see section 30.4.

FFT libraries may use padding, so vectors should be created with `MatCreateVecsFFT`, not with an independent `VecSetSizes`.

The *fftw* library does not scale the output vector, so a forward followed by a backward pass gives a result that is too large by the vector size.

```
// fftsine.c
PetscCall( VecView(signal,PETSC_VIEWER_STDOUT_WORLD) );
PetscCall( MatMult(transform,signal,frequencies) );
PetscCall( VecScale(frequencies,1./Nglobal) );
PetscCall( VecView(frequencies,PETSC_VIEWER_STDOUT_WORLD) );
```

One full cosine wave:

1.	-1. + 1.22465e-16 i
0.809017 + 0.587785 i	-0.809017 - 0.587785 i
0.309017 + 0.951057 i	-0.309017 - 0.951057 i
-0.309017 + 0.951057 i	0.309017 - 0.951057 i
-0.809017 + 0.587785 i	0.809017 - 0.587785 i

```
Frequency n = 1 amplitude ≡ 1:
-2.22045e-17 + 2.33487e-17 i
1. - 9.23587e-17 i
2.85226e-17 + 1.56772e-17 i
-4.44089e-17 + 1.75641e-17 i
                                         -3.35828e-19 + 3.26458e-18 i
                                         0. - 1.22465e-17 i
                                         -1.33873e-17 + 3.26458e-18 i
                                         -4.44089e-17 + 7.59366e-18 i
                                         7.40494e-18 + 1.56772e-17 i
                                         0. + 1.8215e-17 i
```

Strangely enough, the backward pass does not need to be scaled:

```
Vec confirm;
PetscCall( VecDuplicate(signal,&confirm) );
PetscCall( MatMultTranspose(transform,frequencies,confirm) );
PetscCall( VecAXPY(confirm,-1,signal) );
PetscReal nrm;
PetscCall( VecNorm(confirm,NORM_2,&nrm) );
PetscPrintf(MPI_COMM_WORLD,"FFT accuracy %e\n",nrm);
PetscCall( VecDestroy(&confirm) );
```

31.5 Index sets and Vector Scatters

In the PDE type of applications that PETSc was originally intended for, vector data can only be real or complex: there are no vector of integers. On the other hand, integers are used for indexing into vector, for instance for gathering boundary elements into a *halo region*, or for doing the *data transpose* of an *FFT* operation.

To support this, PETSc has the following object types:

- An **IS** object describes a set of integer indices;
- a **VecScatter** object describes the correspondence between a group of indices in an input vector and a group of indices in an output vector.

31.5.1 IS: index sets

An **IS** object contains a set of **PetscInt** values. It can be created with

- **ISCreate** for creating an empty set;
- **ISCreateStride** for a strided set;
- **ISCreateBlock** for a set of contiguous blocks, placed at an explicitly given list of starting indices.
- **ISCreateGeneral** for an explicitly given list of indices.

For example, to describe odd and even indices (on two processes):

```
// oddeven.c
IS oddeven;
if (procid==0) {
    PetscCall( ISCreateStride(comm,Nglobal/2,0,2,&oddeven) );
} else {
    PetscCall( ISCreateStride(comm,Nglobal/2,1,2,&oddeven) );
}
```

After this, there are various query and set operations on index sets.

You can read out the indices of a set by **ISGetIndices** and **ISRestoreIndices**.

Figure 31.34 VecScatterCreate**Synopsis**

Creates a vector scatter context. Collective on Vec

```
#include "petscvec.h"
PetscErrorCode VecScatterCreate(Vec xin,IS ix,Vec yin,IS iy,VecScatter *newctx)
```

Input Parameters:

xin : a vector that defines the layout of vectors from which we scatter

yin : a vector that defines the layout of vectors to which we scatter

ix : the indices of xin to scatter (if NULL scatters all values)

iy : the indices of yin to hold results (if NULL fills entire vector yin)

Output Parameter

newctx : location to store the new scatter context

31.5.2 VecScatter: all-to-all operations

A **VecScatter** object is a generalization of an all-to-all operation. However, unlike MPI **MPI_Alltoall**, which formulates everything in terms of local buffers, a **VecScatter** is more implicit in only describing indices in the input and output vectors.

The **VecScatterCreate** (figure 31.34) call has as arguments:

- An input vector. From this, the parallel layout is used; any vector being scattered from should have this same layout.
- An **IS** object describing what indices are being scattered; if the whole vector is rearranged, **PETSC_NULLPTR** (Fortran: **PETSC_NULL_IS**) can be given.
- An output vector. From this, the parallel layout is used; any vector being scattered into should have this same layout.
- An **IS** object describing what indices are being scattered into; if the whole vector is a target, **PETSC_NULLPTR** can be given.

As a simple example, the odd/even sets defined above can be used to move all components with even index to process zero, and the ones with odd index to process one:

```
VecScatter separate;
PetscCall( VecScatterCreate
           (in,oddeven,out,NULL,&separate) );
PetscCall( VecScatterBegin
           (separate,in,out,INSERT_VALUES,SCATTER_FORWARD) );
PetscCall( VecScatterEnd
           (separate,in,out,INSERT_VALUES,SCATTER_FORWARD) );
```

Note that the index set is applied to the input vector, since it describes the components to be moved. The output vector uses **PETSC_NULLPTR** since these components are placed in sequence.

Exercise 31.3. Modify this example so that the components are still separated odd/even, but now placed in descending order on each process.

Exercise 31.4. Can you extend this example so that process p receives all indices that are multiples of p ? Is your solution correct if n_{global} is not a multiple of n_{procs} ?

31.5.2.1 More VecScatter modes

There is an added complication, in that a `VecScatter` can have both sequential and parallel input or output vectors. Scattering onto process zero is also a popular option.

31.6 AO: Application Orderings

PETSc's decision to partition a matrix by contiguous block rows may be a limitation in the sense an application can have a natural ordering that is different. For such cases the `AO` type can translate between the two schemes.

31.7 Partitionings

By default, PETSc uses partitioning of matrices and vectors based on consecutive blocks of variables. In regular cases that is not a bad strategy. However, for some matrices a permutation and re-division can be advantageous. For instance, one could look at the *adjacency graph*, and minimize the number of *edge cuts* or the sum of the *edge weights*.

This functionality is not built into PETSc, but can be provided by *graph partitioning packages* such as *ParMetis* or *Zoltan*. The basic object is the `MatPartitioning`, with routines for

- Create and destroy: `MatPartitioningCreate`, `MatPartitioningDestroy`;
- Setting the type `MatPartitioningSetType` to an explicit partitioner, or something generated as the dual or a refinement of the current matrix;
- Apply with `MatPartitioningApply`, giving a distributed `IS` object, which can then be used in `MatCreateSubMatrix` to repartition.

Illustrative example:

```
MatPartitioning part;
MatPartitioningCreate(comm,&part);
MatPartitioningSetType(part,MATPARTITIONINGPARMETIS);
MatPartitioningApply(part,&is);
/* get new global number of each old global number */
ISPartitioningToNumbering(is,&isn);
ISBuildTwoSided(is,PETSC_NULLPTR,&isrows);
MatCreateSubMatrix(A,isrows,isrows,MAT_INITIAL_MATRIX,&perA);
```

Other scenario:

```
MatPartitioningSetAdjacency(part,A);
MatPartitioningSetType(part,MATPARTITIONINGHIERARCH);
MatPartitioningHierarchicalSetNcoarseparts(part,2);
MatPartitioningHierarchicalSetNfineparts(part,2);
```

Chapter 32

Grid support

PETSc's `DM` objects raise the abstraction level from the linear algebra problem to the physics problem: they allow for a more direct expression of operators in terms of their domain of definition. In this section we look at the `DMDA` 'distributed array' objects, which correspond to problems defined on Cartesian grids. Distributed arrays make it easier to construct the coefficient matrix of an operator that is defined as a *stencil* on a 1/2/3-dimensional *Cartesian grid*.

The main creation routine exists in three variants that mostly differ their number of parameters. For instance, `DMDACreate2d` has parameters along the x,y axes. However, `DMDACreate1d` has no parameter for the stencil type, since in 1D those are all the same, or for the process distribution.

32.1 Grid definition

A two-dimensional grid is created with `DMDACreate2d` (figure 32.1)

```
DMDACreate2d( communicator,
               x_boundary, y_boundary,
               stenciltypes,
               gridx, gridy, procx, proc当地, dof, width,
               partitionx, partitiony,
               grid);
```

- Boundary type is a value of type `DMBoundaryType`. Values are:
 - `DM_BOUNDARY_NONE`
 - `DM_BOUNDARY_GHOSTED`,
 - `DM_BOUNDARY_PERIODIC`,
- The stencil type is of type `DMStencilType`, with values
 - `DMDA_STENCIL_BOX`,
 - `DMDA_STENCIL_STAR`.(See figure 32.1)
- The `gridx, gridy` values are the global grid size. This can be set with commandline options `-da_grid_x/y/z`.
- The `procx, proc当地` variables are an explicit specification of the processor grid. Failing this specification, PETSc will try to find a distribution similar to the domain grid.
- `dof` indicates the number of 'degrees of freedom', where 1 corresponds to a scalar problem.
- `width` indicates the extent of the stencil: 1 for a 5-point stencil or more general a 2nd order stencil for 2nd order PDEs, 2 for 2nd order discretizations of a 4th order PDE, et cetera.

Figure 32.1 DMDACreate2d

```
#include "petscdmda.h"
PetscErrorCode DMDACreate2d(MPI_Comm comm,
                           DMBoundaryType bx,DMBoundaryType by,DMDAStencilType stencil_type,
                           PetscInt M,PetscInt N,PetscInt m,PetscInt n,PetscInt dof,
                           PetscInt s,const PetscInt lx[],const PetscInt ly[],
                           DM *da)
```

Input Parameters

comm - MPI communicator
 bx,by - type of ghost nodes: DM_BOUNDARY_NONE, DM_BOUNDARY_GHOSTED, DM_BOUNDARY_PERIODIC.
 stencil_type - stencil type: DMDA_STENCIL_BOX or DMDA_STENCIL_STAR.
 M,N - global dimension in each direction of
 m,n - corresponding number of processors in each dimension (or PETSC_DECIDE)
 dof - number of degrees of freedom per node
 s - stencil width
 lx, ly - arrays containing the number of
 nodes in each cell along the x and y coordinates, or NULL.

Output Parameter

da -the resulting distributed array object

- partitionx,partitiony are arrays giving explicit partitionings of the grid over the processors, or PETSC_NULLPTR for default distributions.

Code:

```
// dmrhs.c
DM grid;
PetscCall( DMDACreate2d
           ( comm,
             DM_BOUNDARY_NONE,DM_BOUNDARY_NONE,
             DMDA_STENCIL_STAR,
             100,100,
             PETSC_DECIDE,PETSC_DECIDE,
             1,
             1,
             NULL,NULL,
             &grid
           ) );
PetscCall( DMSetFromOptions(grid) );
PetscCall( DMSetUp(grid) );
PetscCall( DMViewFromOptions(grid,NULL,"-dm_view"
                           ) );
```

Output:

```
[examples/petsc/c] dmcreate:
ld: warning: dylib
  ↗(/Users/eijkhout/Installation/petsc/petsc-3.16.4
  ↗was built for newer macOS
  ↗version (11.5) than being
  ↗linked (11.0)
[0] Local = 0-50 x 0-50, halo =
  ↗0-51 x 0-51
[1] Local = 50-100 x 0-50, halo =
  ↗49-100 x 0-51
[2] Local = 0-50 x 50-100, halo =
  ↗0-51 x 49-100
[3] Local = 50-100 x 50-100, halo
  ↗= 49-100 x 49-100
```

After you define a **DM** object, each process has a contiguous subdomain out of the total grid. You can query its size and location with **DMDAGetCorners**, or query that and all other information with **DMDAGetLocalInfo** (figure 32.2), which returns an **DMDALocalInfo** (figure 32.3) structure.

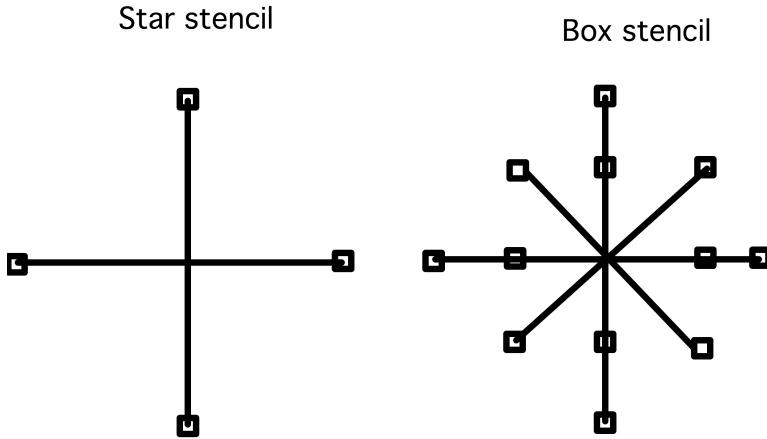


Figure 32.1: Star and box stencils

Figure 32.2 DMDAGetLocalInfo

```
#include "petscdmda.h"
PetscErrorCode DMDAGetLocalInfo(DM da,DMDALocalInfo *info)
```

(A `DMDALocalInfo` struct is the same for 1/2/3 dimensions, so certain fields may not be applicable to your specific PDE.)

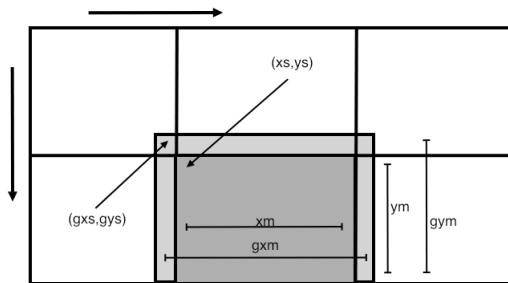


Figure 32.2: Illustration of various fields of the `DMDALocalInfo` structure

32.1.1 Associated vectors

Using the fields in this structure, each process can now iterate over its own subdomain. For instance, the ‘top left’ corner of the owned subdomain is at xs, ys and the number of points is xm, ym (see figure 32.2), so we can iterate over the subdomain as:

```
for (int j=info.ys; j<info.ys+info.ym; j++) {
    for (int i=info.xs; i<info.xs+info.xm; i++) {
        // actions on point i,j
    }
}
```

Figure 32.3 DMDALocalInfo

```
typedef struct {
    PetscInt      dim,dof,sw;
    PetscInt      mx,my,mz; /* global number of grid points in each direction */
    PetscInt      xs,ys,zs; /* starting point of this processor, excluding ghosts */
    PetscInt      xm,ym,zm; /* number of grid points on this processor, excluding ghosts */
    PetscInt      gxs,gys,gzs; /* starting point of this processor including ghosts */
    PetscInt      gxm,gym,gzm; /* number of grid points on this processor including ghosts */
    DMBoundaryType bx,by,bz; /* type of ghost nodes at boundary */
    DMDAStencilType st;
    DM            da;
} DMDALocalInfo;
```

Fortran Notes - This should be declared as

```
DMDALocalInfo :: info(DMDA_LOCAL_INFO_SIZE)
```

and the entries accessed via

```
info(DMDA_LOCAL_INFO_DIM)
info(DMDA_LOCAL_INFO_DOF) etc.
```

The entries `bx,by,bz, st, and da` are not accessible from Fortran.

On each point of the domain, we describe the stencil at that point. First of all, we now have the information to compute the x, y coordinates of the domain points:

```
PetscReal **xyarray;
PetscCall( DMDAVecGetArray(grid,xy,&xyarray) );
for (int j=info.ys; j<info.ys+info.ym; j++) {
    for (int i=info.xs; i<info.xs+info.xm; i++) {
        PetscReal x = i*hx, y = j*hy;
        xyarray[j][i] = x*y;
    }
}
PetscCall( DMDAVecRestoreArray(grid,xy,&xyarray) );
```

In some circumstances, we want to perform stencil operations on the vector of a `DMDA` grid. This requires having the *halo region*. Above, you already saw the `gxs, gxm` and other quantities relating to the halo of each process' subdomain.

What we need is a way to make vectors that contain these halo points.

- You can make a traditional vector corresponding to a grid with `DMCreateGlobalVector`; if you need this vector only for a short while, use `DMGetGlobalVector` and `DMRestoreGlobalVector`.
- You can make a vector including halo points with `DMCreateLocalVector`; if you need this vector only for a short while, use `DMGetLocalVector` and `DMRestoreLocalVector`.
- If you have a 'global' vector, you can make the corresponding 'local' vector, filling in its halo points, with `DMGlobalToLocal`; after operating on a local vector, you can copy its non-halo part back to a global vector with `DMLocalToGlobal`.

Here we set up a local vector for operations:

```
Vec ghostvector;
PetscCall( DMGetLocalVector(grid,&ghostvector) );
PetscCall( DMGlobalToLocal(grid,xy,INSERT_VALUES,ghostvector) );
PetscReal **xyarray,**gh;
PetscCall( DMADeleteArray(grid,xy,&xyarray) );
PetscCall( DMADeleteArray(grid,ghostvector,&gh) );
// computation on the arrays
PetscCall( DMADeleteRestoreArray(grid,xy,&xyarray) );
PetscCall( DMADeleteRestoreArray(grid,ghostvector,&gh) );
PetscCall( DMLocalToGlobal(grid,ghostvector,INSERT_VALUES,xy) );
PetscCall( DMRestoreLocalVector(grid,&ghostvector) );
```

The actual operations involve some tests for the actual presence of the halo:

```
for (int j=info.ys; j<info.ys+info.ym; j++) {
    for (int i=info.xs; i<info.xs+info.xm; i++) {
        if (info.gxs<info.xs && info.gys<info.ys)
            if (i-1>=info.gxs && i+1<=info.gxs+info.gxm &&
                j-1>=info.gys && j+1<=info.gys+info.gym )
                xyarray[j][i] =
                    ( gh[j-1][i] + gh[j][i-1] + gh[j][i+1] + gh[j+1][i] )
                    /4.;
```

32.1.2 Associated matrix

We construct a matrix on a **DMDA** by constructing a stencil on every (i,j) coordinate on a process:

```
for (int j=info.ys; j<info.ys+info.ym; j++) {
    for (int i=info.xs; i<info.xs+info.xm; i++) {
        PetscReal x = i*hx, y = j*hy;
        ...
        // set the row, col, v values
        ierr = MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES);CHKERRQ(ierr);
    }
}
```

Each matrix element row, col is a combination of two **MatStencil** objects. Technically, this is a **struct** with members i, j, k, s for the domain coordinates and the number of the field.

```
MatStencil row;
row.i = i; row.j = j;
```

We could construct the columns in this row one by one, but **MatSetValuesStencil** can set multiple rows or columns at a time, so we construct all columns at the same time:

```
MatStencil col[5];
PetscScalar v[5];
PetscInt ncols = 0;
***** diagonal element *****
col[ncols].i = i; col[ncols].j = j;
v[ncols++] = 4.;
***** off diagonal elements *****
....
```

The other ‘legs’ of the stencil need to be set conditionally: the connection to $(i - 1, j)$ is missing on the top row of the domain, and the connection to $(i, j - 1)$ is missing on the left column. In all:

```
// grid2d.c
for (int j=info.ys; j<info.ys+info.ym; j++) {
    for (int i=info.xs; i<info.xs+info.xm; i++) {
        MatStencil row,col[5];
        PetscScalar v[5];
        PetscInt ncols = 0;
        row.j      = j; row.i = i;
        /**** local connection: diagonal element ****/
        col[ncols].j = j; col[ncols].i = i; v[ncols++] = 4.;
        /* boundaries: top and bottom row */
        if (i>0) {col[ncols].j = j; col[ncols].i = i-1; v[ncols++] = -1.;}
        if (i<info.mx-1) {col[ncols].j = j; col[ncols].i = i+1; v[ncols++] = -1.;}
        /* boundary left and right */
        if (j>0) {col[ncols].j = j-1; col[ncols].i = i; v[ncols++] = -1.;}
        if (j<info.my-1) {col[ncols].j = j+1; col[ncols].i = i; v[ncols++] = -1.;}
        PetscCall( MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES) );
    }
}
```

32.2 Constructing a vector on a grid

A [DMDA](#) object is a description of a grid, so we now need to concern how to construct a linear system defined on that grid.

We start with vectors: we need a solution vector and a right-hand side. Here we have two options:

1. we can build a vector from scratch that has the right structure; or
2. we can use the fact that a grid object has a vector that can be extracted.

32.2.1 Create confirming vector

If we create a vector with [VecCreate](#) and [VecSetSizes](#), it is easy to get the global size right, but the default partitioning will probably not be conformal to the grid distribution. Also, getting the indexing scheme right is not trivial.

First of all, the local size needs to be set explicitly, using information from the [DMDALocalInfo](#) object:

```
Vec xy;
PetscCall( VecCreate(comm,&xy) );
PetscCall( VecSetType(xy,VECMPI) );
PetscInt nlocal = info.xm*info.ym, nglobal = info.mx*info.my;
PetscCall( VecSetSizes(xy,nlocal,nglobal) );
```

After this, you don’t use [VecSetValues](#), but set elements directly in the raw array, obtained by [DMDAVecGetArray](#):

```
PetscReal **xyarray;
PetscCall( DMDAVecGetArray(grid,xy,&xyarray) );
```

```
for (int j=info.ys; j<info.ys+info.ym; j++) {
    for (int i=info.xs; i<info.xs+info.xm; i++) {
        PetscReal x = i*hx, y = j*hy;
        xyarray[j][i] = x*y;
    }
}
PetscCall( DMDAVecRestoreArray(grid,xy,&xyarray) );
```

32.2.2 Extract vector from DMDA

32.2.3 Refinement

The routine `DMDASetRefinementFactor` can be activated with the options `-da_refine` or separately `-da_refine_x/y/z` for the directions.

32.3 Vectors of a distributed array

A distributed array is similar to a distributed vector, so there are routines of extracting the values of the array in the form of a vector. This can be done in two ways: of ways. (The routines here actually pertain to the more general `DM` ‘Data Management’ object, but we will for now discuss them in the context of `DMDA`.)

1. You can create a ‘global’ vector, defined on the same communicator as the array, and which is disjointly partitioned in the same manner. This is done with `DMCreateGlobalVector`:

```
PetscErrorCode DMCreateGlobalVector(DM dm,Vec *vec)
```

2. You can create a ‘local’ vector, which is sequential and defined on `PETSC_COMM_SELF`, that has not only the points local to the process, but also the ‘halo’ region with the extent specified in the definition of the `DMDACreate` call. For this, use `DMCreateLocalVector`:

```
PetscErrorCode DMCreateLocalVector(DM dm,Vec *vec)
```

Values can be moved between local and global vectors by:

- `DMGlobalToLocal`: this establishes a local vector, including ghost/halo points from a disjointly distributed global vector. (For overlapping communication and computation, use `DMGlobalToLocalBegin` and `DMGlobalToLocalEnd`.)
- `DMLocalToGlobal`: this copies the disjoint parts of a local vector back into a global vector. (For overlapping communication and computation use `DMLocalToGlobalBegin` and `DMLocalToGlobalEnd`.)

32.4 Matrices of a distributed array

Once you have a grid, can create its associated matrix:

```
DMSetUp(grid);
DMCreateMatrix(grid,&A)
```

With this subdomain information you can then start to create the coefficient matrix:

```
DM grid;
PetscInt i_first,j_first,i_local,j_local;
DMDAGetCorners(grid,&i_first,&j_first,NULL,&i_local,&j_local,NULL);
for ( PetscInt i_index=i_first; i_index<i_first+i_local; i_index++ ) {
    for ( PetscInt j_index=j_first; j_index<j_first+j_local; j_index++ ) {
        // construct coefficients for domain point (i_index,j_index)
    }
}
}
```

Note that indexing here is in terms of the grid, not in terms of the matrix.

For a simple example, consider 1-dimensional smoothing. From `DMDAGetCorners` we need only the parameters in *i*-direction:

```
// grid1d.c
PetscInt i_first,i_local;
PetscCall( DMDAGetCorners(grid,&i_first,NULL,NULL,&i_local,NULL,NULL) );
for (PetscInt i_index=i_first; i_index<i_first+i_local; i_index++) {
```

We then use a single loop to set elements for the local range in *i*-direction:

```
MatStencil row = {0},col[3] = {{0}};
PetscScalar v[3];
PetscInt ncols = 0;
row.i = i_index;
col[ncols].i = i_index; v[ncols] = 2.;
ncols++;
if (i_index>0) { col[ncols].i = i_index-1; v[ncols] = 1.; ncols++; }
if (i_index<i_global-1) { col[ncols].i = i_index+1; v[ncols] = 1.; ncols++; }
PetscCall( MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES) );
```

Chapter 33

Finite Elements support

33.1 General Data Management

```
ierr = DMCreate(PETSC_COMM_WORLD, &dm);
ierr = DMSetType(dm, DMPLEX);
```

A **DMPLEX** is by default two-dimensional. Use

```
plexprogram -dm_plex_dim k
```

for other dimensions. In two dimensions there are three levels of cells:

- 0-cells are vertices,
- 1-cells are edges, and
- 2-cells are triangles.

The default 2×2 grid has, sequentially:

Code:

```
ierr = DMSetFromOptions(dm);
ierr = PetscObjectSetName((PetscObject) dm, "Sphere");
ierr = DMViewFromOptions(dm, NULL, "-dm_view");
```

Output:

```
[code/petsc/c] sphereviewnp1:
mpiexec -n 1 plexsphere -dm_view
DM Object: Sphere 1 MPI processes
  type: plex
Sphere in 2 dimensions:
  Number of 0-cells per rank: 9
  Number of 1-cells per rank: 16
  Number of 2-cells per rank: 8
Labels:
  celltype: 3 strata with
    ↳ value/size (0 (9), 3 (8), 1
      ↳ (16))
  depth: 3 strata with value/size
    ↳ (0 (9), 1 (16), 2 (8))
  marker: 1 strata with value/size
    ↳ (1 (16))
Face Sets: 1 strata with
  ↳ value/size (1 (8))
```

and parallel:

```
Code:
 ierr = DMSetFromOptions(dm);
 ierr = PetscObjectSetName((PetscObject) dm, "Sphere");
 ierr = DMViewFromOptions(dm, NULL, "-dm_view");
```

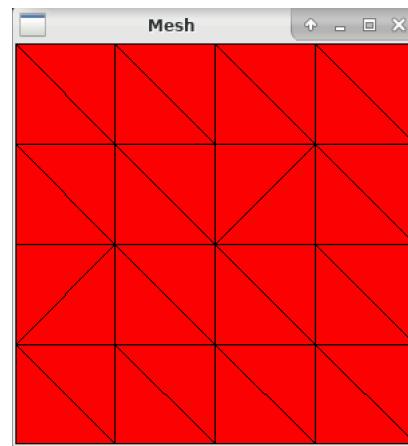
```
Output:
[code/petsc/c] sphereviewnp4:
mpiexec -n 4 plexsphere -dm_view
DM Object: Sphere 4 MPI processes
type: plex
Sphere in 2 dimensions:
Number of 0-cells per rank: 5 5
    ↪6 4
Number of 1-cells per rank: 6 6
    ↪6 5
Number of 2-cells per rank: 2 2
    ↪2 2
Labels:
depth: 3 strata with value/size
    ↪(0 (5), 1 (6), 2 (2))
celltype: 3 strata with
    ↪value/size (0 (5), 1 (6), 3
    ↪(2))
marker: 1 strata with value/size
    ↪(1 (7))
Face Sets: 1 strata with
    ↪value/size (1 (3))
```

For larger grids:

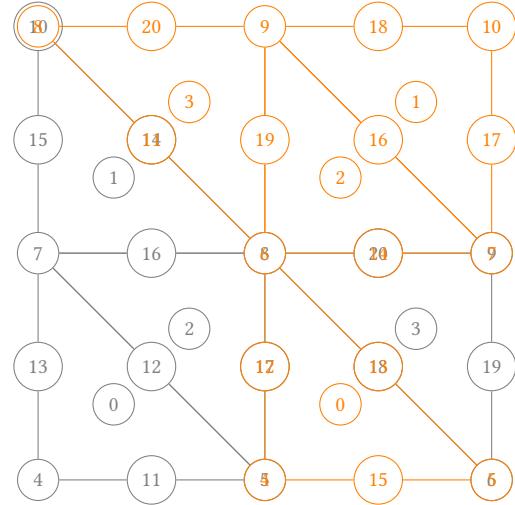
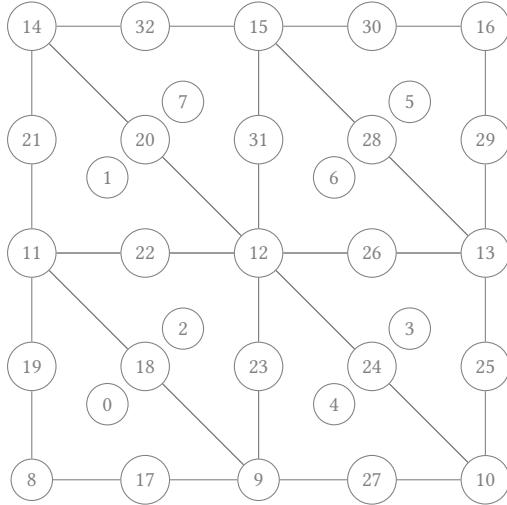
```
plexprogram -dm_plex_box_faces 4,4
```

Graphics output from

```
plexprogram -dm_view draw -draw_pause 20
```



```
plexprogram -dm_view :outputfile.tex:ascii_latex \
-dm_plex_view_scale 4
```



33.1.1 Matrix from dmplex

Loop over batch of elements (e):

Loop over element matrix entries (f,fc,g,gc -> i,j):

Loop over quadrature points (q):

Make u_q and gradU_q (loops over fields,Nb,Ncomp)

```
elemMat[i,j] += ψffc(q)gfc,gc(0)(u, ∇u)φggc(q)
+ψffc(q) · g(1)wfc,gc,dg(u, ∇u)∇φggc(q)
+∇ψffc(q) · gfc,gc,df(2)(u, ∇u)φggc(q)
+∇ψffc(q) · gfc,gc,df,dg(3)(u, ∇u)∇φggc(q)
```

```
// plexsphere.c
 ierr = DM PlexGetDepthStratum(dm, 0, &vStart, &vEnd);
 ierr = PetscSectionCreate(PetscObjectComm((PetscObject) dm), &s);
 ierr = DMSetLocalSection(dm, s);
 ierr = PetscSectionDestroy(&s);

 ierr = PetscSectionSetNumFields(s, 1);
 ierr = PetscSectionSetFieldComponents(s, 0, 1);
 ierr = PetscSectionSetChart(s, vStart, vEnd);
 // printf("start-end: %d -- %d\n", vStart, vEnd);
 for (v = vStart; v < vEnd; ++v) {
    ierr = PetscSectionSetDof(s, v, 1);
    ierr = PetscSectionSetFieldDof(s, v, 0, 1);
 }
 ierr = PetscSectionSetUp(s);
```

Chapter 34

PETSc solvers

Probably the most important activity in PETSc is solving a linear system. This is done through a solver object: an object of the class `KSP`. (This stands for Krylov SPace solver.) The solution routine `KSPSolve` takes a matrix and a right-hand-side and gives a solution; however, before you can call this some amount of setup is needed.

There two very different ways of solving a linear system: through a direct method, essentially a variant of Gaussian elimination; or through an iterative method that makes successive approximations to the solution. In PETSc there are only iterative methods. We will show how to achieve direct methods later. The default linear system solver in PETSc is fully parallel, and will work on many linear systems, but there are many settings and customizations to tailor the solver to your specific problem.

34.1 KSP: linear system solvers

34.1.1 Math background

Many scientific applications boil down to the solution of a system of linear equations at some point:

$$?_x : Ax = b$$

The elementary textbook way of solving this is through an *LU factorization*, also known as *Gaussian elimination*:

$$LU \leftarrow A, \quad Lz = b, \quad Ux = z.$$

While PETSc has support for this, its basic design is geared towards so-called iterative solution methods. Instead of directly computing the solution to the system, they compute a sequence of approximations that, with luck, converges to the true solution:

```
while not converged  
   $x_{i+1} \leftarrow f(x_i)$ 
```

The interesting thing about iterative methods is that the iterative step only involves the *matrix-vector product*:

```
while not converged  
   $r_i = Ax_i - b$   
   $x_{i+1} \leftarrow f(r_i)$ 
```

This *residual* is also crucial in determining whether to stop the iteration: since we (clearly) can not measure the distance to the true solution, we use the size of the residual as a proxy measurement.

Figure 34.1 `KSPCreate`

C:

```
PetscErrorCode KSPCreate(MPI_Comm comm,KSP *v);
```

Python:

```
ksp = PETSc.KSP()
ksp.create()
# or:
ksp = PETSc.KSP().create()
```

The remaining point to know is that iterative methods feature a *preconditioner*. Mathematically this is equivalent to transforming the linear system to

$$M^{-1}Ax = M^{-1}b$$

so conceivably we could iterate on the transformed matrix and right-hand side. However, in practice we apply the preconditioner in each iteration:

```
while not converged
    ri = Axi - b
    zi = M-1ri
    xi+1 ← f(zi)
```

In this schematic presentation we have left the nature of the `f()` update function unspecified. Here, many possibilities exist; the primary choice here is of the iterative method type, such as ‘conjugate gradients’, ‘generalized minimum residual’, or ‘bi-conjugate gradients stabilized’. (We will go into direct solvers in section 34.2.)

Quantifying issues of convergence speed is difficult; see HPC book, section ??.

34.1.2 Solver objects

First we create a KSP object, which contains the coefficient matrix, and various parameters such as the desired accuracy, as well as method specific parameters: `KSPCreate` (figure 34.1).

After this, the basic scenario is:

```
Vec rhs,sol;
KSP solver;
KSPCreate(comm,&solver);
KSPSetOperators(solver,A,A);
KSPSetFromOptions(solver);
KSPSolve(solver,rhs,sol);
KSPDestroy(&solver);
```

using various default settings. The vectors and the matrix have to be conformly partitioned. The `KSPSetOperators` call takes two operators: one is the actual coefficient matrix, and the second the one that the preconditioner is derived from. In some cases it makes sense to specify a different matrix here. (You can retrieve the operators with `KSPGetOperators`.) The call `KSPSetFromOptions` can cover almost all of the settings discussed next.

KSP objects have many options to control them, so it is convenient to call `KSPView` (or use the commandline option `-ksp_view`) to get a listing of all the settings.

Figure 34.2 `KSPSetTolerances`

```
#include "petscksp.h"
PetscErrorCode KSPSetTolerances
(KSP ksp,PetscReal rtol,PetscReal abstol,PetscReal dtol,PetscInt maxits)
```

Logically Collective on `ksp`

Input Parameters:

`ksp`- the Krylov subspace context
`rtol`- the relative convergence tolerance, relative decrease in the
 (possibly preconditioned) residual norm
`abstol`- the absolute convergence tolerance absolute size of the
 (possibly preconditioned) residual norm
`dtol`- the divergence tolerance, amount (possibly preconditioned)
 residual norm can increase before `KSPConvergedDefault()` concludes that
 the method is diverging
`maxits`- maximum number of iterations to use

Options Database Keys

`-ksp_atol <abstol>`- Sets `abstol`
`-ksp_rtol <rtol>`- Sets `rtol`
`-ksp_divtol <dtol>`- Sets `dtol`
`-ksp_max_it <maxits>`- Sets `maxits`

34.1.3 Tolerances

Since neither solution nor solution speed is guaranteed, an iterative solver is subject to some tolerances:

- a relative tolerance for when the residual has been reduced enough;
- an absolute tolerance for when the residual is objectively small;
- a divergence tolerance that stops the iteration if the residual grows by too much; and
- a bound on the number of iterations, regardless any progress the process may still be making.

These tolerances are set with `KSPSetTolerances` (figure 34.2), or options `-ksp_atol`, `-ksp_rtol`, `-ksp_divtol`, `-ksp_max_it`. Specify to `PETSC_DEFAULT` to leave a value unaltered.

In the next section we will see how you can determine which of these tolerances caused the solver to stop.

34.1.4 Why did my solver stop? Did it work?

On return of the `KSPSolve` routine there is no guarantee that the system was successfully solved. Therefore, you need to invoke `KSPGetConvergedReason` (figure 34.3) to get a `KSPConvergedReason` parameter that indicates what state the solver stopped in:

- The iteration can have successfully converged; this corresponds to `reason > 0`;
- the iteration can have diverged, or otherwise failed: `reason < 0`;
- or the iteration may have stopped at the maximum number of iterations while still making progress; `reason = 0`.

For more detail, `KSPConvergedReasonView` (before version 3.14: `KSPReasonView`) can print out the reason in readable form; for instance

```
KSPConvergedReasonView(solver,PETSC_VIEWER_STDOUT_WORLD);
// before 3.14:
KSPReasonView(solver,PETSC_VIEWER_STDOUT_WORLD);
```

Figure 34.3 KSPGetConvergedReason

C:

```
PetscErrorCode KSPGetConvergedReason
  (KSP ksp,KSPConvergedReason *reason)
Not Collective
```

Input Parameter
ksp -the KSP context

Output Parameter
reason -negative value indicates diverged, positive value converged,
see KSPConvergedReason

Python:

```
r = KSP.getConvergedReason(self)
where r in PETSc.KSP.ConvergedReason
```

Figure 34.4 KSPSetType

```
#include "petscksp.h"
PetscErrorCode KSPSetType(KSP ksp, KSPTYPE type)

Logically Collective on ksp

Input Parameters:
ksp : the Krylov space context
type : a known method
```

(This can also be activated with the `-ksp_converged_reason` commandline option.)

In case of successful convergence, you can use `KSPGetIterationNumber` to report how many iterations were taken.

The following snippet analyzes the status of a `KSP` object that has stopped iterating:

```
// shellvector.c
PetscInt its; KSPConvergedReason reason;
Vec Res; PetscReal norm;
ierr = KSPGetConvergedReason(Solve,&reason);
ierr = KSPConvergedReasonView(Solve,PETSC_VIEWER_STDOUT_WORLD);
if (reason<0) {
  PetscPrintf(comm,"Failure to converge: reason=%d\n",reason);
} else {
  ierr = KSPGetIterationNumber(Solve,&its);
  PetscPrintf(comm,"Number of iterations: %d\n",its);
}
```

34.1.5 Choice of iterator

There are many iterative methods, and it may take a few function calls to fully specify them. The basic routine is `KSPSetType` (figure 34.4), or use the option `-ksp_type`.

Here are some values (the full list is in `petscksp.h`:

Figure 34.5 `KSPMatSolve`

```
PetscErrorCode KSPMatSolve(KSP ksp, Mat B, Mat X)
```

Input Parameters

ksp - iterative context

B - block of right-hand sides

Output Parameter

X - block of solutions

- **KSPCG**: only for symmetric positive definite systems. It has a cost of both work and storage that is constant in the number of iterations.
There are variants such as **KSPPIPECG** that are mathematically equivalent, but possibly higher performing at large scale.
- **KSPGMRES**: a minimization method that works for nonsymmetric and indefinite systems. However, to satisfy this theoretical property it needs to store the full residual history to orthogonalize each compute residual to, implying that storage is linear, and work quadratic, in the number of iterations. For this reason, GMRES is always used in a truncated variant, that regularly restarts the orthogonalization. The restart length can be set with the routine **KSPGMRESSetRestart** or the option `-ksp_gmres_restart`.
- **KSPBCGS**: a quasi-minimization method; uses less memory than GMRES.

Depending on the iterative method, there can be several routines to tune its workings. Especially if you're still experimenting with what method to choose, it may be more convenient to specify these choices through commandline options, rather than explicitly coded routines. In that case, a single call to **KSPSetFromOptions** is enough to incorporate those.

34.1.6 Multiple right-hand sides

For the case of multiple right-hand sides, use **KSPMatSolve** (figure 34.5).

34.1.7 Preconditioners

Another part of an iterative solver is the *preconditioner*. The mathematical background of this is given in section 34.1.1. The preconditioner acts to make the coefficient matrix better conditioned, which will improve the convergence speed; it can even be that without a suitable preconditioner a solver will not converge at all.

34.1.7.1 Background

The mathematical requirement that the preconditioner M satisfy $M \approx A$ can take two forms:

1. We form an explicit approximation to A^{-1} ; this is known as a *sparse approximate inverse*.
2. We form an operator M (often given in factored or other implicit) form, such that $M \approx A$, and solving a system $Mx = y$ for x can be done relatively quickly.

In deciding on a preconditioner, we now have to balance the following factors.

1. What is the cost of constructing the preconditioner? This should not be more than the gain in solution time of the iterative method.
2. What is the cost per iteration of applying the preconditioner? There is clearly no point in using a preconditioner that decreases the number of iterations by a certain amount, but increases the cost per iteration much more.

3. Many preconditioners have parameter settings that make these considerations even more complicated: low parameter values may give a preconditioner that is cheaply to apply but does not improve convergence much, while large parameter values make the application more costly but decrease the number of iterations.

34.1.7.2 Usage

Unlike most of the other PETSc object types, a `PC` object is typically not explicitly created. Instead, it is created as part of the `KSP` object, and can be retrieved from it.

```
PC prec;
KSPGetPC(solver,&prec);
PCSetType(prec,PCILU);
```

Beyond setting the type of the preconditioner, there are various type-specific routines for setting various parameters. Some of these can get quite tedious, and it is more convenient to set them through commandline options.

34.1.7.3 Types

Method	PCType	Options Database Name
Jacobi	PCJACOBI	jacobi
Block Jacobi	PCBJACOBI	bjacobi
SOR (and SSOR)	PCSOR	sor
SOR with Eisenstat trick	PCEISENSTAT	eisenstat
Incomplete Cholesky	PCICC	icc
Incomplete LU	PCILU	ilu
Additive Schwarz	PCASM	asm
Generalized Additive Schwarz	PCGASM	gasm
Algebraic Multigrid	PCGAMG	gamg
Balancing Domain Decomposition by Constraints Linear solver	PCBDDC	bddc
Use iterative method	PCKSP	ksp
Combination of preconditioners	PCCOMPOSITE	composite
LU	PCLU	lu
Cholesky	PCCHOLESKY	cholesky
No preconditioning	PCNONE	none
Shell for user-defined PC	PCSHELL	shell

Here are some of the available preconditioner types.

The *Hypre* package (which needs to be installed during configuration time) contains itself several preconditioners. In your code, you can set the preconditioner to `PCHYPRE`, and use `PCHYPRESetType` to one of: euclid, pilut, parasails, boomeramg, ams, ads. However, since these preconditioners themselves have options, it is usually more convenient to use commandline options:

```
-pc_type hypre -pc_hypre_type xxxx
```

34.1.7.3.1 Sparse approximate inverses The inverse of a sparse matrix (at least, those from PDEs) is typically dense. Therefore, we aim to construct a *sparse approximate inverse*.

PETSc offers two such preconditioners, both of which require an external package.

- **PCSPAI**. This is a preconditioner that can only be used in single-processor runs, or as local solver in a block preconditioner; section 34.1.7.3.3.
- As part of the **PCHYPRE** package, the parallel variant *parasails* is available.
`-pc_type hypre -pc_hypre_type parasails`

34.1.7.3.2 Incomplete factorizations The *LU* factorization of a matrix stemming from PDEs problems has several practical problems:

- It takes (considerably) more storage space than the coefficient matrix, and
- it correspondingly takes more time to apply.

For instance, for a three-dimensional PDE in N variables, the coefficient matrix can take storage space $7N$, while the *LU* factorization takes $O(N^{5/3})$.

For this reason, often incompletely *LU* factorizations are popular.

- PETSc has of itself a **PCILU** type, but this can only be used sequentially. This may sound like a limitation, but in parallel it can still be used as the subdomain solver in a block methods; section 34.1.7.3.3.
- As part of *Hypre*, *pilut* is a parallel ILU.

There are many options for the ILU type, such as **PCFactorSetLevels** (option `-pc_factor_levels`), which sets the number of levels of fill-in allowed.

34.1.7.3.3 Block methods Certain preconditioners seem almost intrinsically sequential. For instance, an ILU solution is sequential between the variables. There is a modest amount of parallelism, but that is hard to explore.

Taking a step back, one of the problems with parallel preconditioners lies in the cross-process connections in the matrix. If only those were not present, we could solve the linear system on each process independently. Well, since a preconditioner is an approximate solution to begin with, ignoring those connections only introduces an extra degree of approxomaticity.

There are two preconditioners that operate on this notion:

- **PCBJACOBI**: block Jacobi. Here each process solves locally the system consisting of the matrix coefficients that couple the local variables. In effect, each process solves an independent system on a subdomain. The next question is then what solver is used on the subdomains. Here any preconditioner can be used, in particular the ones that only existed in a sequential version. Specifying all this in code gets tedious, and it is usually easier to specify such a complicated solver through commandline options:

```
-pc_type jacobi -sub_ksp_type preonly \
    -sub_pc_type ilu -sub_pc_factor_levels 1
```

(Note that this also talks about a `sub_ksp`: the subdomain solver is in fact a **KSP** object. By setting its type to `preonly` we state that the solver should consist of solely applying its preconditioner.)

The block Jacobi preconditioner can asymptotically only speed up the system solution by a factor relating to the number of subdomains, but in practice it can be quite valuable.

- **PCASM**: additive Schwarz method. Here each process solves locally a slightly larger system, based on the local variables, and one (or a few) levels of connections to neighboring processes. In effect, the processes solve system on overlapping subdomains. This preconditioner can asymptotically reduce the number of iterations to $O(1)$, but that requires exact solutions on the subdomains, and in practice it may not happen anyway.

Figure 34.1 illustrates these preconditioners both in matrix and subdomain terms.

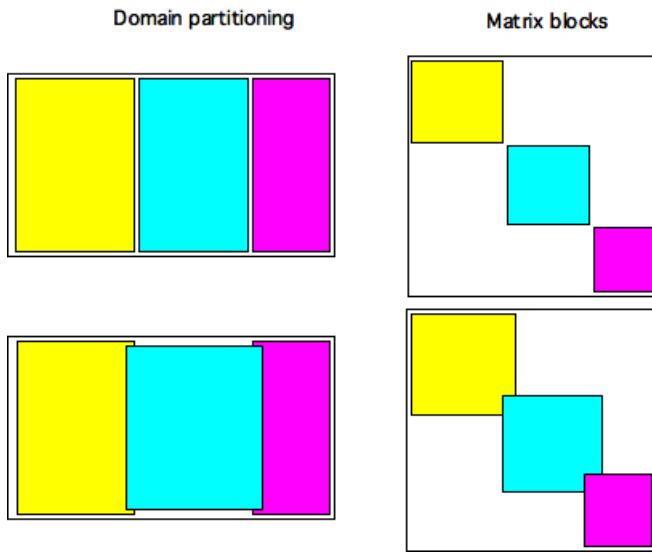


Figure 34.1: Illustration of block Jacobi and Additive Schwarz preconditioners: left domains and subdomains, right the corresponding submatrices

34.1.7.3.4 Multigrid preconditioners

- There is a Algebraic MultiGrid (AMG) type built into PETSc: [PCGAMG](#);
- the external packages *Hypre* and *ML* have AMG methods.
- There is a general Multigrid (MG) type: [PCM](#).

34.1.7.3.5 Field split preconditioners

For background refer to section [31.4.8](#).

Exercise 34.1. The example code `ksp.c` generates a five-point matrix, possibly nonsymmetric, on a unit square. Your assignment is to explore the convergence behavior of different solvers on linear systems with this coefficient matrix.

The example code takes two commandline arguments:

- `-n 123` set the domain size, meaning that the matrix size will be the square of that;
- `-unsymmetry .5` introduces a skew-symmetric component to the matrix.

Investigate the following:

- Some iterative methods, such as Conjugate Gradients (CG), are only mathematically defined for symmetric (and positive definite) matrices. How tolerant are iterative methods actually towards nonsymmetry?
- The number of iterations can sometimes be proved to depend on the *condition number* of the matrix, which is itself related to the size of the matrix. Can you find a relation between the matrix size and the number of iterations?
- A more sophisticated iterative methods (for instance, increasing the GMRES restart length) or a more sophisticated preconditioner (for instance using more fill levels in an ILU preconditioner), may lead to fewer iterations. (Does it, actually?) But it will not necessarily give a faster solution time, since each iteration is now more expensive.

See section [34.1.1](#) for the background on this, as well as the various specific subsections.

34.1.7.3.6 Shell preconditioners You already saw that, in an iterative methods, the coefficient matrix can be given operationally as a *shell matrix*; section 31.4.7. Similarly, the preconditioner matrix can be specified operationally by specifying type `PCSHELL`.

This needs specification of the application routine through `PCShellSetApply`:

```
PCShellSetApply(PC pc,PetscErrorCode (*apply)(PC,Vec,Vec));
```

and probably specification of a context pointer through `PCShellSetContext`:

```
PCShellSetContext(PC pc,void *ctx);
```

The application function then retrieves this context with `PCShellGetContext`:

```
PCShellGetContext(PC pc,void **ctx);
```

If the shell preconditioner requires setup, a routine for this can be specified with `PCShellSetSetUp`:

```
PCShellSetSetUp(PC pc,PetscErrorCode (*setup)(PC));
```

34.1.7.3.7 Combining preconditioners It is possible to combine preconditioners with `PCCOMPOSITE`

```
PCSetType(pc,PCCOMPOSITE);
PCCompositeAddPC(pc,type1);
PCCompositeAddPC(pc,type2);
```

By default, the preconditioners are applied additively; for multiplicative application

```
PCCompositeSetType(PC pc,PCCompositeType PC_COMPOSITE_MULTIPLICATIVE);
```

34.1.8 Customization: monitoring and convergence tests

PETSc solvers can do various *callbacks* to user functions.

34.1.8.1 Convergence tests

For instance, you can set your own convergence test with `KSPSetConvergenceTest`.

```
KSPSetConvergenceTest
(KSP ksp,
 PetscErrorCode (*test)(
   KSP ksp,PetscInt it,PetscReal rnorm,
   KSPConvergedReason *reason,void *ctx),
   void *ctx,PetscErrorCode (*destroy)(void *ctx));
```

This routines accepts

- the custom stopping test function,
- a ‘context’ void pointer to pass information to the tester, and
- optionally a custom destructor for the context information.

By default, PETSc behaves as if this function has been called with `KSPConvergedDefault` as argument.

34.1.8.2 Convergence monitoring

There is also a callback for monitoring each iteration. It can be set with `KSPMonitorSet`.

```
KSPMonitorSet
  (KSP ksp,
   PetscErrorCode (*mon)(
     KSP ksp,PetscInt it,PetscReal rnorm,void *ctx),
   void *ctx,PetscErrorCode (*mondestroy)(void**));
```

By default no monitor is set, meaning that the iteration process runs without output. The option `-ksp_monitor` activates printing a norm of the residual. This corresponds to setting `KSPMonitorDefault` as the monitor.

This actually outputs the ‘preconditioned norm’ of the residual, which is not the L2 norm, but the square root of $r^T M^{-1} r$, a quantity that is computed in the course of the iteration process. Specifying `KSPMonitorTrueResidualNorm` (with corresponding option `-ksp_monitor_true_residual`) as the monitor prints the actual norm $\sqrt{r^T r}$. However, to compute this involves extra computation, since this quantity is not normally computed.

34.1.8.3 Auxiliary routines

```
KSPGetSolution KSPGetRhs KSPBuildSolution KSPBuildResidual
KSPGetSolution(KSP ksp,Vec *x);
KSPGetRhs(KSP ksp,Vec *rhs);
KSPBuildSolution(KSP ksp,Vec w,Vec *v);
KSPBuildResidual(KSP ksp,Vec t,Vec w,Vec *v);
```

34.2 Direct solvers

PETSc has some support for direct solvers, that is, variants of LU decomposition. In a sequential context, the `PCLU` preconditioner can be used for this: a direct solver is equivalent to an iterative method that stops after one preconditioner application. This can be forced by specifying a KSP type of `KSPPREONLY`.

Distributed direct solvers are more complicated. PETSc does not have this implemented in its basic code, but it becomes available by configuring PETSc with the `scalapack` library.

You need to specify which package provides the LU factorization:

```
PCFactorSetMatSolverType(pc, MatSolverType solver )
```

where the solver variable is of type `MatSolverType`, and can be `MATSOLVERMUMPS` and such when specified in source:

```
// direct.c
PetscCall( KSPCreate(comm,&Solver) );
PetscCall( KSPSetOperators(Solver,A,A) );
PetscCall( KSPSetType(Solver,KSPPREONLY) );
{
  PC Prec;
  PetscCall( KSPGetPC(Solver,&Prec) );
  PetscCall( PCSetType(Prec,PCLU) );
  PetscCall( PCFactorSetMatSolverType(Prec,MATSOLVERMUMPS) );
}
```

Figure 34.6 `KSPSetFromOptions`

Synopsis

```
#include "petscksp.h"
PetscErrorCode KSPSetFromOptions(KSP ksp)
```

Collective on `ksp`

Input Parameters

`ksp` – the Krylov space context

As specified on the commandline

```
yourprog -ksp_type preonly -pc_type lu -pc_factor_mat_solver_type mumps
```

the choices are mumps, superlu, umfpack, or a number of others. Note that availability of these packages depends on how PETSc was installed on your system.

34.3 Control through command line options

From the above you may get the impression that there are lots of calls to be made to set up a PETSc linear system and solver. And what if you want to experiment with different solvers, does that mean that you have to edit a whole bunch of code? Fortunately, there is an easier way to do things. If you call the routine `KSPSetFromOptions` (figure 34.6) with the `solver` as argument, PETSc will look at your command line options and take those into account in defining the solver. Thus, you can either omit setting options in your source code, or use this as a way of quickly experimenting with different possibilities. Example:

```
myprogram -ksp_max_it 200 \
           -ksp_type gmres -ksp_type_gmres_restart 20 \
           -pc_type ilu -pc_type_ilu_levels 3
```

Chapter 35

PETSC nonlinear solvers

35.1 Nonlinear systems

Nonlinear system solving means finding the zero of a general nonlinear function, that is:

$$\underset{x}{?} : f(x) = 0$$

with $f : \mathbb{R}^n - \mathbb{R}^n$. In the special case of a linear function,

$$f(x) = Ax - b,$$

we solve this by any of the methods in chapter 34.

The general case can be solved by a number of methods, foremost *Newton's method*, which iterates

$$x_{n+1} = x_n - F(x_n)^{-1} f(x_n)$$

where F is the *Hessian* $F_{ij} = \partial f_i / \partial x_j$.

You see that you need to specify two functions that are dependent on your specific problem: the objective function itself, and its Hessian.

35.1.1 Basic setup

The PETSc nonlinear solver object is of type *SNES*: ‘simple nonlinear equation solver’. As with linear solvers, we create this solver on a communicator, set its type, incorporate options, and call the solution routine *SNESolve* (figure 35.1):

```
Vec value_vector,solution_vector;
/* vector creation code missing */
SNES solver;
SNESCreate( comm,&solver );
SNESSetFunction( solver,value_vector,formfunction, NULL );
SNESSetFromOptions( solver );
SNESolve( solver,NULL,solution_vector );
```

The function has the type

```
PetscErrorCode formfunction(SNES,Vec,Vec,void*)
```

Figure 35.1 SNESolve

```
#include "petscsnes.h"
PetscErrorCode SNESolve(SNES snes,Vec b,Vec x)

Collective on SNES

Input Parameters
snes - the SNES context
b    - the constant part of the equation F(x) = b, or NULL to use zero.
x    - the solution vector.
```

where the parameters are:

- the solver object, so that you can access to its internal parameters
- the x value at which to evaluate the function
- the result vector $f(x)$ for the given input
- a context pointer for further application-specific information.

Example:

```
PetscErrorCode evaluation_function( SNES solver,Vec x,Vec fx, void *ctx ) {
  const PetscReal *x_array;
  PetscReal *fx_array;
  VecGetArrayRead(fx,&fx_array);
  VecGetArray(x,&x_array);
  for (int i=0; i<localsize; i++)
    fx_array[i] = pointfunction( x_array[i] );
  VecRestoreArrayRead(fx,&fx_array);
  VecRestoreArray(x,&x_array);
};
```

Comparing the above to the introductory description you see that the Hessian is not specified here. An analytic Hessian can be dispensed with if you instruct PETSc to approximate it by finite differences:

$$H(x)y \approx \frac{f(x + hy) - f(x)}{h}$$

with h some finite difference. The commandline option `-snes_fd` forces the use of this finite difference approximation. However, it may lead to a large number of function evaluations. The option `-snes_fd_color` applies a coloring to the variables, leading to a drastic reduction in the number of function evaluations.

If you can form the analytic Jacobian / Hessian, you can specify it with `SNESSetJacobian` (figure 35.2), where the Jacobian is a function of type `SNESJacobianFunction` (figure 35.3).

Specifying the Jacobian:

```
Mat J;
ierr = MatCreate(comm,&J); CHKERRQ(ierr);
ierr = MatSetType(J,MATSEQDENSE); CHKERRQ(ierr);
ierr = MatSetSizes(J,n,n,N,N); CHKERRQ(ierr);
ierr = MatSetUp(J); CHKERRQ(ierr);
ierr = SNESSetJacobian(solver,J,J,&Jacobian,NULL); CHKERRQ(ierr);
```

Figure 35.2 SNESSetJacobian

```
#include "petscsnes.h"
PetscErrorCode SNESSetJacobian(SNES snes,Mat Amat,Mat Pmat,PetscErrorCode (*J)(SNES,Vec,Mat,Mat,void*),void *ctx)

Logically Collective on SNES

Input Parameters
snes - the SNES context
Amat - the matrix that defines the (approximate) Jacobian
Pmat - the matrix to be used in constructing the preconditioner, usually the same as Amat.
J - Jacobian evaluation routine (if NULL then SNES retains any previously set value)
ctx - [optional] user-defined context for the Jacobian evaluation routine
```

Figure 35.3 SNESJacobianFunction

```
#include "petscsnes.h"
PetscErrorCode SNESJacobianFunction(SNES snes,Vec x,Mat Amat,Mat Pmat,void *ctx);

Collective on snes

Input Parameters
x - input vector, the Jacobian is to be computed at this value
ctx - [optional] user-defined Jacobian context

Output Parameters
Amat - the matrix that defines the (approximate) Jacobian
Pmat - the matrix to be used in constructing the preconditioner, usually the same as Amat.
```

35.2 Time-stepping

For cases

$$u_t = G(t, u)$$

call *TSSetRHSFunction*.

```
#include "petscts.h"
PetscErrorCode TSSetRHSFunction
(TS ts,Vec r,
 PetscErrorCode (*f)(TS,PetscReal,Vec,Vec,void*),
 void *ctx);
```

For implicit cases

$$F(t, u, u_t) = 0$$

call *TSSetIFunction*

```
#include "petscts.h"
PetscErrorCode TSSetIFunction
(TS ts,Vec r,TSIFunction f,void *ctx)
```

Chapter 36

PETSc GPU support

36.1 Installation with GPUs

PETSc can be configured with options

```
--with-cuda --with-cudac=nvcc?
```

You can test the presence of CUDA with:

```
// cudainstalled.c
#ifndef PETSC_HAVE_CUDA
#error "CUDA is not installed in this version of PETSC"
#endif
```

Some GPUs can accomodate MPI by being directly connected to the network through *GPUDirect* Remote Memory Access (RMA). If not, use this runtime option:

```
-use_gpu_aware_mpi 0
```

More conveniently, add this to your .petscrc file; section [37.3.3](#).

36.2 Setup for GPU

GPUs need to be initialized. This can be done implicitly when a GPU object is created, or explicitly through *PetscDeviceInitialize*. (PETSc versions before PETSc-3.17 had an explicit routine *PetscCUDAIinitialize*.)

```
// cudainit.c
PetscDeviceType cuda = PETSC_DEVICE_CUDA;
 ierr = PetscDeviceInitialize(cuda);
PetscBool has_cuda;
has_cuda = PetscDeviceInitialized(cuda);
```

36.3 Distributed objects

Objects such as matrices and vectors need to be create explicitly with a CUDA type. After that, most PETSc calls are independent of the presence of GPUs.

Should you need to test, there is a CPP macro *PETSC_HAVE_CUDA*.

36.3.1 Vectors

Analogous to vector creation as before, there are specific create calls `VecCreateSeqCUDA`, `VecCreateMPICUDAWithArray`, or the type can be set in `VecSetType`:

```
// kspcu.c
#ifndef PETSC_HAVE_CUDA
    ierr = VecCreateMPICUDA(comm,localsize,PETSC_DECIDE,&Rhs);
#else
    ierr = VecCreateMPI(comm,localsize,PETSC_DECIDE,&Rhs);
#endif
```

The type `VECCUDA` is sequential or parallel dependent on the run; specific types are `VECSEQCUDA`, `VECMPICUDA`.

36.3.2 Matrices

```
ierr = MatCreate(comm,&A);
#ifndef PETSC_HAVE_CUDA
    ierr = MatSetType(A,MATMPIAIJCUSPARSE);
#else
    ierr = MatSetType(A,MATMPIAIJ);
#endif
```

Dense matrices can be created with specific calls `MatCreateDenseCUDA`, `MatCreateSeqDenseCUDA`, or by setting types `MATDENSECUDA`, `MATSEQDENSECUDA`, `MATMPIDENSECUDA`.

Sparse matrices: `MATAIJJCUSPARSE` which is sequential or distributed depending on how the program is started. Specific types are: `MATMPIAIJCUSPARSE`, `MATSEQAIJCUSPARSE`.

36.3.3 Array access

All sorts of ‘array’ operations such as `MatDenseCUDAGetArray`, `VecCUDAGetArray`,

Set `PetscMalloc` to use the GPU: `PetscMallocSetCUDAHost`, and switch back with `PetscMallocResetCUDAHost`.

36.4 Other

The memories of a CPU and GPU are not coherent. This means that routines such as `PetscMalloc1` can not immediately be used for GPU allocation. Use the routines `PetscMallocSetCUDAHost` and `PetscMallocResetCUDAHost` to switch the allocator to GPU memory and back.

```
// cudamatself.c
Mat cuda_matrix;
PetscScalar *matdata;
ierr = PetscMallocSetCUDAHost();
ierr = PetscMalloc1(global_size*global_size,&matdata);
ierr = PetscMallocResetCUDAHost();
ierr = MatCreateDenseCUDA
    (comm,
     global_size,global_size,global_size,global_size,
     matdata,
     &cuda_matrix);
```

Chapter 37

PETSc tools

37.1 Error checking and debugging

37.1.1 Debug mode

During installation (see section 30.3), there is an option of turning on debug mode. An installation with debug turned on:

- Does more runtime checks on numerics, or array indices;
- Does a memory analysis when you insert the `CHKMEMQ` macro (section 37.1.3);
- Has the macro `PETSC_USE_DEBUG` set to 1.

37.1.2 Error codes

PETSc performs a good amount of runtime error checking. Some of this is for internal consistency, but it can also detect certain mathematical errors. To facilitate error reporting, the following scheme is used.

Every PETSc call returns an error code; typically zero for success, and non-zero for various conditions. You should wrap each such function call in the `PetscCall` macro:

```
PetscCall( SomePetscRoutine( arguments ) );
```

(In many codes you may see a macro `CHKERRQ`; which was the mechanism pre-PETSc-3.18; see section 37.1.2.2.) This macro detects any error code, reports it, and exits the current routine.

For a good traceback, surround the executable part of any subprogram with `PetscFunctionBeginUser` and `PetscFunctionReturn`, where the latter has the return value as parameter. (The routine `PetscFunctionBegin` does the same, but should only be used for PETSc library routines.)

37.1.2.1 Error throwing

You can effect your own error return by using the variadic function `SETERRQ` (figure 37.1).

Example. We write a routine that sets an error:

```
// backtrace.c
PetscErrorCode this_function_bombs() {
    PetscFunctionBegin;
    SETERRQ(PETSC_COMM_SELF,1,"We cannot go on like this");
    PetscFunctionReturn(0);
}
```

Figure 37.1 SETERRQ

```
#include <petscsys.h>
PetscErrorCode SETERRQ (MPI_Comm comm,PetscErrorCode ierr,char *message)
PetscErrorCode SETERRQ1(MPI_Comm comm,PetscErrorCode ierr,char *formatmessage,arg1)
PetscErrorCode SETERRQ2(MPI_Comm comm,PetscErrorCode ierr,char *formatmessage,arg1,arg2)
PetscErrorCode SETERRQ3(MPI_Comm comm,PetscErrorCode ierr,char *formatmessage,arg1,arg2,arg3)
```

Input Parameters:

comm - A communicator, so that the error can be collective
ierr - nonzero error code, see the list of standard error codes in include/petscerror.h
message - error message in the printf format
arg1,arg2,arg3 - argument (for example an integer, string or double)

Running this gives, in process zero, the output

```
[0]PETSC ERROR: We cannot go on like this
[0]PETSC ERROR: See https://www.mcs.anl.gov/petsc/documentation/faq.html for trouble shooting.
[0]PETSC ERROR: Petsc Release Version 3.12.2, Nov, 22, 2019
[0]PETSC ERROR: backtrace on a [computer name]
[0]PETSC ERROR: Configure options [all options]
[0]PETSC ERROR: #1 this_function_bombs() line 20 in backtrace.c
[0]PETSC ERROR: #2 main() line 30 in backtrace.c
```

Fortran note 31: Backtrace on error. In Fortran the backtrace is not quite as elegant.

```
!! backtrace.F90
Subroutine this_function_bombs(ierr)
  implicit none
  integer,intent(out) :: ierr

  SETERRQ(PETSC_COMM_SELF,1,"We cannot go on like this")
  ierr = -1

end Subroutine this_function_bombs
```

```
[0]PETSC ERROR: ----- Error Message -----
[0]PETSC ERROR: We cannot go on like this
[....]
[0]PETSC ERROR: #1 User provided function() line 0 in User file
```

Remark In this example, the use of `PETSC_COMM_SELF` indicates that this error is individually generated on a process; use `PETSC_COMM_WORLD` only if the same error would be detected everywhere.

Exercise 37.1. Look up the definition of `SETERRQ`. Write a routine to compute square roots that is used as follows:

```
x = 1.5; ierr = square_root(x,&rootx); CHKERRQ(ierr);
PetscPrintf(PETSC_COMM_WORLD,"Root of %f is %f\n",x,rootx);
x = -2.6; ierr = square_root(x,&rootx); CHKERRQ(ierr);
PetscPrintf(PETSC_COMM_WORLD,"Root of %f is %f\n",x,rootx);
```

This should give as output:

```
Root of 1.500000 is 1.224745
```

```
[0]PETSC ERROR: ----- Error Message -----
```

```
[0]PETSC ERROR: Cannot compute the root of -2.600000
[...]
[0]PETSC ERROR: #1 square_root() line 23 in root.c
[0]PETSC ERROR: #2 main() line 39 in root.c
```

37.1.2.2 Legacy error checking

In PETSc versions pre-PETSc-3.18, errors were handled slightly differently.

1. Every PETSc routine is a function returning a parameter of type `PetscErrorCode`.
2. Calling the macro `CHKERRQ` on the error code will cause an error to be printed and the current routine to be terminated. Recursively this gives a traceback of where the error occurred.

```
PetscErrorCode ierr;
ierr = AnyPetscRoutine( arguments ); CHKERRQ(ierr);
```

3. Other error checking macros are `CHKERRABORT` which aborts immediately, and `CHKERRMPI`.

Fortran note 32: Error code handling. In the main program, use `CHKERRA` and `SETERRA`. Also beware that these error ‘commands’ are macros, and after expansion may interfere with *Fortran line length*, so they should only be used in .F90 files.

C++ note 28: Exception handling. The macro `CHCKERcxx` handles exceptions.

37.1.3 Memory corruption

PETSc has its own memory management (section 37.5) and this facilitates finding memory corruption errors. The macro `CHKMEMQ` (`CHKMEMA` in void functions) checks all memory that was allocated by PETSc, either internally or through the allocation routines, for corruption. Sprinkling this macro through your code can detect memory problems before they lead to a *segfault*.

This testing is only done if the commandline argument `-malloc_debug` (`-malloc_test` in debug mode) is supplied, so it carries no overhead for production runs.

37.1.3.1 Valgrind

Valgrind is rather verbose in its output. To limit the number of processes that run under valgrind:

```
mpiexec -n 3 valgrind --track-origins=yes ./app -args : -n 5 ./app -args
```

37.2 Program output

PETSc has a variety of mechanisms to export or visualize program data. We will consider a few possibilities here.

37.2.1 Screen I/O

Printing screen output in parallel is tricky. If two processes execute a print statement at more or less the same time there is no guarantee as to in what order they may appear on screen. (Even attempts to have them print one after the other may not result in the right ordering.) Furthermore, lines from multi-line print actions on two processes may wind up on the screen interleaved.

Figure 37.2 `PetscPrintf`

C:
PetscErrorCode PetscPrintf(MPI_Comm comm,const char format[],...)

Fortran:
PetscPrintf(MPI_Comm, character(*), PetscErrorCode ierr)

Python:
PETSc.Sys.Print(type cls, *args, **kwargs)
kwargs:
comm : communicator object

Figure 37.3 `PetscSynchronizedPrintf`

C:
PetscErrorCode PetscSynchronizedPrintf(
MPI_Comm comm,const char format[],...)

Fortran:
PetscSynchronizedPrintf(MPI_Comm, character(*), PetscErrorCode ierr)

python:
PETSc.Sys.syncPrint(type cls, *args, **kargs)
kwargs:
comm : communicator object
flush : if True, do synchronizedFlush
other keyword args as for python3 print function

37.2.1.1 *printf replacements*

PETSc has two routines that fix this problem. First of all, often the information printed is the same on all processes, so it is enough if only one process, for instance process 0, prints it. This is done with `PetscPrintf` (figure 37.2).

If all processes need to print, you can use `PetscSynchronizedPrintf` (figure 37.3) that forces the output to appear in process order.

To make sure that output is properly flushed from all system buffers use `PetscSynchronizedFlush` (figure 37.4) where for ordinary screen output you would use `stdout` for the file.

Fortran note 33: Print string construction. Fortran does not have the variable-number-of-arguments mechanism from C, so you can only use `PetscPrintf` on a buffer that you construct with a `Write` statement:

```
Character*80 :: message
write(message,10) xnorm,ynorm
10 format("Norm x: ",f6.3," y: ",f6.3,"\\n")
call PetscPrintf(comm,message,ierr)
```

Fortran note 34: Printing and newlines. The Fortran calls are only wrappers around C routines, so you can use `\n` newline characters in the Fortran string argument to `PetscPrintf`.

The file to flush is typically `PETSC_STDOUT`.

Python note 45: Petsc print and python print. Since the print routines use the python `print` call, they automatically include the trailing newline. You don't have to specify it as in the C calls.

Figure 37.4 `PetscSynchronizedFlush`

```
C:
PetscErrorCode PetscSynchronizedFlush(MPI_Comm comm,FILE *fd)
fd : output file pointer, needs to be valid on process zero

Fortran:
PetscSynchronizedFlush(comm,fd,err)
Integer :: comm
fd is usually PETSC_STDOUT
PetScErrorCode :: err

python:
PETSc.Sys.syncFlush(type cls, comm=None)
```

Figure 37.5 `PetscViewerRead`

Synopsis

```
#include "petscviewer.h"
PetscErrorCode PetscViewerRead(PetscViewer viewer, void *data, PetscInt num, PetscInt *count, PetscDataType dtype)

Collective

Input Parameters
viewer - The viewer
data - Location to write the data
num - Number of items of data to read
datatype - Type of data to read

Output Parameters
count -number of items of data actually read, or NULL
```

37.2.1.2 *scanf replacement*

Using `scanf` in Petsc is tricky, since integers and real numbers can be of different sizes, depending on the installation. Instead, use `PetscViewerRead` (figure 37.5), which operates in terms of `PetScDataType`.

37.2.2 Viewers

In order to export PETSc matrix or vector data structures there is a `PetscViewer` object type. This is a quite general concept of viewing: it encompasses ascii output to screen, binary dump to file, or communication to a running Matlab process. Calls such as `MatView` or `KSPView` accept a `PetscViewer` argument.

In cases where this makes sense, there is also an inverse ‘load’ operation. See section 31.3.5 for vectors.

Some viewers are predefined, such as `PETSC_VIEWER_STDOUT_WORLD` for ascii rendering to standard out. (In C, specifying zero or `NULL` also uses this default viewer; for Fortran use `PETSC_NULL_VIEWER`.)

37.2.2.1 *Viewer types*

For activities such as dumping to file you first need create the viewer with `PetscViewerCreate` and set its type with `PetscViewerSetType`.

```
PetscViewerCreate(comm,&viewer);
PetscViewerSetType(viewer,PETSCVIEWERBINARY);
```

Popular types include PETSCVIEWERASCII, PETSCVIEWERBINARY, PETSCVIEWERSTRING, PETSCVIEWERDRAW, PETSCVIEWERSOCKET, PETSCVIEWERHDF5, PETSCVIEWERVTK; the full list can be found in `include/petscviewer.h`.

37.2.2.2 Viewer formats

Viewers can take further format specifications by using `PetscViewerPushFormat`:

```
PetscViewerPushFormat
(PETSC_VIEWER_STDOUT_WORLD,
 PETSC_VIEWER_ASCII_INFO_DETAIL);
```

and afterwards a corresponding `PetscViewerPopFormat`

Python note 46: HDF5 file generation.

```
## hdf5.py
file_name = "hdf5.dat"
viewer = PETSc.Viewer().createHDF5(file_name, 'w', comm)
x.view(viewer)
viewer = PETSc.Viewer().createHDF5(file_name, 'r', comm)
x.load(viewer)
```

37.2.2.3 Commandline option for viewers

Petsc objects viewers can be activated by calls such as `MatView`, but often it is more convenient to do this through commandline options, such as `-mat_view`, `-vec_view`, or `-ksp_view`. By default, these output to `stdout` in `ascii` form, but this can be controlled by further option values:

```
program -mat_view binary:matrix.dat
```

where `binary` forces a binary dump (`ascii` is the default) and a file name is explicitly given.

Binary dump may not be supported for all datatypes, in particular `DM`. For that case, do

```
program -dm_view draw \
-draw_pause 20
```

which pops up an `X11` window, for the duration of the indicated pause.

If a viewer needs to be triggered at a specific location, calls such as `VecViewFromOptions` can be used. These routines all have a similar calling sequence:

```
#include "petscsys.h"
PetscErrorCode  PetscObjectViewFromOptions(PetscObject obj,PetscObject bobj,const char optionname[])
PetscErrorCode  VecViewFromOptions(Vec A,PetscObject obj,const char name[])

AOViewFromOptions, DMViewFromOptions, ISViewFromOptions, ISLocalToGlobalMappingViewFromOptions,
KSPConvergedReasonViewFromOptions, KSPViewFromOptions, MatPartitioningViewFromOptions,
MatCoarsenViewFromOptions, MatViewFromOptions, PetscObjectViewFromOptions,
PetscPartitionerViewFromOptions, PetscDrawViewFromOptions, PetscRandomViewFromOptions,
PetscDualSpaceViewFromOptions, PetscSFViewFromOptions, PetscFEViewFromOptions,
```

```
PetscFVViewFromOptions, PetscSectionViewFromOptions, PCViewFromOptions, PetscSpaceViewFromOptions,
PFViewFromOptions, PetscLimiterViewFromOptions, PetscLogViewFromOptions, PetscDSViewFromOptions,
PetscViewerViewFromOptions, SNESConvergedReasonViewFromOptions, SNESViewFromOptions,
TSTrajectoryViewFromOptions, TSViewFromOptions, TaoLineSearchViewFromOptions, TaoViewFromOptions,
VecViewFromOptions, VecScatterViewFromOptions,
```

37.2.2.4 Naming objects

A helpful facility for viewing is to name an object: that name will then be displayed when the object is viewed.

```
Vec i_local;
ierr = VecCreate(comm,&i_local); CHKERRQ(ierr);
ierr = PetscObjectSetName((PetscObject)i_local,"space local"); CHKERRQ(ierr);
```

giving:

```
Vec Object: space local 4 MPI processes
  type: mpi
Process [0]
[ ... et cetera ... ]
```

37.3 Commandline options

PETSc has as large number of commandline options, most of which we will discuss later. For now we only mention `-log_summary` which will print out profile of the time taken in various routines. For these options to be parsed, it is necessary to pass `argc`, `argv` to the `PetscInitialize` call.

37.3.1 Adding your own options

You can add custom commandline options to your program. Various routines such as `PetscOptionsGetInt` scan the commandline for options and set parameters accordingly. For instance,

```
// ksp.c
PetscBool flag;
PetscInt domain_size = 100;
PetscCall( PetscOptionsGetInt(NULL,NULL,"-n",&domain_size,&flag) );
PetscPrintf(comm,"Using domain size %d\n",domain_size);
```

declares the existence of an option `-n` to be followed by an integer. (The first argument is an options database or `NULL`; the second argument is a prefix for the option or `NULL`.)

Now executing

```
mpiexec yourprogram -n 5
```

will

1. set the `flag` to true, and
2. set the parameter `domain_size` to the value on the commandline.

Omitting the `-n` option will leave the default value of `domain_size` unaltered.

For flags, use `PetscOptionsHasName`.

Python note 47: Petsc options. In Python, do not specify the initial hyphen of an option name. Also, the functions such as `getInt` do not return the boolean flag; if you need to test for the existence of the commandline option, use:

```
hasn = PETSc.Options().hasName("n")
```

There is a related mechanism using `PetscOptionsBegin / PetscOptionsEnd`:

```
// optionsbegin.c
PetscOptionsBegin(comm,NULL,"Parameters",NULL);
PetscCall( PetscOptionsInt("-i","i value",__FILE__,i_value,&i_value,&i_flag) );
PetscCall( PetscOptionsInt("-j","j value",__FILE__,j_value,&j_value,&j_flag) );
PetscOptionsEnd();
if (i_flag)
    PetscPrintf(comm,"Option `i' was used\n");
if (j_flag)
    PetscPrintf(comm,"Option `j' was used\n");
```

The selling point for this approach is that running your code with

```
mpiexec yourprogram -help
```

will display these options as a block. Together with a ton of other options, unfortunately.

37.3.2 Options prefix

In many cases, your code will have only one `KSP` solver object, so specifying `-ksp_view` or `-ksp_monitor` will display / trace that one. However, you may have multiple solvers, or nested solvers. You may then not want to display all of them.

As an example of the nest solver case, consider the case of a *block jacobi preconditioner*, where the block is itself solved with an iterative method. You can trace that one with `--sub_ksp_monitor`.

The `sub_` is an *option prefix*, and you can defined your own with `KSPSetOptionsPrefix`. (There are similar routines for other PETSc object types.)

Example:

```
KSPCreate(comm,&time_solver);
KSPCreate(comm,&space_solver);
KSPSetOptionsPrefix(time_solver,"time_");
KSPSetOptionsPrefix(space_solver,"space_");
```

You can then use options `-time_ksp_monitor` and such. Note that the prefix does not have a leading dash, but it does have the trailing underscore.

Similar routines: `MatSetOptionsPrefix`, `PCSetOptionsPrefix`, `PetscObjectSetOptionsPrefix`, `PetscViewerSetOptionsPrefix`, `SNESSetOptionsPrefix`, `TSSetOptionsPrefix`, `VecSetOptionsPrefix`, and some more obscure ones.

Figure 37.6 `PetscTime`**Synopsis**

Returns the CPU time in seconds used by the process.

```
#include "petscsys.h"
#include "petsctime.h"
PetscErrorCode PetscGetCPUTime(PetscLogDouble *t)
PetscErrorCode PetscTime(PetscLogDouble *v)
```

37.3.3 Where to specify options

Commandline options can obviously go on the commandline. However, there are more places where they can be specified.

Options can be specified programmatically with `PetscOptionsSetValue`:

```
PetscOptionsSetValue( NULL, // for global options
    "-some_option", "value_as_string");
```

Options can be specified in a file `.petscrc` in the user's home directory or the current directory.

Finally, an environment variable `PETSC_OPTIONS` can be set.

The `rc` file is processed first, then the environment variable, then any commandline arguments. This parsing is done in `PetscInitialize`, so any values from `PetscOptionsSetValue` override this.

37.4 Timing and profiling

PETSc has a number of timing routines that make it unnecessary to use system routines such as `getrusage` or MPI routines such as `MPI_Wtime`. The main (wall clock) timer is `PetscTime` (figure 37.6). Note the return type of `PetscLogDouble` which can have a different precision from `PetscReal`.

The routine `PetscGetCPUTime` is less useful, since it measures only time spent in computation, and ignores things such as communication.

37.4.1 Logging

Petsc does a lot of logging on its own operations. Additionally, you can introduce your own routines into this log.

The simplest way to display statistics is to run with an option `-log_view`. This takes an optional file name argument:

```
mpiexec -n 10 yourprogram -log_view :statistics.txt
```

The corresponding routine is `PetscLogView`.

37.5 Memory management

Allocate the memory for a given pointer: `PetscNew`, allocate arbitrary memory with `PetscMalloc`, allocate a number of objects with `PetscMalloc1` (figure 37.7) (this does not zero the memory allocated, use `PetscCalloc1` to obtain memory that has been zeroed); use `PetscFree` (figure 37.8) to free.

Figure 37.7 `PetscMalloc1`

Synopsis

Allocates an array of memory aligned to PETSC_MEMALIGN

C:

```
#include <petscsys.h>
PetscErrorCode PetscMalloc1(size_t m1,type **r1)
```

Input Parameter:

m1 - number of elements to allocate (may be zero)

Output Parameter:

r1 - memory allocated

Figure 37.8 `PetscFree`

Synopsis

Frees memory, not collective

C:

```
#include <petscsys.h>
PetscErrorCode PetscFree(void *memory)
```

Input Parameter:

memory - memory to free (the pointer is ALWAYS set to NULL upon success)

```
PetscInt *idxs;
PetscMalloc1(10,&idxs);
// better than:
// PetscMalloc(10*sizeof(PetscInt),&idxs);
for (PetscInt i=0; i<10; i++)
  idxs[i] = f(i);
PetscFree(idxs);
```

Allocated memory is aligned to PETSC_MEMALIGN.

The state of memory allocation can be written to file or standard out with `PetscMallocDump`. The commandline option `-malloc_dump` outputs all not-freed memory during `PetscFinalize`.

37.5.1 GPU allocation

The memories of a CPU and GPU are not coherent. This means that routines such as `PetscMalloc1` can not immediately be used for GPU allocation. See section 36.4 for details.

Chapter 38

PETSc topics

38.1 Communicators

PETSc has a ‘world’ communicator, which by default equals `MPI_COMM_WORLD`. If you want to run PETSc on a subset of processes, you can assign a subcommunicator to the variable `PETSC_COMM_WORLD` in between the calls to `MPI_Init` and `PetscInitialize`. Petsc communicators are of type `PetscComm`.

PART IV

OTHER PROGRAMMING MODELS

There are many other programming systems for parallelism. In particular threading models are a dime a dozen.

Here we discuss:

- Co-array Fortran (CAF): a distributed parallel mode for Fortran arrays; chapter 39.
- Kokkos and Sycl, two heterogeneous programming models that target both multicore and accelerator programming (and Field Programmable Gate Arrays (FPGAs) in the case of Sycl); chapters 40 and 41 respectively.
- Python multiprocessing; chapter 43.

Comparing these to the systems already discussed above we can remark:

- CAF is one of the very few alternatives to MPI where it comes to distributed memory programming. Its treatment of Cartesian arrays is more elegant; otherwise it lacks much MPI functionality.
- Kokkos and Sycl are competitors to OpenMP offloading; chapter 27. Switching between CPU and GPU modes is easier in these systems than in OpenMP.
- The python multiprocessing toolbox is more task-base than the *mpi4py* module in Python.

Among the various threading models in existence we mention

- *pthreads*, which is more geared to systems programming than scientific computing; see HPC book, section ??.
- C++ has a native thread library; see *Introduction to Scientific Programming book*, section ???. There is also the execution policy mechanism for the C++ standard library ‘algorithms’.
- Intel TBB is often used as a lower layer for other threading implementations, such as the native C++ parallel execution policies.

Parallelism models we do not discuss include

- *Compute-Unified Device Architecture (CUDA)* which targets GPUs.
- *NVidia Collective Communication Library (NCCL)*, which optimized MPI collectives for GPUs.
- Chapel, a completely independent language for parallel computing; see HPC book, section ???. On the topic of parallel languages, see more generally HPC book, section ??.

Chapter 39

Co-array Fortran

This chapter explains the basic concepts of CAF, and helps you get started on running your first program.

39.1 History and design

https://en.wikipedia.org/wiki/Coarray_Fortran

39.2 Compiling and running

CAF is built on the same SPMD design as MPI. Where MPI talks about processes or ranks, CAF calls the running instances of your program *images*.

The Intel compiler uses the flag `-coarray=xxx` with values `single`, `shared`, `distributed` `gpu`.

It is possible to bake the number of ‘images’ into the executable, but by default this is not done, and it is determined at runtime by the variable `FOR_COARRAY_NUM_IMAGES`.

CAF can not be mixed with OpenMP.

39.3 Basics

Co-arrays are defined by giving them, in addition to the `Dimension`, a `Codimension`

```
Complex,codimension(*) :: number
Integer,dimension(:,:,:),codimension[-1:1,*] :: grid
```

This means we are respectively declaring an array with a single number on each image, or a three-dimensional grid spread over a two-dimensional processor grid.

Traditional-like syntax can also be used:

```
Complex :: number[*]
Integer :: grid(10,20,30)[-1:1,*]
```

Unlike MPI, which normally only supports a linear process numbering, CAF allows for multi-dimensional process grids. The last dimension is always specified as `*`, meaning it is determined at runtime.

39.3.1 Image identification

As in other models, in CAF one can ask how many images/processes there are, and what the number of the current one is, with `num_images` and `this_image` respectively.

```
!! hello.F90
write(*,*) "Hello from image ", this_image(), &
"out of ", num_images()," total images"
```

If you call `this_image` with a co-array as argument, it will return the image index, as a tuple of `cosubscripts`, rather than a linear index. Given such a set of subscripts, `image_index` will return the linear index.

The functions `lcobound` and `ucobound` give the lower and upper bound on the image subscripts, as a linear index, or a tuple if called with a co-array variable.

39.3.2 Remote operations

The appeal of CAF is that moving data between images looks (almost) like an ordinary copy operation:

```
real :: x(2)[*]
integer :: p
p = this_image()
x(1)[ p+1 ] = x(2)[ p ]
```

Exchanging grid boundaries is elegantly done with array syntax:

```
Real,Dimension( 0:N+1,0:N+1 )[*] :: grid
grid( N+1,: )[p] = grid( 0,: )[p+1]
grid( 0,: )[p] = grid( N,: )[p-1]
```

39.3.3 Synchronization

The fortran standard forbids *race conditions*:

If a variable is defined on an image in a segment, it shall not be referenced, defined or become undefined in a segment on another image unless the segments are ordered.

That is, you should not cause them to happen. The language and runtime are certainly not going to help you with that.

Well, a little. After remote updates you can synchronize images with the `sync` call. The easiest variant is a global synchronization:

```
sync all
```

Compare this to a wait call after MPI nonblocking calls.

More fine-grained, one can synchronize with specific images:

```
sync images( / p-1,p,p+1 / )
```

While remote operations in CAF are nicely one-sided, synchronization is not: if image `p` issues a call

```
sync(q)
```

then q also needs to issue a mirroring call to synchronize with p.

As an illustration, the following code is not a correct implementation of a *ping-pong*:

```
!! pingpong.F90
sync all
if (procid==1) then
    number[procid+1] = number[procid]
else if (procid==2) then
    number[procid-1] = 2*number[procid]
end if
sync all
```

We can solve this with a global synchronization:

```
sync all
if (procid==1) &
    number[procid+1] = number[procid]
sync all
if (procid==2) &
    number[procid-1] = 2*number[procid]
sync all
```

or a local one:

```
if (procid==1) &
    number[procid+1] = number[procid]
if (procid<=2) sync images( (/1,2/) )
if (procid==2) &
    number[procid-1] = 2*number[procid]
if (procid<=2) sync images( (/2,1/) )
```

Note that the local sync call is done on both images involved.

Example of how you would synchronize a collective:

```
if ( this_image() .eq. 1 ) sync images( * )
if ( this_image() .ne. 1 ) sync images( 1 )
```

Here image 1 synchronizes with all others, but the others don't synchronize with each other.

```
if (procid==1) then
    sync images( (/procid+1/) )
else if (procid==nprocs) then
    sync images( (/procid-1/) )
else
    sync images( (/procid-1,procid+1/) )
end if
```

39.3.4 Collectives

Collectives are not part of CAF as of the 2008 Fortran standard.

Chapter 40

Kokkos

Much of this material is based on the Kokkos Tutorial that Jeff Miles and Christian Trott gave April 21-24, 2020.

40.1 Compilation

Include file:

```
// hello.cxx
#include "Kokkos_Core.hpp"
```

Discoverable in CMake:

```
find_package(Kokkos REQUIRED)
target_link_libraries(myTarget Kokkos::kokkos)
```

Either set *CMAKE_PREFIX_PATH* or add

```
-DKokkos_ROOT=<Kokkos Install Directory>/lib64/cmake/Kokkos
```

Maybe:

```
-DCMAKE_CXX_COMPILER=<Kokkos Install Directory>/bin/nvcc_wrapper
```

See <https://kokkos.org/kokkos-core-wiki/ProgrammingGuide/Compiling.html>

40.2 Parallel code execution

In parallel execution we basically have two issues:

1. The parallel structure of the algorithm; that's what we discuss in this section.
2. the memory structure of how the data is laid out; this will be discussed in section 40.3.

The algorithmic parallel structure is indicated with the following constructs.

```
Kokkos::parallel_for
Kokkos::parallel_reduce
Kokkos::parallel_scan
```

40.2.1 Example: 1D loop

Hello world:

```
Kokkos::parallel_for
( 10,
[](int i){ cout << "hello " << i << "\n"; }
);
```

The two arguments of `parallel_for` are:

- The number of iterations,
- A function of the iteration number. You can use a function pointer here, but often we will use lambda expressions.

Optionally, the parallel construct takes a string argument that can be used for naming.

40.2.2 Reduction

Reductions add a parameter to the construct: the reduction variable. Here is the traditional calculation of π by integration:

```
double pi{0.};
int n{100};
Kokkos::parallel_reduce
( "PI",
n,
KOKKOS_LAMBDA ( int i, double& partial ) {
    double h = 1./n, x = i*h;
    partial += h * sqrt( 1-x*x );
},
pi
);
```

- The parallel construct has an optional name. This is useful for profiling and debugging.
- Instead of an explicit lambda capture, we use `KOKKOS_LAMBDA` which does a [=] capture, and adds clauses for GPU execution, if needed.
- The lambda expression now takes two parameters: the iteration number, and the reduction variable. This is the thread-private variable, not the final one.
- The final argument is the global reduction variable.

For reductions other than summing, a `reducer` is needed.

```
// reduxmax.cxx
double max=0.;
Kokkos::parallel_reduce
( npoints,
KOKKOS_LAMBDA ( int i,double& m) {
    if (x(i)>m)
        m = x(i);
},
Kokkos::Max<double>(max)
);
cout << "max: " << max << "\n";
```

40.2.3 Examples: Multi-D loops

You can of course parallelize over the outer loop, and do the inner loops in the functor. This code computes $r \leftarrow y^t Ax$:

```
Kokkos::parallel_reduce( "yAx", N,
    KOKKOS_LAMBDA ( int j, double &update ) {
        double temp2 = 0;

        for ( int i = 0; i < M; ++i ) {
            temp2 += A[ j * M + i ] * x[ i ];
        }

        update += y[ j ] * temp2;
    },
    result
);
```

You can also leave all the loops to Kokkos, with an *RangePolicy* or *MDRangePolicy*. Here you indicate the rank (as in: number of dimensions) of the object, as well as arrays of first/last values. In the above examples

```
Kokkos::parallel_reduce( N, ... );
// equivalent:
Kokkos::parallel_reduce( Kokkos::RangePolicy<>(0,N), ... );
```

An example with a higher rank than one:

```
// matyax.cxx
Kokkos::parallel_reduce
( "ytAx product",
  Kokkos::MDRangePolicy<Kokkos::Rank<2>>( {0,0}, {m,n} ),
  KOKKOS_LAMBDA ( int i,int j,double &partial ) {
    partial += yvec(i) * matrix(i,j) * xvec(j); },
  sum
);
```

Note the multi-D indexing in this example: this parenthesis notation gets translated to the correct row/column-major depending on whether the code runs on a CPU or GPU; see section [19.6.2](#).

40.3 Data

One of the problems Kokkos addresses is the coherence of data between main processor and attached devicees such GPUs. This is handled through the *Kokkos::View* mechanism.

```
// matsum.cxx
int m=10,n=100;
Kokkos::View<double*> matrix("flat",m,n);
assert( matrix.extent(0)==10 );
```

These act like C++ *shared_ptr*, so capturing them by value gives you the data by reference anyway. Storage is automatically freed, RAII-style, when they go out of scope.

Indexing is best done with a Fortran-style notation:

```
matrix(i,j)
```

which makes indexing in your algorithm independent of the actual layout.

Compile-time dimensions can be accommodated:

```
View<double*[2]> tallskinny("tallthin",100);
View<double*[2][3]> tallthin(100);
```

with the compile-time dimensions trailing. Naming is optional.

Methods:

- `extent(int)` gives the extent in a certain dimensions;
- `data` gives a raw pointer to the data.

40.3.1 Data layout

The view declaration has an optional template argument for the data layout.

```
View<double***, Layout, Space> name(...);
```

Values are

- `LayoutLeft` where, Fortran-style, the leftmost index is stride 1; this is the default for `CudaSpace`.
- `LayoutRight` where, C-style, the leftmost index is stride 1; this is the default for `HostSpace`.
- `LayoutStride`, `LayoutTiled` and others.
- User-defined.

Practically speaking, the traversal of a two-dimensional array is now a function of

- the layout, possibly determined by the memory space, and
- the indexing in in the functor:

```
Kokkos::parallel_whatever(
    N,
    KOKKOS_LAMBDA ( size_t i ) {
        matrix(i,j) or matrix(j,i); }
);
```

It is probably best to stick with this Rule of Thumb:

With a layout determined by the memory space,
let the iterator index be first,
and let loops inside the functor range over subsequent indexes.

40.4 Execution and memory spaces

The body of the functor can be executed on the CPU or on a GPU. Those are the *execution spaces*. Kokkos needs to be installed with support for such spaces.

To indicate that a function or lambda expression can be executed on more than one possible execution space:

- use `KOKKOS_LAMBDA` as the capture for lambda expressions, or
- prefix explicitly defined functions with `KOKKOS_INLINE_FUNCTION`.

Execution spaces can be explicitly indicated using the *RangePolicy* keyword:

```
Kokkos::parallel_for
( Kokkos::RangePolicy<>( 0,10 ), # default execution space
[] (int i) {} );
Kokkos::parallel_for
( Kokkos::RangePolicy<SomeExecutionSpace>( 0,10 ),
[] (int i) {} );
```

The default

```
Kokkos::parallel_for( N, ...
```

is equivalent to

```
Kokkos::parallel_for( RangePolicy<>(N), ...
```

40.4.1 Memory spaces

Where data is stored is an independent story. Each execution space has a *memory space*. When creating a *View*, you can optionally indicate a memory space argument:

```
View<double***,MemorySpace> data(...);
```

Available memory spaces include: *HostSpace*, *CudaSpace*, *CudaUVMSpace*. Leaving out the memory space argument is equivalent to

```
View<double**,DefaultExecutionSpace::memory_space> x(1,2);
```

Examples:

```
View<double*,HostSpace> hostarray(5);
View<double*,CudaSpace> cudaarray(5);
```

The *CudaSpace* is only available if Kokkos has been configured with CUDA

40.4.2 Space coherence

Kokkos never makes implicit deep copies, so you can not immediately run a functor in the Cuda execution space on a view in Host space.

You can create a mirror of CUDA data on the host:

```
CuMatrix matrix(m,n);
CuMatrix::HostMirror hostmatrix =
    Kokkos::create_mirror_view(matrix);
// populate matrix on the host
for (i) for (j) hostmatrix(i,j) = ....;
// deep copy to GPU
Kokkos::deep_copy(matrix,hostmatrix);
// do something on the GPU
Kokkos::parallel_whatever(
    RangePolicy<CudaSpace>( 0,n ),
    some_lambda );
// if needed, deep copy back.
```

40.5 Configuration

An accelerator-free installation with OpenMP:

```
cmake \
    -D Kokkos_ENABLE_SERIAL=ON -D Kokkos_ENABLE_OPENMP=ON
```

Threading is not compatible with OpenMP:

```
-D Kokkos_ENABLE_THREADS=ON
```

Cuda installation:

```
cmake \
    -D Kokkos_ENABLE_CUDA=ON -D Kokkos_ARCH_TURING75=ON -D Kokkos_ENABLE_CUDA_LAMBDA=ON
```

40.6 Stuff

There are init/finalize calls, which are not always needed.

```
// pi.cu
Kokkos::initialize(argc,argv);
Kokkos::finalize();
```

40.6.1 OpenMP integration

Cmake flag to enable OpenMP: -D Kokkos_ENABLE_OPENMP=ON

After that, all the usual OpenMP environment variables work.

Alternatively:

```
int nthreads = Kokkos::OpenMP::concurrency();
Kokkos::initialize(Kokkos::InitializationSettings().set_num_threads(nthreads))
```

Parallelism control:

```
--kokkos-threads=123 # threads
--kokkos-numa=45      # numa regions
--kokkos-device=6     * GPU id to use
```

Chapter 41

Sycl, OneAPI, DPC++

This chapter explains the basic concepts of Sycl/Dpc++, and helps you get started on running your first program.

- *SYCL* is a C++-based language for portable parallel programming.
- *Data Parallel C++ (DPCPP)* is Intel's extension of Sycl.
- *OneAPI* is Intel's compiler suite, which contains the DPCPP compiler.

Intel DPC++ extension. The various Intel extensions are listed here: <https://spec.oneapi.com/versions/latest/elements/dpcpp/source/index.html#extensions-table>

41.1 Logistics

Headers:

```
#include <CL/sycl.hpp>
```

You can now include namespace, but with care! If you use

```
using namespace cl;
```

you have to prefix all SYCL class with *sycl::*, which is a bit of a bother. However, if you use

```
using namespace cl::sycl;
```

you run into the fact that SYCL has its own versions of many Standard Template Library (STL) commands, and so you will get name collisions. The most obvious example is that the *cl::sycl* name space has its own versions of *cout* and *endl*. Therefore you have to use explicitly *std::cout* and *std::end*. Using the wrong I/O will cause tons of inscrutable error messages. Additionally, SYCL has its own version of *free*, and of several math routines.

Intel DPC++ extension.

```
using namespace sycl;
```

41.2 Platforms and devices

Since DPCPP is cross-platform, we first need to discover the devices.

First we list the platforms:

```
// devices.cxx
std::vector<sycl::platform> platforms = sycl::platform::get_platforms();
for (const auto &plat : platforms) {
    // get_info is a template. So we pass the type as an 'arguments'.
    std::cout << "Platform: "
    << plat.get_info<sycl::info::platform::name>() << " "
    << plat.get_info<sycl::info::platform::vendor>() << " "
    << plat.get_info<sycl::info::platform::version>()
    << '\n';
```

Then for each platform we list the devices:

```
std::vector<sycl::device> devices = plat.get_devices();
for (const auto &dev : devices) {
    std::cout << "-- Device: "
    << dev.get_info<sycl::info::device::name>()
    // << (dev.is_host() ? ": is the host" : "")
    << (dev.is_cpu() ? ": is a cpu" : "")
    << (dev.is_gpu() ? ": is a gpu" : "")
    << std::endl;
```

You can query what type of device you are dealing with by *is_cpu*, *is_gpu*. (The function *is_host* was deprecated in SYCL-2020.)

41.3 Queues

The execution mechanism of SYCL is the *queue*: a sequence of actions that will be executed on a selected device. The only user action is submitting actions to a queue; the queue is executed at the end of the scope where it is declared.

Queue execution is asynchronous with host code.

41.3.1 Device selectors

You need to select a device on which to execute the queue. A single queue can only dispatch to a single device.

A queue is coupled to one specific device, so it can not spread work over multiple devices. You can find a default device for the queue with

```
sycl::queue myqueue;
```

The following example explicitly assigns the queue to the CPU device using the *sycl::cpu_selector*.

```
// cpuname.cxx
sycl::queue myqueue( sycl::cpu_selector_v );
```

Remark Pre-SYCL-2020: the `sycl::host_selector` bypasses any devices and make the code run on the host.

`cpu_selector` is deprecated in SYCL-2020, replaced by `cpu_selector_v`.

It is good for your sanity to print the name of the device you are running on:

```
// devname.cxx
std::cout << myqueue.get_device().get_info<sycl::info::device::name>()
    << std::endl;
```

If you try to select a device that is not available, a `sycl::runtime_error` exception will be thrown.

Intel DPC++ extension.

```
#include "CL/sycl/intel/fpga_extensions.hpp"
fpga_selector
```

41.3.2 Queue submission and execution

It seems that queue kernels will also be executed when only they go out of scope, but not the queue:

```
// doubler.cxx
sycl::range<1> mySize{SIZE};
sycl::buffer<int, 1> bufferA(myArray.data(), mySize);
myqueue.submit
( [&] (sycl::handler &myHandle) {
    auto deviceAccessorA =
        bufferA.get_access<sycl::access::mode::read_write>(myHandle);
} // queue goes out of scope, executes
```

41.3.3 Kernel ordering

Kernels are not necessarily executed in the order in which they are submitted. You can enforce this by specifying an *in-order queue*:

```
sycl::queue myqueue{property::queue::inorder()};
```

41.4 Kernels

One kernel per submit.

```
myqueue.submit( [&] ( handler &commandgroup ) {
    commandgroup.parallel_for<uniquename>
    ( range<1>{N},
      [=] ( id<1> idx ) { ... idx }
    )
});
```

Note that the lambda in the kernel captures by value. Capturing by reference makes no sense, since the kernel is executed on a device.

```
cgh.single_task(
[=]() {
    // kernel function is executed EXACTLY once on a SINGLE work-item
});
```

The `submit` call results in an event object:

```
auto myevent = myqueue.submit( /* stuff */ );
```

This can be used for two purposes:

1. It becomes possible to wait for this specific event:

```
myevent.wait();
```

2. It can be used to indicate kernel dependencies:

```
myqueue.submit( [=] (handler &h) {
    h.depends_on(myevent);
    /* stuff */
} );
```

41.5 Parallel operations

41.5.1 Loops

```
cgh.parallel_for(
    range<3>(1024,1024,1024),
    // using 3D in this example
    [=](id<3> myID) {
        // kernel function is executed on an n-dimensional range (NDrange)
    });

cgh.parallel_for(
    nd_range<3>( {1024,1024,1024}, {16,16,16} ),
    // using 3D in this example
    [=](nd_item<3> myID) {
        // kernel function is executed on an n-dimensional range (NDrange)
    });

cgh.parallel_for_work_group(
    range<2>(1024,1024),
    // using 2D in this example
    [=](group<2> myGroup) {
        // kernel function is executed once per work-group
    });

grp.parallel_for_work_item(
    range<1>(1024),
    // using 1D in this example
    [=](h_item<1> myItem) {
        // kernel function is executed once per work-item
    });

```

In SYCL-2020 offsets on `nd_range` are deprecated.

41.5.1.1 Loop bounds: ranges

SYCL adopts the modern C++ philosophy that one does not iterate over by explicitly enumerating indices, but by indicating their range. This is realized by the `range` class, which is templated over the number of space dimensions.

```
sycl::range<2> matrix{10,10};
```

Some compilers are sensitive to the type of the integer arguments:

```
sycl::range<1> array{ static_cast<size_t>(size) } ;
```

41.5.1.2 Loop indices

Kernels such as `parallel_for` expects two arguments:

- a `range` over which to index; and
- a lambda of one argument: an index.

There are several ways of indexing. The `id<nd>` class of multi-dimensional indices.

```
myHandle.parallel_for<class uniqueID>
( mySize,
  [=]( id<1> index ) {
    float x = index.get(0) * h;
    deviceAccessorA[index] *= 2. ;
  }
)

cgh.parallel_for<class foo>(
  range<1>{D*D*D},
  [=](id<1> item) {
    xx[ item[0] ] = 2 * item[0] + 1;
  }
)
```

While the C++ vectors remain one-dimensional, DPCPP allows you to make multi-dimensional buffers:

```
std::vector<int> y(D*D*D);
buffer<int,1> y_buf(y.data(), range<1>(D*D*D));
cgh.parallel_for<class foo2D>
( range<2>{D,D*D},
  [=](id<2> item) {
    yy[ item[0] + D*item[1] ] = 2;
  }
);
```

Intel DPC++ extension. There is an implicit conversion from the one-dimensional `sycl::id<1>` to `size_t`, so

```
[=](sycl::id<1> i) {
  data[i] = i;
}
```

is legal, which in SYCL requires

```
data[i[0]] = i[0];
```

41.5.1.3 Multi-dimensional indexing

```
// stencil2d.cxx
sycl::range<2> stencil_range(N, M);
sycl::range<2> alloc_range(N + 2, M + 2);
std::vector<float>
    input(alloc_range.size()),
    output(alloc_range.size());
sycl::buffer<float, 2> input_buf(input.data(), alloc_range);
sycl::buffer<float, 2> output_buf(output.data(), alloc_range);

constexpr size_t B = 4;
sycl::range<2> local_range(B, B);
sycl::range<2> tile_range = local_range + sycl::range<2>(2, 2); // Includes boundary cells
auto tile = local_accessor<float, 2>(tile_range, h); // see templated def'n above
```

We first copy global data into an array local to the work group:

```
sycl::id<2> offset(1, 1);
h.parallel_for
( sycl::nd_range<2>(stencil_range, local_range, offset),
 [=] ( sycl::nd_item<2> it ) {
// Load this tile into work-group local memory
    sycl::id<2> lid = it.get_local_id();
    sycl::range<2> lrange = it.get_local_range();
    for (int ti = lid[0]; ti < B + 2; ti += lrange[0]) {
        for (int tj = lid[1]; tj < B + 2; tj += lrange[1]) {
            int gi = ti + B * it.get_group(0);
            int gj = tj + B * it.get_group(1);
            tile[ti][tj] = input[gi][gj];
        }
    }
}
```

Global coordinates in the input are computed from the `nd_item`'s coordinate and group:

```
[=] ( sycl::nd_item<2> it ) {
for (int ti ... ) {
    for (int tj ... ) {
        int gi = ti + B * it.get_group(0);
        int gj = tj + B * it.get_group(1);
        ... = input[gi][gj];
```

Local coordinates in the tile, including boundary, I DON'T QUITE GET THIS YET.

```
[=] ( sycl::nd_item<2> it ) {
sycl::id<2> lid = it.get_local_id();
sycl::range<2> lrange = it.get_local_range();
for (int ti = lid[0]; ti < B + 2; ti += lrange[0]) {
    for (int tj = lid[1]; tj < B + 2; tj += lrange[1]) {
        tile[ti][tj] = ..
```

41.5.2 Task dependencies

Each `submit` call can be said to correspond to a ‘task’. Since it returns a token, it becomes possible to specify task dependencies by referring to a token as a dependency in a later specified task.

```
queue myQueue;
auto myTokA = myQueue.submit
( [&](handler& h) {
    h.parallel_for<class taskA>(...);
}
);
auto myTokB = myQueue.submit
( [&](handler& h) {
    h.depends_on(myTokA);
    h.parallel_for<class taskB>(...);
}
);
```

41.5.3 Race conditions

Sycl has the same problems with *race conditions* that other shared memory system have:

```
// sum1d.cxx
auto array_accessor =
array_buffer.get_access<sycl::access::mode::read>(h);
auto scalar_accessor =
scalar_buffer.get_access<sycl::access::mode::read_write>(h);
h.parallel_for<class uniqueID>
( array_range,
[=](sycl::id<1> index)
{
    scalar_accessor[0] += array_accessor[index];
}
); // end of parallel for
```

To get this working correctly would need either a reduction primitive or atomics on the accumulator. The 2020 proposed standard has improved atomics.

```
// reduct1d.cxx
auto input_values = array_buffer.get_access<sycl::access::mode::read>(h);
auto sum_reduction = sycl::reduction( scalar_buffer,h, std::plus<>() );
h.parallel_for
( array_range,sum_reduction,
[=]( sycl::id<1> index,auto& sum )
{
    sum += input_values[index];
}
); // end of parallel for
```

41.5.4 Reductions

Reduction operations were added in the the SYCL 2020 Provisional Standard, meaning that they are not yet finalized.

Here is one approach, which works in *hipsycl*:

```
// reductscalar.cxx
auto reduce_to_sum =
    sycl::reduction( sum_array, static_cast<float>(0.), std::plus<float>() );
myqueue.parallel_for// parallel_for<reduction_kernel<T,BinaryOp,__LINE__>>
( array_range, // sycl::range<1>(input_size),
    reduce_to_sum, // sycl::reduction(output, identity, op),
    [=] (sycl::id<1> idx, auto& reducer) { // type of reducer is impl-dependent, so use auto
        reducer.combine(shared_array[idx[0]]); // (input[idx[0]]);
    //reducer += shared_array[idx[0]]; // see line 216: add_reducer += input0[idx[0]];
} ).wait();
```

Here a `sycl::reduction` object is created from the target data and the reduction operator. This is then passed to the `parallel_for` and its `combine` method is called.

41.6 Memory access

Memory treatment in SYCL is a little complicated, because is (at the very least) host memory and device memory, which are not necessarily coherent.

There are also three mechanisms:

- Unified Shared Memory, based on ordinary C/C++ ‘star’-pointers.
- Buffers, using the `buffer` class; this needs the `accessor` class to access the data.
- Images.

Table 41.1: Memory types and treatments

Location	allocation	coherence	copy to/from device
Host	<code>malloc</code>	explicit transfer	<code>queue::memcpy</code>
	<code>malloc_host</code>	coherent host/device	
Device	<code>malloc_device</code>	explicit transfer	<code>queue::memcpy</code>
Shared	<code>malloc_shared</code>	coherent host/device	

41.6.1 Unified shared memory

Memory allocated with `malloc_host` is visible on the host:

```
// outshared.cxx
floattype
*host_float = (floattype*)malloc_host( sizeof(floattype),ctx ),
*shar_float = (floattype*)malloc_shared( sizeof(floattype),dev,ctx );
cgh.single_task
( [=] () {
    shar_float[0] = 2 * host_float[0];
    sout << "Device sets " << shar_float[0] << sycl::endl;
} );
```

Device memory is allocated with `malloc_device`, passing the queue as parameter:

```
// reductimpl.cxx
floattype
*host_float = (floattype*)malloc( sizeof(floattype) ),
*devc_float = (floattype*)malloc_device( sizeof(floattype), dev, ctx );
[&](sycl::handler &cgh) {
    cgh.memcpy(devc_float, host_float, sizeof(floattype));
}
```

Note the corresponding `free` call that also has the queue as parameter.

Note that you need to be in a parallel task. The following gives a segmentation error:

```
[&](sycl::handler &cgh) {
    shar_float[0] = host_float[0];
}
```

Ordinary memory, for instance from `malloc`, has to be copied in a kernel:

```
[&](sycl::handler &cgh) {
    cgh.memcpy(devc_float, host_float, sizeof(floattype));
}
[&](sycl::handler &cgh) {
    sycl::stream sout(1024, 256, cgh);
    cgh.single_task(
        [&] () {
            sout << "Number " << devc_float[0] << sycl::endl;
        }
    );
}
} // end of submitted lambda
free(host_float);
sycl::free(devc_float, myqueue);
```

41.6.2 Buffers and accessors

Arrays need to be declared in a way such that they can be accessed from any device.

```
// forloop.cxx
std::vector<int> myArray(SIZE);
sycl::range<1> mySize{myArray.size()};
sycl::buffer<int, 1> bufferA(myArray.data(), myArray.size());
```

Remark `sycl::range` takes a `size_t` parameter; specifying an `int` may give a compiler warning about a narrowing conversion.

Inside the kernel, the array is then unpacked from the buffer:

```
myqueue.submit( [&] ( sycl::handler &h ) {
    auto deviceAccessorA =
        bufferA.get_access<sycl::access::mode::read_write>(h);
```

However, the `get_access` function results in a `sycl::accessor`, not a pointer to a simple type. The precise type is templated and complicated, so this is a good place to use `auto`.

Accessors can have a mode associated: `sycl::access::mode::read` `sycl::access::mode::write`

Intel DPC++ extension.

```
array<floattype,1> leftsum{0.};
#ifndef __INTEL_CLANG_COMPILER
    sycl::buffer leftbuf(leftsum);
#else
    sycl::range<1> scalar{1};
    sycl::buffer<floattype,1> leftbuf(leftsum.data(),scalar);
```

Intel DPC++ extension. There are modes

```
// standard
sycl::accessor acc = buffer.get_access<sycl::access::mode::write>(h);
// dpcpp extension
sycl::accessor acc( buffer,h,sycl::read_only );
sycl::accessor acc( buffer,h,sycl::write_only );
```

41.6.2.1 Multi-D buffers

To create a multi-dimensional buffer object, use a `sycl::range` to specify the dimensions:

```
// jordan.cxx
vector<double> matrix(vecsize*vecsize);
sycl::range<2> mat_range{vecsize,vecsize};
sycl::buffer<double,2> matrix_buffer( matrix.data(),mat_range );
```

41.6.3 Querying

The function `get_range` can query the size of either a buffer or an accessor:

```
// range2.cxx
sycl::buffer<int, 2>
a_buf(a.data(), sycl::range<2>(N, M)),
b_buf(b.data(), sycl::range<2>(N, M)),
c_buf(c.data(), sycl::range<2>(N, M));

sycl::range<2>
a_range = a_buf.get_range(),
b_range = b_buf.get_range();

if (a_range==b_range) {

    sycl::accessor c = c_buf.get_access<sycl::access::mode::write>(h);

    sycl::range<2> c_range = c.get_range();
    if (a_range==c_range) {
        h.parallel_for
        ( a_range,
          [=]( sycl::id<2> idx ) {
            c[idx] = a[idx] + b[idx];
          } );
    }
}
```

41.7 Parallel output

There is a `sycl::cout` and `sycl::endl`.

```
// hello.cxx
[&](sycl::handler &cgh) {
    sycl::stream sout(1024, 256, cgh);
    cgh.parallel_for<class hello_world>
    (
        sycl::range<1>(global_range), [=](sycl::id<1> idx) {
            sout << "Hello, World: World rank " << idx << sycl::endl;
        });
    } // End of the kernel function
}
```

Since the end of a queue does not flush stdout, it may be necessary to call `sycl::queue::wait`
`myQueue.wait();`

41.8 Other

Exceptions:

```
try {
    sycl::whatever
} catch ( sycl::errc::runtime &e ) { .... }
```

Deprecated as of SYCL-2020:

```
} catch ( sycl::runtime_error &e ) { ... }
```

41.9 DPCPP extensions

Intel has made some extensions to SYCL:

- Unified Shared Memory,
- Ordered queues.

41.10 Intel devcloud notes

`qsub -I` for interactive session.

`gdb-oneapi` for debugging.

<https://community.intel.com/t5/Intel-oneAPI-Toolkits/ct-p/oneapi> for support.

41.11 Examples

41.11.1 Kernels in a loop

The following idiom works:

```
sycl::event last_event = queue.submit( [&] (sycl::handler &h) {
    for (int iteration=0; iteration<N; iteration++) {
        last_event = queue.submit( [&] (sycl::handler &h) {
            h.depends_on(last_event);
```

41.11.2 Stencil computations

The problem with stencil computations is that only interior points are updated. Translated to SYCL: we need to iterate over a subrange of the range over which the buffer is defined. First let us define these ranges:

```
// jacobi1d.cxx
sycl::range<1> unknowns(N);
sycl::range<1> with_boundary(N + 2);
std::vector<float>
old_values(with_boundary.size(), 0.),
new_values(with_boundary.size(), 0.);
old_values.back() = 1.; new_values.back() = 1.;
```

Note the boundary value 1. on the right boundary.

Restricting the iteration to the interior points is done through the *offset* parameter of the *parallel_for*:

```
sycl::id<1> offset(1);
h.parallel_for
(
    unknowns, offset,
    [=] (sycl::id<1> idx) {
        int i = idx[0];
        float self = old_array[i];
        float left = old_array[i - 1];
        float right = old_array[i + 1];
        new_array[i] = (self + left + right) / 3.0f;
    });
}
```

Chapter 42

CUDA

42.1 Host and device

Analogous to how OpenMP marked certain regions as parallel, a CUDA program marks certain function calls as executable on a device.

- Host: the CPU where the execution starts
- Device: one of possibly several attached GPUs.

Device code is recognizable by a keyword prefix, either `__global__` or `__device__`. The `__host__` keyword marks host code, but that's the default.

```
--global__ void some_cuda_function( /* parameters */ );
```

Device code needs have type `void`: return results have to be passed explicitly through memory.

42.1.1 Hello world

A device function that prints hello:

```
// hello.cu
__global__ void hello_cuda() {
    printf("hello world!\n");
}
```

And the main program launches this kernel on a grid of one block of size 1.

```
hello_cuda<<<1,1>>>();
cudaDeviceSynchronize();
```

We need the `cudaDeviceSynchronize` function so that the host waits until all device actions are completed. Normally, host and device actions can take place concurrently; see figure 42.1.

The ‘triple chevron’ syntax indicates how many block, and how many threads per block, are used to launch the kernel. You can put a 2D or 3D structure on the blocks by using the `dim3` coordinate structure:

```
int nx=6,ny=6;
dim3 block(2,2);
dim3 grid( nx/block.x,ny/block.y );
hello_cuda<<<grid,block>>>();
cudaDeviceSynchronize();
```

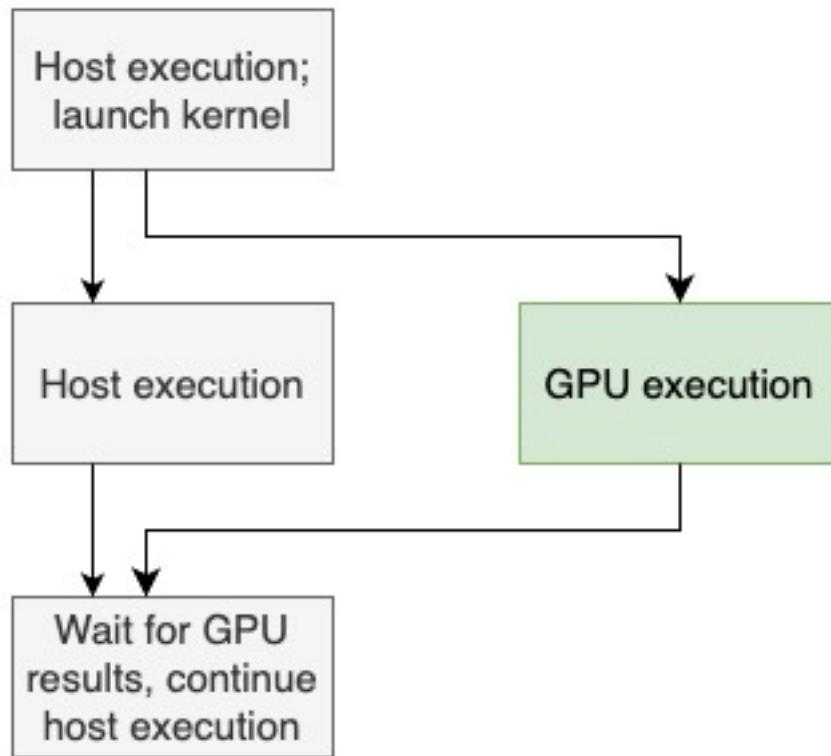


Figure 42.1: Host and device activity

The coordinate of the block and the thread within the block can now be found through the `blockIdx` and `threadIdx` coordinate structures.

While the number of blocks can be large, the number of threads is limited. The x, y sides of the block cannot exceed 1024, and the z side can not exceed 64; the total size of the block is limited to 1024.

42.2 Architecture

You have seen the basic design of a grid of blocks, where each block is a set of threads. The blocks are distributed among the Streaming Multiprocessors (SMs), but even a single block can have more threads than the SM has cores, so there is a further division in *warps*. A warp is typically 32 threads.

42.2.1 Thread indexing

We need a way of identifying which CUDA thread executes a kernel. Each kernel execution can query the (globally defined) variables `blockIdx` and `threadIdx`.

It is possible to launch a kernel on more threads than there are data points. In that case, your code could include

```
int global_thread_id = blockIdx.x * blockDim.x + threadIdx.x; // or 2d, 3d variant
if ( global_thread_id > datasize ) return;
```

42.2.2 Data transfer

The host and device have separate memory, so you may need to synchronize host data to the device.

42.2.2.1 Explicit copy

One strategy is allocate data separately for host and device.

Device memory is allocated with `cudaMalloc`. While the resulting pointer is visible to the host code, the actual memory is on the device and you need to transfer data explicitly between host and device with `cudaMemcpy`. This transfer can be between host and device in either direction, or between two devices.

```
cudaMemcpy( dest_ptr, orig_ptr, byte_size, direction );
// direction:
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice
```

Example:

```
// add1.cu
float
*x, *y, *z;
size_t nbytes = N*sizeof(float);
x = (float*)malloc(nbytes);
y = (float*)malloc(nbytes);
z = (float*)malloc(nbytes);
float
*dev_x, *dev_y, *dev_z;
cudaMalloc( &dev_x,nbytes );
cudaMalloc( &dev_y,nbytes );
cudaMalloc( &dev_z,nbytes );
```

The data is then copied to the GPU:

```
cudaMemcpy( dev_x,x,nbytes,cudaMemcpyHostToDevice );
cudaMemcpy( dev_y,y,nbytes,cudaMemcpyHostToDevice );
```

and after the kernel execution resulting data is copied back:

```
cudaMemcpy( z,dev_z,nbytes,cudaMemcpyDeviceToHost );
```

42.2.2.2 Managed memory

You can also allocate memory in a more abstract way: ‘managed memory’ can be written on the host, but accessed from the device. The actual mechanism that moves data back and forth is based on *paging*.

```
// add1m.cu
size_t nbytes = N*sizeof(float);
CU_ASSERT( cudaMallocManaged( &x,nbytes ) );
CU_ASSERT( cudaMallocManaged( &y,nbytes ) );
CU_ASSERT( cudaMallocManaged( &z,nbytes ) );
```

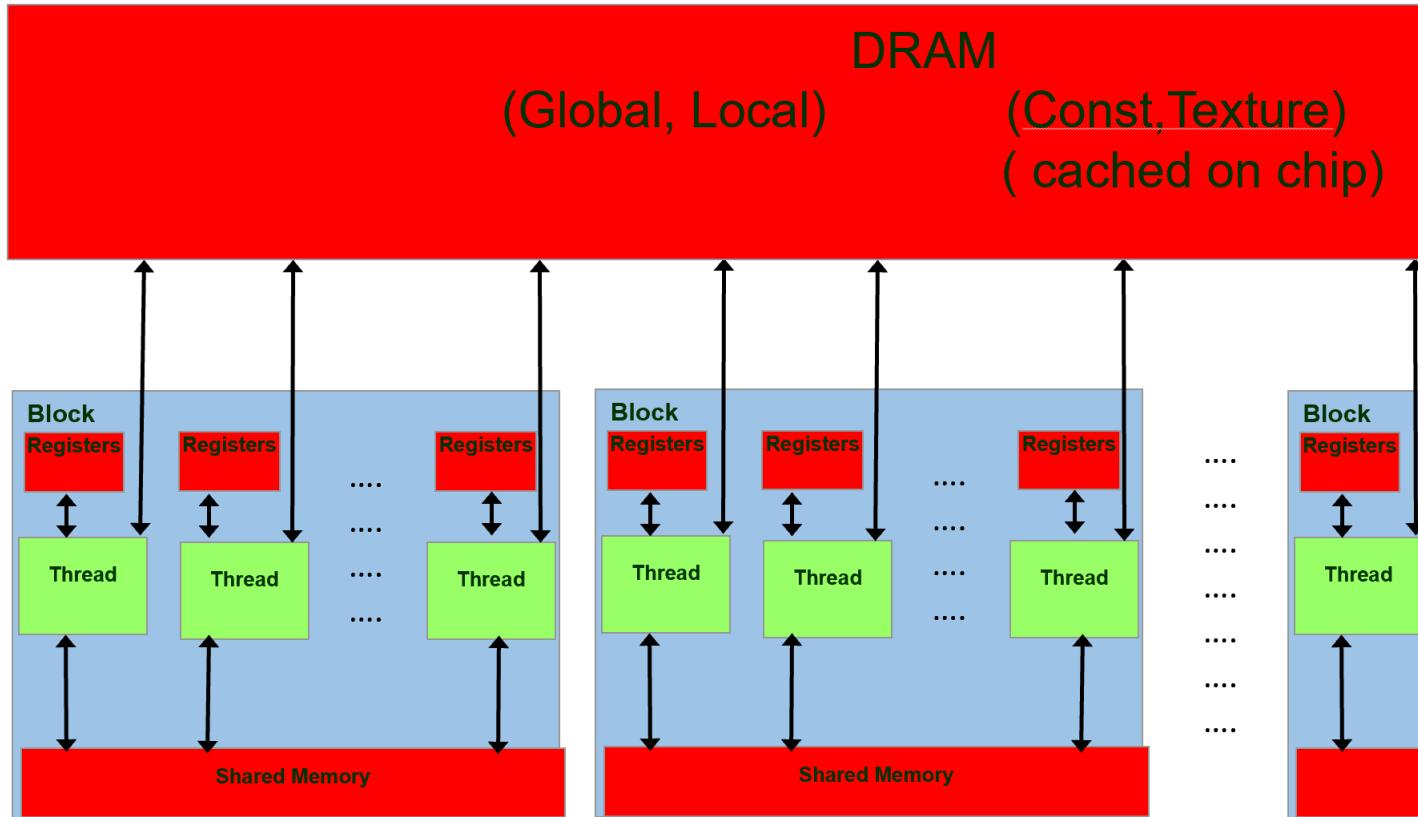


Figure 42.2: GPU memory structure

42.2.3 Shared memory

The way we have coded so far, data is streamed from the main memory of the GPU. For higher performance it is necessary to use the *shared memory* of the threads in a block. See figure 42.2.

Shared memory is declared by adding the keyword `__shared__` to a regular array declaration:

```
__shared__ float tmp[BLOCKSIZE];
```

42.3 Querying

Use `cudaGetDeviceCount` to report how many GPUs are attached:

```
// device.cu
int ndev;
auto status = cudaGetDeviceCount(&ndev);
```

Possible errors: `cudaErrorNoDevice` if there are no devices, and `cudaErrorInsufficientDriver` if the driver is too old.

Use `cudaGetDeviceProperties` to return a `cudaDeviceProp` object:

MISSING SNIPPET `cudevprop` in `codesnippetsdir=snippets`

Here are some of the properties:

MISSING SNIPPET `cudevprops` in `codesnippetsdir=snippets`

42.4 Performance

The term *warp divergence* describes the situation that the threads in a warp do not execute the same instruction.

```
if ( threadIdx.x%2==0 )
    // do something
else
    // do something else
```

Since threads in a warp are tightly synchronized this has a performance penalty.

42.4.1 Profiling

`ncu` <https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html>

Remark Legacy: `nvprof`.

`nvprof --metrics branch_efficiency your_program`

Chapter 43

Python multiprocessing

Python has a *multiprocessing* toolbox. This is a parallel processing library that relies on subprocesses, rather than threads.

43.1 Software and hardware

The *multiprocessing* toolbox does its own hardware detection; it uses a single node, and however many cores the system tells it there are.

```
## pool.py
nprocs = mp.cpu_count()
print(f"I detect {nprocs} cores")
```

43.2 Process

A process is an object that will execute a python function:

```
## quicksort.py
import multiprocessing as mp
import random
import os

def quicksort( numbers ) :
    if len(numbers)==1:
        return numbers
    else:
        median = numbers[0]
        left = [ i for i in numbers if i<median ]
        right = [ i for i in numbers if i>=median ]
        with mp.Pool(2) as pool:
            [sortleft,sortright] = pool.map( quicksort,[left,right] )
        return sortleft.append( sortright )

if __name__ == '__main__':
    numbers = [ random.randint(1,50) for i in range(32) ]
    process = mp.Process(target=quicksort,args=[numbers])
```

43. Python multiprocessing

```
process.start()  
process.join()
```

Creating a process does not start it: for that use the `start` function. Execution of the process is not guaranteed until you call the `join` function on it:

```
if __name__ == '__main__':  
    for p in processes:  
        p.start()  
    for p in processes:  
        p.join()
```

By making the start and join calls less regular than in a loop like this, arbitrarily complicated code can be produced.

43.2.1 Arguments

Arguments can be passed to the function of the process with the `args` keyword. This accepts a list (or tuple) of arguments, leading to a somewhat strange syntax for a single argument:

```
proc = Process(target=print_func, args=(name,))
```

43.2.2 Process details

Note the test on `__main__`: the processes started read the current file in order to execute the function specified. Without this clause, the import would first execute more process start calls, before getting to the function execution.

Processes have a name that you can retrieve as `current_process().name`. The default is `Process-5` and such, but you can specify custom names:

```
Process(name="Your name here")
```

The target function of a process can get hold of that process with the `current_process` function.

Of course you can also query `os.getpid()` but that does not offer any further possibilities.

```
def say_name(iproc):  
    print(f"Process {os.getpid()} has name: {mp.current_process().name}")  
if __name__ == '__main__':  
    processes = [ mp.Process(target=say_name, name=f"proc{iproc}", args=[iproc])  
                 for iproc in range(nprocs) ]
```

43.3 Pools and mapping

Often you want a number of processes to do apply to a number of arguments, for instance in a *parameter sweep*. For this, create a `Pool` object, and apply the `map` method to it:

```
pool = mp.Pool( nprocs )  
results = pool.map( print_value, range(1,2*nprocs) )
```

Note that this is also the easiest way to get return values from a process, which is not directly possible with a `Process` object. Other approaches are using a shared object, or an object in a `Queue` or `Pipe` object; see below.

43.4 Shared data

The best way to deal with shared data is to create a *Value* or *Array* object, which come equipped with a lock for safe updating.

```
pi = mp.Value('d')
pi.value = 0
```

For instance, one could stochastically calculate π by

1. generating random points in $[0, 1]^2$, and
2. recording how many fall in the unit circle, after which
3. π is $4 \times$ the ratio between points in the circle and the total number of points.

```
## pi.py
def calc_pi(pi,n):
    for i in range(n):
        x = random.random()
        y = random.random()
        with pi.get_lock():
            if x*x+y*y<1:
                pi.value += 1.
```

Exercise 43.1. Do you see a way to improve the speed of this calculation?

43.4.1 Pipes

A *pipe*, object type *Pipe*, corresponds to what used to be called a *channel* in older parallel programming systems: a First-in-first-out (FIFO) object into which one process can place items, and from which another process can take them. However, a pipe is not associated with any particular pair: creating the pipe gives the entrance and exit from the pipe

```
q_entrance,q_exit = mp.Pipe()
```

And they can be passed to any process

```
producer1 = mp.Process(target=add_to_pipe,args=([1,q_entrance]))
producer2 = mp.Process(target=add_to_pipe,args=([2,q_entrance]))
printer = mp.Process(target=print_from_pipe,args=(q_exit,))
```

which can then can put and get items, using the `send` and `recv` commands.

```
## pipemulti.py
def add_to_pipe(v,q):
    for i in range(10):
        print(f"put {v}")
        q.send(v)
        time.sleep(1)
    q.send("END")
```

```
def print_from_pipe(q):
    ends = 0
    while True:
        v = q.recv()
        print(f"Got: {v}")
        if v=="END":
            ends += 1
        if ends==2:
            break
    print("pipe is empty")
```

43.4.2 Queues

PART V

THE REST

Chapter 44

Exploring computer architecture

There is much that can be said about computer architecture. However, in the context of parallel programming we are mostly concerned with the following:

- How many networked nodes are there, and does the network have a structure that we need to pay attention to?
- On a compute node, how many sockets (or other Non-Uniform Memory Access (NUMA) domains) are there?
- For each socket, how many cores and hyperthreads are there? Are caches shared?

44.1 Tools for discovery

An easy way for discovering the structure of your parallel machine is to use tools that are written especially for this purpose.

44.1.1 Intel cpufreq

The *Intel compiler suite* comes with a tool *cpufreq* that reports on the structure of the node you are running on. It reports on the number of *packages*, that is: sockets, cores, and threads.

44.1.2 hwloc

The open source package *hwloc* does similar reporting to *cpufreq*, but it has been ported to many platforms. Additionally, it can generate ascii and pdf graphic renderings of the architecture.

Chapter 45

Hybrid computing

So far, you have learned to use MPI for distributed memory and OpenMP for shared memory parallel programming. However, distributed memory architectures actually have a shared memory component, since each cluster node is typically of a multicore design. Accordingly, you could program your cluster using MPI for inter-node and OpenMP for intra-node parallelism.

You now have to find the right balance between processes and threads, since each can keep a core fully busy. Complicating this story, a node can have more than one *socket*, and corresponding *NUMA* domain. Figure 45.1

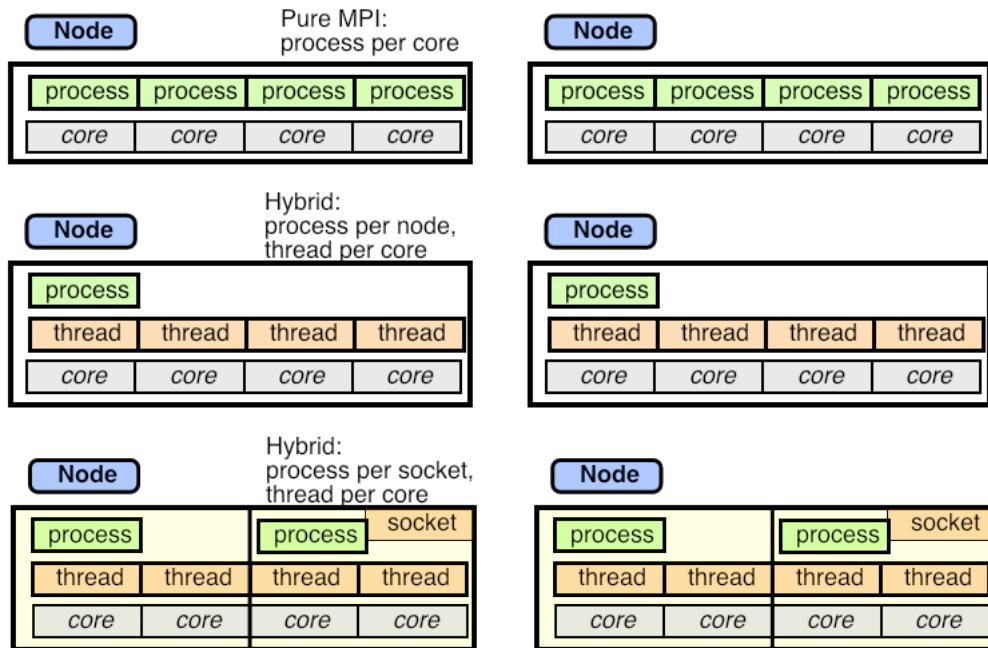


Figure 45.1: Three modes of MPI/OpenMP usage on a multi-core cluster

illustrates three modes: pure MPI with no threads used; one MPI process per node and full multi-threading; two MPI processes per node, one per socket, and multiple threads on each socket.

45.1 Concurrency

With hybrid multi-process / multi-thread computing, one thing that goes out the door is the sequential semantics of each MPI process. For instance, the fact that messages between a single sender and a single receiver are *non-overtaking* no longer holds if the messages originated in different threads.

```
// anytag.c
#pragma omp parallel sections
{
#pragma omp section
MPI_Isend
( &x,1,MPI_DOUBLE,
  receiver,xtag,comm,requests+0);
#pragma omp section
MPI_Isend
( &y,1,MPI_DOUBLE,
  receiver,ytag,comm,requests+1);
}
MPI_Waitall(2,requests,MPI_STATUSES_IGNORE);

#pragma omp section
MPI_Irecv
( &xy1,1,MPI_DOUBLE,
  sender, MPI_ANY_TAG, comm, requests+0);
#pragma omp section
MPI_Irecv
( &xy2,1,MPI_DOUBLE,
  sender, MPI_ANY_TAG, comm, requests+1);
}
MPI_Waitall(2,requests,statuses);
```

Messages from simultaneous threads are said to be *concurrent*: there is no temporal or causal relationship between them.

45.2 Affinity

In the preceding chapters we mostly considered all MPI nodes or OpenMP thread as being in one flat pool. However, for high performance you need to worry about *affinity*: the question of which process or thread is placed where, and how efficiently they can interact.

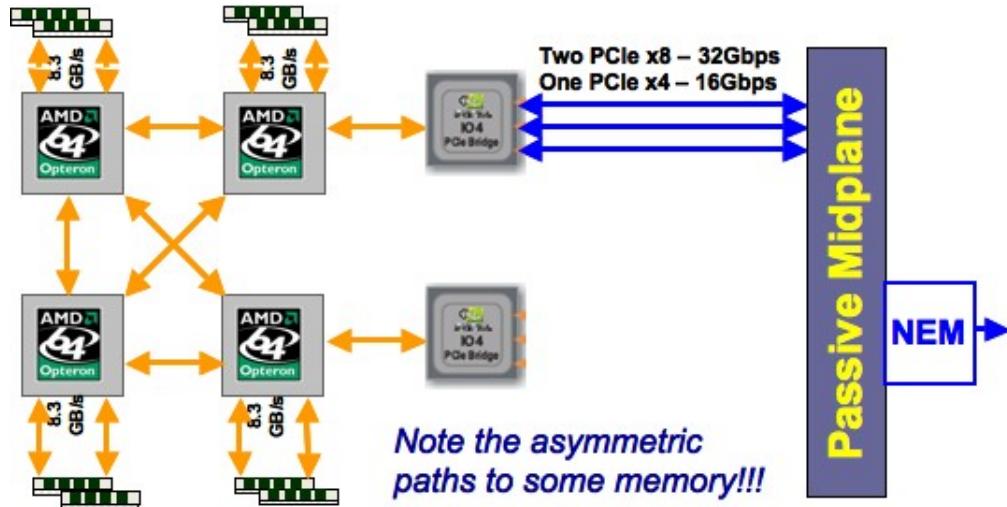


Figure 45.2: The NUMA structure of a Ranger node

Here are some situations where you affinity becomes a concern.

- In pure MPI mode processes that are on the same node can typically communicate faster than processes on different nodes. Since processes are typically placed sequentially, this means that a scheme where process p interacts mostly with $p + 1$ will be efficient, while communication with large jumps will be less so.
- If the cluster network has a structure (*processor grid* as opposed to *fat-tree*), placement of processes has an effect on program efficiency. MPI tries to address this with *graph topology*; section 11.2.
- Even on a single node there can be asymmetries. Figure 45.2 illustrates the structure of the four sockets of the *Ranger* supercomputer (no longer in production). Two cores have no direct connection. This asymmetry affects both MPI processes and threads on that node.
- Another problem with multi-socket designs is that each socket has memory attached to it. While every socket can address all the memory on the node, its local memory is faster to access. This asymmetry becomes quite visible in the *first-touch* phenomenon; section 25.2.
- If a node has fewer MPI processes than there are cores, you want to be in control of their placement. Also, the operating system can migrate processes, which is detrimental to performance since it negates data locality. For this reason, utilities such as *numactl* (and at TACC *tacc_affinity*) can be used to *pin a thread* or process to a specific core.
- Processors with *hyperthreading* or *hardware threads* introduce another level of worry about where threads go.

45.3 What does the hardware look like?

If you want to optimize affinity, you should first know what the hardware looks like. The *hwloc* utility is valuable here [11] (<https://www.open-mpi.org/projects/hwloc/>).

Figure 45.3 depicts a *Stampede compute node*, which is a two-socket *Intel Sandybridge* design; figure 45.4 shows a *Stampede largemem node*, which is a four-socket design. Finally, figure 45.5 shows a *Lonestar5* compute node, a two-socket design with 12-core *Intel Haswell* processors with two hardware threads each.

45.4 Affinity control

See chapter 25 for OpenMP affinity control.

45.5 Discussion

The performance implications of the pure MPI strategy versus hybrid are subtle.

- First of all, we note that there is no obvious speedup: in a well balanced MPI application all cores are busy all the time, so using threading can give no immediate improvement.
- Both MPI and OpenMP are subject to Amdahl's law that quantifies the influence of sequential code; in hybrid computing there is a new version of this law regarding the amount of code that is MPI-parallel, but not OpenMP-parallel.
- MPI processes run unsynchronized, so small variations in load or in processor behavior can be tolerated. The frequent barriers in OpenMP constructs make a hybrid code more tightly synchronized, so load balancing becomes more critical.
- On the other hand, in OpenMP codes it is easier to divide the work into more tasks than there are threads, so statistically a certain amount of load balancing happens automatically.
- Each MPI process has its own buffers, so hybrid takes less buffer overhead.

45. Hybrid computing

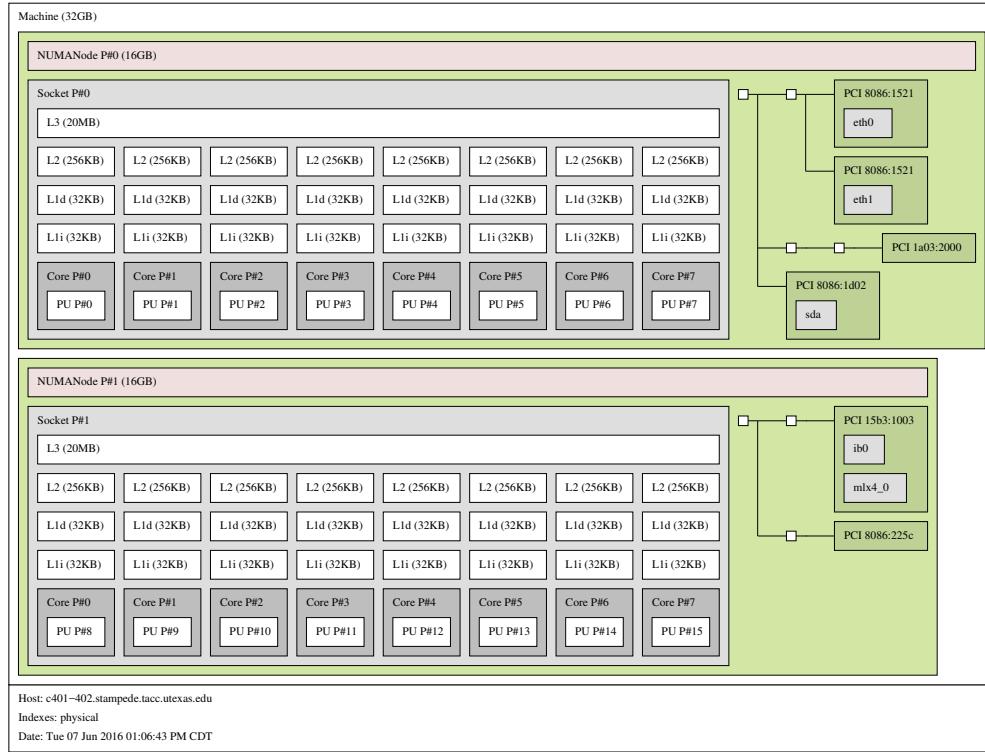


Figure 45.3: Structure of a Stampede compute node

Exercise 45.1. Review the scalability argument for 1D versus 2D matrix decomposition in HPC book, section ???. Would you get scalable performance from doing a 1D decomposition (for instance, of the rows) over MPI processes, and decomposing the other directions (the columns) over OpenMP threads?

Another performance argument we need to consider concerns message traffic. If let all threads make MPI calls (see section 13.1) there is going to be little difference. However, in one popular hybrid computing strategy we would keep MPI calls out of the OpenMP regions and have them in effect done by the master thread. In that case there are only MPI messages between nodes, instead of between cores. This leads to a decrease in message traffic, though this is hard to quantify. The number of messages goes down approximately by the number of cores per node, so this is an advantage if the average message size is small. On the other hand, the amount of data sent is only reduced if there is overlap in content between the messages.

Limiting MPI traffic to the master thread also means that no buffer space is needed for the on-node communication.

45.6 Processes and cores and affinity

In OpenMP, threads are purely a software construct and you can create however many you want. The hardware limit of the available cores can be queried with `omp_get_num_procs` (section 17.5). How does that work in a hybrid context? Does the ‘proc’ count return the total number of cores, or does the MPI scheduler limit it to a number exclusive to each MPI process?

The following code fragment explore this:

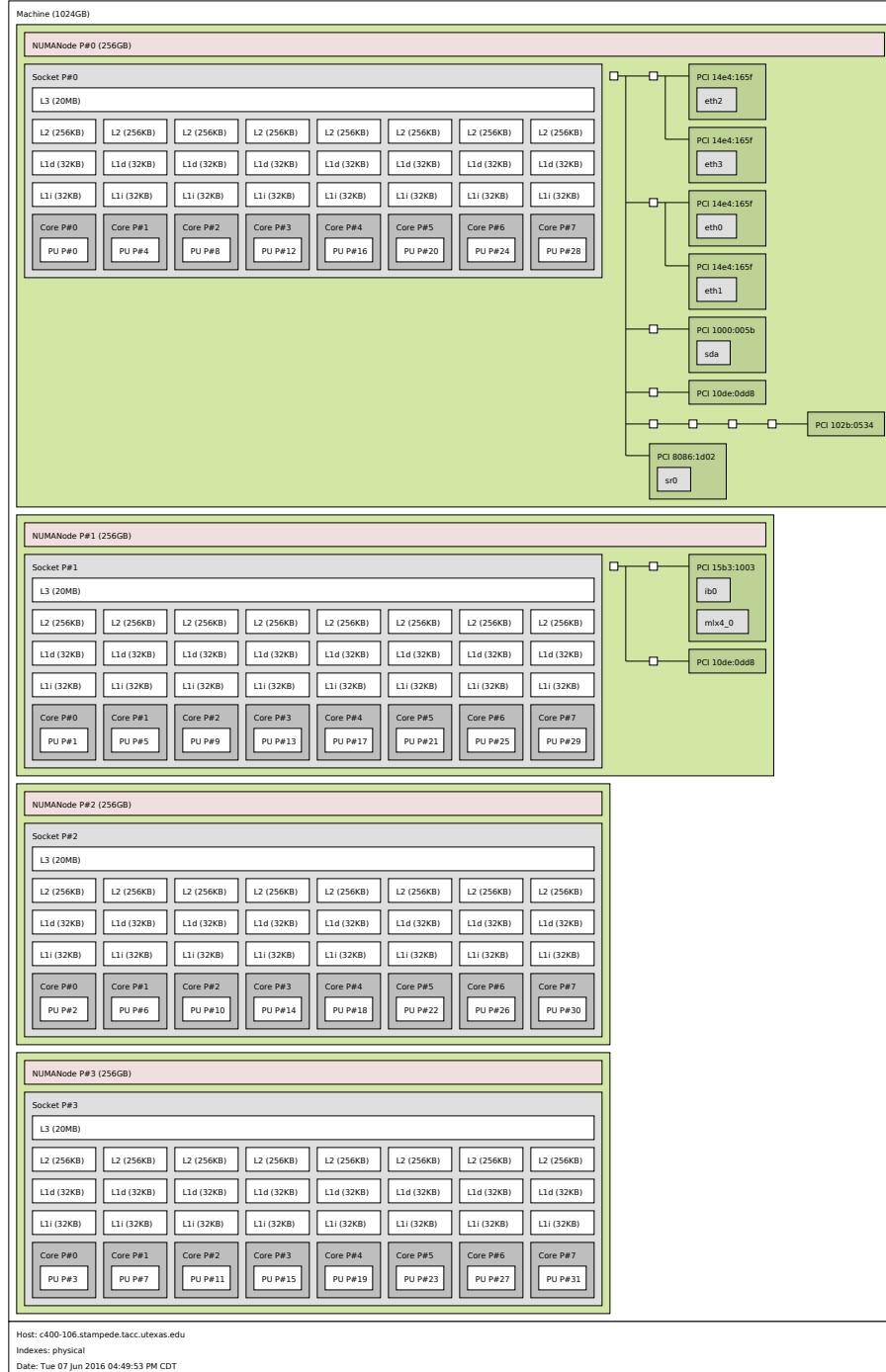


Figure 45.4: Structure of a Stampede largemem four-socket compute node

45. Hybrid computing

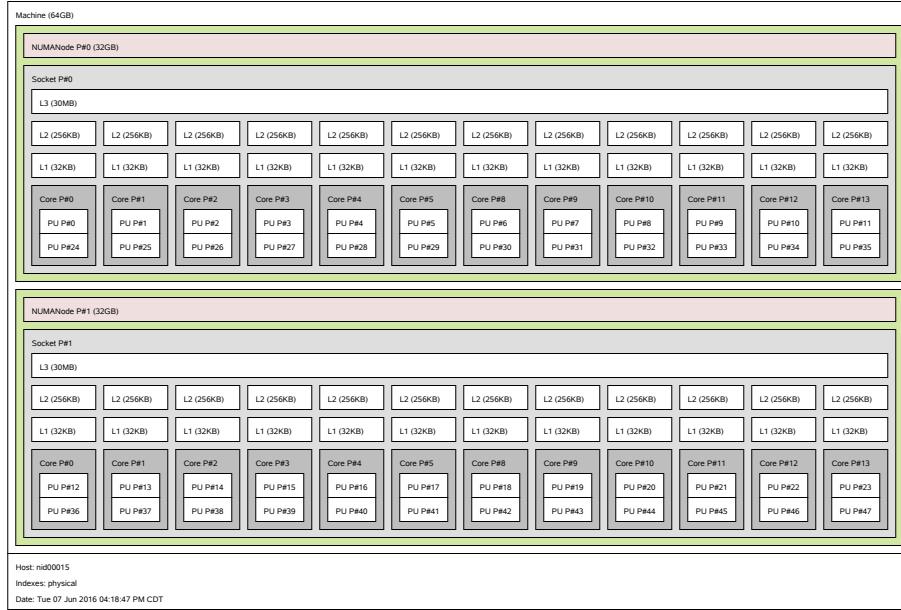


Figure 45.5: Structure of a Lonestar5 compute node

```
// procthread.c
int ncores;
#pragma omp parallel
#pragma omp master
ncores = omp_get_num_procs();

int totalcores;
MPI_Reduce(&ncores,&totalcores,1,MPI_INT,MPI_SUM,0,comm);
if (procid==0) {
    printf("Omp procs on this process: %d\n",ncores);
    printf("Omp procs total: %d\n",totalcores);
}
```

Running this with *Intel MPI* (version 19) gives the following:

```
---- nprocs: 14
Omp procs on this process: 4
Omp procs total: 56
---- nprocs: 15
Omp procs on this process: 3
Omp procs total: 45
---- nprocs: 16
Omp procs on this process: 3
Omp procs total: 48
```

We see that

- Each process get an equal number of cores, and
- Some cores will go unused.

While the OpenMP ‘proc’ count is such that the MPI processes will not oversubscribe cores, the actual placement of processes and threads is not expressed here. This assignment is known as *affinity* and it is determined by the MPI/OpenMP runtime system. Typically it can be controlled through environment variables, but one hopes the default assignment makes sense. Figure 45.6 illustrates this for the *Intel Knights Landing*:

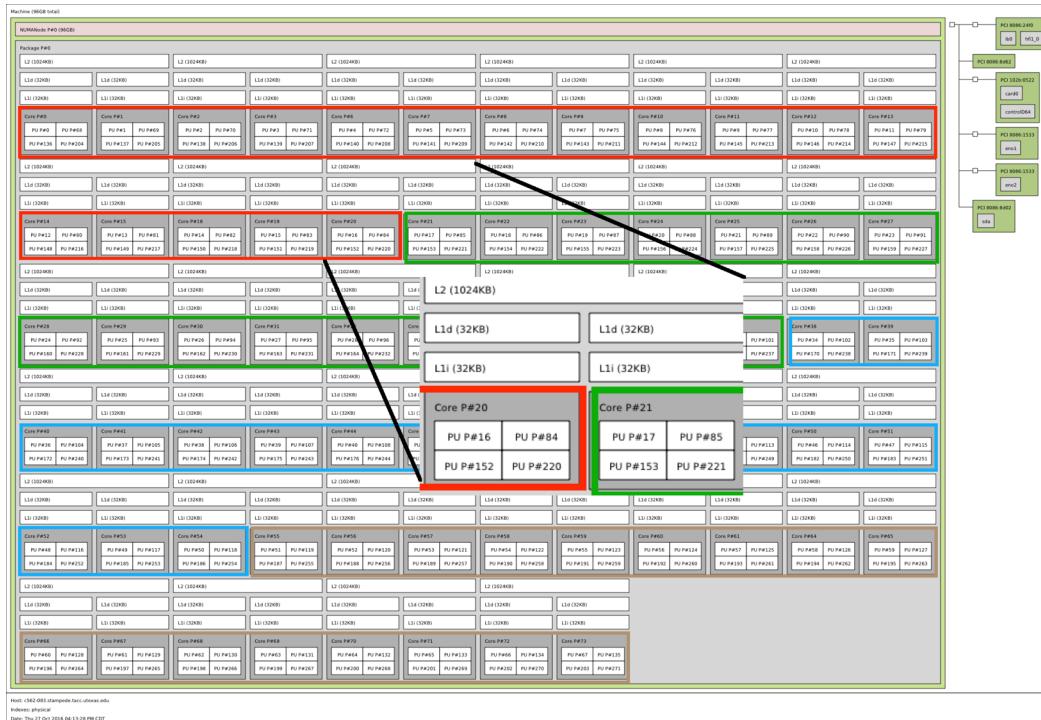


Figure 45.6: Process and thread placement on an Intel Knights Landing

- Placing four MPI processes on 68 cores gives 17 cores per process.
- Each process receives a contiguous set of cores.
- However, cores are grouped in ‘tiles’ of two, so processes 1 and 3 start halfway a tile.
- Therefore, thread zero of that process is bound to the second core.

45.7 Practical specification

Say you use 100 cluster nodes, each with 16 cores. You could then start 1600 MPI processes, one for each core, but you could also start 100 processes, and give each access to 16 OpenMP threads.

In your slurm scripts, the first scenario would be specified `-N 100 -n 1600`, and the second as

```
#$ SBATCH -N 100
#$ SBATCH -n 100
```

```
export OMP_NUM_THREADS=16
```

There is a third choice, in between these extremes, that makes sense. A cluster node often has more than one socket, so you could put one MPI process on each *socket*, and use a number of threads equal to the number of cores per socket.

The script for this would be:

```
#$ SBATCH -N 100
#$ SBATCH -n 200

export OMP_NUM_THREADS=8
ibrun tacc_affinity yourprogram
```

The *tacc_affinity* script unsets the following variables:

```
export MV2_USE_AFFINITY=0
export MV2_ENABLE_AFFINITY=0
export VIADEV_USE_AFFINITY=0
export VIADEV_ENABLE_AFFINITY=0
```

If you don't use *tacc_affinity* you may want to do this by hand, otherwise *mvapich2* will use its own affinity rules.

Chapter 46

Support libraries

There are many libraries related to parallel programming to make life easier, or at least more interesting, for you.

46.1 SimGrid

SimGrid [17] is a simulator for distributed systems. It can for instance be used to explore the effects of architectural parameters. It has been used to simulate large scale operations such as High Performance Linpack (HPL) [4].

46.2 Other

ParaMesh

Global Arrays

Hdf5 and Silo

PART VI

CLASS PROJECTS

Chapter 47

A Style Guide to Project Submissions

Here are some guidelines for how to submit assignments and projects. As a general rule, consider programming as an experimental science, and your writeup as a report on some tests you have done: explain the problem you're addressing, your strategy, your results.

47.1 General approach

As a general rule, consider programming as an experimental science, and your writeup as a report on some tests you have done: explain the problem you're addressing, your strategy, your results.

Turn in a writeup in pdf form (Word and text documents are not acceptable) that was generated from a text processing program such (preferably) \LaTeX . For a tutorial, see *Tutorials book, section ??*.

47.2 Style

Your report should obey the rules of proper English.

- Observing correct spelling and grammar goes without saying.
- Use full sentences.
- Try to avoid verbiage that is disparaging or otherwise inadvisable. The *Google developer documentation style guide* [12] is a great resource.

47.3 Structure of your writeup

47.3.1 Write as if it's an article

Consider this project writeup an opportunity to practice writing a scientific article.

Start with the obvious stuff.

- Your writeup should have a title. Not ‘Project’ or ‘parallel programming’, but something like ‘Parallelization of Chronosynclastic Enfundibula in MPI’.
- Author and contact information. This differs per publication. Here it is: your name, EID, TACC username, and email.
- Introductory section that is high level: what is the problem, what did you do, what did you find.
- Conclusion: what do your findings mean, what are limitations, opportunities for future extensions.
- Bibliography.

47.3.2 Consider your audience

An article is written for a specific audience: a journal, a conference, or in this case: your instructors. So don't go into details that mean nothing to your audience, and try giving them what they find interesting.

In other words: give enough background on your application, but not too much. You're not writing for your thesis supervisor, you're writing to interested outsiders to your field.

On the other hand, your instructors know everything about parallelism. So don't show a differential equation and say 'and I made this parallel with OpenMP'. Go into detail how you translated your problem into something computational, and then show relevant bits of code.

That does not mean that turning in the code is sufficient, nor code plus sample output. Write an article.

47.3.3 Observe, measure, hypothesize, deduce

Treat your project as if it is a scientific investigation of some phenomenon. Formulate hypotheses as to what you expect to observe, report on your observations, and draw conclusions.

Quite often your program will display unexpected behaviour. It is important to note this, and hypothesize what the reason might be for your observed behaviour.

In most applications of computing machinery we care about the efficiency with which we find the solution. Thus, make sure that you do measurements. In general, make observations that allow you to judge whether your program behaves the way you would expect it to.

47.3.4 Reporting

Your report should include both code snippets and graphs.

Screenshots of code snippets are not acceptable. Use at least a verbatim/monospace mode in your text processor, but better, use the `LATEX listings` package or equivalent.

Graphs can be generated any number of ways. Kudos if you can figure out the `LATEX tikz` package, but Matlab, gnuplot, Excel, or Google Sheets are all acceptable. Again: no screenshots.

For parallel runs you can, but are not required to, use TAU plots; see *Tutorials book, section ??*.

47.3.5 Repository organization

If you submit your work through a repository have your pdf file at the top level; organize your sources in clearly named subdirectories. Object files and binaries should not be in a repository since they are dependent on hardware and things like compilers.

47.4 The parallel part

The parallelization part is the most important of your writeup. So don't write 3 pages about your application and 1 about the parallel code. Discuss in detail:

- What is the parallel structure of your problem? Relate the code structure to the application structure.
- Did you use MPI or OpenMP? Why?
- What kind of parallelism did you use? Mostly MPI collectives or point-to-point operations? OpenMP loop parallelism or tasks? Why?

47.4.1 Performance

The most important reason for parallel programming is to achieve faster performance.

To measure the efficiency of your code, run for a range of processor/core counts. Do you aim for strong or weak scaling?

Make sure your problem is large enough to overcome the overhead of parallelization. Can you run several problem sizes at a given core/process count?

How much speedup are you expecting for your application, given its structure? How much are you getting? If your speedup is low, reason where the problem might lie.

47.4.2 Running your code

A single run doesn't prove anything. For a good report, you need to run your code for more than one input dataset (if available) and in more than one processor configuration. When you choose problem a size, bear the following factors in mind.

- Parallelism introduces overhead, hundreds of cycles for an OpenMP barrier, or a few microseconds for an MPI message. So the amount of work in a parallel region, or between messages, probably needs to be in the thousands of operations.
- Various things in the system introduce random fluctuations, timings that are too small may be meaningless. As a rule of thumb, a timed sections needs to take at least on the order of a second.
- There are various startup phenomena in a parallel code, such as OpenMP thread creation, or allocation of dynamic memory. Make sure you only time the relevant bit of your code; if in doubt, put a loop around it to take multiple measurements, and use an average time.

When you run a code in parallel, beware that on clusters the behaviour of a parallel code will always be different between one node and multiple nodes. On a single node the MPI implementation is likely optimized to use the shared memory. This means that results obtained from a single node run will be unrepresentative. In fact, in timing and scaling tests you will often see a drop in (relative) performance going from one node to two. Therefore you need to run your code in a variety of scenarios, using more than one node.

47.4.3 Graphs and tables

In parallel programming, speedup and scaling are the test of how good your work is. So it's up to you to report this as well as you can. Mere numbers in a table or graph are not enough: they need to tell a story.

Thus, if you do a scaling analysis, a graph reporting runtimes should make this point visible. In particular, do not use a linear time axis, as curved graphs are hard to read. Try to find a way to compare your results to a straight line, such as constant time, or linearly increasing speedup.

It is up to you to decide what quantity to report. This may depend on your application.

Use enough data points! Writing a short script to run your program multiple times takes very little time.

Supply a caption for your graph or table: 'weak scaling efficiency of problem 2' or 'speedup as a function of thread count'. Graphs and tables need to have a number that you refer to in the text of your writeup.

47.5 Remarks

47.5.1 Parallel performance or the lack thereof

In a perfect world, the performance of your code should grow with the number of available resources. If your program shows disappointing performance, consider the following.

Synchronizing OpenMP threads at the end of a parallel region takes maybe a few hundred cycles. This means that the amount of work in that region should be considerably more.

If your OpenMP program stops scaling at a certain core count, consider affinity settings; section 25.1.

MPI messages takes a couple of microseconds. Again, this implies that the amount of work between messages needs to be large enough.

47.5.2 Code formatting

Included code snippets should be readable. At a minimum you could indent the code correctly in an editor before you include it in a *verbatim* environment. (Screenshots of your terminal window are a decidedly suboptimal solution.) But it's better to use the *listing* package which formats your code, include syntax coloring. For instance,

```
\lstset{language=C++} % or Fortran or so
\begin{lstlisting}
for (int i=0; i<N; i++)
    s += 1;
\end{lstlisting}
```

With output:

```
for (int i=0; i<N; i++)
    s += 1;
```

Chapter 48

Warmup Exercises

We start with some simple exercises.

48.1 Hello world

For background, see section 2.3.

First of all we need to make sure that you have a working setup for parallel jobs. The example program `helloworld.c` does the following:

```
// helloworld.c
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&ntids);
MPI_Comm_rank(MPI_COMM_WORLD,&mytid);
printf("Hello, this is processor %d out of %d\n",mytid,ntids);
MPI_Finalize();
```

Compile this program and run it in parallel. Make sure that the processors do *not* all say that they are processor 0 out of 1!

48.2 Collectives

It is a good idea to be able to collect statistics, so before we do anything interesting, we will look at MPI collectives; section 3.1.

Take a look at `time_max.cxx`. This program sleeps for a random number of seconds:

```
wait = (int) ( 6.*rand() / (double)RAND_MAX );
tstart = MPI_Wtime();
sleep(wait);
tstop = MPI_Wtime();
jitter = tstop-tstart-wait;
```

and measures how long the sleep actually was:

```
if (mytid==0)
    sendbuf = MPI_IN_PLACE;
else sendbuf = (void*)&jitter;
MPI_Reduce(sendbuf,(void*)&jitter,1,MPI_DOUBLE,MPI_MAX,0,comm);
```

In the code, this quantity is called ‘jitter’, which is a term for random deviations in a system.

Exercise 48.1. Change this program to compute the average jitter by changing the reduction operator.

Exercise 48.2. Now compute the standard deviation

$$\sigma = \sqrt{\frac{\sum_i (x_i - m)^2}{n}}$$

where m is the average value you computed in the previous exercise.

- Solve this exercise twice: once by following the reduce by a broadcast operation and once by using an Allreduce.
- Run your code both on a single cluster node and on multiple nodes, and inspect the TAU trace. Some MPI implementations are optimized for shared memory, so the trace on a single node may not look as expected.
- Can you see from the trace how the allreduce is implemented?

Exercise 48.3. Finally, use a gather call to collect all the values on processor zero, and print them out. Is there any process that behaves very differently from the others?

48.3 Linear arrays of processors

In this section you are going to write a number of variations on a very simple operation: all processors pass a data item to the processor with the next higher number.

- In the file `linear-serial.c` you will find an implementation using blocking send and receive calls.
- You will change this code to use non-blocking sends and receives; they require an `MPI_Wait` call to finalize them.
- Next, you will use `MPI_Sendrecv` to arrive at a synchronous, but deadlock-free implementation.
- Finally, you will use two different one-sided scenarios.

In the reference code `linear-serial.c`, each process defines two buffers:

```
// linear-serial.c
int my_number = mytid, other_number=-1.;
```

where `other_number` is the location where the data from the left neighbour is going to be stored.

To check the correctness of the program, there is a gather operation on processor zero:

```
int *gather_buffer=NULL;
if (mytid==0) {
    gather_buffer = (int*) malloc(ntids*sizeof(int));
    if (!gather_buffer) MPI_Abort(comm,1);
}
MPI_Gather(&other_number,1,MPI_INT,
           gather_buffer,1,MPI_INT, 0,comm);
if (mytid==0) {
    int i,error=0;
    for (i=0; i<ntids; i++)
        if (gather_buffer[i]!=i-1) {
            printf("Processor %d was incorrect: %d should be %d\n",
                   i,gather_buffer[i],i-1);
            error =1;
        }
}
```

```

        }
        if (!error) printf("Success!\n");
        free(gather_buffer);
    }
}

```

48.3.1 Coding with blocking calls

Passing data to a neighbouring processor should be a very parallel operation. However, if we code this naively, with MPI_Send and MPI_Recv, we get an unexpected serial behaviour, as was explained in section 4.1.4.

```

if (mytid<ntids-1)
    MPI_Ssend( /* data: */ &my_number,1,MPI_INT,
               /* to: */ mytid+1, /* tag: */ 0, comm);
if (mytid>0)
    MPI_Recv( /* data: */ &other_number,1,MPI_INT,
              /* from: */ mytid-1, 0, comm, &status);

```

(Note that this uses an Ssend; see section 15.8 for the explanation why.)

Exercise 48.4. Compile and run this code, and generate a TAU trace file. Confirm that the execution is serial. Does replacing the Ssend by Send change this?

Let's clean up the code a little.

Exercise 48.5. First write this code more elegantly by using MPI_PROC_NULL.

48.3.2 A better blocking solution

The easiest way to prevent the serialization problem of the previous exercises is to use the MPI_Sendrecv call. This routine acknowledges that often a processor will have a receive call whenever there is a send. For border cases where a send or receive is unmatched you can use MPI_PROC_NULL.

Exercise 48.6. Rewrite the code using MPI_Sendrecv. Confirm with a TAU trace that execution is no longer serial.

Note that the Sendrecv call itself is still blocking, but at least the ordering of its constituent send and recv are no longer ordered in time.

48.3.3 Non-blocking calls

The other way around the blocking behaviour is to use Isend and Irecv calls, which do not block. Of course, now you need a guarantee that these send and receive actions are concluded; in this case, use MPI_Waitall.

Exercise 48.7. Implement a fully parallel version by using MPI_Isend and MPI_Irecv.

48.3.4 One-sided communication

Another way to have non-blocking behaviour is to use one-sided communication. During a Put or Get operation, execution will only block while the data is being transferred out of or into the origin process, but it is not blocked by the target. Again, you need a guarantee that the transfer is concluded; here use MPI_Win_fence.

Exercise 48.8. Write two versions of the code: one using MPI_Put and one with MPI_Get. Make TAU traces.

Investigate blocking behaviour through TAU visualizations.

Exercise 48.9. If you transfer a large amount of data, and the target processor is occupied, can you see any effect on the origin? Are the fences synchronized?

Chapter 49

Mandelbrot set

If you've never heard the name *Mandelbrot set*, you probably recognize the picture; figure 49.1 Its formal definition

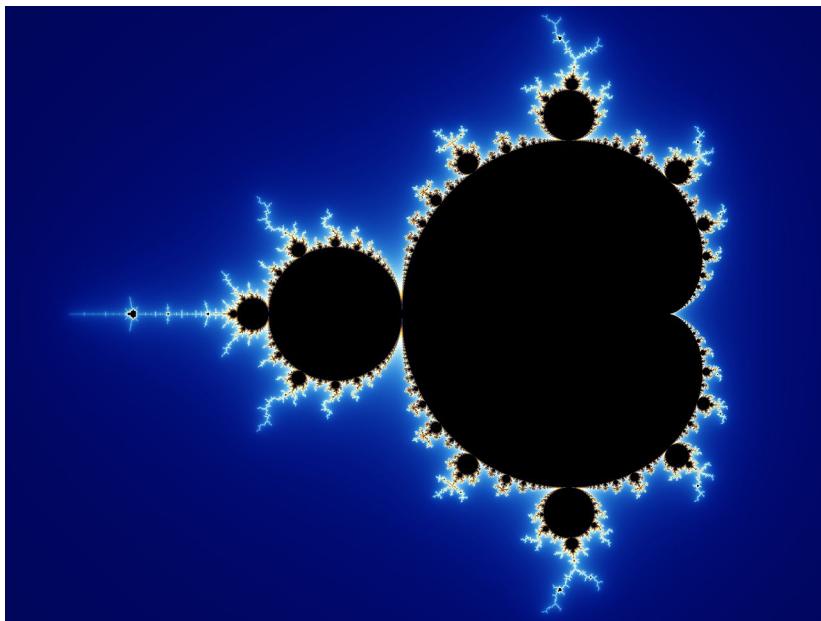


Figure 49.1: The Mandelbrot set

is as follows:

A point c in the complex plane is part of the Mandelbrot set if the series x_n defined by

$$\begin{cases} x_0 = 0 \\ x_{n+1} = x_n^2 + c \end{cases}$$

satisfies

$$\forall n : |x_n| \leq 2.$$

It is easy to see that only points c in the bounding circle $|c| < 2$ qualify, but apart from that it's hard to say much without a lot more thinking. Or computing; and that's what we're going to do.

In this set of exercises you are going to take an example program `mandel_main.cxx` and extend it to use a variety of MPI programming constructs. This program has been set up as a *manager-worker* model: there is one manager processor (for a change this is the last processor, rather than zero) which gives out work to, and accepts results from, the worker processors. It then takes the results and constructs an image file from them.

49.1 MPI solutions

49.1.1 Invocation

The `mandel_main` program is called as

```
mpirun -np 123 mandel_main steps 456 iters 789
```

where the `steps` parameter indicates how many steps in x, y direction there are in the image, and `iters` gives the maximum number of iterations in the `belong` test.

If you forget the parameter, you can call the program with

```
mandel_serial -h
```

and it will print out the usage information.

49.1.2 Tools

The driver part of the Mandelbrot program is simple. There is a circle object that can generate coordinates

```
class circle {
public :
    circle(double stp,int bound);
    void next_coordinate(struct coordinate& xy);
    int is_valid_coordinate(struct coordinate xy);
    void invalid_coordinate(struct coordinate& xy);
```

and a global routine that tests whether a coordinate is in the set, at least up to an iteration bound. It returns zero if the series from the given starting point has not diverged, or the iteration number in which it diverged if it did so.

```
int belongs(struct coordinate xy,int itbound) {
    double x=xy.x, y=xy.y; int it;
    for (it=0; it<itbound; it++) {
        double xx,yy;
        xx = x*x - y*y + xy.x;
        yy = 2*x*y + xy.y;
        x = xx; y = yy;
        if (x*x+y*y>4.) {
            return it;
        }
    }
    return 0;
}
```

In the former case, the point could be in the Mandelbrot set, and we colour it black, in the latter case we give it a colour depending on the iteration number.

```

if (iteration==0)
    memset(colour,0,3*sizeof(float));
else {
    float rfloat = ((float) iteration) / workcircle->infty;
    colour[0] = rfloat;
    colour[1] = MAX((float)0,(float)(1-2*rfloat));
    colour[2] = MAX((float)0,(float)(2*(rfloat-.5)));
}

```

We use a fairly simple code for the worker processes: they execute a loop in which they wait for input, process it, return the result.

```

void queue::wait_for_work(MPI_Comm comm,circle *workcircle) {
    MPI_Status status; int ntids;
    MPI_Comm_size(comm,&ntids);
    int stop = 0;

    while (!stop) {
        struct coordinate xy;
        int res;

        MPI_Recv(&xy,1,coordinate_type,ntids-1,0, comm,&status);
        stop = !workcircle->is_valid_coordinate(xy);
        if (stop) break; //res = 0;
        else {
            res = belongs(xy,workcircle->infty);
        }
        MPI_Send(&res,1,MPI_INT,ntids-1,0, comm);
    }
    return;
}

```

A very simple solution using blocking sends on the manager is given:

```

// mandel_serial.cxx
class serialqueue : public queue {
private :
    int free_processor;
public :
    serialqueue(MPI_Comm queue_comm,circle *workcircle)
        : queue(queue_comm,workcircle) {
        free_processor=0;
    };
    /**
     The `addtask' routine adds a task to the queue. In this
     simple case it immediately sends the task to a worker
     and waits for the result, which is added to the image.

     This routine is only called with valid coordinates;
     the calling environment will stop the process once
     an invalid coordinate is encountered.
    */
    int addtask(struct coordinate xy) {
        MPI_Status status; int contribution, err;

```

```

err = MPI_Send(&xy,1,coordinate_type,
               free_processor,0,comm); CHK(err);
err = MPI_Recv(&contribution,1,MPI_INT,
               free_processor,0,comm, &status); CHK(err);

coordinate_to_image(xy,contribution);
total_tasks++;
free_processor = (free_processor+1)%(ntids-1);

return 0;
};

```

Exercise 49.1. Explain why this solution is very inefficient. Make a trace of its execution that bears this out.

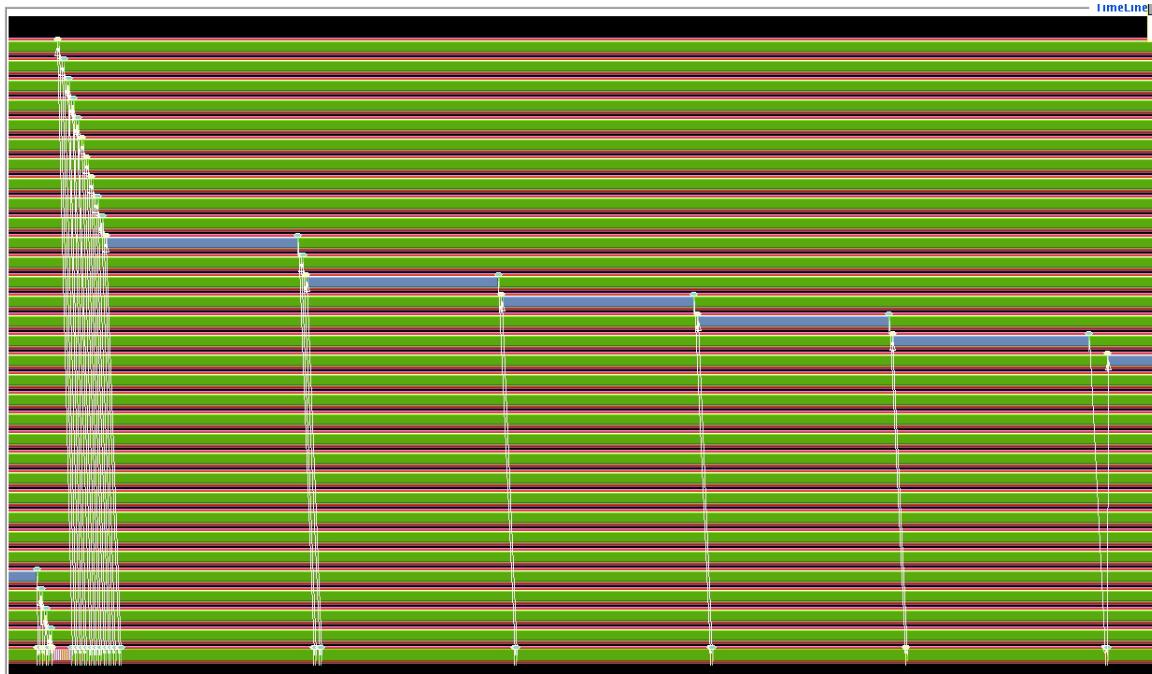


Figure 49.2: Trace of a serial Mandelbrot calculation

49.1.3 Bulk task scheduling

The previous section showed a very inefficient solution, but that was mostly intended to set up the code base. If all tasks take about the same amount of time, you can give each process a task, and then wait on them all to finish. A first way to do this is with non-blocking sends.

Exercise 49.2. Code a solution where you give a task to all worker processes using non-blocking sends and receives, and then wait for these tasks with `MPI_Waitall` to finish before you give a new round of data to all workers. Make a trace of the execution of this and report on the total time.

You can do this by writing a new class that inherits from `queue`, and that provides its own `addtask` method:

```
// mandel_bulk.cxx
class bulkqueue : public queue {
public :
    bulkqueue(MPI_Comm queue_comm,circle *workcircle)
        : queue(queue_comm,workcircle) {
```

You will also have to override the `complete` method: when the circle object indicates that all coordinates have been generated, not all workers will be busy, so you need to supply the proper `MPI_Waitall` call.

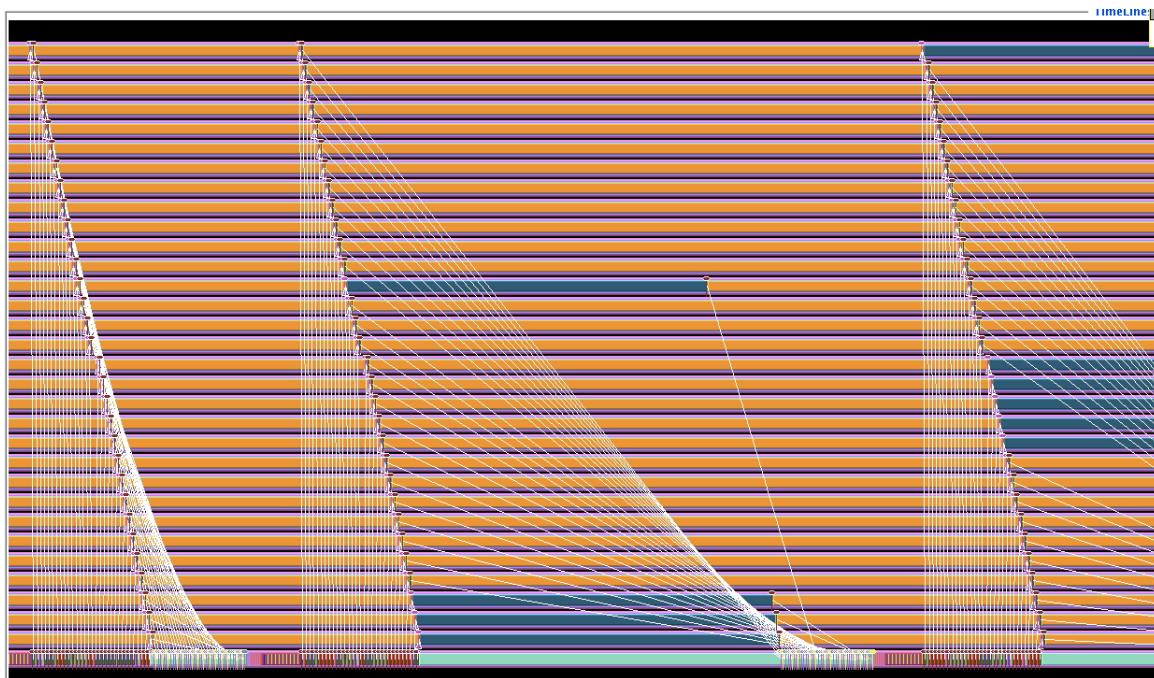


Figure 49.3: Trace of a bulk Mandelbrot calculation

49.1.4 Collective task scheduling

Another implementation of the bulk scheduling of the previous section would be through using collectives.

Exercise 49.3. Code a solution which uses scatter to distribute data to the worker tasks, and gather to collect the results. Is this solution more or less efficient than the previous?

49.1.5 Asynchronous task scheduling

At the start of section 49.1.3 we said that bulk scheduling mostly makes sense if all tasks take similar time to complete. In the Mandelbrot case this is clearly not the case.

Exercise 49.4. Code a fully dynamic solution that uses `MPI_Probe` or `MPI_Waitany`. Make an execution trace and report on the total running time.

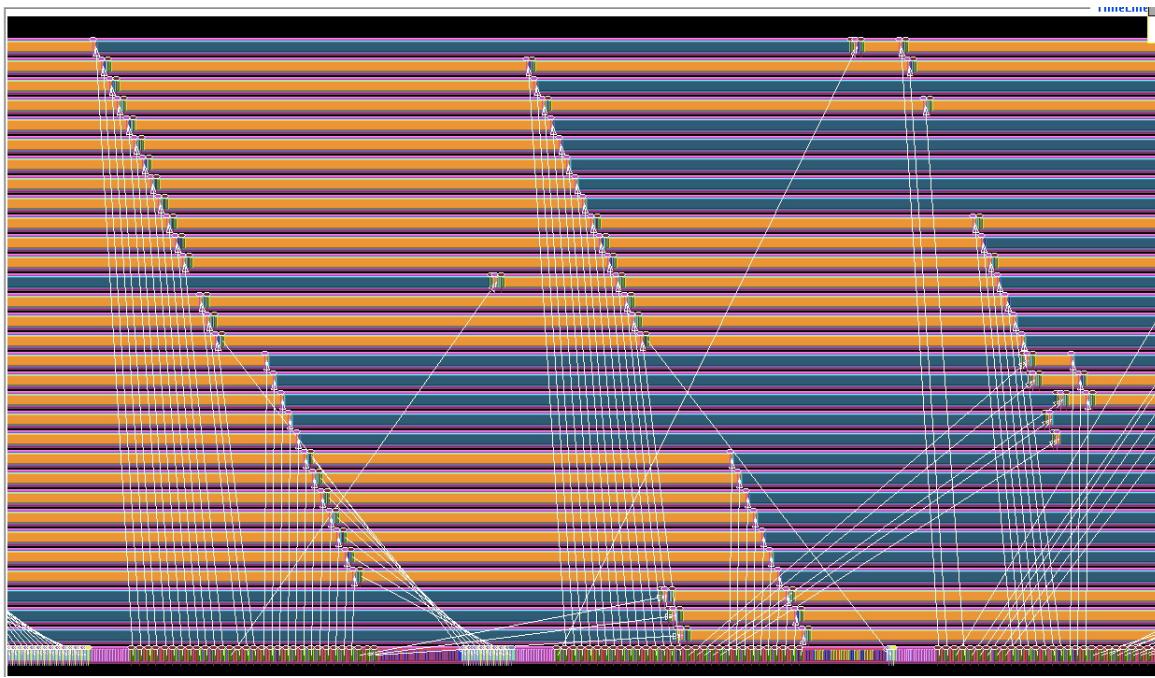


Figure 49.4: Trace of an asynchronous Mandelbrot calculation

49.1.6 One-sided solution

Let us reason about whether it is possible (or advisable) to code a one-sided solution to computing the Mandelbrot set. With active target synchronization you could have an exposure window on the host to which the worker tasks would write. To prevent conflicts you would allocate an array and have each worker write to a separate location in it. The problem here is that the workers may not be sufficiently synchronized because of the differing time for computation.

Consider then passive target synchronization. Now the worker tasks could write to the window on the manager whenever they have something to report; by locking the window they prevent other tasks from interfering. After a worker writes a result, it can get new data from an array of all coordinates on the manager.

It is hard to get results into the image as they become available. For this, the manager would continuously have to scan the results array. Therefore, constructing the image is easiest done when all tasks are concluded.

49.2 OpenMP solutions

49.2.1 Loop based

One could make a double loop over all coordinates, and use a dynamic or guided schedule.

49.2.2 Producer-consumer

You could also use a *producer-consumer* model: one thread generates coordinates, which the other threads then process. To make this as asynchronous as possible, we keep a single FIFO object (in C++ one could use a *deque*) with a lock on it: both the writing and reading threads lock the FIFO.

This constant locking probably means that threads will keep each other from doing useful work part of the time. This can be alleviated by writing and reading the FIFO with a block of values at a time.

```
[c205-013 cxx:30] for t in 2 3 4 5 6 ; do OMP_NUM_THREADS=$t ./mandeldeque -s .001 -l 10000 -b 10 ; do
===== #threads = 2 =====
Time: 62.9568
Number of points: 16000000
Interior: 1506920
Compute time: 60.288
===== #threads = 3 =====
Time: 33.2471
Compute time: 60.5359
===== #threads = 4 =====
Time: 23.6858
Compute time: 60.4749
===== #threads = 5 =====
Time: 19.2046
Compute time: 60.4396
===== #threads = 6 =====
Time: 16.7634
Compute time: 60.4717
```

Note that the aggregate compute time is more or less constant, and equals to the case with just one consumer thread.

Lower limits destroy scalability, and the aggregate time stays again constant, roughly equal to the external time.

Frontera:

```
[c208-022 cxx:14] for t in 2 3 4 5 6 ; do OMP_NUM_THREADS=$t ./mandeldeque -t $t -s .001 -l 1000 -b 20
===== #threads = 2 =====
Time: 8.01974
Number of points: 16000000
Interior: 1510209
Compute time: 6.47284
===== #threads = 3 =====
Time: 4.85859
Number of points: 16000000
Interior: 1510209
Compute time: 6.37375
===== #threads = 4 =====
Time: 4.31056
Number of points: 16000000
Interior: 1510209
Compute time: 6.40085
===== #threads = 5 =====
Time: 4.26642
Number of points: 16000000
Interior: 1510209
Compute time: 6.37777
===== #threads = 6 =====
Time: 4.18028
Number of points: 16000000
```

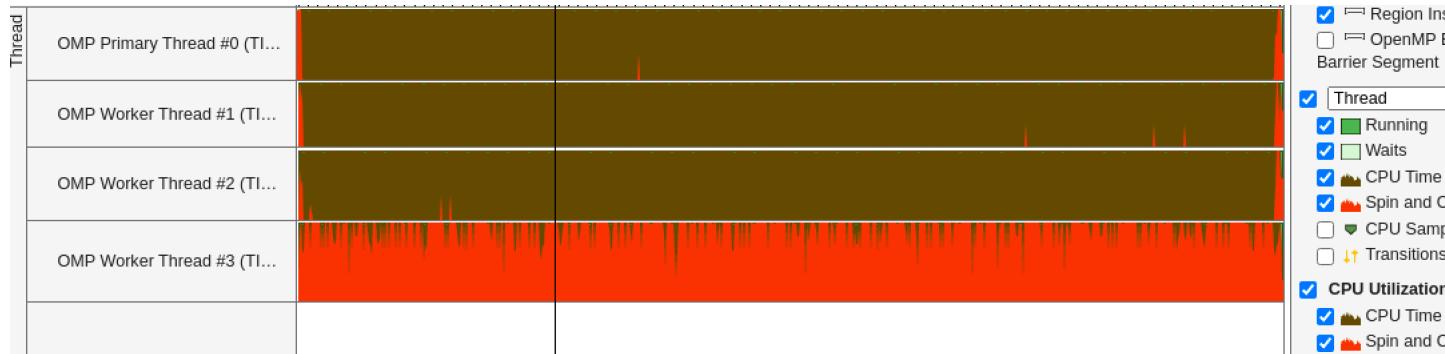
49. Mandelbrot set

```
Interior: 1510209
Compute time: 6.41134

Blocksize 200:
===== #threads = 2 =====
Time: 6.51065
Number of points: 16000000
Interior: 1510209
Compute time: 6.19076
===== #threads = 3 =====
Time: 3.44035
===== #threads = 4 =====
Time: 2.42226
===== #threads = 5 =====
Time: 1.90706
===== #threads = 6 =====
Time: 1.60447

Blocksize 2000:
===== #threads = 2 =====
Time: 6.50357
Number of points: 16000000
Interior: 1510209
Compute time: 6.43529
===== #threads = 3 =====
Time: 3.23416
===== #threads = 4 =====
Time: 2.21621
===== #threads = 5 =====
Time: 1.66643
===== #threads = 6 =====
Time: 1.36181
```

Four threads:



Eight:



It's not clear where the idle time in the beginning and end comes from.

Chapter 50

Data parallel grids

In this section we will gradually build a semi-realistic example program. To get you started some pieces have already been written: as a starting point look at `code/mpi/c/grid.cxx`.

50.1 Description of the problem

With this example you will investigate several strategies for implementing a simple iterative method. Let's say you have a two-dimensional grid of datapoints $G = \{g_{ij} : 0 \leq i < n_i, 0 \leq j < n_j\}$ and you want to compute G' where

$$g'_{ij} = 1/4 \cdot (g_{i+1,j} + g_{i-1,j} + g_{i,j+1} + g_{i,j-1}). \quad (50.1)$$

This is easy enough to implement sequentially, but in parallel this requires some care.

Let's divide the grid G and divide it over a two-dimension grid of $p_i \times p_j$ processors. (Other strategies exist, but this one scales best; see section HPC book, section ??.) Formally, we define two sequences of points

$$0 = i_0 < \dots < i_{p_i} < i_{p_i+1} = n_i, \quad 0 < j_0 < \dots < j_{p_j} < j_{p_j+1} = n_j$$

and we say that processor (p, q) computes g_{ij} for

$$i_p \leq i < i_{p+1}, \quad j_q \leq j < j_{q+1}.$$

From formula (50.1) you see that the processor then needs one row of points on each side surrounding its part of the grid. A picture makes this clear; see figure 50.1. These elements surrounding the processor's own part are called the *halo* or *ghost region* of that processor.

The problem is now that the elements in the halo are stored on a different processor, so communication is needed to gather them. In the upcoming exercises you will have to use different strategies for doing so.

50.2 Code basics

The program needs to read the values of the grid size and the processor grid size from the commandline, as well as the number of iterations. This routine does some error checking: if the number of processors does not add up to the size of MPI_COMM_WORLD, a nonzero error code is returned.

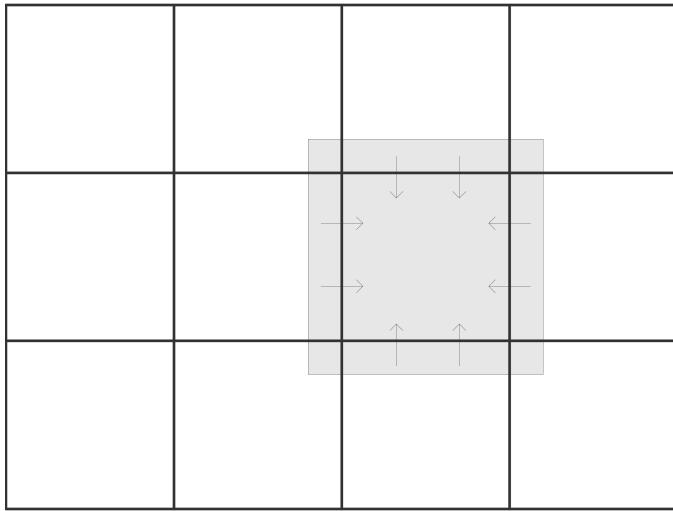


Figure 50.1: A grid divided over processors, with the ‘ghost’ region indicated

```
ierr = parameters_from_commandline
(argc,argv,comm,&ni,&nj,&pi,&pj,&nit);
if (ierr) return MPI_Abort(comm,1);
```

From the processor parameters we make a processor grid object:

```
processor_grid *pgrid = new processor_grid(comm,pi,pj);
```

and from the numerical parameters we make a number grid:

```
number_grid *grid = new number_grid(pgrid,ni,nj);
```

Number grids have a number of methods defined. To set the value of all the elements belonging to a processor to that processor’s number:

```
grid->set_test_values();
```

To set random values:

```
grid->set_random_values();
```

If you want to visualize the whole grid, the following call gathers all values on processor zero and prints them:

```
grid->gather_and_print();
```

Next we need to look at some data structure details.

The definition of the `number_grid` object starts as follows:

```
class number_grid {
public:
    processor_grid *pgrid;
    double *values,*shadow;
```

where `values` contains the elements owned by the processor, and `shadow` is intended to contain the values plus the ghost region. So how does `shadow` receive those values? Well, the call looks like

```
grid->build_shadow();
```

and you will need to supply the implementation of that. Once you’ve done so, there is a routine that prints out the `shadow` array of each processor

```
grid->print_shadow();
```

In the file `code/mpi/c/grid_impl.cxx` you can see several uses of the macro INDEX. This translates from a two-dimensional coordinate system to one-dimensional. Its main use is letting you use (i, j) coordinates for indexing the processor grid and the number grid: for processors you need the translation to the linear rank, and for the grid you need the translation to the linear array that holds the values.

A good example of the use of INDEX is in the `number_grid::relax` routine: this takes points from the shadow array and averages them into a point of the values array. (To understand the reason for this particular averaging, see HPC book, section ?? and HPC book, section ??.) Note how the INDEX macro is used to index in a $\text{ilength} \times \text{jlength}$ target array `values`, while reading from a $(\text{ilength} + 2) \times (\text{jlength} + 2)$ source array `shadow`.

```
for (i=0; i<ilength; i++) {
    for (j=0; j<jlength; j++) {
        int c=0;
        double new_value=0.;
        for (c=0; c<5; c++) {
            int ioff=i+1+ioffsets[c],joff=j+1+joffsets[c];
            new_value += coefficients[c] *
                shadow[ INDEX(ioff,joff,ilength+2,jlength+2) ];
        }
        values[ INDEX(i,j,ilength,jlength) ] = new_value/8.;
    }
}
```

Chapter 51

N-body problems

N-body problems describe the motion of particles under the influence of forces such as gravity. There are many approaches to this problem, some exact, some approximate. Here we will explore a number of them.

For background reading see HPC book, section ??.

51.1 Solution methods

It is not in the scope of this course to give a systematic treatment of all methods for solving the N-body problem, whether exactly or approximately, so we will just consider a representative selection.

1. Full N^2 methods. These compute all interactions, which is the most accurate strategy, but also the most computationally demanding.
2. Cutoff-based methods. These use the basic idea of the N^2 interactions, but reduce the complexity by imposing a cutoff on the interaction distance.
3. Tree-based methods. These apply a coarsening scheme to distant interactions to lower the computational complexity.

51.2 Shared memory approaches

51.3 Distributed memory approaches

PART VII

DIDACTICS

Chapter 52

Teaching guide

Based on two lectures per week, here is an outline of how MPI can be taught in a college course. Links to the relevant exercises.

Topic	Exercises	Week
Block 1: SPMD and collectives		
Intro: cluster structure	hello: 2.1 , 2.2	
Functional parallelism	commrank: 2.4 , 2.5 , prime: 2.6	week 1
Allreduce, broadcast	3.1 , randommax: 3.2 jordan: 3.9	week 2
Scan, Gather	3.14 , scangather: 3.12, 3.16	
Block 2: Two-sided point-to-point		week 3
Send and receive	pingpong: 4.1 , rightsend: 4.4	
Sendrecv	bucketblock: 4.6 , sendrecv: 4.8, 4.9	
Nonblocking	isendirecv: 4.13 , isendirecvarray: 4.14 bucketpipenonblock: 4.11	week 4
Block 3: Derived datatypes		week 5
Contiguous, Vector, Indexed	stridesend: 6.4 , cubegather: 6.6	
Extent and resizing		
Block 4: Communicators		
Duplication, split Groups	procgrid: 7.1 , 7.2	week 6
Block 5: I/O		
File open, write, views	blockwrite: 10.1 , viewwrite 10.4	
Block 6: Neighborhood collectives		week 7
Neighbor allgather	rightgraph: 11.2	

Chapter 53

Teaching from mental models

Distributed memory programming, typically through the MPI library, is the *de facto* standard for programming large scale parallelism, with up to millions of individual processes. Its dominant paradigm of Single Program Multiple Data (SPMD) programming is different from threaded and multicore parallelism, to an extent that students have a hard time switching models. In contrast to threaded programming, which allows for a view of the execution with central control and a central repository of data, SPMD programming has a symmetric model where all processes are active all the time, with none privileged, and where data is distributed.

This model is counterintuitive to the novice parallel programmer, so care needs to be taken how to instill the proper ‘mental model’. Adoption of an incorrect mental model leads to broken or inefficient code.

We identify problems with the currently common way of teaching MPI, and propose a structuring of MPI courses that is geared to explicit reinforcing the symmetric model. Additionally, we advocate starting from realistic scenarios, rather than writing artificial code just to exercise newly-learned routines.

53.1 Introduction

The MPI library [25, 22] is the *de facto* tool for large scale parallelism as it is used in engineering sciences. In this paper we want to discuss the manner it is usually taught, and propose a rethinking.

We argue that the topics are typically taught in a sequence that is essentially dictated by level of complexity in the implementation, rather than by conceptual considerations. Our argument will be for a sequencing of topics, and use of examples, that is motivated by typical applications of the MPI library, and that explicitly targets the required mental model of the parallelism model underlying MPI.

We have written an open-source textbook [9] with exercise sets that follows the proposed sequencing of topics and the motivating applications.

53.1.1 Short background on MPI

The MPI library dates back to the early days of cluster computing, the first half of the 1990s. It was an academic/industrial collaboration to unify earlier, often vendor-specific, message passing libraries. MPI is typically used to code large-scale Finite Element Method (FEM) and other physical simulation applications, which share characteristics of a relatively static distribution of large amounts of data – hence the use of clusters to increase size of the target problem – and the need for very efficient exchange of small amounts of data.

The main motivation for MPI is the fact that it can be scaled to more or less arbitrary scales, currently up to millions of cores [1]. Contrast this with threaded programming, which is limited more or less by the core count on a single node, currently about 70.

Considering this background, the target audience for MPI teaching consists of upper level undergraduate students, graduate students, and even post-doctoral researchers who are engaging for the first time in large scale simulations. The typical participant in an MPI course is likely to understand more than the basics of linear algebra and some amount of numerics of Partial Differential Equation (PDE).

53.1.2 Distributed memory parallelism

Corresponding to its origins in cluster computing, MPI targets distributed memory parallelism¹. Here, network-connected cluster nodes run codes that share no data, but synchronize through explicit messages over the network. Its main model for parallelism is described as Single Program Multiple Data (SPMD): multiple instances of a single program run on the processing elements, each operating on their own data. The MPI library then implements the communication calls that allow processes to combine and exchange data.

While MPI programs can solve many or all of the same problems that can be solved with a multicore approach, the programming approach is different, and requires an adjustment in the programmer's 'mental model' [7, 30] of the parallel execution. This paper addresses the question of how to teach MPI to best effect this shift in mindset.

Outline of this paper. We use section 53.2 to address explicitly the mental models that govern parallel thinking and parallel programming, pointing out why MPI is different, and difficult initially. In section 53.3 we consider the way MPI is usually taught, while in section 53.4 we offer an alternative that is less likely to lead to an incorrect mental model.

Some details of our proposed manner of teaching are explored in sections 53.5, 53.6, 53.7. We conclude with discussion in sections 53.8 and 53.9.

53.2 Implied mental models

Denning [8] argued how computational thinking consists in finding an abstract machine (a 'computational model') that solves the problem in a simple algorithmic way. In our case of teaching parallel programming, the complication to this story is that the problem to be solved is already a computational system. That doesn't lessen the need to formulate an abstract model, since the full explanation of MPI's workings are unmanageable for a beginning programmer, and often not needed for practical purposes.

In this section we consider in more detail the mental models that students may implicitly be working under, and the problems with them; targeting the right mental model will then be the subject of later sections. The two (interrelated) aspects of a correct mental model for distributed memory programming are control and synchronization. We here discuss how these can be misunderstood by students.

53.2.1 The traditional view of parallelism

The problem with mastering the MPI library is that beginning programmers take a while to overcome a certain mental model for parallelism. In this model, which we can call 'sequential semantics' (or more whimsically the 'big index finger' model), there is only a single strand of execution², which we may think of as a big index finger going down the source code.

1. Recent additions to the MPI standard target shared memory too.

2. We carefully avoid the word 'thread' which carries many connotations in the context of parallel programming.

This mental model corresponds closely to the way algorithms are described in the mathematical literature of parallelism, and it is actually correct to an extent in the context of threaded libraries such as OpenMP, where there is indeed initially a single thread of execution, which in some places spawns a team of threads to execute certain sections of code in parallel. However, in MPI this model is factually incorrect, since there are always multiple processes active, with none essentially privileged over others, and no shared or central data store.

53.2.2 The misconceptions of centralized control

The sequential semantics mental model that, as described above, underlies much of the theoretical discussion of parallelism, invites the student to adopt certain programming techniques, such as the master-worker approach to parallel programming. While this is often the right approach with thread-based coding, where we indeed have a master thread and spawned threads, it is usually incorrect for MPI. The strands of execution in an MPI run are all long-living processes (as opposed to dynamically spawned threads), and are *symmetric* in their capabilities and execution.

Lack of recognition of this process symmetry also induces students to solve problems by having a form of ‘central data store’ on one process, rather than adopting a symmetric, distributed, storage model. For instance, we have seen a student solve a data transposition problem by collecting all data on process 0, and subsequently distributing it again in transposed form. While this may be reasonable³ in shared memory with OpenMP, with MPI it is unrealistic in that no process is likely to have enough storage for the full problem. Also, this introduces a sequential bottleneck in the execution.

In conclusion, we posit that beginning MPI programmers may suffer from a mental model that makes them insufficiently realize the symmetry of MPI processes, and thereby arrive at inefficient and nonscalable solutions.

53.2.3 The reality of distributed control

An MPI program run consists of multiple independent threads of control. One problem in recognizing this is that there is only a single source code, so there is an inclination to envision the program execution as a single thread of control: the above-mentioned ‘index finger’ going down the statements of the source. A second factor contributing to this view is that a parallel code incorporates statements with values (`int x = 1.5;`) that are replicated over all processes. It is easy to view these as centrally executed.

Interestingly, work by Ben-David Kolikant [2] shows that students with no prior knowledge of concurrency, when invited to consider parallel activities, will still think in terms of centralized solutions. This shows that distributed control, such as it appears in MPI, is counterintuitive and needs explicit enforcement in its mental model. In particular, we explicitly target process symmetry and process differentiation.

The centralized model can still be maintained in MPI to an extent, since the scalar operations that would be executed by a single thread become replicated operations in the MPI processes. The distinction between sequential execution and replicated execution escapes many students at first, and in fact, since nothing it gained by explaining this, we do not do so.

53.2.4 The misconception of synchronization

Even with multiple threads of control and distributed data, there is still a temptation to see execution as ‘bulk synchronous processing’ (BSP [29]). Here, the execution proceeds by supersteps, implying that processes are largely synchronized. (The BSP model has several components more, which are usually ignored, notably one-sided communication and processor oversubscription.)

3. To first order; second order effects such as affinity complicate this story.

Supersteps as a computational model allow for small differences in control flow, for instance conditional inside a big parallelizable loop, but otherwise imply a form of centralized control (as above) on the level of major algorithm steps. However, codes using the pipeline model of parallelism, such idioms as

```
MPI_Recv( /* from: */ my_process-1)
// do some major work
MPI_Send( /* to : */ my_process+1)
```

fall completely outside either the sequential semantics or BSP model and require an understanding of one process' control being dependent on another's. Gaining a mental model for this sort of unsynchronized execution is nontrivial to achieve. We target this explicitly in section [53.5.1](#).

53.3 Teaching MPI, the usual way

The MPI library is typically taught as follows. After an introduction about parallelism, covering concepts such as speedup and shared versus distributed memory parallelism, students learn about the initialization and finalization routines, and the MPI_Comm_size and MPI_Comm_rank calls for querying the number of processes and the rank of the current process.

After that, the typical sequence is

1. two-sided communication, with first blocking and later nonblocking variants;
2. collectives; and
3. any number of advanced topics such as derived data types, one-sided communication, subcommunicators, MPI I/O et cetera, in no particular order.

This sequence is defensible from a point of the underlying implementation: the two-sided communication calls are a close map to hardware behavior, and collectives are both conceptually equivalent to, and can be implemented as, a sequence of point-to-point communication calls. However, this is not a sufficient justification for teaching this sequence of topics.

53.3.1 Criticism

We offer three points of criticism against this traditional approach to teaching MPI.

First of all, there is no real reason for teaching collectives after two-sided routines. They are not harder, nor require the latter as prerequisite. In fact, their interface is simpler for a beginner, requiring one line for a collective, as opposed to at least two for a send/receive pair, probably surrounded by conditionals testing the process rank. More importantly, they reinforce the symmetric process view, certainly in the case of the MPI_All... routines.

Our second point of criticism is regarding the blocking and nonblocking two-sided communication routines. The blocking routines are typically taught first, with a discussion of how blocking behavior can lead to load unbalance and therefore inefficiency. The nonblocking routines are then motivated from a point of latency hiding and solving the problems inherent in blocking. In our view such performance considerations should be secondary. Nonblocking routines should instead be taught as the natural solution to a conceptual problem, as explained below.

Thirdly, starting with point-to-point routines stems from a Communicating Sequential Processes (CSP)[\[14\]](#) view of a program: each process stands on its own, and any global behavior is an emergent property of the run. This may make sense for the teacher who know how concepts are realized 'under the hood', but it does not lead to additional insight with the students. We believe that a more fruitful approach to MPI programming starts from the global behavior, and then derives the MPI process in a top-down manner.

53.3.2 Teaching MPI and OpenMP

In scientific computing, another commonly used parallel programming system is OpenMP [23]. OpenMP and MPI are often taught together, with OpenMP taught earlier because it is supposedly easier, or because its parallelism would be easier to grasp. Regardless our opinion on the first estimate, we argue that OpenMP should be taught *after* MPI because of its ‘central control’ parallelism model. If students come to associate parallelism with a model that has a ‘master thread’ and ‘parallel regions’ they will find it much harder to make idiomatic use of the symmetric model of MPI.

53.4 Teaching MPI, our proposal

As alternative to the above sequence of introducing MPI concepts, we propose a sequence that focuses on practical scenarios, and that actively reinforces the mental model of SPMD execution.

Such reinforcement is often an immediate consequence of our strategy of illustrating MPI constructs in the context of an application: most MPI applications (as we shall briefly discuss next) operate on large ‘distributed objects’. This immediately leads to a mental model of the workings of each process being the ‘projection’ onto that process of the global calculation. The opposing view, where the overall computation is emergent from the individual processes, is the CSP model mentioned above.

53.4.1 Motivation from applications

The typical application for MPI comes from Computational Science and Engineering, such as N-body problems, aerodynamics, shallow water equations, Lattice Boltzman methods, weather modeling with Fast Fourier Transform. Of these, the PDE based applications can readily be explained to need a number of MPI mechanisms.

Nonnumeric applications exist:

- Graph algorithms such as shortest-path or PageRank are straightforward to explain sequentially. However, the distributed memory algorithms need to be approached fundamentally different from the more naive shared memory variants. Thus they require a good amount of background knowledge. Additionally, they do not feature the regular communications that one-dimensional PDE applications have. Scalability arguments make this story even more complicated. Thus, these algorithms are in fact a logical next topic *after* discussion of parallel PDE algorithm.
- N-body problems, in their naive implementation, are easy to explain to any student who knows inverse-square laws such as gravity. It is a good illustration of some collectives, but nothing beyond that.
- Sorting. Sorting algorithms based on a sorting network (this includes bubblesort, but not quicksort) can be used as illustration. In fact, we use odd-even transposition sort as a ‘midterm’ exam assignment, which can be solved with MPI_Sendrecv. Algorithms such as bitonic sort can be used to illustrate some advanced concepts, but quicksort, which is relatively easy to explain as a serial algorithm, or even in shared memory, is quite hard in MPI.
- Point-to-point operations can also be illustrated by graphics operations such as a ‘blur’, since these correspond to a ‘stencil’ applied to a cluster of pixels. Unfortunately, this example suffers from the fact that neither collectives, nor irregular communications have a use in this application. Also, using graphics to illustrate simple MPI point-to-point communication is unrealistic in two ways: first, to start out simple we have to posit a one-dimensional pixel array; secondly, graphics is hardly ever of the scale that necessitates distributed memory, so this example is far from ‘real world’. (Ray tracing is naturally done distributed, but that has a completely different computational structure.)

Based on this discussion of possible applications, and in view of the likely background of course attendants, we consider Finite Difference solution of PDEs as a prototypical application that exercises both the simplest and more sophisticated mechanisms. During a typical MPI training, even a one-day short course, we insert a lecture on sparse matrices and their computational structure to motivate the need for various MPI constructs.

53.4.2 Process symmetry

Paradoxically, the first way to get students to appreciate the notion of process symmetry in MPI is to run a non-MPI program. Thus, students are asked to write a ‘hello world’ program, and execute this with `mpiexec`, as if it were an MPI program. Every process executes the print statement identically, bearing out the total symmetry between the processes.

Next, students are asked to insert the initialize and finalize statements, with three different ‘hello world’ statements before, between, and after them. This will prevent any notion of the code between initialization and finalization being considered as an OpenMP style ‘parallel region’.

A simple test to show that while processes are symmetric they are not identical is offered by the exercise of using the `MPI_Get_processor_name` function, which will have different output for some or all of the processes, depending on how the hostfile was arranged.

53.4.3 Functional parallelism

The `MPI_Comm_rank` function is introduced as a way of distinguishing between the MPI processes. Students are asked to write a program where only one process prints the output of `MPI_Comm_size`.

Having different execution without necessarily different data is a case of ‘functional parallelism’. At this point there are few examples that we can assign. For instance, in order to code the evaluation of an integral by Riemann sums ($\pi/4 = \int_0^1 \sqrt{1-x^2} dx$ is a popular one) would need a final sum collective, which has not been taught at this point.

A possible example would be primality testing, where each process tries to find a factor of some large integer N by traversing a subrange of $[2, \sqrt{N}]$, and printing a message if a factor is found. Boolean satisfiability problems form another example, where again a search space is partitioned without involving any data space; a process finding a satisfying input can simply print this fact. However, this example requires background that students typically don’t have.

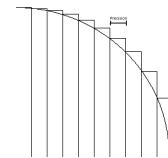


Figure 53.1: Calculation of $\pi/4$ by Riemann sums

53.4.4 Introducing collectives

At this point we can introduce collectives, for instance to find the maximum of a random value that is computed locally on each process. This requires teaching the code for random number generation and, importantly, setting a process-dependent random number seed. Generating random 2D or 3D coordinates and finding the center of mass is an examples that requires a send and receive buffer of length greater than 1, and illustrates that reductions are then done pointwise.

These examples evince both process symmetry and a first form of local data. However, a thorough treatment of distributed parallel data will come in the discussion of point-to-point routines.

It is an interesting question whether we should dispense with ‘rooted’ collectives such as `MPI_Reduce` at first, and start with `MPI_Allreduce`⁴. The latter is more symmetric in nature, and has a buffer treatment that is easier to

4. The `MPI_Reduce` call performs a reduction on data found on all processes, leaving the result on a ‘root’ process. With `MPI_Allreduce` the result is left on *all* processes.

explain; it certainly reinforces the symmetric mindset. There is also essentially no difference in efficiency.

Certainly, in most applications the ‘allreduce’ is the more common mechanism, for instance where the algorithm requires computations such as

$$\bar{y} \leftarrow \bar{x}/\|\bar{x}\|$$

where x, y are distributed vectors. The quantity $\|\bar{x}\|$ is then needed on all processes, making the Allreduce the natural choice. The rooted reduction is typically only used for final results. Therefore we advocate introducing both rooted and nonrooted collectives, but letting the students initially do exercises with the nonrooted variants.

This has the added advantage of not bothering the students initially with the asymmetric treatment of the receive buffer between the root and all other processes.

53.4.5 Distributed data

As motivation for the following discussion of point-to-point routines, we now introduce the notion of distributed data. In its simplest form, a parallel program operates on a linear array the dimensions of which exceed the memory of any single process.

```
int n;
double data[n];
```

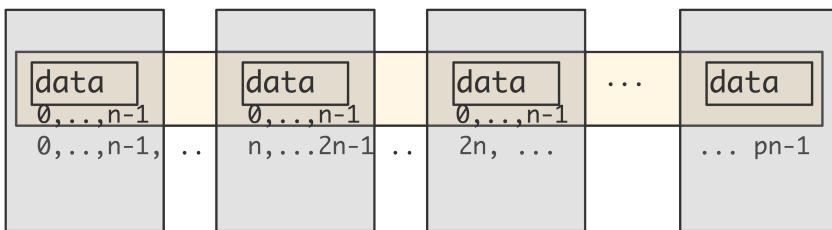


Figure 53.2: A distributed array versus multiple local arrays

The lecturer stresses that the global structure of the distributed array is only ‘in the programmer’s mind’: each MPI process sees an array with indexing starting at zero. The following snippet of code is given for the students to use in subsequent exercises:

```
int myfirst = ....;
for (int ilocal=0; ilocal<nlocal; ilocal++) {
    int iglobal = myfirst+ilocal;
    array[ilocal] = f(iglobal);
}
```

At this point, the students can code a second variant of the primality testing exercise above, but with an array allocated to store the integer range. Since collectives are now known, it becomes possible to have a single summary statement from one process, rather than a partial result statement from each.

The inner product of two distributed vectors is a second illustration of working with distributed data. In this case, the reduction for collecting the global result is slightly more useful than the collective in the previous examples. For this example no translation from local to global numbering is needed.

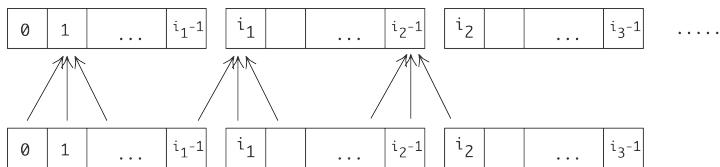
53.4.6 Point-to-point motivated from operations on distributed data

We now state the importance of local combining operations such as

$$y_i = (x_{i-1} + x_i + x_{i+1})/3 : i = 1, \dots, N - 1$$

applied to an array. Students who know about PDEs will recognize that with different coefficients this is the heat equation; for others a graphics ‘blur’ operation can be used as illustration, if they accept that a one-dimensional pixel array is a stand-in for a true graphic.

Under the ‘owner computes’ regime, where the process that stores location y_i performs the full calculation of that quantity, we see the need for communication in order to compute the first and last element of the local part of y :



We then state that this data transfer is realized in MPI by two-sided send/receive pairs.

53.4.7 Detour: deadlock and serialization

The concept of ‘blocking’ is now introduced, and we discuss how this can lead to deadlock. A more subtle behavior is ‘unexpected serialization’: processes interacting to give serial behavior on a code that conceptually should be parallel. (The classroom protocol is discussed in detail in section 53.5.1.) For completeness, the ‘eager limit’ can be discussed.

This introduces students to an interesting phenomenon in the concept of parallel correctness: a program may give the right result, but not with the proper parallel efficiency. Asking a class to come up with a solution that does not have a running time proportional to the number of processes, will usually lead to at least one student suggesting splitting processes in odd and even subsets. The limits to this approach, code complexity and the reliance on regular process connectivity, are explained to the students as a preliminary to the motivation for nonblocking sends; section 53.4.10.

53.4.8 Detour: ping-pong

At this point we briefly abandon the process symmetry, and consider the ping-pong operation between two processes A and B⁵. We ask students to consider what the ping-pong code looks like for A and, for B. Since we are working with SPMD code, we arrive at a program where the A code and B code are two branches of a conditional.

We ask the students to implement this, and do timing with MPI_Wtime. The implementation of the ping-pong is itself a good exercise in SPMD thinking; finding the right sender/receiver values usually takes the students a nontrivial amount of time. Many of them will initially write a code that deadlocks.

The concepts of latency and bandwidth can be introduced, as the students test the ping-pong code on messages of increasing size. The concept of halfbandwidth can be introduced by letting half of all processes execute a ping-pong with a partner process in the other half.

5. In this operation, process A sends to B, and B subsequently sends to A. Thus the time for a message is half the time of a ping-pong. It is not possible to measure a single message directly, since processes can not be synchronized that finely.

53.4.9 Back to data exchange

The foregoing detours into the behavior of two-sided send and receive calls were necessary, but they introduced asymmetric behavior in the processes. We return to the averaging operation given above, and with it to a code that treats all processes symmetrically. In particular, we argue that, except for the first and last, each process exchanges information with its left and right neighbor.

This could be implemented with blocking sends and receive calls, but students recognize how this could be somewhere between tedious and error-prone. Instead, to prevent deadlock and serialization as described above, we now offer the MPI_Sendrecv routine⁶. Students are asked to implement the classroom exercise above with the sendrecv routine. Ideally, they use timing or tracing to gather evidence that no serialization is happening.

As a nontrivial example (in fact, this takes enough programming that one might assign it as an exam question, rather than an exercise during a workshop) students can now implement an odd-even transposition sort algorithm using MPI_Sendrecv as the main tool. For simplicity they can use a single array element per process. (If each process has a subarray one has to make sure their solution has the right parallel complexity. It is easy to make errors here and implement a correct algorithm that, however, performs too slowly.)

Note that students have at this point not done any serious exercises with the blocking communication calls, other than the ping-pong. No such exercises will in fact be done.

53.4.10 Nonblocking sends

Nonblocking sends are now introduced as the solution to a specific problem: the above schemes required paired-up processes, or careful orchestration of send and receive sequences. In the case of irregular communications this is no longer possible or feasible. Life would be easy if we could declare ‘this data needs to be sent’ or ‘these messages are expected’, and then wait for these messages collectively. Given this motivation, it is immediately clear that multiple send or receive buffers are needed, and that requests need to be collected.

Implementing the three-point averaging with nonblocking calls is at this point an excellent exercise.

Note that we have here motivated the nonblocking routines to solve a symmetric problem. Doing this should teach the students the essential point that each nonblocking call needs its own buffer and generates its own request. Viewing nonblocking routines as a performance alternative to blocking routines is likely to lead to students reusing buffers or failing to save the request objects. Doing so is a correctness bug that is very hard to find, and at large scale it induces a memory leak since many requests objects are lost.

53.4.11 Taking it from here

At this point various advanced topics can be discussed. For instance, Cartesian topologies can be introduced, extending the linear averaging operation to a higher dimensional one. Subcommunicators can be introduced to apply collectives to rows and columns of a matrix. The recursive matrix transposition algorithm is also an excellent application of subcommunicators.

However, didactically these topics do not require the careful attention that the introduction of the basic concepts needs, so we will not go into further detail here.

6. The MPI_Sendrecv call combines a send a receive operation, specifying for each process both a sending and receiving communication. The execution guarantees that no deadlock or serialization will occur.

53.5 ‘Parallel computer games’

Part of the problem in developing an accurate mental model of parallel computation is that there is no easy way to visualize the execution. While sequential execution can be imagined with the ‘big index finger’ model (see section 53.2.1), the possibly unsynchronized execution of an MPI program makes this a gross simplification. Running a program in a parallel graphical environment (such as the DDT debugger or the Eclipse PTP IDE) would solve this, but they introduce much learning overhead. Ironically, the low tech solution of

```
mpiexec -n 4 xterm -e gdb program
```

is fairly insightful, but having to learn gdb is again a big hurdle.

We have arrived at the somewhat unusual solution of having students act out the program in front of the class. With each student acting out the program, any interaction is clearly visible to an extent that is hard to achieve any other way.

53.5.1 Sequentialization

Our prime example is to illustrate the blocking behavior of MPI_Send and MPI_Recv⁷. Deadlock is easy enough to understand as a consequence of blocking – in the simplest case of deadlock two processes are both blocked expecting a receive from the other – but there are more subtle effects that will come as a surprise to students. (This was alluded to in section 53.4.7.)

Consider the following basic program:

- Pass a data item to the next higher numbered process.

Note that this is conceptually a fully parallel program, so it should execute in time $O(1)$ in terms of the number of processes.

In terms of send and receive calls, the program becomes

- Send data to the next higher process;
- Receive data from the next lower process.

The final detail concerns the boundary conditions: the first process has nothing to receive and the last one has nothing to send. This makes the final version of the program:

- If you are not the last process, send data to the next higher process; then
- If you are not the first process, receive data from the next lower process.

To have students act this out, we tell them to hold a pen in their right hand, and put the left hand in a pocket or behind their back. Thus, they have only one ‘communication channel’. The ‘send data’ instruction becomes ‘turn to your right and give your pen’, and ‘receive data’ becomes ‘turn to your left and receive a pen’.

Executing this program, the students first all turn to the right, and they see that giving data to a neighbor is not possible because no one is executing the receive instruction. The last process is not sending, so moves on to the receive instruction, after which the penultimate process can receive, et cetera.

This exercise makes the students see, better than any explanation or diagram, how a parallel program can compute the right result, but with unexpectedly low performance because of the interaction of the processes. (In fact, we have had explicit feedback that this game was the biggest lightbulb moment of the class.)

7. Blocking is defined as the process executing a send or receive call halting until the corresponding operation is executing.

53.5.2 Ping-pong

While in general we emphasize the symmetry of MPI processes, during the discussion of send and receive calls we act out the ping-pong operation (one process sending data to another, followed by the other sending data back), precisely to demonstrate how asymmetric actions are handled. For this, two students throw a pen back and forth between them, calling out ‘send’ and ‘receive’ when they do so.

The teacher then asks each student what program they executed, which is ‘send-receive’ for the one, and ‘receive-send’ for the other student. Incorporating this in the SPMD model then leads to a code with conditionals to determine the right action for the right process.

53.5.3 Collectives and other games

Other operations can be acted out by the class. For instance, the teacher can ask one student to add the grades of all students, as a proxy for a reduction operation. The class quickly sees that this will take a long time, and strategies such as taking by-row sums in the classroom quickly suggest themselves.

We have at one point tried to have a pair of student act out a ‘race condition’ in shared memory programming, but modeling this quickly became too complicated to be convincing.

53.5.4 Remaining questions

Even with our current approach, however, we still see students writing idioms that are contrary to the symmetric model. For instance, they will write

```
for (p=0; p<nprocs; p++)
    if (p==myrank)
        // do some function of p
```

This code computes the correct result, and with the correct performance behavior, but it still shows a conceptual misunderstanding. As one of the ‘parallel computer games’ (section 53.5) we have put a student stand in front of the class with a sign ‘I am process 5’, and go through the above loop out loud (‘Am I process zero? No. Am I process one? No.’) which quickly drives home the point about the futility of this construct.

53.6 Further course summary

We have taught MPI based on the above ideas in two ways. First, we teach an academic class, that covers MPI, OpenMP, and general theory of parallelism in one semester. The typical enrollment is around 30 students, who do lab exercises and a programming project of their own choosing. We also teach a two-day intensive workshop (attendance 10–40 students depending on circumstances) of 6–8 hours per day. Students of the academic class are typically graduate or upper level undergraduate students; the workshops get attendance from post-docs, academics, and industry too. The typical background is applied math, engineering, physical sciences.

We cover the following topics, with division over two days in the workshop format:

- Day 1: familiarity with SPMD, collectives, blocking and nonblocking two-sided communication.
- Day 2: exposure to: sub-communicators, derived datatypes. Two of the following: MPI-I/O, one-sided communication, process management, the profiling and tools interfaces, neighborhood collectives.

53.6.1 Exercises

On day 1 the students do approximately 10 programming exercises, mostly finishing a skeleton code given by the instructor. For the day 2 material students do two exercises per topic, again starting with a given skeleton. (Skeleton codes are available as part of the repository [9].)

The design of these skeleton codes is an interesting problem in view of our concern with mental models. The skeletons are intended to take the grunt work away from the students, to both indicate a basic code structure and relieve them from making elementary coding errors that have no bearing on learning MPI. On the other hand, the skeletons should leave enough unspecified that multiple solutions are possible, including wrong ones: we want students to be confronted with conceptual errors in their thinking, and a too-far-finished skeleton would prevent them from doing that.

Example: the prime finding exercise mentioned above (which teaches the notion of functional parallelism) has the following skeleton:

```
int myfactor;
// Specify the loop header:
// for ( ... myfactor ... )
for (
    **** your code here ****/
    ) {
if (bignum%myfactor==0)
    printf("Process %d found factor %d\n",
           procno,myfactor);
}
```

This leaves open the possibility of both a blockwise and a cyclic distribution of the search space, as well as incorrect solutions where each process runs through the whole search space.

53.6.2 Projects

Students in our academic course do a programming project in place of a final exam. Students can choose between one of a set of standard projects, or doing a project of their own choosing. In the latter case, some students will do a project in context of their graduate research, which means that they have an existing codebase; others will write code from scratch. It is this last category, that will most clearly demonstrate their correct understanding of the mental model underlying SPMD programs. However, we note that this is only a fraction of the students in our course, a fraction made even smaller by the fact that we also give a choice of doing a project in OpenMP rather than MPI. Since OpenMP is, at least to the beginning programmer, simpler to use, there is an in fact a clear preference for it among the students who pick their own project.

53.7 Prospect for an online course

Currently the present author teaches MPI in the form of an academic course or short workshop, as outlined in section 53.6. In both cases, lecture time is far less than lab time, making the setup very intensive in teacher time. It also means that this setup is not scalable to a larger number of students. Indeed, while the workshops are usually webcast, we have not sufficiently solved the problem of supporting remote students. (The Pittsburgh Supercomputing Center offers courses that have remotely located teaching assistants, which seems a promising approach.) Such problems of support would be even more severe with an online course, where in-person support is completely absent.

One obvious solution to online teaching is automated grading: a student submits an exercise, which is then run through a checker program that tests the correct output. Especially if the programming assignment takes input, a checker script can uncover programming errors, notably in boundary cases.

However, the whole target of this paper is to uncover conceptual misunderstandings, for instance such as can lead to correct results with sub-optimal performance. In a classroom situation such misunderstandings are quickly caught and cleared up, but to achieve this in a context of automated grading we need to go further.

We have started experiments with actually parsing code submitted by the students. This effort started in a beginning programming class taught by the present author, but is now being extended to the MPI courses.

It is possible to uncover misconceptions in students' understanding by detecting the typical manifestations of such misconceptions. For instance, the code in section 53.5.4 can be uncovered by detecting a loop where the upper bound involves a variable that was set by `MPI_Comm_size`. Many MPI codes have no need for such a loop over all processes, so detecting one leads to an alert for the student.

Note that no tools exist for such automated evaluation. The source code analysis needed falls far short of full parsing. On the other hand, the sort of constructs it supposed to detect, are normally not of interest to the writers of compilers and source translators. This means that by writing fairly modest parsers (say, less than 200 lines of python) we can perform a sophisticated analysis of the students' codes. We hope to report on this in more detail in a follow-up paper.

53.8 Evaluation and discussion

At the moment, no rigorous evaluation of the efficacy of the above ideas has been done. We intend to perform a comparison between outcomes of the proposed way of teaching and the traditional way by comparing courses at two (or more) different institutions and from different syllabi. The evaluation will then be based on evaluating the independent programming project.

However, anecdotal evidence suggests that students are less likely to develop 'centralized' solutions as described in section 53.2.2. This was especially the case in our semester-long course, where the students have to design and implement a parallel programming project of their own choosing. After teaching the 'symmetric' approach, no students wrote code based on a manager-worker model, or using centralized storage. In earlier semesters, we had seen students do this, even though this model was never taught as such.

53.9 Summary

In this paper we have introduced a nonstandard sequence for presenting the basic mechanisms in MPI. Rather than starting with sends and receives and building up from there, we start with mechanisms that emphasize the inherent symmetry between processes in the SPMD programming model. This symmetry requires a substantial shift in mindset of the programmer, and therefore we target it explicitly.

In general, it is the opinion of this author that it pays off to teach from the basis of instilling a mental model, rather than of presenting topics in some order of (perceived) complexity or sophistication.

Comparing our presentation as outlined above to the standard presentation, we recognize the downplaying of the blocking send and receive calls. While students learn these, and in fact learn them before other send and receive mechanisms, they will recognize the dangers and difficulties in using them, and will have the combined `sendrecv` call as well as nonblocking routines as standard tools in their arsenal.

PART VIII

BIBLIOGRAPHY, INDEX, AND LIST OF ACRONYMS

Chapter 54

Bibliography

- [1] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Larsson Träff. MPI on millions of cores. *Parallel Processing Letters*, 21(01):45–60, 2011. [Cited on page 586.]
- [2] Y. Ben-David Kolikant. Gardeners and cinema tickets: High schools’ preconceptions of concurrency. *Computer Science Education*, 11:221–245, 2001. [Cited on page 587.]
- [3] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19:1749–1783, 2007. [Cited on page 79.]
- [4] Tom Cornebize, Franz C Heinrich, Arnaud Legrand, and Jérôme Vienne. Emulating High Performance Linpack on a Commodity Server at the Scale of a Supercomputer. working paper or preprint, December 2017. [Cited on page 559.]
- [5] Lisandro Dalcin. MPI for Python, homepage. <https://github.com/mpi4py/mpi4py>. [Cited on page 11.]
- [6] Lisandro Dalcin and Yao-Lung L. Fang. mpi4py: Status update after 12 years of development. *Computing in Science & Engineering*, 23(4):47–54, 2021. [Cited on page 11.]
- [7] Saeed Dehnadi, Richard Bornat, and Ray Adams. Meta-analysis of the effect of consistency on success in early learning of programming. In *Psychology of Programming Interest Group PPIG 2009*, pages 1–13. University of Limerick, Ireland, 2009. [Cited on page 586.]
- [8] Peter J. Denning. Computational thinking in science. *American Scientist*, pages 13–17, 2017. [Cited on page 586.]
- [9] Victor Eijkhout. *Parallel Programming in MPI and OpenMP*. 2016. available for download: <https://bitbucket.org/VictorEijkhout/parallel-computing-book/src>. [Cited on pages 585 and 596.]
- [10] Victor Eijkhout. Performance of MPI sends of non-contiguous data. *arXiv e-prints*, page arXiv:1809.10778, Sep 2018. [cited on page 312.]
- [11] Brice Goglin. Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc). In *International Conference on High Performance Computing & Simulation (HPCS 2014)*, Bologna, Italy, July 2014. IEEE. [Cited on page 553.]
- [12] Google. Google developer documentation style guide. <https://developers.google.com/style/>. [Cited on page 562.]
- [13] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1994. [Cited on page 318.]
- [14] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. ISBN-10: 0131532715, ISBN-13: 978-0131532717. [Cited on page 588.]
- [15] Torsten Hoefler, Prabhanjan Kambadur, Richard L. Graham, Galen Shipman, and Andrew Lumsdaine. A case for standard non-blocking collective operations. In *Proceedings, Euro PVM/MPI*, Paris, France, October 2007. [Cited on page 72.]
- [16] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. *SIGPLAN Not.*, 45(5):159–168, January 2010. [Cited on page 72.]
- [17] INRIA. SimGrid homepage. <http://simgrid.gforge.inria.fr/>. [Cited on page 559.]

-
- [18] L. V. Kale and S. Krishnan. Charm++: Parallel programming with message-driven objects. In *Parallel Programming using C++, G. V. Wilson and P. Lu, editors*, pages 175–213. MIT Press, 1996. [Cited on page 8.]
- [19] M. Li, H. Subramoni, K. Hamidouche, X. Lu, and D. K. Panda. High performance mpi datatype support with user-mode memory registration: Challenges, designs, and benefits. In *2015 IEEE International Conference on Cluster Computing*, pages 226–235, Sept 2015. [Cited on page 312.]
- [20] Zhenying Liu, Barbara Chapman, Tien-Hsiung Weng, and Oscar Hernandez. Improving the performance of openmp by array privatization. In *Proceedings of the OpenMP Applications and Tools 2003 International Conference on OpenMP Shared Memory Parallel Programming*, WOMPAT'03, pages 244–259, Berlin, Heidelberg, 2003. Springer-Verlag. [Cited on page 410.]
- [21] Robert McLay. T3pio: TACC’s terrific too for parallel I/O. <https://github.com/TACC/t3pio>. [Cited on page 261.]
- [22] MPI forum: MPI documents. <http://www.mpi-forum.org/docs/docs.html>. [Cited on page 585.]
- [23] The OpenMP API specification for parallel programming. <http://openmp.org/wp/openmp-specifications/>. [Cited on page 589.]
- [24] Amit Ruhela, Hari Subramoni, Sourav Chakraborty, Mohammadreza Bayatpour, Pouya Kousha, and Dhabeleswar K. Panda. Efficient asynchronous communication progress for mpi without dedicated resources. EuroMPI’18, New York, NY, USA, 2018. Association for Computing Machinery. [Cited on page 308.]
- [25] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference, Volume 1, The MPI-1 Core*. MIT Press, second edition edition, 1998. [Cited on page 585.]
- [26] Jeff Squyres. Mpi-request-free is evil. Cisco Blogs, January 2013. https://blogs.cisco.com/performance/mpi_request_free_is_evil. [Cited on page 116.]
- [27] R. Thakur, W. Gropp, and B. Toonen. Optimizing the synchronization operations in MPI one-sided communication. *Int'l Journal of High Performance Computing Applications*, 19:119–128, 2005. [Cited on page 252.]
- [28] Universitat Politècnica de Valencia. SLEPC – Scalable software for Eigenvalue Problem Computations. <http://www.grycap.upv.es/slepc/>. [Cited on page 451.]
- [29] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990. [Cited on page 587.]
- [30] P.C. Wason and P.N. Johnson-Laird. *Thinking and Reasoning*. Harmondsworth: Penguin, 1968. [Cited on page 586.]

Chapter 55

List of acronyms

ABI	Application Binary Interface	FEM	Finite Element Method
ADL	Argument-Dependent Lookup	FMA	Fused Multiply-Add
AMG	Algebraic MultiGrid	FEM	Finite Element Method
AMR	Adaptive Mesh Refinement	FIFO	First-in-first-out
AOS	Array-Of-Structures	FMM	Fast Multipole Method
API	Application Programmer Interface	FOM	Full Orthogonalization Method
AVX	Advanced Vector Extensions	FPGA	Field Programmable Gate Array
BEM	Boundary Element Method	FPU	Floating Point Unit
BFS	Breadth-First Search	FFT	Fast Fourier Transform
BLAS	Basic Linear Algebra Subprograms	FSA	Finite State Automaton
BM	Bowers-Moore	FSB	Front-Side Bus
BSP	Bulk Synchronous Parallel	GMRES	Generalized Minimum Residual
BVP	Boundary Value Problem	GPU	Graphics Processing Unit
CAF	Co-array Fortran	GPGPU	General Purpose Graphics Processing Unit
CCS	Compressed Column Storage	GS	Gram-Schmidt
CG	Conjugate Gradients	GSL	Guideline Support Library
CGS	Classical Gram-Schmidt	GnuSL	GNU Scientific Library
COO	Coordinate Storage	GUI	Graphical User Interface
CPP	C Preprocessor	HDFS	Hadoop File System
CPU	Central Processing Unit	HPC	High-Performance Computing
CRS	Compressed Row Storage	HPF	High Performance Fortran
CSP	Communicating Sequential Processes	HPL	High Performance Linpack
CSV	comma-separated values	IBVP	Initial Boundary Value Problem
CTAD	Class Template Argument Deduction	ICV	Internal Control Variable
TAD	Template Argument Deduction	IDE	Integrated Development Environment
CUDA	Compute-Unified Device Architecture	ILP	Instruction Level Parallelism
DAG	Directed Acyclic Graph	ILU	Incomplete LU
DFS	Depth First Search	IMP	Integrative Model for Parallelism
DL	Deep Learning	IVP	Initial Value Problem
DPCPP	Data Parallel C++	LAPACK	Linear Algebra Package
DRAM	Dynamic Random-Access Memory	LAN	Local Area Network
DSP	Digital Signal Processing	LBM	Lattice Boltzmann Method
FD	Finite Difference	LRU	Least Recently Used
FDM	Finite Difference Method	MG	Multigrid
FE	Finite Elements	MGS	Modified Gram-Schmidt

MIC	Many Integrated Cores	SCS	Shortest Common Superset
MIMD	Multiple Instruction Multiple Data	SFC	Space-Filling Curve
MGS	Modified Gram-Schmidt	SGD	Stochastic Gradient Descent
MKL	Math Kernel Library	SIMD	Single Instruction Multiple Data
ML	Machine Learning	SIMT	Single Instruction Multiple Thread
MPI	Message Passing Interface	SLURM	Simple Linux Utility for Resource Management
MPL	Message Passing Library	SM	Streaming Multiprocessor
MPMD	Multiple Programm Multiple Data	SMP	Symmetric Multi Processing
MSI	Modified-Shared-Invalid	SMT	Symmetric Multi Threading
MTA	Multi-Threaded Architecture	SOA	Structure-Of-Arrays
MTSP	Multiple Traveling Salesman Problem	SOR	Successive Over-Relaxation
NCCL	NVidia Collective Communication Library	SSOR	Symmetric Successive Over-Relaxation
NIC	Network Interface Card	SP	Streaming Processor
NTTP	Non-Type Template Parameter	SPMD	Single Program Multiple Data
NUMA	Non-Uniform Memory Access	SPD	symmetric positive definite
ODE	Ordinary Differential Equation	SRAM	Static Random-Access Memory
OO	Object-Oriented	SSE	SIMD Streaming Extensions
OOP	Object-Oriented Programming	SSSP	Single Source Shortest Path
OS	Operating System	STL	Standard Template Library
PGAS	Partitioned Global Address Space	TBB	Threading Building Blocks
PDE	Partial Differential Equation	TDD	Test-Driven Development
PRAM	Parallel Random Access Machine	TLB	Translation Look-aside Buffer
RDMA	Remote Direct Memory Access	TSP	Traveling Salesman Problem
RMA	Remote Memory Access	UB	Undefined Behavior
RNG	Random Number Generator	UMA	Uniform Memory Access
RVO	Return Value Optimization	UPC	Unified Parallel C
SAN	Storage Area Network	URI	Uniform Resource Identifier
SAS	Software As a Service	WAN	Wide Area Network

Chapter 56

General Index

accelerator, 325
active target synchronization, 223, 229
address
 physical, 285, 408
 virtual, 285, 408
adjacency
 graph, 477
affinity, 552, 557
 process and thread, 552–553
 thread
 on multi-socket nodes, 404
alignment, 177
all-to-all, 32
allocate
 and private/shared data, 380
allreduce, 31
AMD
 Milan, 411
array
 static, 89
atomic operation, 386, 442
 file, 263
 MPI, 238–243
 OpenMP, 388–390
bandwidth, 78
 bisection, 83
barrier
 for timing, 311
 implicit, 388
 nonblocking, 76
Basic Linear Algebra Subprograms (BLAS), 454
batch
 job, 8
 scheduler, 8
Beowulf cluster, 7

block row, 465
Boolean satisfiability, 27
boost, 10
broadcast, 30
bucket brigade, 55, 82, 97
buffer
 MPI, in C, 36
 MPI, in Fortran, 37
 MPI, in MPL, 38
 MPI, in Python, 37
receive, 90

C

C11, 145
C99, 144
MPI bindings, *see MPI, C bindings*

C++

 bindings, *see MPI, C++ bindings*
 C++17, 352
 C++20, 348, 352
 C++23, 348
 C++32, 352
 first-touch, *see first-touch, in C++ standard library*, 167
 vector, 167

C++ iterators

 in OMP reduction, 367

cacheline, 361

callback, 497

cast, 36

channel, 547

Charmpp, 8

chunk, 351

Clang, 416

clang, 9

client, 214

cluster, 326

Codimension, 519

coherent memory, *see memory, coherence*

collective

 split, 259

collectives, 30

 neighborhood, 274, 313

 nonblocking, 72

 cancelling, 318

column-major storage, 160

combiner, 185

commandline arguments

 broadcast, 43

of spawned process, 210
communication
 asynchronous, 137–138
 blocking, 92–97
 vs nonblocking, 312
 buffered, 138, 139, 312
 local, 138
 nonblocking, 103–116
 nonlocal, 138
 one-sided, 223–252
 one-sided, implementation of, 252
 partitioned, *see* partitioned, communication
 persistent, 312
 synchronous, 137–138
 two-sided, 129
communicator, 23, 196–209
 info object, 299
 inter, 205, 205, 210
 from socket, 218
 intra, 205, 207
 object, 24
 peer, 205
 variable, 23
compare-and-swap, 101
compiler, 171
completion, 245
 local, 245
 remote, 246
Compute-Unified Device Architecture (CUDA), 518
concurrency
 and MPI, *see* MPI, concurrency and
condition number, 496
construct, 331
contention, 83
contention group, 338
contiguous
 data type, 152
control variable
 access, 293
 handle, 292
core, 15, 326, 404
cosubscript, 520
cpp, 328
cpuinfo, 550
Cray
 MPI, 9
 T3E, 318
critical section, 360, 388, 423
 flush at, 393

curly braces, 329

Dalcin

- Lisandro, 11, 446

data dependency, 399

Data Parallel C++ (DPCPP), 528

datatype, 143–189

- big, 172–176
- derived, 143, 152–254
- different on sender and receiver, 157
- predefined, 143–152
 - in C, 144
 - in Fortran, 145
 - in Python, 148
- signature, 169

deadlock, 75, 89, 93, 103, 312, 314

Dekker’s algorithm, 427

dense linear algebra, 199

Depth First Search (DFS), 435

destructor, 196

directive, 329–330

- C++ syntax, 329
- end-of, 330

displacement unit, 249

distributed array, 28

distributed shared memory, 223

doubling

- recursive, *see* recursive doubling

eager

- limit, 93
- send, 93

eager limit, 93–95, 311

eager send

- and non-blocking, 95

edge

- cuts, 477
- weight, 477

envelope, *see* message, envelope

environment variables, 289

epoch, 229

- access, 229, 231, 244
- communication, 229
- completion, 247
- exposure, 229, 230
- passive target, 243

error return, 10

ethernet, 10

execution policy, 352, 369

execution space, 525

false sharing, 361, 362
Fast Fourier Transform (FFT), 48, 474, 475
fat-tree, 553
fence, 229
fftw, 451, 474
Fibonacci sequence, 393–395
file
 pointer
 advance by write, 260
 individual, 259
file system
 shared, 254
first-touch, 407, 553
 in C++, 409
five-point stencil, 75
fork/join model, 327, 333, 399
Fortran
 1-based indexing, 110
 2008, 24
 array syntax, 373, 374
 assumed-shape arrays in MPI, 308
 fixed-form source, 330
 forall loops, 374
 Fortran2003, 365
 Fortran2008, 118, 143, 153, 159, 234
 MPI bindings, *see* MPI, Fortran2008 bindings
 Fortran2018, 308, 330
 Fortran77, 69, 330
 PETSc interface, 445
 Fortran90, 12, 119, 146, 159
 bindings, 11
 PETSc interface, 445
 line length, 507
 MPI bindings, *see* MPI, Fortran bindings
 MPI equivalences of scalar types, 146
 MPI issues, 307–308

gather, 30
Gauss-Jordan algorithm, 45
Gaussian elimination, 489
GCC, 416
gcc
 thread affinity, 410
ghost region, 578
Google
 developer documentation style guide, 562
GPUDirect, 503
Gram-Schmidt, 35
graph

partitioning
 packages, 477
topology, 274, 313, 553
unweighted, 275

grid
 Cartesian, 266, 478
 periodic, 266
 processor, 553

group, 230

group of
 processors, 232

halo, 578
 update, 235

halo region, 475, 481

handshake, 314

hdf5, 254

heap, 332

heat equation, 406

Hessian, 500

hipsycl, 535

histogram, 429

hostname, 302

hwloc, 553

hydra, 217

hyper-thread, 404

hyperthread, 346

hyperthreading, 553

Hypre, 451, 494, 496
 pilut, 495

I/O
 in OpenMP, 372

IBM
 Power9, 412

ICC, 416

image, 519

image_index, 520

immediate operation, 103

incomplete operation, *see* operation, incomplete

indexed
 data type, 152

inner product, 35

input redirection
 shell, 314

Intel, 313
 Cascade Lake, 345, 352, 408, 411, 431

compiler
 optimization report, 346
 thread affinity, 410

compiler suite, 550
Haswell, 553
Knight's Landing, 418
 thread placement, 407
Knights Landing, 346, 411, 557
MPI, 9, 84, 95, 213, 308, 315, 556
Paragon, 308
Sandybridge, 326, 553
Skylake, 411
TBB, 518
Xeon PHI, 325
Internal Control Variable (ICV), 422–423

Java, 7

Kokkos
 and OpenMP, 527

Laplace equation, 467
latency, 78, 319
 hiding, 114, 312, 460, 462, 469
lcobound, 520
league(OpenMP), 421
lexical scope, 375
Linear Algebra Package (LAPACK), 454
linked list, 397
linker
 weak symbol, 311
load
 balancing, 349
 imbalance, 349
local
 operation, 138, 139
local operation, 115
local refinement, 76
lock, 390, 390–393
 flush at, 393
 nested, 393
Lonestar5, 553
LU factorization, 350, 489

macports, 303
malloc
 and private/shared data, 380
manager-worker, 115, 118, 123, 570
Mandelbrot set, 27, 82, 383, 392, 569
matching, 314
matching queue, 125
matrix
 sparse, 72, 470

transposition, 200
matrix-vector product, 489
 dense, 54
 sparse, 57
Mellanox, 313
memory
 coherent, 247
 high-bandwidth, 384
 model, *see* window, memory, model
 non-volatile, 384
 page, 408
 shared (GPU), 543
 shared, MPI, 226
memory leak, 116
memory space, 526
message
 collision, 313
 count, 120
 envelope, 93, 126
 non-overtaking, 119, 552
 source, 91
 status, 92, 116–124
 error, 120
 source, 118
 tag, 119
 synchronous, 93
 tag, 89, 290
Message Passing Library (MPL), 10, 26
messsage
 target, 89
MKL, 454
mkl, 451
ML, 496
Monte Carlo codes, 27
MPI
 accelerator memory allocation, 247
 C bindings, 10
 C++ bindings, 10
 concurrency and, 552
 concurrent Accumulate calls, 250
 concurrent file operations, 256
 constants, 316–317
 compile-time, 316
 link-time, 316
datatype
 extent, 176
 size, 151
 subarray, 177
 vector, 151

Fortran bindings, 10–11
Fortran issues, *see* Fortran, MPI issues
Fortran2008 bindings, 10–11
I/O, 306
initialization, 18
MPI-1, 266, 280
MPI-2, 210, 300, 305
MPI-3, 34, 72, 163, 167, 174, 238, 282, 308
 C++ bindings removed, 10
 Fortran2008 interface, 10
MPI-3.0, 291
MPI-3.1, 291, 316
MPI-3.2, 318
MPI-4, 10, 34, 152, 172, 201, 305
MPI-4.0, 291
MPI-4.1, 18, 116, 135, 140, 174, 201, 227, 247, 284, 301
Python bindings, 11
semantics, 313–314
tools interface, 291–296, 311
version, 303
MPI/O, 254–264
mpi4py, 518
mpi4py, 11
mpi_f08, 10
mpicc, 9
mpich, 9
mpiexec
 and environment variables, 289
 options, 9
 stdout/err of, 315
MPICH, 313
mpirun, 8, 21
MPL, 10
 compiling and linking, 10
multicore, 327
Multiple Programm Multiple Data (MPMD), 17, 315, 316
multiprocessing, 545
Mumps, 451
mumps, 451
mvapich, 289
mvapich2, 95, 558

N-body problem, 429
name server, 217
nested parallelism, 337–339
netcdf, 254
network
 card, 313
 contention, 313

port
 oversubscription, 313

Newton's method, 500

node, 15
 cluster, 325

non-blocking communication, 101

Non-Uniform Memory Access (NUMA), 551

norm
 one, 71

null terminator, 297

null-terminator, 210, 214

num_images, 520

numactl, 410

numerical integration, 362

Numpy, 148
 1.20, 148

numpy, 11, 37, 150, 226

NVidia Collective Communication Library (NCCL), 518

offloading
 vs onloading, 313

omp
 reduction, 359–369
 roundoff, 360
 user-defined, 364–368

OneAPI, 528

onloading, *see* offloading, vs onloading

opaque handle, 24, 36

OpenMP, 282
 accelerator support in, 425
 co-processor support in, 425
 compiling, 328
 directive, *see* directive
 environment variables, 329, 333, 422–423
 library routines, 333
 library routines, 422–423
 macro, 328
 OpenMP-3, 344
 OpenMP-3.1, 363, 426
 OpenMP-4, 438
 OpenMP-4.0, 326, 402, 420, 425, 426
 OpenMP-4.5, 363, 426
 OpenMP-5, 384
 OpenMP-5.0, 347, 368, 401, 426
 OpenMP-5.1, 351, 373, 427
 OpenMP-5.2, 363, 373, 427
 places, 404
 running, 329
 standard, 328

standard versions, 426–427
tasks, 396–403
 data, 397–398
 dependencies, 399–401
 synchronization, 398–399
OpenMPI, 9, 95, 202
operating system, 425
operation
 non-local, 93
operator, 68–71
 predefined, 68
 user-defined, 69
option
 prefix, 512
origin, 223, 231
overlapping computation and communication, *see* latency, hiding
owner computes, 86

package, 550
packing, 186
page
 small, 285
 table, 285, 408
page, memory, *see* memory, page
paging, 542
parallel
 data, 410
 embarrassingly, 410
 prefix, *see* prefix
parallel region, 327, 335–339, 371
 dynamic scope, 338, 377
 flush at, 393
parallel regions
 nested, 423
parallelism
 nested, 337
parameter sweep, 546
parasails, 495
ParMetis, 313, 477
partitioned communication, 135–137
passive target synchronization, 224, 240, 243
persistent
 collectives, 133–135
 communication, 135
 point-to-point, 131–133
persistent communication, 101, *see* communication, persistent
PETSc, 313
 interoperability with BLAS, 454
 interoperability with MPI, 454

log files, 450
PETSc-3.17, 503
PETSc-3.18, 505, 507
pin a thread, 553
ping-pong, 87, 311, 521
pipe, 547
point-to-point, 86
pointer
 null, 53
polling, 111, 309
posting
 of send/receive, 104
pragma, 329
preconditioner, 490, 493
 block jacobi, 512
 field-split, 474
prefix
 operation, 368
prefix operation, 47
private variables, 332
process, 15
 set, 220
processes status of, 315
producer-consumer, 441, 574
progress
 asynchronous, 114, 308
protocol, 93
 rendezvous, 93
pthreads, 308, 518
PVM, 8, 210
Python
 MPI bindigs, *see* MPI, Python bindings
 PETSc interface, 446
queue
 in-order, 530
 SYCL, 529
race condition, 238, 288, 331, 359, 424, 439, 442
 in co-array Fortran, 520
 in MPI/OpenMP, 288
 in SYCL, 534
radix sort, 56
random number generator, 383, 424
Ranger, 553
ray tracing, 286
recursive doubling, 83
redirection, *see* shell, input redirection
reduction, 30
region of code, 331

register
 SSE2, 418

residual, 489

Riemann
 sum, 359

Riemann sums, 362

RMA
 active, 223
 passive, 224

root, 39

root process, 30

row-major, 356

scalable
 in space, 82
 in time, 82

scalapack, 498

scan, 32
 exclusive, 48
 inclusive, 47

scatter, 30

scope
 lexical, 331
 of variables, 331

segfault, 507

segmented scan, 50

send
 buffer, 89
 ready mode, 138
 synchronous, 138

sentinel, 329

sequential
 semantics, 445

sequential consistency, 427

serialization, 96

server, 214

session, 219
 performance experiment, 294

session model, 218, 218

sessions model, 19

shared data, 327

shared memory, *see* memory, shared

shared variables, 332

shell
 matrix, 497

shmem, 318

silo, 254

simd
 width, 351

SimGrid, 80
Single Program Multiple Data (SPMD), 330, 335
Slepc, 451
SLURM, 265
socket, 15, 218, 284, 325, 404, 551, 557
 dual, 408
sort
 odd-even transposition, 101
 radix, 56
 swap, 101
sparse approximate inverse, 493, 494
sparse matrix vector product, 50
spin loop, 308
spin-lock, 423
ssh, 8
stack, 332, 423
 overflow, 377
 per thread, 377
Stampede
 compute node, 553
 largemem node, 553
standard deviation, 31
status
 of received message, 117
stderr, 315
stdout, 315
stencil, 478
storage association, 376, 379
stride, 160
struct
 data type, 152
structured block, 331
Sun
 compiler, 410
superstep, 224
SYCL, 528
 SYCL-2020, 529–531, 538
sync, 520
synchronization
 in OpenMP, 386–395

T3PIO, 261
TACC
 Frontera, 345, 352, 374, 408, 431
 portal, 446
 Stampede, 325
 Stampede2, 346
tacc_affinity, 410
tag, *see* message, tag

bound on value, 301
target, 223, 231
 active synchronization, *see* active target synchronization
 passive synchronization, *see* passive target synchronization
task
 generating, 420
 initial, 420
 scheduler, 397
 scheduling point, 401
 target, 420
TBB, *see* Intel, TBB
team, 341
team(OpenMP), 421
this_image, 520
thread
 affinity, 404–407
 initial, 336
 master, 336
 migrating a, 407
 primary, 337
 private data, 382
thread-safe, 424
threads, 327
 hardware, 327, 553
 master, 327
 team of, 327, 336
time slicing, 15, 327
time-slicing, 210
timing
 MPI, 309–311
topology
 virtual, 266
transpose, 75
 and all-to-all, 55–56
 data, 475
 recursive, 200
 through derived types, 185
tree
 traversal
 post-order, 434
tunnel
 ssh, 314

ucobound, 520
Unix
 process, 377

vector
 data type, 152
 instructions, 416

virtual shared memory, 223
wall clock, 309
warp, 541
 divergence, 544
weak symbol, *see* linker, weak symbol
while loop, 397
while loops, 357
window, 223–228
 consistency, 245, 246
 displacement, 232
 displacement unit, 250
 info object, 299
 memory, *see also* memory model
 model, 247
 separate, 247
 memory allocation, 225–228
 private, 247
 public, 247
work sharing, 327
work sharing construct, 333, 371
workshare
 flush after, 393
worksharing
 construct, 341
world model, 218
wormhole routing, 83
wraparound connections, 266

X11, 470, 510
XSEDE
 portal, 446

Zoltan, 313, 477

Chapter 57

Lists of notes

57.1 MPI-4 notes

List of MPI-4 notes

1	Non-blocking/persistent sendrecv	101
2	Status access	117
3	Persistent collectives	133
4	Neighborhood collectives, init	135
5	Partitioned communication	135
6	Buffer per communicator/session	139
7	Buffer on communicator	140
8	U	141
9	MPI Count type	172
10	No more x routines	174
11	Extent result as count	179
12	Split by guided types	201
13	Session model	218
14	Info key for alignment	227
15	Memory allocation kinds	247
16	Window memory alignment	285
17	Tools callback interface	293
18	Info with null terminator	297
19	Memory alignment	300
20	Error code for aborted process	304
21	Error handler for session	305
22	Abort on communicator	305

57.2 Fortran notes

List of Fortran notes

MPI	6
1 Formatting of Fortran notes	10
2 MPI module	18
3 Processor name	23
4 Communicator type	24
5 MPI send/recv buffers	37
6 In-place operations	42
7 Min/maxloc types	69
8 Index of requests	110
9 Status object in f08	117
10 Derived types for handles	143
11 Byte counting types in Fortran	149
12 Subarrays	163
13 Extent as Aint	183
14 Displacement unit	233
15 Offset literals	262
16 Attribute querying	302
17 Fortran-only compile-time constants	317
OpenMP	324
18 OpenMP version	328
19 OpenMP sentinel	329
20 OMP do pragma	342
21 Missing sentinel for loops	342
22 Reduction declaration	365
23 Reductions on derived types	368
24 Private variables in parallel region	376
25 Saved variables	378
26 Private common blocks	383
27 Array sizes in map clause	421
PETSc	444
28 Petsc Initialization	448
29 Setting values	461
30 F90 array access through pointer	463
31 Backtrace on error	506
32 Error code handling	507
33 Print string construction	508
34 Printing and newlines	508
Other	518

57.3 C++ notes

List of C++ notes

MPI	6
1 Buffer treatment	36
OpenMP	324
2 Bracket syntax	329
3 Output streams in parallel	336
4 Parallel regions in lambdas	336
5 Dynamic scope for class methods	339
6 Custom iterators	343
7 Range syntax	347
8 C++20 ranges header	348
9 C++20 ranges speedup	348
10 Ranges and indices	348
11 Parallel standard algorithms	352
12 Performance comparison	354
13 Reductions on vectors	363
14 Lambda expressions in declared reductions	366
15 Reduction over iterators	367
16 Templated reductions	367
17 Example: reduction over a map	367
18 Reduction on class objects	368
19 Reductions on parallel standard algorithms	369
20 Privatizing class members	376
21 Vectors are copied, unlike arrays	381
22 Threadprivate random number generators	384
23 Threadprivate random use	384
24 Lock inside overloaded operator	392
25 Use mdspan for submatrices	402
26 Uninitialized containers	409
27 List filtering example	438
PETSc	444
28 Exception handling	507

57.4 The MPL C++ interface

List of Mpl notes

1	Notes format	10
2	Header file	18
3	Init, finalize	19
4	Processor name	22
5	World communicator	24
6	Communicator copying	24
7	Communicator passing	24
8	Rank and size	27
9	Allreduce operator	35
10	Scalar buffers	38
11	Vector buffers	38
12	Array buffers	38
13	Iterator buffers	38
14	Send vs recv buffer	39
15	Reduce in place	43
16	Broadcast	45
17	Scan operations	48
18	Gather/scatter	53
19	Gather on nonroot	53
20	Operators	69
21	User-defined operators	71
22	Lambda operator	71
23	Nonblocking collectives	73
24	Null processor	89
25	Buffer type safety	89
26	Blocking send and receive	89
27	Sending automatic arrays	89
28	Sending arrays	90
29	Iterator layout	90
30	Any source	91
31	Send-recv call	99
32	Requests from nonblocking calls	105
33	Request pools	111
34	Request pool status	111
35	Wait any	111
36	Request handling	111
37	Status object	118
38	Status querying	119
39	Message tag	119

40	Tag types	119
41	Receive count	121
42	Persistent requests	130
43	Buffered send	141
44	Buffer attach and detach	141
45	Datatype handling	144
46	Native MPI data types	144
47	Derived type handling	153
48	Layouts	154
49	Contiguous type	156
50	Contiguous composing	156
51	Vector type	159
52	Subarray layout	162
53	Indexed type	166
54	Layouts for gatherv	167
55	Indexed block type	167
56	Struct type scalar	171
57	Struct type general	171
58	Large counts	172
59	Extent resizing	184
60	Predefined communicators	192
61	Raw communicator handles	192
62	Communicator duplication	193
63	Communicator comparing	194
64	Communicator splitting	200
65	Raw group handles	204
66	File opening	255
67	File writing	258
68	Cartesian communicator	267
69	Dims create	267
70	Cartesian communicator create	268
71	Get the dimensions object	269
72	Rank to coord translation	270
73	Cartesian shifting	271
74	Distributed graph creation	275
75	Graph communicators	278
76	Graph communicator querying	278
77	Split by shared memory	283
78	Threading support	289
79	Info objects	297
80	Communicator errhandler	306
81	Timing	310

57.5 Python notes

List of Python notes

MPI	6
1 Running mpi4py programs	10
2 Python notes	11
3 Import mpi module	18
4 Initialize/finalize	19
5 Communicator objects	24
6 Communicator rank and size	26
7 Buffers from numpy	37
8 Buffers from subarrays	37
9 In-place collectives	43
10 Sending objects	44
11 Define reduction operator	70
12 Reduction function	70
13 Message tags	91
14 Handling a single request	109
15 Request arrays	109
16 Status object	117
17 Data types	143
18 Predefined data types	148
19 Size of numpy types	150
20 Derived type handling	153
21 Vector type	159
22 Sending from the middle of a matrix	161
23 Big data	176
24 Communicator types	192
25 Communicator duplication	192
26 Comm split key is optional	199
27 Displacement byte computations	226
28 Window buffers	228
29 MPI one-sided transfer routines	234
30 File open is class method	256
31 Graph communicators	278
32 Thread level	288
33 Universe size	302
34 MPI Version	303
35 Utility functions	303
36 Error policy	305
OpenMP	324
PETSc	444

LIST OF PYTHON NOTES

37 Init, and with commandline options	449
38 Communicator object	449
39 petsc4py interface	451
40 Vector creation	455
41 Vector size	457
42 Vector operations	459
43 Setting vector values	461
44 Vector access	464
45 Petsc print and python print	508
46 HDF5 file generation	510
47 Petsc options	512
Other	518

Chapter 58

Index of MPI commands and keywords

58.1 From the standard document

This is an automatically generated list of every function, type, and constant in the MPI standard document. Where these appear in this book, a page reference is given.

58.1.1 List of all functions

- MPI_Abort 20
- MPI_Accumulate 236
- MPI_Address ??
- MPI_Add_error_class 306
- MPI_Add_error_code 307
- MPI_Add_error_string 307
- MPI_Aint_add 149
- MPI_Aint_diff 149
- MPI_Allgather 54
- MPI_Allgatherv 63
- MPI_Allgatherv_init 135
- MPI_Allgather_init 135
- MPI_Alloc_mem 227
- MPI_Alloc_mem_cptr ??
- MPI_Allreduce 33
- MPI_Allreduce_init 133
- MPI_Alltoall 55
- MPI_Alltoallv 56
- MPI_Alltoallv_init 135
- MPI_Alltoallw ??
- MPI_Alltoallw_init 135
- MPI_Alltoalll_init 135
- MPI_Attr_delete ??
- MPI_Attr_get 300
- MPI_Attr_put ??
- MPI_Accumulate 236
- MPI_Barrier 62
- MPI_Barrier_init 134
- MPI_Bcast 43
- MPI_Bcast_init 134
- MPI_Bsend 139
- MPI_Bsend_init 142
- MPI_Buffer_attach 140
- MPI_Buffer_detach 141
- MPI_Buffer_flush 140
- MPI_Buffer_iflush ??
- MPI_Cancel 317
- MPI_Cartdim_get 269
- MPI_Cart_coords 269
- MPI_Cart_create 267
- MPI_Cart_get 269
- MPI_Cart_map 273
- MPI_Cart_rank 269
- MPI_Cart_shift ??
- MPI_Cart_sub 272
- MPI_Close_port 215
- MPI_Comm_accept 214
- MPI_Comm_attach_buffer 140
- MPI_Comm_call_errhandler ??
- MPI_Comm_compare 193
- MPI_Comm_connect 215
- MPI_Comm_create 202
- MPI_Comm_create_errhandler 306
- MPI_Comm_create_from_group ??
- MPI_Comm_create_group 203
- MPI_Comm_create_keyval 302
- MPI_Comm_delete_attr 302
- MPI_Comm_detach_buffer 140
- MPI_Comm_disconnect 197
- MPI_Comm_dup 192
- MPI_Comm_dup_with_info 192
- MPI_Comm_flush_buffer 140
- MPI_Comm_free 197
- MPI_Comm_free_keyval 302
- MPI_Comm_get_attr 300
- MPI_Comm_get_errhandler 305
- MPI_Comm_get_info 299
- MPI_Comm_get_name ??
- MPI_Comm_get_parent 207
- MPI_Comm_group 202
- MPI_Comm_idup 192
- MPI_Comm_idup_with_info 192
- MPI_Comm_iflush_buffer ??
- MPI_Comm_join 218
- MPI_Comm_rank 25
- MPI_Comm_remote_size 207
- MPI_Comm_set_attr 300
- MPI_Comm_set_errhandler 305
- MPI_Comm_set_info 299
- MPI_Comm_set_name 192
- MPI_Comm_size 25

- MPI_Comm_spawn 210
- MPI_Comm_spawn_multiple 214
- MPI_Comm_split 198
- MPI_Comm_split_type 201
- MPI_Comm_test_inter 207
- MPI_Compare_and_swap 241
- MPI_Compare_and_swap 241
- MPI_Dims_create 267
- MPI_Dist_graph_create 275
- MPI_Dist_graph_create_adjacent ??
- MPI_Dist_graph_neighbors 280
- MPI_Dist_graph_neighbors_count 280
- MPI_Errhandler_create 305
- MPI_Errhandler_free 305
- MPI_Errhandler_get ??
- MPI_Errhandler_set ??
- MPI_Error_class ??
- MPI_Error_string 306
- MPI_Exscan 48
- MPI_Exscan_init 135
- MPI_Fetch_and_op 238
- MPI_File_call_errhandler 305
- MPI_File_close 255
- MPI_File_create_errhandler ??
- MPI_File_delete 256
- MPI_File_get_amode ??
- MPI_File_get_atomicity ??
- MPI_File_get_byte_offset ??
- MPI_File_get_errhandler ??
- MPI_File_get_group ??
- MPI_File_get_info ??
- MPI_File_get_position ??
- MPI_File_get_position_shared ??
- MPI_File_get_size 262
- MPI_File_get_type_extent ??
- MPI_File_get_view 262
- MPI_File_iread 259
- MPI_File_iread_all 259
- MPI_File_iread_at 259
- MPI_File_iread_at_all 259
- MPI_File_iread_shared 262
- MPI_File_iwrite 259
- MPI_File_iwrite_all 259
- MPI_File_iwrite_at 259
- MPI_File_iwrite_at_all 259
- MPI_File_iwrite_shared 262
- MPI_File_open 255
- MPI_File_reallocate 262
- MPI_File_read 257
- MPI_File_read_all 257
- MPI_File_read_all_begin 259
- MPI_File_read_all_end 259
- MPI_File_read_at 257
- MPI_File_read_at_all 257
- MPI_File_read_at_all_begin ??
- MPI_File_read_at_all_end ??
- MPI_File_read_errhandler 305
- MPI_File_set_atomicity 263
- MPI_File_set_errhandler ??
- MPI_File_set_size 262
- MPI_File_set_view 261
- MPI_File_sync 256
- MPI_File_write 257
- MPI_File_write_all 257
- MPI_File_write_all_begin 259
- MPI_File_write_all_end 259
- MPI_File_write_at 260
- MPI_File_write_at_all 258
- MPI_File_write_at_all_begin ??
- MPI_File_write_at_all_end ??
- MPI_File_write_ordered 262
- MPI_File_write_ordered_begin ??
- MPI_File_write_ordered_end ??
- MPI_File_write_shared 262
- MPI_Finalize 18
- MPI_Finalized 20
- MPI_Free_mem 227
- MPI_F_sync_reg ??
- MPI_Gather 52
- MPI_Gatherv 63
- MPI_Gatherv_init 134
- MPI_Gather_init 134
- MPI_Get 234
- MPI_Get_accumulate 238
- MPI_Get_address 167
- MPI_Get_count 120
- MPI_Get_elements 121
- MPI_Get_elements_x 175
- MPI_Get_hw_resource_info 201
- MPI_Get_library_version 303
- MPI_Get_processor_name 21
- MPI_Get_version 303
- MPI_Graphdims_get 280
- MPI_Graph_create 280
- MPI_Graph_get 280
- MPI_Graph_map 280
- MPI_Graph_neighbors 280
- MPI_Graph_neighbors_count 280
- MPI_Grequest_complete ??
- MPI_Grequest_start ??
- MPI_Group_compare ??
- MPI_Group_excl 203
- MPI_Group_free ??
- MPI_Group_from_session_pset ??
- MPI_Group_incl 203
- MPI_Group_range_excl ??
- MPI_Group_range_incl ??
- MPI_Group_rank ??
- MPI_Group_size ??
- MPI_Group_translate_ranks ??
- MPI_Get_elements_c ??
- MPI_Iallgather 73
- MPI_Iallgatherv 73
- MPI_Iallreduce 73
- MPI_Ialltoall 73
- MPI_Ialltoallv 73
- MPI_Ialltoallw 73
- MPI_Ibarrier 76
- MPI_Ibcast 73
- MPI_Ibsend ??
- MPI_Iexscan 73
- MPI_Igather 73
- MPI_Igatherv 73
- MPI_Improbe ??
- MPI_Imrecv ??
- MPI_Ineighbor_allgather ??
- MPI_Ineighbor_allgatherv ??
- MPI_Ineighbor_alltoall ??
- MPI_Ineighbor_alltoallv ??
- MPI_Ineighbor_alltoallw ??
- MPI_Info_create 297
- MPI_Info_create_env ??
- MPI_Info_delete 297
- MPI_Info_dup 297
- MPI_Info_free 297
- MPI_Info_get 297
- MPI_Info_get_nkeys 297
- MPI_Info_get_nthkey 297
- MPI_Info_get_string 297
- MPI_Info_get_valuelen ??
- MPI_Info_set 297
- MPI_Init 18
- MPI_Initialized 20
- MPI_Init_thread 288
- MPI_Intercomm_create 205
- MPI_Intercomm_create_from_groups ??
- MPI_Intercomm_merge 208
- MPI_Iprobe 124
- MPI_Irecv 103
- MPI_Ireduce 73
- MPI_Ireduce_scatter 73
- MPI_Ireduce_scatter_block 73
- MPI_Irsend ??

- MPI_Iscan 73
- MPI_Iscatter 73
- MPI_Iscatterv 73
- MPI_Isend 103
- MPI_Isendrecv 101
- MPI_Isendrecv_replace 101
- MPI_Issend 138
- MPI_Is_thread_main 288
- MPI_Keyval_create ??
- MPI_Keyval_free ??
- MPI_Lookup_name ??
- MPI_Mprobe 125
- MPI_Mrecv 125
- MPI_Neighbor_allgather 278
- MPI_Neighbor_allgatherv ??
- MPI_Neighbor_allgatherv_init 135
- MPI_Neighbor_allgather_init 135
- MPI_Neighbor_alltoall ??
- MPI_Neighbor_alltoallv ??
- MPI_Neighbor_alltoallv_init 135
- MPI_Neighbor_alltoallw ??
- MPI_Neighbor_alltoallw_init 135
- MPI_Neighbor_alltoalll_init 135
- MPI_Open_port 214
- MPI_Op_commutative 71
- MPI_Op_create 69
- MPI_Pack 186
- MPI_Pack_external ??
- MPI_Pack_size 188
- MPI_Parived 137
- MPI_Pcontrol ??
- MPI_Pready 136
- MPI_Pready_list 136
- MPI_Pready_range 136
- MPI_Precv_init 137
- MPI_Probe 124
- MPI_Psend_init 135
- MPI_Publish_name 216
- MPI_Put 232
- MPI_Put 232
- MPI_Query_thread 288
- MPI_Raccumulate ??
- MPI_Recv 90
- MPI_Recv_init 131
- MPI_Reduce 39
- MPI_Reduce_init 134
- MPI_Reduce_local 71
- MPI_Reduce_scatter 60
- MPI_Reduce_scatter_block 57
- MPI_Reduce_scatter_block_init 134
- MPI_Reduce_scatter_init 134
- MPI_Register_datarep ??
- MPI_Remove_error_class ??
- MPI_Remove_error_code ??
- MPI_Remove_error_string ??
- MPI_Request_free 116
- MPI_Request_get_status 116
- MPI_Request_get_status_all 116
- MPI_Request_get_status_any 116
- MPI_Request_get_status_some 116
- MPI_Rget ??
- MPI_Rget_accumulate ??
- MPI_Rput 237
- MPI_Rsend ??
- MPI_Scan 47
- MPI_Scan_init 135
- MPI_Scatter 51
- MPI_Scatterv 63
- MPI_Scatterv_init 135
- MPI_Scatter_init 135
- MPI_Send 88
- MPI_Sendrecv 97
- MPI_Send_init 131
- MPI_Session_attach_buffer 140
- MPI_Session_call_errhandler 305
- MPI_Session_create_errhandler 220
- MPI_Session_detach_buffer 140
- MPI_Session_finalize 219
- MPI_Session_flush_buffer 140
- MPI_Session_get_info 219
- MPI_Session_get_nth_pset 220
- MPI_Session_get_num_pssets 220
- MPI_Session_iflush_buffer ??
- MPI_Session_init 219
- MPI_Sizeof 152
- MPI_Ssend 138
- MPI_Start 131
- MPI_Startall 131
- MPI_Status_get_error ??
- MPI_Status_get_source ??
- MPI_Status_get_tag ??
- MPI_Status_set_cancelled ??
- MPI_Status_set_elements ??
- MPI_Status_set_elements_x ??
- MPI_Status_set_error ??
- MPI_Status_set_source ??
- MPI_Status_set_tag ??
- MPI_Send 88
- MPI_Status_set_elements_c ??
- MPI_Test 115
- MPI_Testall 115
- MPI_Testany 115
- MPI_Testsome 115
- MPI_Test_cancelled ??
- MPI_Topo_test 268
- MPI_Type_commit 154
- MPI_Type_contiguous 155
- MPI_Type_create_darray ??
- MPI_Type_create_hindexed ??
- MPI_Type_create_hindexed_block 167
- MPI_Type_create_hvector ??
- MPI_Type_create_indexed_block ??
- MPI_Type_create_keyval 302
- MPI_Type_create_resized 180
- MPI_Type_create_struct 167
- MPI_Type_create_subarray 162
- MPI_Type_delete_attr 302
- MPI_Type_dup ??
- MPI_Type_extent ??
- MPI_Type_free 154
- MPI_Type_get_attr ??
- MPI_Type_get_contents 185
- MPI_Type_get_envelope 185
- MPI_Type_get_extent 176
- MPI_Type_get_extent_x 176
- MPI_Type_get_name ??
- MPI_Type_get_true_extent 178
- MPI_Type_get_true_extent_x 176
- MPI_Type_get_value_index ??
- MPI_Type_hindexed ??
- MPI_Type_hvector ??
- MPI_Type_indexed 164
- MPI_Type_lb ??
- MPI_Type_match_size 150
- MPI_Type_set_attr 302
- MPI_Type_set_name ??
- MPI_Type_size 151
- MPI_Type_size_x ??
- MPI_Type_struct ??
- MPI_Type_ub ??
- MPI_Type_vector 157
- MPI_T_category_changed 296
- MPI_T_category_get_categories 295
- MPI_T_category_get_cvars 295
- MPI_T_category_get_events ??
- MPI_T_category_get_index 295
- MPI_T_category_get_info 295
- MPI_T_category_get_num_events ??
- MPI_T_category_get_pvars 295
- MPI_T_cvar_get_index 292
- MPI_T_cvar_get_info 291
- MPI_T_cvar_get_num 291
- MPI_T_cvar_handle_alloc ??
- MPI_T_cvar_handle_free 292
- MPI_T_cvar_write 293
- MPI_T_enum_get_info ??
- MPI_T_enum_get_item ??
- MPI_T_event_callback_get_info ??
- MPI_T_event_callback_set_info ??

- MPI_T_event_copy ??
- MPI_T_event_get_index ??
- MPI_T_event_get_info ??
- MPI_T_event_get_source ??
- MPI_T_event_get_timestamp ??
- MPI_T_event_handle_alloc ??
- MPI_T_event_handle_free ??
- MPI_T_event_handle_get_info ??
- MPI_T_event_handle_set_info ??
- MPI_T_event_read ??
- MPI_T_event_register_callback ??
- MPI_T_event_set_dropped_handler ??
- MPI_T_finalize 291
- MPI_T_init_thread 291
- MPI_T_pvar_get_index 294
- MPI_T_pvar_get_info 293
- MPI_T_pvar_get_num 293
- MPI_T_pvar_handle_alloc 294
- MPI_T_pvar_handle_free 294
- MPI_T_pvar_read 295
- MPI_T_pvar_readreset ??
- MPI_T_pvar_reset ??
- MPI_T_pvar_start 294
- MPI_T_pvar_stop 294
- MPI_T_pvar_write ??
- MPI_T_source_get_info ??
- MPI_T_source_get_num ??
- MPI_T_source_get_timestamp ??
- MPI_Type_get_extent_c ??
- MPI_Type_get_true_extent_c ??
- MPI_Type_size_c ??
- MPI_Unpack 186
- MPI_Unpack_external ??
- MPI_Unpublish_name 217
- MPI_Wait 104
- MPI_Waitall 107
- MPI_Waitany 109
- MPI_Waitsome ??
- MPI_Win_allocate 226
- MPI_Win_allocate_cptr ??
- MPI_Win_allocate_shared 284
- MPI_Win_allocate_shared_cptr ??
- MPI_Win_attach 248
- MPI_Win_call_errhandler 305
- MPI_Win_complete 231
- MPI_Win_create 225
- MPI_Win_create_dynamic 248
- MPI_Win_create_errhandler ??
- MPI_Win_create_keyval 302
- MPI_Win_detach 249
- MPI_Win_fence 229
- MPI_Win_flush 246
- MPI_Win_flush_all ??
- MPI_Win_flush_local 246
- MPI_Win_flush_local_all ??
- MPI_Win_free 228
- MPI_Win_get_attr 247
- MPI_Win_get_group ??
- MPI_Win_get_info 299
- MPI_Win_get_name ??
- MPI_Win_lock 243
- MPI_Win_lock_all 245
- MPI_Win_post ??
- MPI_Win_set_attr 302
- MPI_Win_set_info 299
- MPI_Win_shared_query 286
- MPI_Win_shared_query_cptr ??
- MPI_Win_start 231
- MPI_Win_sync 247
- MPI_Win_test 231
- MPI_Win_unlock 244
- MPI_Win_unlock_all 244
- MPI_Win_wait 231
- MPI_Wtick 310
- MPI_Wtime 309
- PMPI_ ??
- PMPI_Aint_add ??
- PMPI_Aint_diff ??
- PMPI_Isend ??
- PMPI_Wtick ??
- PMPI_Wtime ??

58.1.2 List of all dtypes

58.1.3 List of all ctypes

- MPI_Offset 260
- _Bool ??
- bool ??
- char ??
- double ??
- enum ??
- float ??
- int ??
- long ??
- short ??
- wchar_t ??

58.1.4 List of all ftypes

- ALLOCATABLE ??
- ASYNCHRONOUS ??
- BLOCK ??
- CHARACTER ??
- COMMON ??
- COMPLEX ??
- CONTAINS ??
- CONTIGUOUS ??
- C_F_POINTER ??
- C_PTR ??
- EXTERNAL ??
- FUNCTION ??
- IN ??
- INCLUDE ??
- INOUT ??
- INTEGER ??
- INTENT ??
- INTERFACE ??
- ISO_C_BINDING ??
- ISO_FORTRAN_ENV ??
- KIND ??
- LOGICAL ??
- MODULE ??
- MPI_User_function ??
- OPTIONAL ??
- OUT ??
- POINTER ??
- PROCEDURE ??
- REAL ??
- SEQUENCE ??
- TARGET ??
- TYPE ??
- USER_FUNCTION ??
- VOLATILE ??

58.1.5 List of all constants

- MPI_ADDRESS_KIND 149
- MPI_ANY_SOURCE 91
- MPI_ANY_TAG 91
- MPI_APPNUM 214
- MPI_ARGVS_NULL 214
- MPI_ARGV_NULL 211
- MPI_ASYNC_PROTECTS_NONBLOCKING ??
- MPI_BAND 68
- MPI_BOR 68
- MPI_BOTTOM ??
- MPI_BSEND_OVERHEAD 140
- MPI_BUFFER_AUTOMATIC 141
- MPI_BXOR 68
- MPI_CART 266
- MPI_COMBINER_CONTIGUOUS ??
- MPI_COMBINER_DARRAY ??
- MPI_COMBINER_DUP ??
- MPI_COMBINER_HINDEXED ??
- MPI_COMBINER_HINDEXED_BLOCK ??
- MPI_COMBINER_HINDEXED_INTEGER ??
- MPI_COMBINER_HVECTOR ??
- MPI_COMBINER_HVECTOR_INTEGER ??
- MPI_COMBINER_INDEXED ??
- MPI_COMBINER_INDEXED_BLOCK ??
- MPI_COMBINER_NAMED ??
- MPI_COMBINER_RESIZED ??
- MPI_COMBINER_STRUCT ??
- MPI_COMBINER_STRUCT_INTEGER ??
- MPI_COMBINER_SUBARRAY ??
- MPI_COMBINER_VALUE_INDEX ??
- MPI_COMBINER_VECTOR 186
- MPI_COMM_NULL ??
- MPI_COMM_SELF ??
- MPI_COMM_TYPE_HW_GUIDED ??
- MPI_COMM_TYPE_HW_UNGUIDED ??
- MPI_COMM_TYPE_RESOURCE_GUIDED ??
- MPI_COMM_TYPE_SHARED ??
- MPI_COMM_WORLD ??
- MPI_CONGRUENT 194
- MPI_COUNT_KIND ??
- MPI_DATATYPE_NULL 154
- MPI_DISPLACEMENT_CURRENT 261
- MPI_DISTRIBUTE_BLOCK 164
- MPI_DISTRIBUTE_CYCLIC 164
- MPI_DISTRIBUTE_DFLT_DARG 164
- MPI_DISTRIBUTE_NONE 164
- MPI_DIST_GRAPH 266
- MPI_ERRCODES_IGNORE 210
- MPI_ERRHANDLER_NULL ??
- MPI_ERROR 120
- MPI_ERRORS_ABORT 305
- MPI_ERRORS_ARE_FATAL 305
- MPI_ERRORS_RETURN 305
- MPI_ERR_LASTCODE 304
- MPI_FILE_NULL ??
- MPI_FLOAT_INT 68
- MPI_F_ERROR ??
- MPI_F_SOURCE ??
- MPI_F_STATUSES_IGNORE ??
- MPI_F_STATUS_SIZE ??
- MPI_F_TAG ??
- MPI_GRAPH 266
- MPI_GROUP_EMPTY 204
- MPI_GROUP_NULL 204
- MPI_HOST ??
- MPI_IDENT ??
- MPI_INFO_ENV 299
- MPI_INFO_NULL ??
- MPI_INTEGER_KIND ??
- MPI_IN_PLACE 41
- MPI_IO ??
- MPI_KEYVAL_INVALID ??
- MPI_LAND 68
- MPI_LASTUSEDCODE 307
- MPI_LOCK_EXCLUSIVE ??
- MPI_LOCK_SHARED ??
- MPI_LOR 68
- MPI_MAX 68
- MPI_MAXLOC 68
- MPI_MAX_DATAREP_STRING ??
- MPI_MAX_ERROR_STRING 306
- MPI_MAX_INFO_KEY 297
- MPI_MAX_INFO_VAL ??
- MPI_MAX_LIBRARY_VERSION_STRING 308
- MPI_MAX_OBJECT_NAME ??
- MPI_MAX_PORT_NAME 214
- MPI_MAX_PROCESSOR_NAME ??
- MPI_MAX_PSET_NAME_LEN ??
- MPI_MAX_STRINGTAG_LEN ??
- MPI_MESSAGE_NO_PROC ??
- MPI_MESSAGE_NULL ??
- MPI_MIN 68
- MPI_MINLOC 68
- MPI_MODE_APPEND 256
- MPI_MODE_CREATE 256
- MPI_MODE_DELETE_ON_CLOSE 256
- MPI_MODE_EXCL 256
- MPI_MODE_NOCHECK 245
- MPI_MODE_NOPRECEDE 230
- MPI_MODE_NOPUT 230
- MPI_MODE_NOSTORE 230
- MPI_MODE_NOSUCCEED 230
- MPI_MODE_RDONLY 256
- MPI_MODE_RDWR 256
- MPI_MODE_SEQUENTIAL 256
- MPI_MODE_UNIQUE_OPEN 256
- MPI_MODE_WRONLY 256
- MPI_NO_OP 68
- MPI_OFFSET_KIND 144
- MPI_OP_NULL ??
- MPI_ORDER_C 163
- MPI_ORDER_FORTRAN 163
- MPI_PROC_NULL 100
- MPI_PROD 68
- MPI_REPLACE 236
- MPI_REQUEST_NULL ??
- MPI_ROOT ??
- MPI_SEEK_CUR 257
- MPI_SEEK_END 257
- MPI_SEEK_SET 257
- MPI_SESSION_NULL ??
- MPI_SHORT_INT 68
- MPI_SIMILAR 194
- MPI_SOURCE 118
- MPI_STATUSES_IGNORE ??
- MPI_STATUS_IGNORE 92
- MPI_STATUS_SIZE ??
- MPI_SUBARRAYS_SUPPORTED 163
- MPI_SUBVERSION 303
- MPI_SUCCESS 305
- MPI_SUM 68
- MPI_TAG 119
- MPI_TAG_UB 301
- MPI_THREAD_FUNNELED 288
- MPI_THREAD_MULTIPLE 288
- MPI_THREAD_SERIALIZED 288
- MPI_THREAD_SINGLE 288
- MPI_TYPECLASS_COMPLEX ??
- MPI_TYPECLASS_INTEGER ??
- MPI_TYPECLASS_REAL ??
- MPI_T_BIND_MPI_COMM ??
- MPI_T_BIND_MPI_DATATYPE ??
- MPI_T_BIND_MPI_ERRHANDLER ??
- MPI_T_BIND_MPI_FILE ??
- MPI_T_BIND_MPI_GROUP ??
- MPI_T_BIND_MPI_INFO ??
- MPI_T_BIND_MPI_MESSAGE ??
- MPI_T_BIND_MPI_OP ??
- MPI_T_BIND_MPI_REQUEST ??
- MPI_T_BIND_MPI_SESSION ??
- MPI_T_BIND_MPI_WIN ??

- MPI_T_BIND_NO_OBJECT 291
- MPI_T_PVAR_SESSION_NULL 294
- MPI_UNDEFINED ??
- MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE 291
- MPI_T_SCOPE_ALL ??
- MPI_UNEQUAL 194
- MPI_T_CB_REQUIRE_MPI_RESTRICTED ??
- MPI_T_SCOPE_ALL_EQ ??
- MPI_UNWEIGHTED ??
- MPI_T_CB_REQUIRE_NONE ??
- MPI_T_SCOPE_CONSTANT ??
- MPI_UNVERSION 303
- MPI_T_CB_REQUIRE_THREAD_SAFE ??
- MPI_T_SCOPE_GROUP ??
- MPI_WEIGHTS_EMPTY 275
- MPI_T_CVAR_HANDLE_NULL ??
- MPI_T_SCOPE_GROUP_EQ ??
- MPI_WIN_BASE ??
- MPI_T_ENUM_NULL ??
- MPI_T_SCOPE_LOCAL ??
- MPI_WIN_CREATE_FLAVOR ??
- MPI_T_PVAR_ALL_HANDLES 295
- MPI_T_SCOPE_READONLY ??
- MPI_WIN_DISP_UNIT ??
- MPI_T_PVAR_CLASS_AGGREGATE 293
- MPI_T_SOURCE_ORDERED ??
- MPI_WIN_FLAVOR_ALLOCATE ??
- MPI_T_PVAR_CLASS_COUNTER 293
- MPI_T_SOURCE_UNORDERED ??
- MPI_WIN_FLAVOR_CREATE ??
- MPI_T_PVAR_CLASS_GENERIC 293
- MPI_T_VERBOSITY_MPIDEV_ALL ??
- MPI_WIN_FLAVOR_DYNAMIC ??
- MPI_T_PVAR_CLASS_HIGHWATERMARK 293
- MPI_T_VERBOSITY_MPIDEV_BASIC ??
- MPI_WIN_FLAVOR_SHARED ??
- MPI_T_PVAR_CLASS_LEVEL 293
- MPI_T_VERBOSITY_MPIDEV_DETAIL ??
- MPI_WIN_MODEL 247
- MPI_T_PVAR_CLASS_LOWWATERMARK 293
- MPI_T_VERBOSITY_TUNER_ALL ??
- MPI_WIN_NULL ??
- MPI_T_PVAR_CLASS_PERCENTAGE 293
- MPI_T_VERBOSITY_TUNER_BASIC ??
- MPI_WIN_SEPARATE ??
- MPI_T_PVAR_CLASS_SIZE 293
- MPI_T_VERBOSITY_TUNER_DETAIL ??
- MPI_WIN_SIZE ??
- MPI_T_PVAR_CLASS_STATE 293
- MPI_T_VERBOSITY_USER_ALL ??
- MPI_WIN_UNIFIED ??
- MPI_T_PVAR_CLASS_TIMER 293
- MPI_T_VERBOSITY_USER_BASIC ??
- MPI_WTIME_IS_GLOBAL ??
- MPI_T_PVAR_HANDLE_NULL 294
- MPI_T_VERBOSITY_USER_DETAIL ??

58.1.6 List of all callbacks

- COMM_COPY_ATTR_FUNCTION ??
- MPI_Comm_errhandler_function ??
- MPI_Handler_function ??
- COMM_DELETE_ATTR_FUNCTION ??
- MPI_Copy_function ??
- MPI_Session_errhandler_function ??
- COPY_FUNCTION ??
- MPI_Datarep_conversion_function ??
- MPI_Type_delete_attr_function ??
- DELETE_FUNCTION ??
- MPI_Datarep_conversion_function_c
- MPI_User_function ??
- MPI_Comm_copy_attr_function ??
- MPI_Delete_function ??
- MPI_User_function_c ??
- MPI_Comm_delete_attr_function ??
- MPI_File_errhandler_function ??
- MPI_Win_errhandler_function ??

58.2 MPI for Python

58.2.1 Buffer specifications

58.2.2 Listing of python routines

Class Comm:	Class Cartcomm:	Class Request:	Class Grequest:
Class Distgraphcomm:	Class Graphcomm:	Class Prequest:	Class Status:
Class Intercomm:	Class Intracomm:		
Class Topocomm:	Class Group:	Class Win:	Class Datatype:
			Class File:
			Class Info:
			Class Op:
			Class Errhandler:
			Class Message:

Chapter 59

Index of OpenMP keywords

Chapter 60

Index of PETSc keywords

Chapter 61

Index of KOKKOS keywords

Chapter 62

Index of SYCL keywords

