

# **Parallel Programming for Science and Engineering**

Using MPI, OpenMP, and the PETSc library

Victor Eijkhout

2nd edition 2020, formatted October 13, 2020

---

**Public draft - open for comments**

---

This book is open source under the CC-BY 4.0 license.

Download location for the latest pdf is: <https://web.corral.tacc.utexas.edu/CompEdu/pdf/pcse/>

---

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. (A Java interface to MPI is rumored to exist but we lend this no credit.) For MPI and the PETSc library we will also discuss the Python interfaces.

# Contents

<b>I MPI</b>	<b>13</b>
1	<b>Getting started with MPI</b> 14
1.1	<i>Distributed memory and message passing</i> 14
1.2	<i>History</i> 14
1.3	<i>Basic model</i> 15
1.4	<i>Making and running an MPI program</i> 16
1.5	<i>Language bindings</i> 17
1.6	<i>Review</i> 21
1.7	<i>Sources used in this chapter</i> 22
2	<b>MPI topic: Functional parallelism</b> 23
2.1	<i>The SPMD model</i> 23
2.2	<i>Starting and running MPI processes</i> 25
2.3	<i>Processor identification</i> 28
2.4	<i>Functional parallelism</i> 31
2.5	<i>Review questions</i> 33
2.6	<i>Sources used in this chapter</i> 34
3	<b>MPI topic: Collectives</b> 35
3.1	<i>Working with global information</i> 35
3.2	<i>Reduction</i> 38
3.3	<i>Rooted collectives: broadcast, reduce</i> 42
3.4	<i>Rooted collectives: gather and scatter</i> 49
3.5	<i>All-to-all</i> 55
3.6	<i>Reduce-scatter</i> 56
3.7	<i>Barrier</i> 60
3.8	<i>Variable-size-input collectives</i> 61
3.9	<i>Scan operations</i> 64
3.10	<i>MPI Operators</i> 67
3.11	<i>Non-blocking collectives</i> 72
3.12	<i>Performance of collectives</i> 77
3.13	<i>Collectives and synchronization</i> 78
3.14	<i>Performance considerations</i> 81
3.15	<i>Review questions</i> 83
3.16	<i>Sources used in this chapter</i> 86

4	<b>MPI topic: Point-to-point</b>	111
4.1	<i>Distributed computing and distributed data</i>	111
4.2	<i>Blocking point-to-point operations</i>	112
4.3	<i>Non-blocking point-to-point operations</i>	129
4.4	<i>More about point-to-point communication</i>	142
4.5	<i>Review questions</i>	156
4.6	<i>Sources used in this chapter</i>	160
5	<b>MPI topic: Data types</b>	178
5.1	<i>Data type handling</i>	178
5.2	<i>Elementary data types</i>	179
5.3	<i>Derived datatypes</i>	184
5.4	<i>Type size</i>	203
5.5	<i>More about data</i>	209
5.6	<i>Review questions</i>	213
5.7	<i>Sources used in this chapter</i>	214
6	<b>MPI topic: Communicators</b>	245
6.1	<i>Communicator basics</i>	245
6.2	<i>Making new communicators</i>	246
6.3	<i>Duplicating communicators</i>	248
6.4	<i>Splitting a communicator</i>	250
6.5	<i>Communicators and groups</i>	253
6.6	<i>Inter-communicators</i>	254
6.7	<i>Review questions</i>	259
6.8	<i>Sources used in this chapter</i>	260
7	<b>MPI topic: Process management</b>	268
7.1	<i>Process spawning</i>	268
7.2	<i>Socket-style communications</i>	271
7.3	<i>Sources used in this chapter</i>	274
8	<b>MPI topic: One-sided communication</b>	278
8.1	<i>Windows</i>	279
8.2	<i>Active target synchronization: epochs</i>	283
8.3	<i>Put, get, accumulate</i>	284
8.4	<i>Passive target synchronization</i>	298
8.5	<i>More about window memory</i>	303
8.6	<i>Assertions</i>	306
8.7	<i>Implementation</i>	307
8.8	<i>Review questions</i>	309
8.9	<i>Sources used in this chapter</i>	310
9	<b>MPI topic: File I/O</b>	324
9.1	<i>File handling</i>	325
9.2	<i>File reading and writing</i>	326
9.3	<i>Consistency</i>	334
9.4	<i>Constants</i>	334

9.5	<i>Review questions</i>	335
9.6	<i>Sources used in this chapter</i>	336
10	<b>MPI topic: Topologies</b>	337
10.1	<i>Cartesian grid topology</i>	337
10.2	<i>Distributed graph topology</i>	339
10.3	<i>Sources used in this chapter</i>	344
11	<b>MPI topic: Shared memory</b>	346
11.1	<i>Recognizing shared memory</i>	346
11.2	<i>Shared memory for windows</i>	347
11.3	<i>Sources used in this chapter</i>	352
12	<b>MPI leftover topics</b>	358
12.1	<i>Contextual information, attributes, etc.</i>	358
12.2	<i>Error handling</i>	362
12.3	<i>Fortran issues</i>	364
12.4	<i>Fault tolerance</i>	365
12.5	<i>Asynchronous progress</i>	366
12.6	<i>Performance, tools, and profiling</i>	366
12.7	<i>Determinism</i>	371
12.8	<i>Subtleties with processor synchronization</i>	372
12.9	<i>Shell interaction</i>	372
12.10	<i>The origin of one-sided communication in ShMem</i>	374
12.11	<i>Leftover topics</i>	374
12.12	<i>Literature</i>	377
12.13	<i>Sources used in this chapter</i>	378
13	<b>MPI Reference</b>	382
13.1	<i>Leftover topics</i>	382
13.2	<i>Sources used in this chapter</i>	384
 II OpenMP 385		
14	<b>Getting started with OpenMP</b>	386
14.1	<i>The OpenMP model</i>	386
14.2	<i>Compiling and running an OpenMP program</i>	389
14.3	<i>Your first OpenMP program</i>	390
14.4	<i>Thread data</i>	393
14.5	<i>Creating parallelism</i>	394
14.6	<i>Sources used in this chapter</i>	396
15	<b>OpenMP topic: Parallel regions</b>	397
15.1	<i>Nested parallelism</i>	399
15.2	<i>Cancel parallel construct</i>	401
15.3	<i>Sources used in this chapter</i>	402
16	<b>OpenMP topic: Loop parallelism</b>	404
16.1	<i>Loop parallelism</i>	404

16.2	<i>Loop schedules</i>	406
16.3	<i>Reductions</i>	409
16.4	<i>Collapsing nested loops</i>	409
16.5	<i>Ordered iterations</i>	410
16.6	<i>nowait</i>	411
16.7	<i>While loops</i>	411
16.8	<i>Sources used in this chapter</i>	413
17	<b>OpenMP topic: Work sharing</b>	414
17.1	<i>Sections</i>	414
17.2	<i>Single/master</i>	415
17.3	<i>Fortran array syntax parallelization</i>	416
17.4	<i>Sources used in this chapter</i>	417
18	<b>OpenMP topic: Controlling thread data</b>	418
18.1	<i>Shared data</i>	418
18.2	<i>Private data</i>	418
18.3	<i>Data in dynamic scope</i>	419
18.4	<i>Temporary variables in a loop</i>	420
18.5	<i>Default</i>	420
18.6	<i>Array data</i>	421
18.7	<i>First and last private</i>	422
18.8	<i>Persistent data through <code>threadprivate</code></i>	422
18.9	<i>Sources used in this chapter</i>	425
19	<b>OpenMP topic: Reductions</b>	427
19.1	<i>Built-in reduction operators</i>	429
19.2	<i>Initial value for reductions</i>	429
19.3	<i>User-defined reductions</i>	430
19.4	<i>Reductions and floating-point math</i>	431
19.5	<i>Sources used in this chapter</i>	432
20	<b>OpenMP topic: Synchronization</b>	434
20.1	<i>Barrier</i>	434
20.2	<i>Mutual exclusion</i>	435
20.3	<i>Locks</i>	436
20.4	<i>Example: Fibonacci computation</i>	438
20.5	<i>Sources used in this chapter</i>	441
21	<b>OpenMP topic: Tasks</b>	442
21.1	<i>Task data</i>	443
21.2	<i>Task synchronization</i>	444
21.3	<i>Task dependencies</i>	446
21.4	<i>More</i>	447
21.5	<i>Examples</i>	448
21.6	<i>Sources used in this chapter</i>	450
22	<b>OpenMP topic: Affinity</b>	452
22.1	<i>OpenMP thread affinity control</i>	452

22.2	<i>First-touch</i>	456
22.3	<i>Affinity control outside OpenMP</i>	457
22.4	<i>Sources used in this chapter</i>	458
23	<b>OpenMP topic: Memory model</b>	459
23.1	<i>Thread synchronization</i>	459
23.2	<i>Data races</i>	460
23.3	<i>Relaxed memory model</i>	461
23.4	<i>Sources used in this chapter</i>	462
24	<b>OpenMP topic: SIMD processing</b>	463
24.1	<i>Sources used in this chapter</i>	467
25	<b>OpenMP remaining topics</b>	468
25.1	<i>Runtime functions and internal control variables</i>	468
25.2	<i>Timing</i>	470
25.3	<i>Thread safety</i>	470
25.4	<i>Performance and tuning</i>	471
25.5	<i>Accelerators</i>	472
25.6	<i>Sources used in this chapter</i>	473
26	<b>OpenMP Review</b>	474
26.1	<i>Concepts review</i>	474
26.2	<i>Review questions</i>	475
26.3	<i>Sources used in this chapter</i>	484

### III PETSc 485

27	<b>PETSc basics</b>	486
27.1	<i>What is PETSc and why?</i>	486
27.2	<i>Basics of running a PETSc program</i>	488
27.3	<i>PETSc installation</i>	491
27.4	<i>Sources used in this chapter</i>	493
28	<b>PETSc objects</b>	494
28.1	<i>Distributed objects</i>	494
28.2	<i>Scalars</i>	495
28.3	<i>Vec: Vectors</i>	496
28.4	<i>Mat: Matrices</i>	506
28.5	<i>DMDA: distributed arrays</i>	516
28.6	<i>Index sets and Vector Scatters</i>	518
28.7	<i>AO: Application Orderings</i>	520
28.8	<i>Partitionings</i>	520
28.9	<i>Sources used in this chapter</i>	522
29	<b>Finite Elements support</b>	529
29.1	<i>Sources used in this chapter</i>	531
30	<b>PETSc solvers</b>	532
30.1	<i>KSP: linear system solvers</i>	532

30.2	<i>Direct solvers</i>	542
30.3	<i>Control through command line options</i>	542
30.4	<i>Sources used in this chapter</i>	543
31	<b>PETSc tools</b>	544
31.1	<i>Error checking and debugging</i>	544
31.2	<i>Program output</i>	546
31.3	<i>Commandline options</i>	549
31.4	<i>Timing and profiling</i>	551
31.5	<i>Memory management</i>	552
31.6	<i>Sources used in this chapter</i>	553
32	<b>PETSc topics</b>	557
32.1	<i>Communicators</i>	557
32.2	<i>Scalars</i>	557
32.3	<i>Sources used in this chapter</i>	558

IV	<b>Other programming models</b>	559
33	<b>Co-array Fortran</b>	560
33.1	<i>History and design</i>	560
33.2	<i>Compiling and running</i>	560
33.3	<i>Basics</i>	560
33.4	<i>Sources used in this chapter</i>	564
34	<b>Sycl, OneAPI, DPC++</b>	565
34.1	<i>Logistics</i>	565
34.2	<i>Platforms and devices</i>	565
34.3	<i>Queues</i>	566
34.4	<i>Kernels</i>	566
34.5	<i>Unified memory</i>	568
34.6	<i>Parallel output</i>	569
34.7	<i>Sources used in this chapter</i>	570

V	<b>The Rest</b>	571
35	<b>Exploring computer architecture</b>	572
35.1	<i>Tools for discovery</i>	572
35.2	<i>Sources used in this chapter</i>	573
36	<b>Process and thread affinity</b>	574
36.1	<i>What does the hardware look like?</i>	575
36.2	<i>Affinity control</i>	577
36.3	<i>Sources used in this chapter</i>	578
37	<b>Hybrid computing</b>	579
37.1	<i>Discussion</i>	580
37.2	<i>Hybrid MPI-plus-threads execution</i>	581
37.3	<i>Sources used in this chapter</i>	584

38	<b>Random number generation</b>	585
38.1	<i>Sources used in this chapter</i>	586
39	<b>Parallel I/O</b>	587
39.1	<i>Sources used in this chapter</i>	588
40	<b>Support libraries</b>	589
40.1	<i>SimGrid</i>	589
40.2	<i>Other</i>	589
40.3	<i>Sources used in this chapter</i>	590

VI	Tutorials	591
40.4	<i>Debugging</i>	593
40.5	<i>Tracing and profiling with TAU</i>	601
40.6	<i>SimGrid</i>	606
40.7	<i>Batch systems</i>	610

VII	Projects, index	619
41	<b>Class projects</b>	620
41.1	<i>A Style Guide to Project Submissions</i>	620
41.2	<i>Warmup Exercises</i>	623
41.3	<i>Mandelbrot set</i>	627
41.4	<i>Data parallel grids</i>	634
41.5	<i>N-body problems</i>	637
42	<b>Bibliography, index, and list of acronyms</b>	638
42.1	<i>Bibliography</i>	638
42.2	<i>List of acronyms</i>	640
42.3	<i>General Index</i>	641
42.4	<i>Index of MPI commands</i>	650
42.5	<i>MPL commands and topics</i>	656
42.6	<i>MPI4Python notes</i>	657
42.7	<i>Index of OpenMP commands</i>	658
42.8	<i>Index of PETSc commands</i>	660

## Contents

---

## **PART I**

### **MPI**

# **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 interface to C/C++ or Fortran; there are even bindings to Python. A related point is that it is 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.

### **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 mecha-

nisms. 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* [11], 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`<sup>1</sup> and supplies

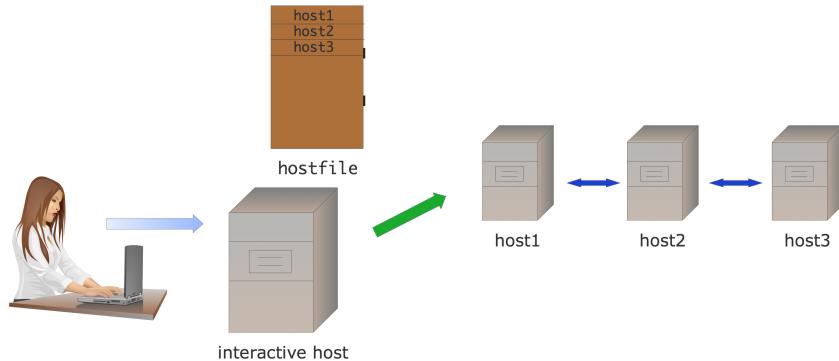


Figure 1.1: Interactive MPI setup

- 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 `mpirun` 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 `mpirun` 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 `mpirun` 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.

You see that in both scenarios the parallel program is started by the `mpirun` command using an 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.

---

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

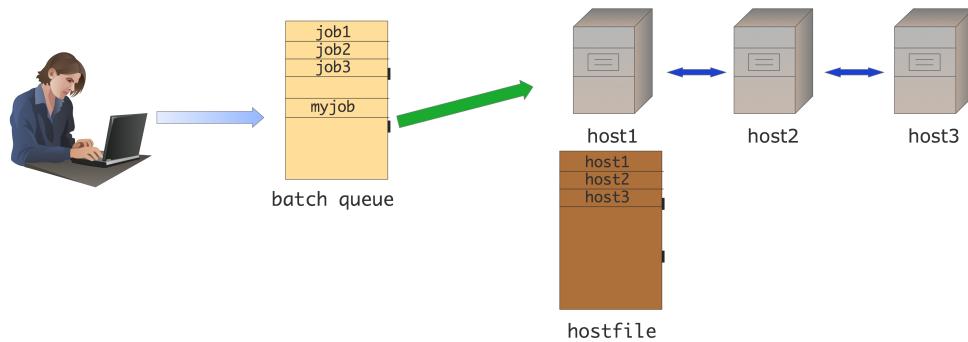


Figure 1.2: Batch MPI setup

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:

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

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 *mpirun* 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, *mpirun* 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.* 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. Thus, its bindings are the most natural for that language.

### 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*

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 Layer (MPL)* <https://github.com/rabauke/mp1>. This book has an index of MPL notes and commands: section 42.5.

MPL is a C++ header-only library. If your program includes the line

```
|| #include <mp1/mp1.hpp>
```

it is enough to add the include path when compiling with `mpicxx`. There is no further library to link.

*MPL note.* MPL-specific notes will be indicated with a note like this.

### 1.5.3 Fortran

*Fortran note.* 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 6.1, Datatype 5.3.1.1, Info 12.1.1, Op 3.10.2, Request 4.3.1, Status 4.4.2, Window 8.1.

- 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

For details see <http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node409.htm>.

### 1.5.4 Python

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

The `mpi4py` package [3] 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 native 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 native 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 prototypes

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 prototypes’ look like code but they are not! Here is how you translate them.

### 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 12.2.1):

```
|| 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 12.2.

- 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 pre-2008 legacy mode:

## 1. Getting started with MPI

---

```
|| 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 (`mpiexec`, `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

**1.7 Sources used in this chapter**

**1.7.1 Listing of code header**

## 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

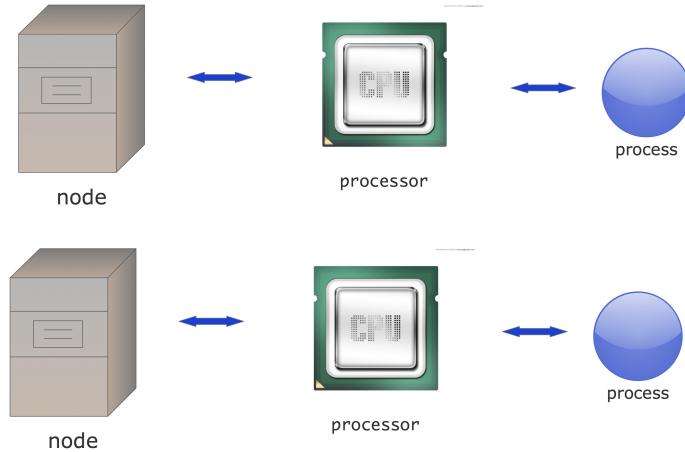


Figure 2.1: Cluster structure as of the mid 1990s

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

---

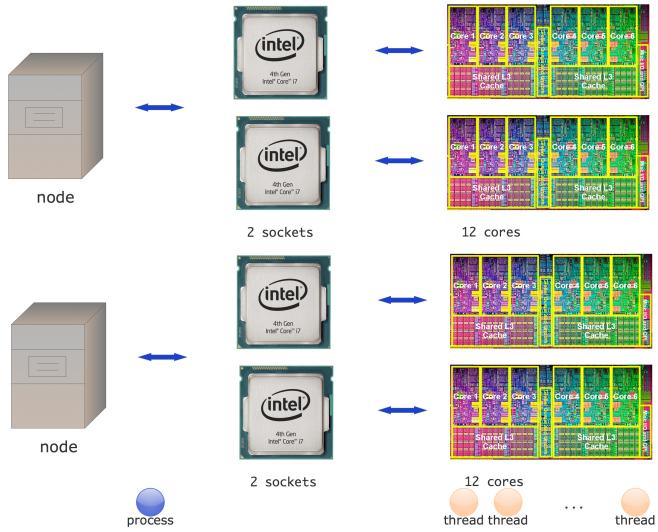


Figure 2.2: Hybrid cluster structure

of figure 2.3, which we will use in most of this part of the book. (Hybrid computing will be discussed in chapter 37.)

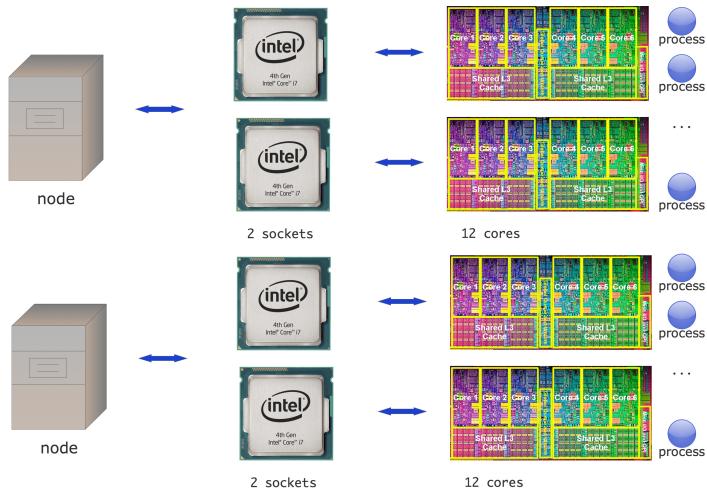


Figure 2.3: MPI-only cluster structure

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 1** In some rare cases you may want to run in an Multiple Program Multiple Data (MPMD) mode, rather than SPMD. This can be achieved either on the OS level (see section 12.9.4), using options of the `mpiexec` mechanism, or you can use MPI's built-in process management; chapter 7. 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'.

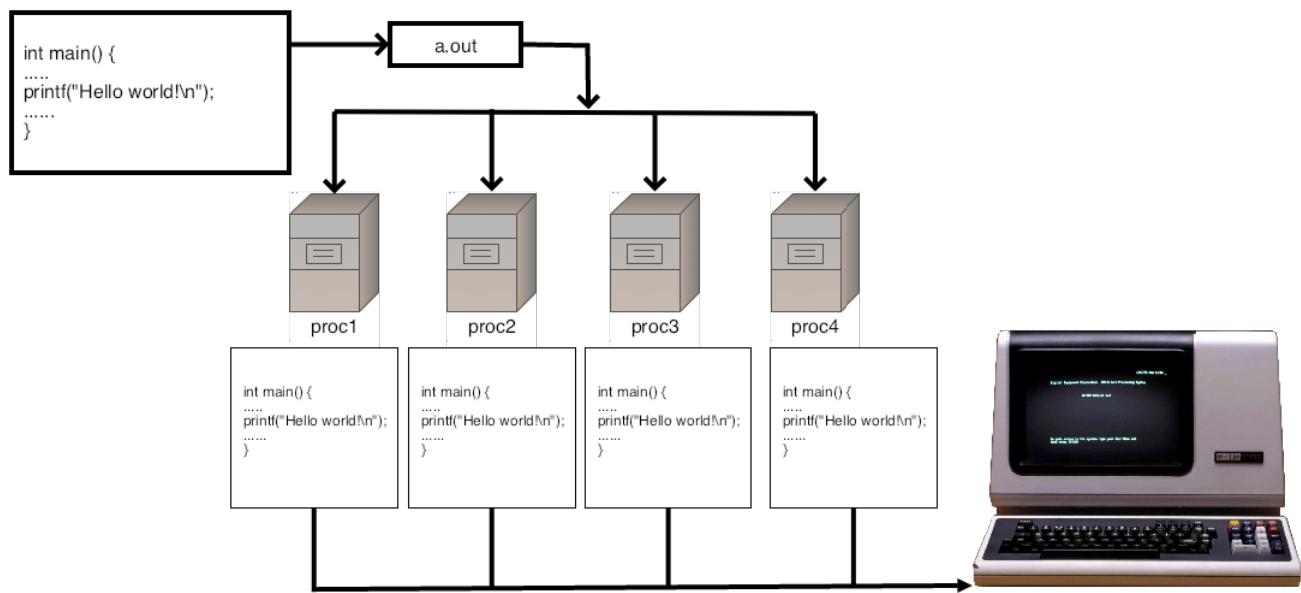


Figure 2.4: Running a hello world program in parallel

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.

This exercise is illustrated in figure 2.4.

### 2.2.1 Headers

If you use MPI commands in a program file, be sure to include the proper header file, `mpi.h` or `mpif.h`.

### 2.1 MPI\_Init

```
C:
int MPI_Init(int *argc, char ***argv)

Fortran:
MPI_Init(ierr)
INTEGER, OPTIONAL, INTENT(OUT) :: ierr
```

### 2.2 MPI\_Finalize

```
C:
int MPI_Finalize(void)

Fortran:
MPI_Finalize(ierr)
INTEGER, OPTIONAL, INTENT(OUT) :: ierr
```

```
#include "mpi.h" // for C
#include "mpif.h" ! for Fortran
```

For *Fortran90*, many MPI installations also have an MPI module, so you can write

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

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.

*Python note.* It's easiest to

```
|| from mpi4py import MPI
```

*MPL note.* To compile MPL programs, add a line

```
|| #include <mpl/mp1.hpp>
```

to your file.

#### 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:

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

### 2.3 MPI\_Abort

Synopsis:

```
int MPI_Abort (MPI_Comm comm, int errorcode)
```

Input Parameters

comm : communicator of tasks to abort

errorcode : error code to return to invoking environment

Python:

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

MPL:

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

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

Fortran (before 2008) lacks this commandline argument handling, so **MPI\_Init** lacks those arguments.

*Python note.* There are no initialize and finalize calls: the `import` statement performs the initialization.

*MPL note.* There is no initialization or finalize call.

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 2** For hybrid MPI-plus-threads programming there is also a call **MPI\_Init\_thread**. For that, see section 37.2.

#### 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 aborts execution on all processes associated with the communicator, but many implementations simply abort all processes. The value parameter is returned to the environment.

#### 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).

## 2.4 MPI\_Initialized

```
C:  
int MPI_Initialized(int *flag)  
  
Fortran:  
MPI_Initialized(flag, ierror)  
LOGICAL, INTENT(OUT) :: flag  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

## 2.5 MPI\_Finalized

```
C:  
int MPI_Finalized( int *flag )  
  
Fortran:  
MPI_Finalized(flag, ierror)  
LOGICAL, INTENT(OUT) :: flag  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### 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 12.1.1.1.

### 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 12.1.1.

## 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.

**Exercise 2.3.** Now use the command `MPI_Get_processor_name` in between the init and finalize statement, and print out on what processor your process runs. Confirm that you are able to run a program that uses two different nodes.

## 2.6 MPI\_Get\_processor\_name

```
C:
int MPI_Get_processor_name(char *name, int *resultlen)
    name : buffer char[MPI_MAX_PROCESSOR_NAME]

Fortran:
MPI_Get_processor_name(name, resultlen, ierror)
CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
INTEGER, INTENT(OUT) :: resultlen
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Python:
MPI.Get_processor_name()
```

The character buffer needs to be allocated by you, it is not created by MPI, with size at least `MPI_MAX_PROCESSOR_NAME`.

The character storage is provided by the user: 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.

*MPL note.* The `processor_name` call is an environment method returning a `std::string`:

```
|| std::string mpl::environment::processor_name();
```

## 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 procedural languages (C/Fortran), a *communicator* is a *variable* that is passed to most routines:

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

- 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 )
```

- 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 )
```

## 2. MPI topic: Functional parallelism

---

### 2.7 MPI\_Comm\_size

Semantics:  
`MPI_COMM_SIZE(comm, size)`  
IN `comm`: communicator (handle)  
OUT `size`: number of processes in the group of `comm` (integer)

C:  
`int MPI_Comm_size(MPI_Comm comm, int *size)`

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

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

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

*MPL note.* In MPL, which is based on C++, the naive construct would be:

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

*For the source of this example, see section 2.6.2*

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();
```

*For the source of this example, see section 2.6.2*

You will learn much more about communicators in chapter 6.

### 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.

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’.

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

## 2.8 MPI\_Comm\_rank

Semantics:

```
MPI_COMM_RANK(comm, rank)
IN comm: communicator (handle)
OUT rank: rank of the calling process in group of comm (integer)
```

C:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran:

```
MPI_Comm_rank(comm, rank, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: rank
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

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

MPL:

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

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.*

**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 , the **MPI\_Comm\_rank** and **MPI\_Comm\_size** routines are methods of the communicator class:

*Python note.*

```
procid = comm.Get_rank()
nprocs = comm.Get_size()
```

*MPL note.* The rank of a process (by *rank*) and the size of a communicator (by *size*) 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();
```

Note the predefined environment method *comm\_world*.

## 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

## 2. MPI topic: Functional parallelism

---

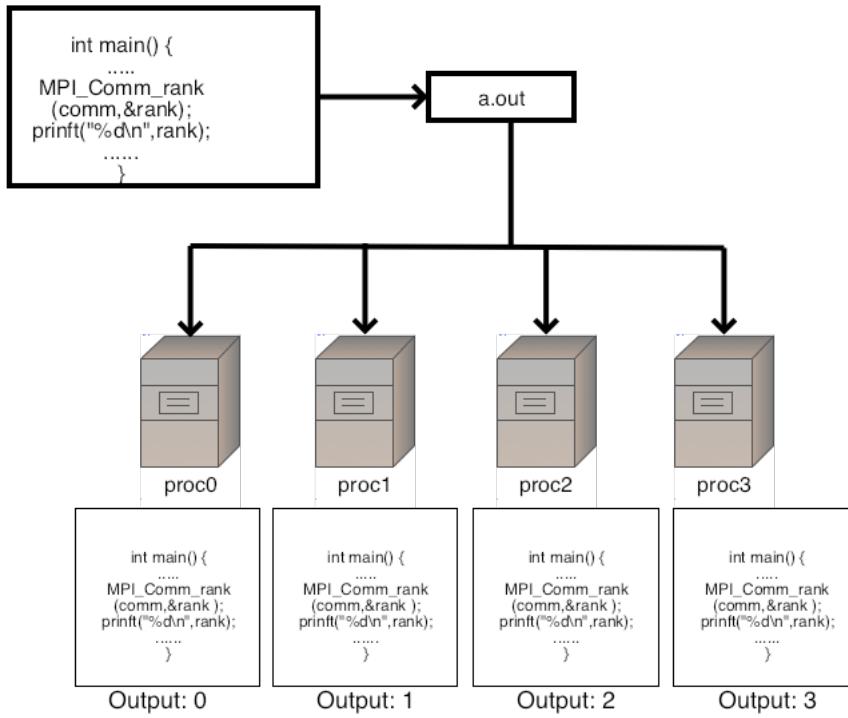


Figure 2.5: Parallel program that prints process rank

```

// 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 range 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.)

**Remark 3** 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 Review questions

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

**Exercise 2.7.** True or false: `mpicc` is a compiler.

**Exercise 2.8.** 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 2.9.** What is the function of a hostfile?

## 2.6 Sources used in this chapter

### 2.6.1 Listing of code header

### 2.6.2 Listing of code examples/mpi/mpl/commrank.cxx

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

int main() {
#ifndef _MSC_VER
    mpl::communicator comm_world =
        mpl::environment::comm_world();
#else
    const mpl::communicator &comm_world =
        mpl::environment::comm_world();
#endif
    std::cout << "Hello world! I am running on \""
        << mpl::environment::processor_name()
        << "\". My rank is "
        << comm_world.rank()
        << " out of "
        << comm_world.size() << " processes.\n" << std::endl;
    return EXIT_SUCCESS;
}
```

## 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 is a little different from what happens with MPI processes, because these are more symmetric; the process doing the reducing and broadcasting is no different from the others. Any process can function as the *root process* in such a collective.

**Exercise 3.1.** 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.

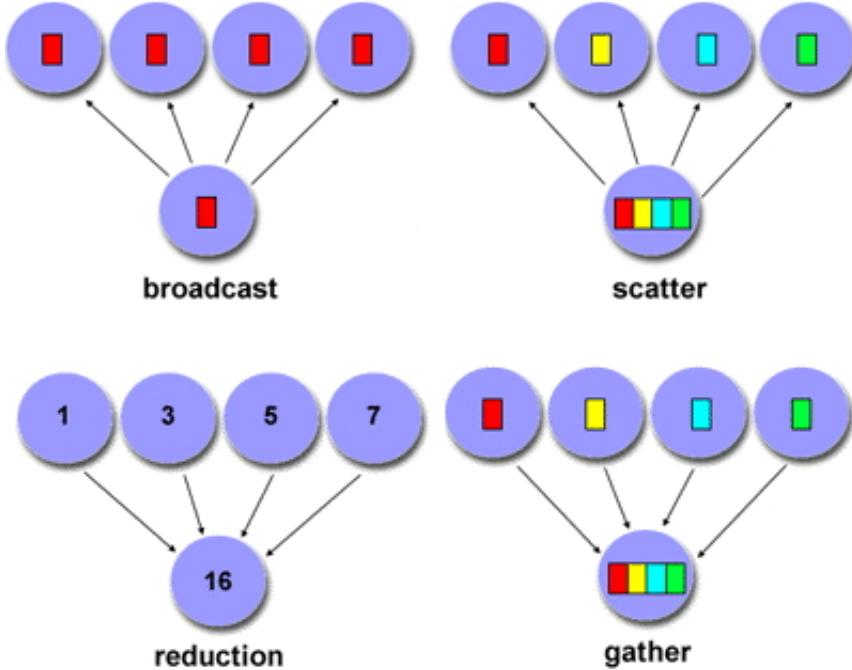


Figure 3.1: The four most common collectives

- Let each process compute a random number. You want to print on what processor the maximum value is computed.

### 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.

- The calculation of the average  $\mu$  is a reduction, since all the distributed values need to be added.
- Now every process needs to compute  $x_i - \mu$  for its value  $x_i$ , so  $\mu$  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  $\sigma$ , we can again use an allreduce.

As another examples, if  $x, y$  are distributed vector objects, and you want to compute

$$y - (x^t y)x$$

which is part of the Gramm-Schmidt algorithm; see HPSC-???. Again you need to use an allreduce to reduce the inner product value on all processors.

```
// compute local value
localvalue = innerproduct( x[ localpart ], y[ localpart ] );
// compute inner product on the every process
AllReduce( localvalue, reducedvalue );
```

### 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 3.2.1. 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.

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

There are more collectives or variants on the above.

- 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 3.8).
- 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 3.9).
- If every processor needs to broadcast to every other, you use an *all-to-all* operation (section 3.5).

- A barrier is an operation that makes all processes wait until every process has reached the barrier (section 3.7).

Finally, there are some advanced topics in collectives.

- User-defined reduction operators; section 3.10.2.
- Non-blocking collectives; section 3.11.

## 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 %6.9f compared to .5\n", sumrandom/nprocs);
```

*For the source of this example, see section 3.16.2*

For Python we illustrate both the native and the numpy variant. In the numpy variant we create an array for the receive buffer, even though only one element is used.

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

*For the source of this example, see section 3.16.3*

**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.

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

### 3.1 MPI\_Allreduce

C:

```
int MPI_Allreduce(const void* sendbuf,
                  void* recvbuf, int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm)
```

Semantics:

```
IN sendbuf: starting address of send buffer (choice)
OUT recvbuf: starting address of receive buffer (choice)
IN count: number of elements in send buffer (non-negative integer)
IN datatype: data type of elements of send buffer (handle)
IN op: operation (handle)
IN comm: communicator (handle)
```

Fortran:

```
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python native:

```
recvobj = MPI.Comm.allreduce(self, sendobj, op=SUM)
Python numpy:
MPI.Comm.Allreduce(self, sendbuf, recvbuf, Op op=SUM)
```

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
```

### 3. MPI topic: Collectives

---

Compute the sum of the scaled numbers

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

and check that it is 1.

*MPL note.* In MPL, the reduction operator is a `std::function`:

```
|| float
|| xrank = static_cast<float>( comm_world.rank() ),
|| xreduce;
|| // separate recv buffer
|| comm_world.allreduce(mpl::plus<float>(), xrank, xreduce);
|| // in place
|| comm_world.allreduce(mpl::plus<float>(), xrank);
|| if ( comm_world.rank() == comm_world.size() - 1 )
|| std::cout << "Allreduce got: separate=" << xreduce
|| << ", inplace=" << xrank << std::endl;
```

For the source of this example, see section [3.16.4](#)

Note the parentheses after the operator. Also note that the operator comes first, not last.

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 3.3.** 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.

#### 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:

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

and for longer buffers:

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

You could cast the buffers and write:

```
|| MPI_Allreduce( (void*)&x, (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.

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++ Application Programmer Interface (API) has been removed from the standard.

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

*Python note.* In Python, all *buffers* are *numpy* objects, which carry information about their type and size.

```
|| x = numpy.zeros(1, dtype=np.float32)
|| comm.Allreduce( x )
|| xs = numpy.zeros(2, dtype=np.float64)
|| comm.Allreduce( xx )
```

*MPL note.* The C++ package MPL uses polymorphism in its *buffer* handling, in addition to the templating mechanism. Thus, there are polymorphic methods for scalars, and for buffers. For the latter, a layout specification is needed.

```

|| float x;
|| comm.Allreduce( x );
|| float xx[2];
|| comm.Allreduce( xx, mpl::contiguous_layout<float>(2) );

```

The *contiguous\_layout* is a ‘derived type’; this will be discussed in more detail in section 5.3. For now, interpret it as a way of indicating the count/type part of a buffer specification.

If your buffer is a *std::vector* you need to take the *.data()* component of it:

```

|| vector<float> xx(2);
|| comm.Allreduce( xx.data(), mpl::contiguous_layout<float>(2) );

```

C++ has improved handling of *static arrays*, so MPL can send those 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

```

For the source of this example, see section 3.16.5

### 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.

### 3.2 MPI\_Reduce

```
C:
int MPI_Reduce(
    const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype,
    MPI_Op op, int root, MPI_Comm comm)

Fortran:
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: count, root
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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

*For the source of this example, see section 3.16.6*

**Exercise 3.4.** 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 3.5.** 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. MPI topic: Collectives

---

#### 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);
```

For the source of this example, see section [3.16.7](#)

Now every process only has a receive buffer, so this has the advantage of saving half the memory. The input value is put in the receive buffer.

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 ...
- ... 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);
```

For the source of this example, see section [3.16.7](#)

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);
```

For the source of this example, see section [3.16.7](#)

In Fortran the code is less elegant because you can not do these address calculations:

```
// 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, err)
else
    mynumber = mytid
    call MPI_Reduce(mynumber, result, 1, MPI_REAL, MPI_SUM, &
                    target_proc, comm, err)
end if
! the result should be ntids*(ntids-1)/2:
if (mytid.eq.target_proc) then
    write(*, ("Result ",f5.2," compared to n(n-1)/2=",f5.2)) &
        result, ntids*(ntids-1)/2.
end if

```

For the source of this example, see section 3.16.8

*Python note.* The value `MPI.IN_PLACE` can be used for the send buffer:

```

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

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

```

For the source of this example, see section 3.16.9

*MPL note.* In MPL, the in-place variant is activated by specifying only one instead of two buffer arguments.

```

float
xrank = static_cast<float>(comm_world.rank() ),
xreduce;
// separate recv buffer
comm_world.allreduce(mpl::plus<float>(), xrank, xreduce);
// in place
comm_world.allreduce(mpl::plus<float>(), xrank);
if (comm_world.rank()==comm_world.size()-1)
std::cout << "Allreduce got: separate=" << xreduce
<< ", inplace=" << xrank << std::endl;

```

For the source of this example, see section 3.16.4

Reducing a buffer requires specification of a `contiguous_layout`:

```

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

```

### 3. MPI topic: Collectives

---

#### 3.3 MPI\_Bcast

```
C:  
int MPI_Bcast(  
    void* buffer, int count, MPI_Datatype datatype,  
    int root, MPI_Comm comm)  
  
Fortran:  
MPI_Bcast(buffer, count, datatype, root, comm, ierror)  
TYPE(*), DIMENSION(..) :: buffer  
INTEGER, INTENT(IN) :: count, root  
TYPE(MPI_Datatype), INTENT(IN) :: datatype  
TYPE(MPI_Comm), INTENT(IN) :: comm  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror  
  
Python native:  
rbuf = MPI.Comm.bcast(self, obj=None, int root=0)  
Python numpy:  
MPI.Comm.Bcast(self, buf, int root=0)  
  
MPL:  
template<typename T >  
void mpl::communicator::bcast  
( int root, T & data ) const  
( int root, T * data, const layout< T > & l ) const
```

For the source of this example, see section [3.16.10](#)

Similarly for the rooted reduce:

```
int root = 1;  
float  
xrank = static_cast<float>( comm_world.rank() ),  
xreduce;  
// separate receive buffer  
comm_world.reduce(mpl::plus<float>(), root, xrank,xreduce);  
// in place  
comm_world.reduce(mpl::plus<float>(), root, xrank);  
if ( comm_world.rank()==root )  
    std::cout << "Allreduce got: separate=" << xreduce  
        << ", inplace=" << xrank << std::endl;
```

For the source of this example, see section [3.16.4](#)

#### 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.

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);
```

For the source of this example, see section [3.16.11](#)

*Python note.* 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 native and numpy variants. In the native 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.")
```

For the source of this example, see section [3.16.12](#)

### 3. MPI topic: Collectives

---

Initial:

matrix	sol	rhs	action
2 2 13	1	17	
4 5 32	1	41	
-2 -3 -16	1	-21	

Step 1:

matrix	sol	rhs	action
2 2 13	1	17	take this row
4 5 32	1	41	
-2 -3 -16	1	-21	

Step 2:

matrix	sol	rhs	action
2 2 13	1	17	take this row
↓ ↓			
4 5 32	1	41	minus × 2
-2 -3 -16	1	-21	

Step 3:

matrix	sol	rhs	action
2 2 13	1	17	take this row
0 1 6	1	7	
-2 -3 -16	1	-21	

Step 4:

matrix	sol	rhs	action
2 2 13	1	17	take this row
↓ ↓ ↓			
0 1 6	1	7	
-2 -3 -16	1	-21	plus × 1

Step 5:

matrix	sol	rhs	action
2 2 13	1	17	take this row
0 1 6	1	7	
0 -1 -3	1	-4	

Step 6:

matrix	sol	rhs	action
2 2 13	1	17	first column done
0 1 6	1	7	
0 -1 -3	1	-4	

Step 7:

matrix	sol	rhs	action
2 2 13	1	17	
0 1 6	1	7	take this row
0 -1 -3	1	-4	

Step 14:

matrix	sol	rhs	action
2 0 1	1	3	minus × 1/3
0 1 6	1	7	
↑↑↑ 0 0 3	1	3	take this row

Step 8:

matrix	sol	rhs	action
2 2 13	1	17	minus × 2
↑↑↑ 0 1 6	1	7	take this row
0 -1 -3	1	-4	

Step 15:

matrix	sol	rhs	action
2 0 0	1	2	
0 1 6	1	7	
0 0 3	1	3	take this row

Step 9:

matrix	sol	rhs	action
2 0 1	1	3	
0 1 6	1	7	take this row
0 -1 -3	1	-4	

Step 16:

matrix	sol	rhs	action
2 0 0	1	2	
0 1 6	1	7	minus × 2
↑↑↑ 0 0 3	1	3	take this row

Step 10:

matrix	sol	rhs	action
2 0 1	1	3	
0 1 6	1	7	take this row
0 -1 -3	1	-4	plus × 1

Step 17:

matrix	sol	rhs	action
2 0 0	1	2	
0 1 0	1	1	
0 0 3	1	3	take this row

Step 11:

matrix	sol	rhs	action
2 0 1	1	3	
0 1 6	1	7	
0 0 3	1	3	

Step 18:

matrix	sol	rhs	action
2 0 0	1	2	
0 1 0	1	1	
0 0 3	1	3	third column done

Step 12:

matrix	sol	rhs	action
2 0 1	1	3	
0 1 6	1	7	second column done
0 0 3	1	3	

Step 13:

matrix	sol	rhs	action
2 0 1	1	3	
0 1 6	1	7	
0 0 3	1	3	take this row

Finished:

matrix	sol	rhs	action
2 0 0	1	2	
0 1 0	1	1	
0 0 3	1	3	

Figure 3.2: Gauss-Jordan elimination example

For the following exercise, study figure 3.2.

**Exercise 3.6.** 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  $c = 1, \dots, n$   
         $A_{rc} \leftarrow A_{rc} - \ell_r^{(k)} A_{rc}$

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.

**Exercise 3.7.** 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 Rooted collectives: gather and scatter

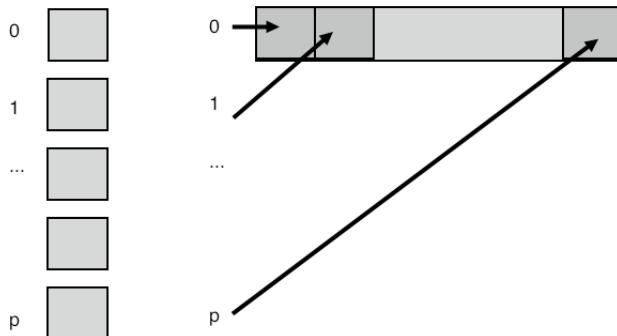


Figure 3.3: Gather collects all data onto a root

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 3.4.

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

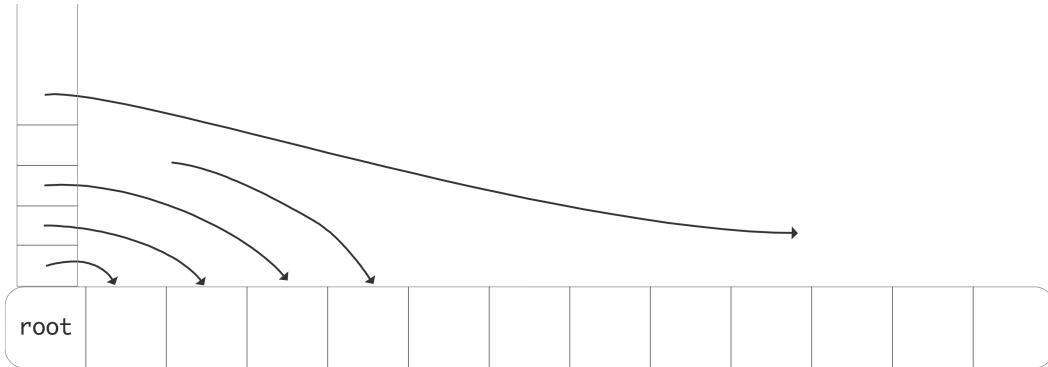


Figure 3.4: A scatter operation

- 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);
```

*For the source of this example, see section 3.16.13*

This will also be the basis of a more elaborate example in section 3.8.

**Exercise 3.8.** Let each process compute a random number. You want to print the 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:

### 3.4 MPI\_Gather

C:

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

Semantics:

```
IN sendbuf: starting address of send buffer (choice)
IN sendcount: number of elements in send buffer (non-negative integer)
IN sendtype: data type of send buffer elements (handle)
OUT recvbuf: address of receive buffer (choice, significant only at root)
IN recvcount: number of elements for any single receive (non-negative integer, significant
IN recvtype: data type of recv buffer elements (significant only at root) (handle)
IN root: rank of receiving process (integer)
IN comm: communicator (handle)
```

Fortran:

```
MPI_Gather
    (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
     root, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcount, root
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

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

- The prototype for `MPI_Gather` (figure 3.4) 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. Non-root processes can use a *null pointer* here.
- More elegantly, the `MPI_IN_PLACE` option can be used buffers that are not applicable. See section 3.3.2.

*MPL note.* Gathering (by *gather*) or scattering (by *scatter*) a single scalar takes a scalar argument and a raw array:

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

For the source of this example, see section 3.16.10

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);
mpi::contiguous_layout<float> twonums(2);
comm_world.scatter
(0, vsend.data(), twonums, vrecv.data(), twonums );
```

For the source of this example, see section 3.16.10

### 3.4.1 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 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 3.5 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 5.3.4.

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 );
}
```

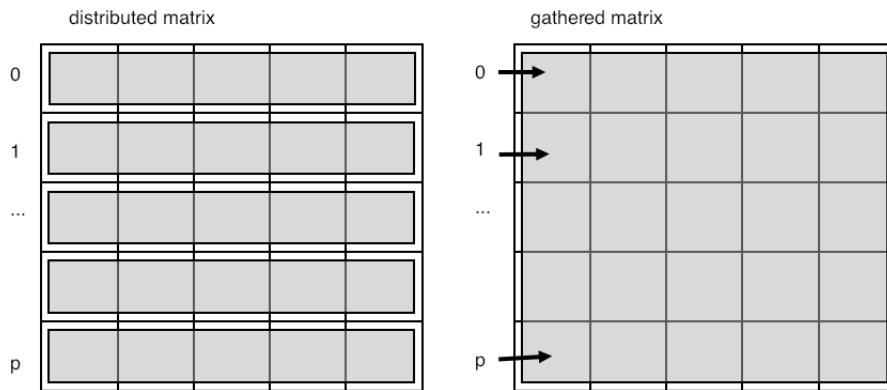


Figure 3.5: Gather a distributed matrix onto one process

*For the source of this example, see section 3.16.14*

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 3.9.** 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?

### 3.4.2 Allgather

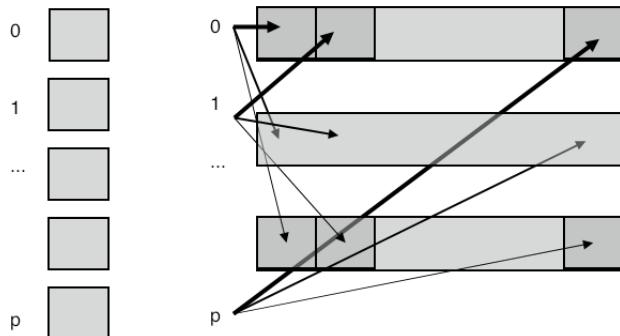


Figure 3.6: All gather collects all data onto every process

The **MPI\_Allgather** (figure 3.5) routine does the same gather onto every process: each process winds up with the totality of all data; figure 3.6.

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

Some cases look like an all-gather but can be implemented more efficiently. Suppose you have two distributed vectors, and you want to create a new vector that contains those elements of the one that do not

### 3.5 MPI\_Allgather

C:

```
int MPI_Allgather(const void *sendbuf, int sendcount,
                  MPI_Datatype sendtype, void *recvbuf, int recvcount,
                  MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Iallgather(const void *sendbuf, int sendcount,
                   MPI_Datatype sendtype, void *recvbuf, int recvcount,
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

Fortran:

```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT,
               RECVTYPE, COMM, IERROR)
<type>    SENDBUF (*), RECVBUF (*)
INTEGER     SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, COMM,
INTEGER     IERROR
MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT,
                RECVTYPE, COMM, REQUEST, IERROR)
<type>    SENDBUF (*), RECVBUF (*)
INTEGER     SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, COMM
INTEGER     REQUEST, IERROR
```

C++ Syntax

Parameters:

```
sendbuf : Starting address of send buffer (choice).
sendcount: Number of elements in send buffer (integer).
sendtype: Datatype of send buffer elements (handle).
recvbuf: Starting address of recv buffer (choice).
recvcount: Number of elements received from any process (integer).
recvtype: Datatype of receive buffer elements (handle).
comm; Communicator (handle).

recvbuf: Address of receive buffer (choice).
request: Request (handle, non-blocking only).
```

### 3.6 MPI\_Alltoall

```
int MPI_Alltoallv
    (void *sendbuf, int sendcnt, MPI_Datatype sendtype,
     void *recvbuf, int recvcnt, MPI_Datatype recvtype,
     MPI_Comm comm)
```

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 3.10.** Can you think of another algorithm for taking the set difference of two distributed vectors. Hint: look up ‘bucket-brigade algorithm’ in [5]. What is the time and space complexity of this algorithm? Can you think of other advantages beside a reduction in workspace?

## 3.5 All-to-all

The all-to-all operation **MPI\_Alltoall** (figure 3.6) 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.

### 3.5.1 All-to-all as data transpose

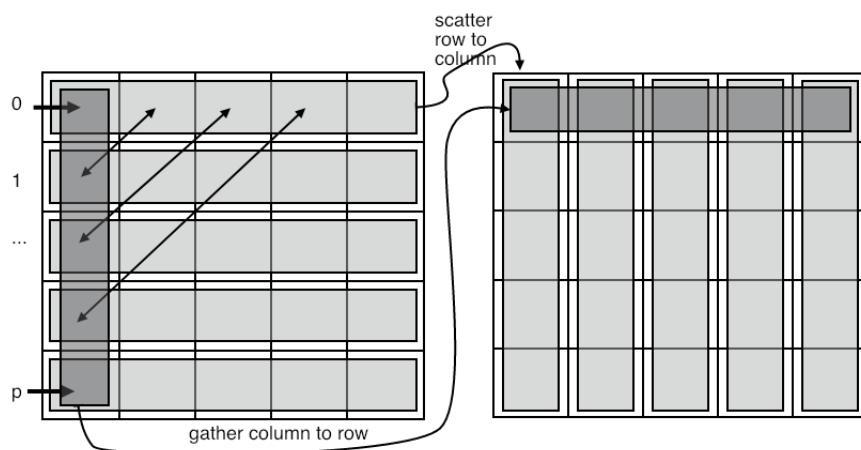


Figure 3.7: All-to-all transposes data

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  $P \times P$ , denoting

### 3. MPI topic: Collectives

---

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 Fast Fourier Transform (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 3.11.** In the initial stage of *radix sorting*, 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.

#### 3.5.2 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.

**Exercise 3.12.** The actual data shuffle of a *radix sort* can be done with `MPI_Alltoallv`.

Finish the code of exercise 3.11.

### 3.6 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 HPSC-??.

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 3.7), using an important example: the *sparse matrix-vector product* (see HPSC-?? 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:

```
// 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 determin:
*procs_to_send_to, nprocs_to_send_to;
```

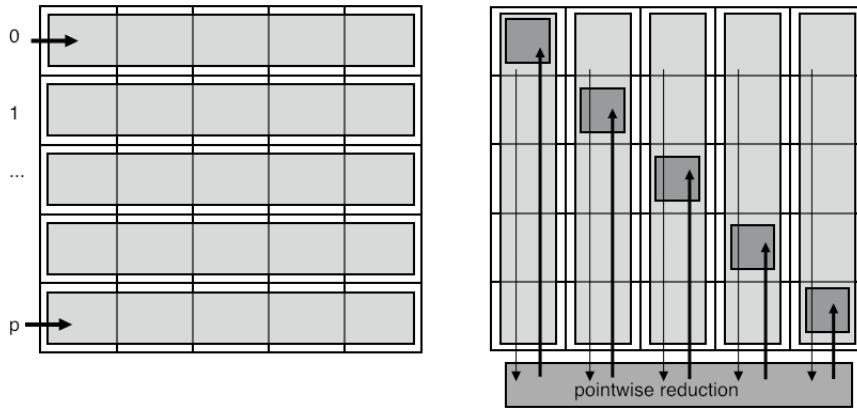


Figure 3.8: Reduce scatter

*For the source of this example, see section 3.16.15*

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,
   MPI_SUM, comm);
```

*For the source of this example, see section 3.16.15*

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.
 */
```

### 3. MPI topic: Collectives

---

```
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);
```

For the source of this example, see section [3.16.15](#)

The **MPI\_Reduce\_scatter** (figure 3.7) 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 3.5.1). 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 3.8).

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

#### 3.6.1 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);
```

For the source of this example, see section [3.16.15](#)

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

### 3.7 MPI\_Reduce\_scatter

Semantics:

```
MPI_Reduce_scatter
    ( sendbuf, recvbuf, recvcounts, datatype, op, comm)
MPI_Reduce_scatter_block
    ( sendbuf, recvbuf, recvcount, datatype, op, comm)
```

Input parameters:

```
sendbuf: starting address of send buffer (choice)
recvcount: element count per block (non-negative integer)
recvcounts: non-negative integer array (of length group size)
    specifying the number of elements of the result distributed to each
process.
datatype: data type of elements of send and receive buffers (handle)
op: operation (handle)
comm: communicator (handle)
```

Output parameters:

```
recvbuf: starting address of receive buffer (choice)
```

C:

```
int MPI_Reduce_scatter
    (const void* sendbuf, void* recvbuf, const int recvcounts[],
     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

F:

```
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: recvcounts(*)
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

PY:

```
comm.Reduce_scatter(sendbuf, recvbuf, recvcounts=None, Op op=SUM)
```

### 3. MPI topic: Collectives

---

#### 3.8 MPI\_BARRIER

```
C:  
int MPI_BARRIER( MPI_Comm comm )  
  
Fortran2008:  
MPI_BARRIER(COMM, IERROR)  
Type(MPI_Comm), intent(int) :: COMM  
INTEGER,intent(out) :: IERROR  
  
Fortran 95:  
MPI_BARRIER(COMM, IERROR)  
INTEGER :: COMM, IERROR  
  
Input parameter:  
comm : Communicator (handle)  
  
Output parameter:  
ierror : Error status (integer), Fortran only  
  
/*  
 * 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);
```

For the source of this example, see section [3.16.15](#)

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);
```

For the source of this example, see section [3.16.16](#)

## 3.7 Barrier

A barrier call, **MPI\_BARRIER** (figure 3.8) 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 non-blocking ones; section 3.11) introduce a barrier of sorts themselves.

## 3.8 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 3.9) 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` 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` does the same, but leaves its result on all processes; `MPI_Alltoallv` does a different variable-sized gather on each process.

```
int MPI_Scatterv
  (void* sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype,
   void* recvbuf, int recvcount, MPI_Datatype recvtype,
   int root, MPI_Comm comm)
```

### 3.9 MPI\_Gatherv

C:

```
int MPI_Gatherv(
    const void* sendbuf, int sendcount, MPI_Datatype sendtype,
    void* recvbuf, const int recvcounts[], const int displs[],
    MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Semantics:

IN sendbuf: starting address of send buffer (choice)  
IN sendcount: number of elements in send buffer (non-negative integer)  
IN sendtype: data type of send buffer elements (handle)  
OUT recvbuf: address of receive buffer (choice, significant only at root)  
IN recvcounts: non-negative integer array (of length group size) containing the number of elements to receive from each process  
IN displs: integer array (of length group size). Entry i specifies the displacement relative to the start of the receive buffer for process i  
IN recvtype: data type of recv buffer elements (significant only at root) (handle)  
IN root: rank of receiving process (integer)  
IN comm: communicator (handle)

Fortran:

```
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

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

```

|| int MPI_Allgatherv
||   (void *sendbuf, int sendcount, MPI_Datatype sendtype,
||    void *recvbuf, int *recvcounts, int *displs,
||    MPI_Datatype recvtype, MPI_Comm comm)

```

### 3.8.1 Example of Gatherv

We use **MPI\_Gatherv** to do an irregular gather onto a root. We first need an **MPI\_Gather** to determine offsets.

```

// 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) );
}
// everyone contributes their local size info
MPI_Gather(&localsize,1,MPI_INT,
            localsizes,1,MPI_INT,root,comm);
// the root constructs the offsets array
if (procno==root) {
    int total_data = 0;
    for (int i=0; i<nprocs; i++) {
        offsets[i] = total_data;
        total_data += localsizes[i];
    }
    alldata = (int*) malloc( total_data*sizeof(int) );
}
// everyone contributes their data
MPI_Gatherv(localdata,localsize,MPI_INT,
            alldata,localsizes,offsets,MPI_INT,root,comm);

```

For the source of this example, see section 3.16.17

```

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

```

For the source of this example, see section 3.16.18

### 3.8.2 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.

### 3. MPI topic: Collectives

---

#### 3.10 MPI\_Alltoallv

```
int MPI_Alltoallv
    (void *sendbuf, int *sendcnts, int *sdispls, MPI_Datatype sendtype,
     void *recvbuf, int *recvcnts, int *rdispls, MPI_Datatype recvtype,
      MPI_Comm comm)

// 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]; }
MPI_Allgatherv
( my_array, procno+1, MPI_INT,
  global_array, recv_counts, recv_displs, MPI_INT, comm );
```

For the source of this example, see section 3.16.19

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)
```

For the source of this example, see section 5.7.3

```
my_count = np.empty(1, dtype=np.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]
  )
```

For the source of this example, see section 5.7.3

#### 3.8.3 Variable all-to-all

**MPI\_Alltoallv** (figure 3.10)

### 3.9 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 HPSC-??.

### 3.11 MPI\_Scan

```
C:
int MPI_Scan(const void* sendbuf, void* recvbuf,
    int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
IN sendbuf: starting address of send buffer (choice)
OUT recvbuf: starting address of receive buffer (choice)
IN count: number of elements in input buffer (non-negative integer)
IN datatype: data type of elements of input buffer (handle)
IN op: operation (handle)
IN comm: communicator (handle)

Fortran:
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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

The **MPI\_Scan** (figure 3.11) routine is an *inclusive scan* operation.

```
// 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);
```

For the source of this example, see section 3.16.21

In python native 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=np.int)
rbuf = np.empty(1, dtype=np.int)
sbuf[0] = mycontrib
comm.Scan(sbuf, rbuf)
```

For the source of this example, see section 3.16.22

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.

### 3. MPI topic: Collectives

---

#### 3.12 MPI\_Exscan

C:

```
int MPI_Exscan(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Iexscan(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
    MPI_Request *request)
```

Fortran:

```
MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
<type>    SENDBUF(*), RECVBUF(*)
INTEGER    COUNT, DATATYPE, OP, COMM, IERROR
MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
<type>    SENDBUF(*), RECVBUF(*)
INTEGER    COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
```

Input Parameters

sendbuf: Send buffer (choice).  
count: Number of elements in input buffer (integer).  
datatype: Data type of elements of input buffer (handle).  
op: Operation (handle).  
comm: Communicator (handle).

Output Parameters

recvbuf: Receive buffer (choice).  
request: Request (handle, non-blocking only).

#### 3.9.1 Exclusive scan

Often, the more useful variant is the *exclusive scan* `MPI_Exscan` (figure 3.12) with the same prototype.

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

#### 3.9.2 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  $0 \dots n_p - 1$  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.

**Exercise 3.14.** Do you use `MPI_Scan` or `MPI_Exscan` for this operation? How would you describe the result of the other scan operation, given the same input?

Exclusive scan examples:

```
// exscan.c
int my_first=0,localsize;
// localsize = ..... result of local computation ....
// find myfirst location based on the local sizes
err = MPI_Exscan(&localsize,&my_first,
                  1,MPI_INT,MPI_SUM,comm); CHK(err);
```

For the source of this example, see section 3.16.23

```
## exscan.py
localsize = 10+random.randint(1,nprocs)
myfirst = 0
mypartial = comm.exscan(localsize,0)
```

For the source of this example, see section 3.16.24

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 HPSC-??.) 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

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

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

## 3.10 MPI Operators

MPI operators are used in reduction operators. Most common operators, such as sum or maximum, have been built into the MPI library, but it is possible to define new operators.

### 3.10.1 Pre-defined operators

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

### 3. MPI topic: Collectives

---

pi-opstable

*MPL note.* In MPL, operators are classes with an `operator()` method. For example:

```
|| float
|| xrank = static_cast<float>( comm_world.rank() ),
|| xreduce;
|| // separate recv buffer
|| comm_world.allreduce(mpl::plus<float>(), xrank, xreduce);
|| // in place
|| comm_world.allreduce(mpl::plus<float>(), xrank);
|| if ( comm_world.rank()==comm_world.size()-1 )
|| std::cout << "Allreduce got: separate=" << xreduce
||           << ", inplace=" << xrank << std::endl;
```

For the source of this example, see section [3.16.4](#)

(Note the parentheses after the operator; without that, `mpl::min<int>` is a type, and you get an error message about ‘type name is not allowed’.)

#### 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.

The Fortran interface to MPI was designed to use pre-Fortran90 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:

```
|| Real8,dimension(2,N) :: input,output
|| call MPI_Reduce( input,output, N, MPI_2DOUBLE_PRECISION, &
||                   MPI_MAXLOC, root, comm )
```

#### 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 3.13), which takes a user function and turns it into an object of type `MPI_Op`, which can then be used in any reduction:

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

For the source of this example, see section [3.16.25](#)

### 3.13 MPI\_Op\_create

Semantics:

```
MPI_OP_CREATE( function, commute, op)
[ IN function] user defined function (function)
[ IN commute] true if commutative; false otherwise.
[ OUT op] operation (handle)
```

C:

```
int MPI_Op_create
    (MPI_User_function *function, int commute,
     MPI_Op *op)
```

Fortran 2008:

```
USE mpi_f08
MPI_Op_create(user_fn, commute, op, ierror)
PROCEDURE(MPI_User_function) :: user_fn
LOGICAL, INTENT(IN) :: commute
TYPE(MPI_Op), INTENT(OUT) :: op
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran90:

```
MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR)
EXTERNAL FUNCTION
LOGICAL COMMUTE
INTEGER OP, IERROR
```

Python:

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

### 3. MPI topic: Collectives

---

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

For the source of this example, see section [3.16.26](#)

The user function needs to have the following prototype:

```
|| 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 non-zero number in an array of nonnegative integers:

```
// reductpositive.c
void reduce_without_zero(void *in, void *inout, int *len, MPI_Datatype *type) {
    // r is the already reduced value, n is the new value
    int n = *(int*)in, r = *(int*)inout;
    int m;
    if (n==0) { // new value is zero: keep r
        m = r;
    } else if (r==0) {
        m = n;
    } else if (n<r) { // new value is less but not zero: use n
        m = n;
    } else { // new value is more: use r
        m = r;
    };
    *(int*)inout = m;
}
```

For the source of this example, see section [3.16.25](#)

*Python note.* 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
```

**3.14 MPI\_Op\_commutative**

Semantics:

```
MPI_Op_commutative(op, commute)
IN  op : handle
OUT commute : true/false
```

C:

```
int MPI_Op_commutative(MPI_Op op, int *commute)
```

Fortran:

```
MPI_OP_COMMUTATIVE( op, commute)
TYPE(MPI_Op), INTENT(IN) :: op
LOGICAL, INTENT(OUT) :: commute
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
||| elif n<r:
|||     m = n
||| else:
|||     m = r
||| inout_array[0] = m
```

For the source of this example, see section [3.16.26](#)

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

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

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

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 3.14).

**3.10.3 Local reduction**

The application of an `MPI_Op` can be performed with the routine `MPI_Reduce_local` (figure 3.15). 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

---

#### 3.15 MPI\_Reduce\_local

Semantics:

```
MPI_REDUCE_LOCAL( inbuf, inoutbuf, count, datatype, op)
```

Input parameters:

inbuf: input buffer (choice)

count: number of elements in inbuf and inoutbuf buffers  
(non-negative integer)

datatype: data type of elements of inbuf and inoutbuf buffers  
(handle)

op: operation (handle)

Input/output parameters:

inoutbuf: combined input and output buffer (choice)

C:

```
int MPI_Reduce_local  
(void* inbuf, void* inoutbuf, int count,  
 MPI_Datatype datatype, MPI_Op op)
```

Fortran:

```
MPI_REDUCE_LOCAL(INBUF, INOUBUF, COUNT, DATATYPE, OP, IERROR)  
<type> INBUF(*), INOUTBUF(*)  
INTEGER :: COUNT, DATATYPE, OP, IERROR
```

### 3.11 Non-blocking 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 *non-blocking 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 [8]. Occasionally, a non-blocking collective can be used for non-obvious purposes, such as the **MPI\_IBARRIER** in [9].

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.

Non-blocking collectives offer a number of performance advantages:

**3.16 MPI\_Iallreduce**

Semantics

```
int MPI_Iallreduce(
    const void *sendbuf, void *recvbuf,
    int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
    MPI_Request *request)
```

Input Parameters

sendbuf : starting address of send buffer (choice)  
 count : number of elements in send buffer (integer)  
 datatype : data type of elements of send buffer (handle)  
 op : operation (handle)  
 comm : communicator (handle)

Output Parameters

recvbuf : starting address of receive buffer (choice)  
 request : communication request (handle)

- 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 non-blocking collective with a blocking one.

**Remark 4** *Blocking and non-blocking don't match: either all processes call the non-blocking 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 non-blocking calls: you can catch a message with MPI\_Irecv that was sent with MPI\_Send.*

**Exercise 3.16.** Revisit exercise 6.1. 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 non-blocking broadcasts, and time the difference between a blocking and a non-blocking solution.

**MPI\_Iallreduce** (figure 3.16)

### 3.17 MPI\_Iallgather

Semantics

```
int MPI_Iallgather(
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    MPI_Comm comm, MPI_Request *request)
```

Input Parameters

```
sendbuf : starting address of send buffer (choice)
sendcount : number of elements in send buffer (integer)
sendtype : data type of send buffer elements (handle)
recvcount : number of elements received from any process (integer)
recvtype : data type of receive buffer elements (handle)
comm : communicator (handle)
```

Output Parameters

```
recvbuf : address of receive buffer (choice)
request : communication request (handle)
```

**MPI\_Igatherv, MPI\_Igathervv, MPI\_Iallgather** (figure 3.17), **MPI\_Iallgatherv, MPI\_Iscatter, MPI\_Iscatterv, MPI\_Ireduce, MPI\_Iallreduce, MPI\_Ireduce\_scatter, MPI\_Ireduce\_scatter\_block, MPI\_Ialltoall, MPI\_Ialltoallv, MPI\_Ialltoallw, MPI\_Ibarrier, MPI\_Ibcast, MPI\_Iexscan, MPI\_Iscan,**

*MPL note.* Non-blocking collectives have the same argument list as the corresponding blocking variant, except that instead of a **void** result, they return an *irequest*.

### 3.11.1 Examples

#### 3.11.1.1 Array transpose

To illustrate the overlapping of multiple non-blocking 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);
}
```

For the source of this example, see section 3.16.27

Introducing the non-blocking `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);
```

For the source of this example, see section 3.16.27

**Exercise 3.17.** Can you implement the same algorithm with `MPI_Igather`?

### 3.11.1.2 Stencils

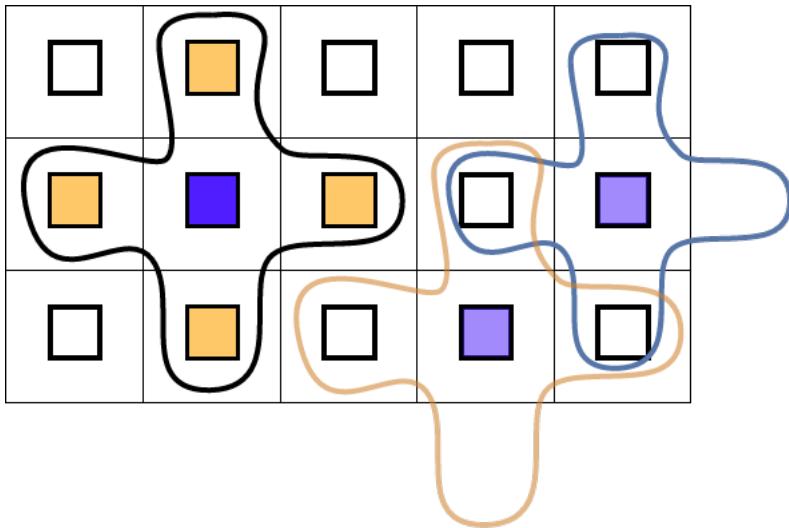


Figure 3.9: Illustration of five-point stencil gather

The ever-popular *five-point stencil* evaluation does not look like a collective operation, and indeed, it is usually evaluated with (non-blocking) send/recv operations. However, if we create a subcommunicator on each subdomain that contains precisely that domain and its neighbors, (see figure 3.9) 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 non-blocking collectives make this feasible.

We will see an even more elegant formulation of this operation in section 10.2.

## 3.11.2 Non-blocking barrier

Probably the most surprising non-blocking collective is the *non-blocking barrier* `MPI_Ibarrier` (figure 3.18). 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 non-blocking barrier:

### 3.18 MPI\_Ibarrier

```
C:
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)

Input Parameters
comm : communicator (handle)

Output Parameters
request : communication request (handle)

Fortran2008:
MPI_Ibarrier(comm, request, ierror)
Type(MPI_Comm), intent(int) :: comm
TYPE(MPI_Request), intent(out) :: request
INTEGER, intent(out), optional :: ierror
```

- 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.

```
// ibarrierprobe.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-block barrier
 * and gets a request to test/wait for
 */
MPI_Request barrier_request;
MPI_Ibarrier(comm, &barrier_request);
```

For the source of this example, see section [3.16.28](#)

Now every process alternately probes for messages and tests for completion of the barrier. Probing is done through the non-blocking **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 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!

```

For the source of this example, see section 3.16.28

We can use a non-blocking 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);

```

For the source of this example, see section 3.16.29

## 3.12 Performance of collectives

It is easy to visualize a broadcast as in figure 3.10: see figure 3.10. the root sends all of its data directly

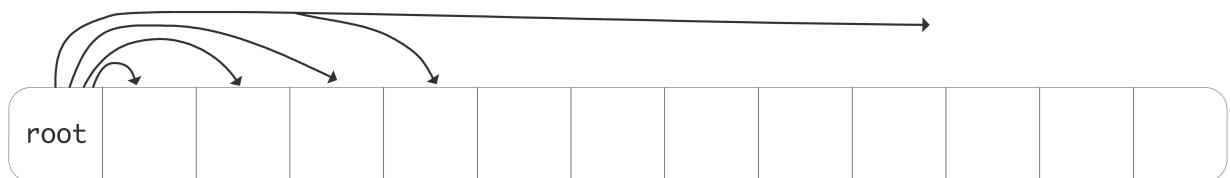


Figure 3.10: A simple broadcast

to 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  $\alpha$  is the *latency*, a one time delay from establishing the communication between two processes, and  $\beta$  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 3.10 would take a time proportional to the number of processors.

**Exercise 3.18.** What is the total time required for a broadcast involving  $p$  processes? Give  $\alpha$  and  $\beta$  terms separately.

One way to ameliorate that is to structure the broadcast in a tree-like fashion. This is depicted in figure 3.11.

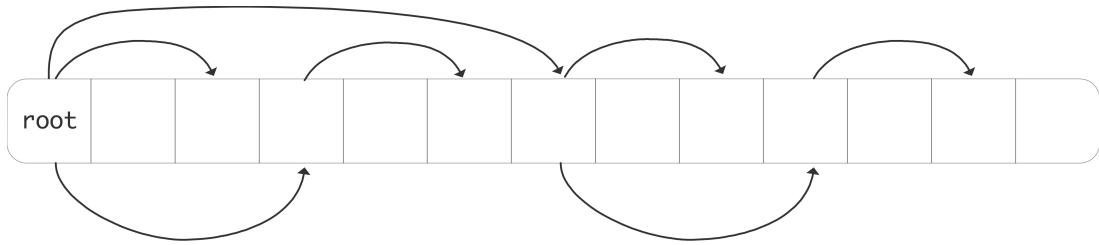


Figure 3.11: A tree-based broadcast

**Exercise 3.19.** How does the communication time now depend on the number of processors, again  $\alpha$  and  $\beta$  terms separately.

What would be a lower bound on the  $\alpha, \beta$  terms?

The theory of the complexity of collectives is described in more detail in HPSC-??; see also [1].

### 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 reduce operation the root may have to wait for other processors. This is illustrated in figure 3.12, which gives a TAU trace of a reduction operation on two nodes, with two six-core sockets (processors) each. We see that<sup>1</sup>:

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

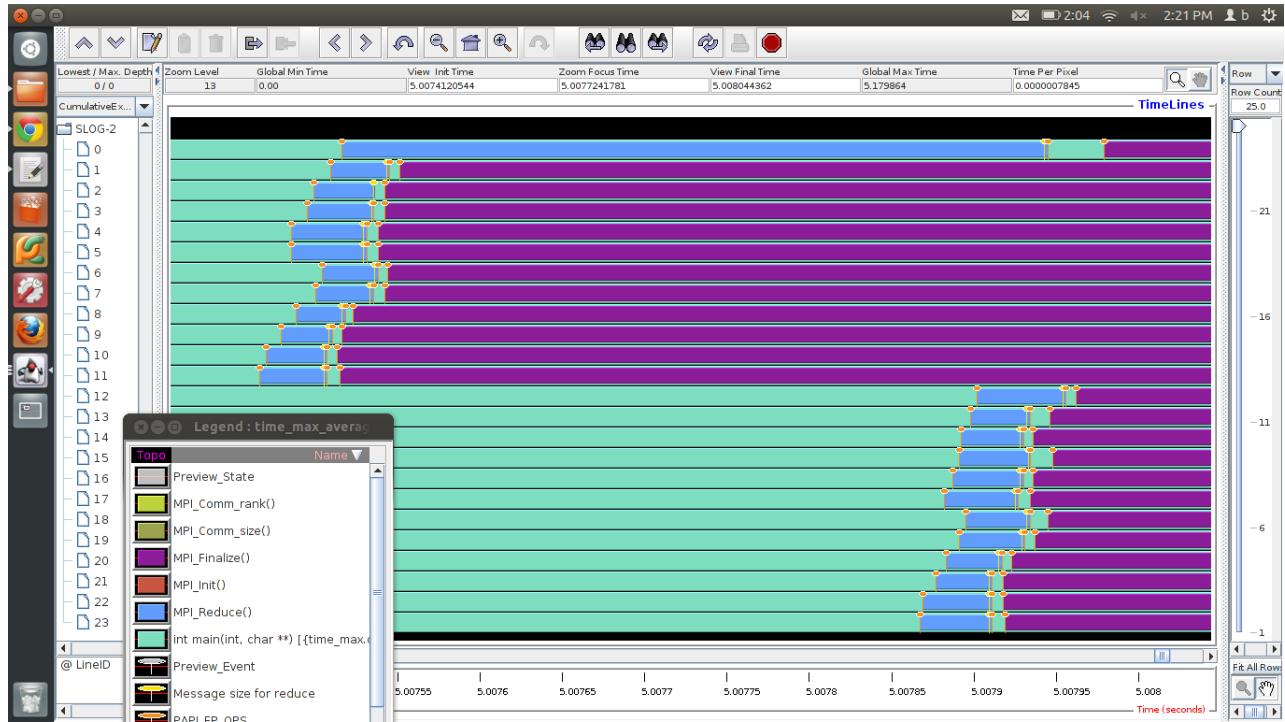


Figure 3.12: Trace of a reduction operation between two dual-socket 12-core nodes

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:

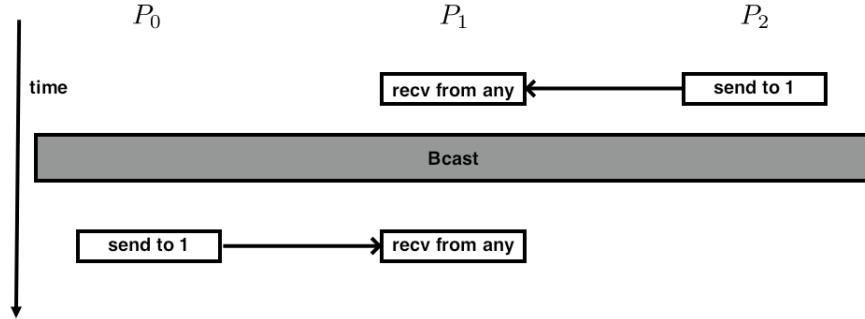
```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, &status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count,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

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

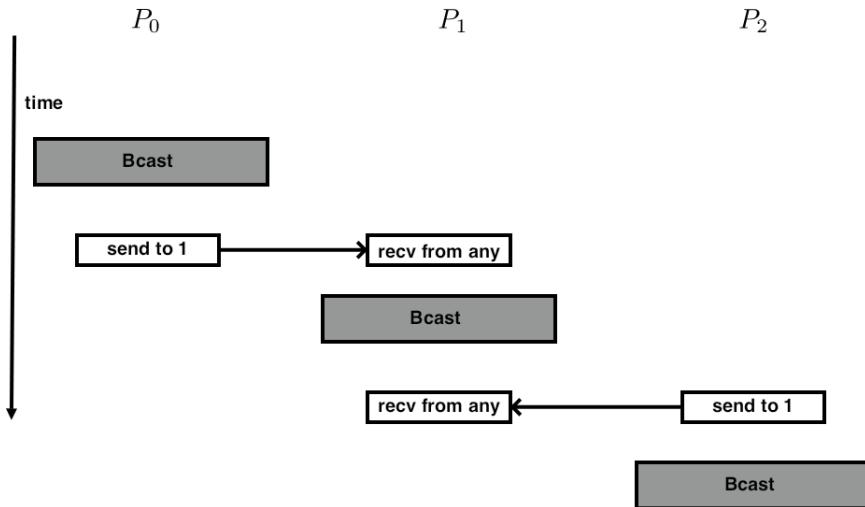
### 3. MPI topic: Collectives

---

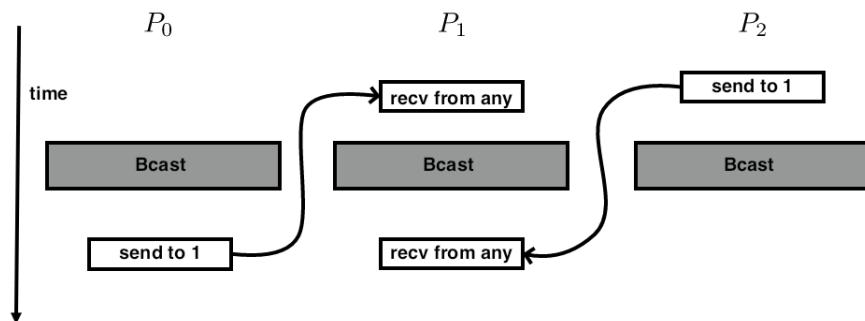
The most logical execution is:



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 3.13: Possible temporal orderings of send and collective calls

```
    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 3.13.

## 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 40.6).

### 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 3.20.** Discuss scalability of the following algorithms:

- You have an array of floating point numbers. You need to compute the sine of each
- You 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 3.21.** 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  $\alpha, \beta$ ?

Run some tests and confirm.

**Simple ring** Let the root only send to the next process, and that one send to its neighbour. This scheme is known as a *bucket brigade*; see also section 4.2.3.

What is the expected performance of this in terms of  $\alpha, \beta$ ?

Run some tests and confirm.

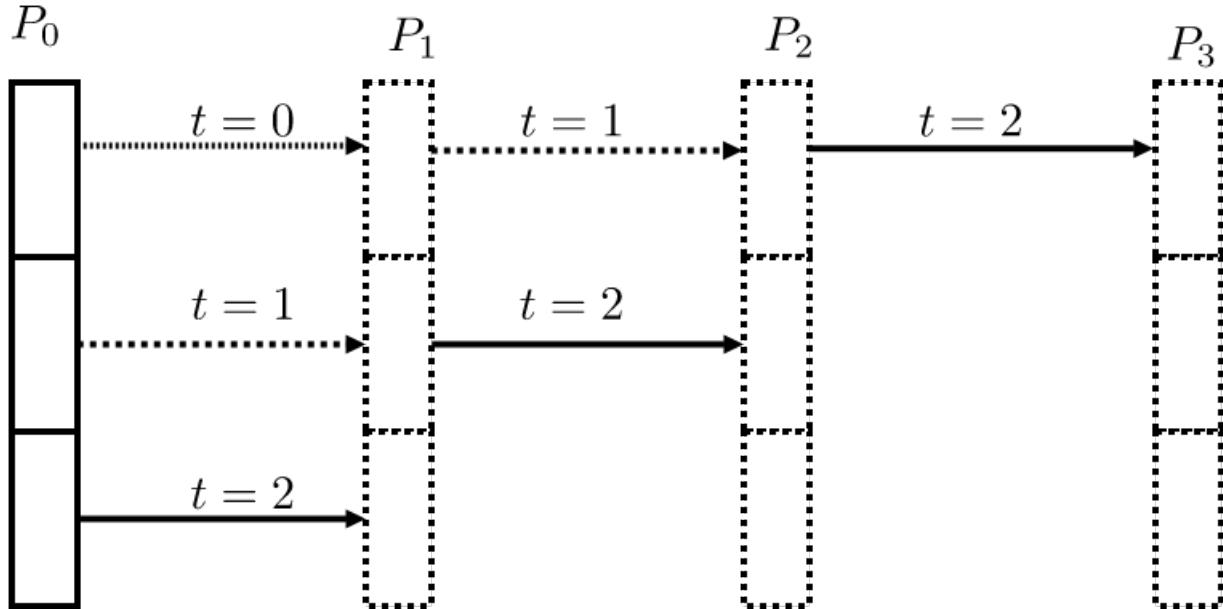


Figure 3.14: A pipelined bucket brigade

**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 3.14.) 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  $\alpha, \beta$ ? Why is this better than the simple ring?

Run some tests and confirm.

**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 3.22.** 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 3.23.** Analyze this variant of recursive doubling. Code it and measure runtimes on SimGrid.

**Exercise 3.24.** Revisit exercise 3.22 and replace the blocking calls by non-blocking `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-linear-algebra>

## 3.15 Review questions

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

**Review 3.25.** 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.

### 3. MPI topic: Collectives

---

- Let each process compute a random number. You want to print on what processor the maximum value is computed.

**Review 3.26.** MPI collectives can be sorted in at least the following categories

- rooted vs rootless
- using uniform buffer lengths vs variable length buffers
- blocking vs non-blocking.

Give examples of each type.

**Review 3.27.** True or false: there are collective routines that do not communicate user data. If true, give an example.

**Review 3.28.** True or false: an `MPI_Scatter` call puts the same data on each process.

**Review 3.29.** True or false: using the option `MPI_IN_PLACE` you only need space for a send buffer in `MPI_Reduce`.

**Review 3.30.** True or false: using the option `MPI_IN_PLACE` you only need space for a send buffer in `MPI_Gather`.

**Review 3.31.** 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 3.32.**

```
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?

**Review 3.33.**

```
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?

**Review 3.34.** 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 3.35.** 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 3.36.** Think of at least two algorithms for doing a broadcast? Compare them with regards to asymptotic behavior.

### 3.16 Sources used in this chapter

#### 3.16.1 Listing of code header

#### 3.16.2 Listing of code examples/mpi/c/allreduce.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char **argv) {

    #include "globalinit.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 %6.9f compared to .5\n",sumrandom/nprocs);

    MPI_Finalize();
    return 0;
}
```

#### 3.16.3 Listing of code examples/mpi/p/allreduce.py

```
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

random_number = random.randint(1,nprocs*nprocs)
print("[%d] random=%d" % (procid,random_number))

# native mode send
max_random = comm.allreduce(random_number,op=MPI.MAX)

if procid==0:
    print("Python native:\n max=%d" % max_random)
```

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

if procid==0:
    print("Python numpy:\n max=%d" % allrandom[0])
```

### 3.16.4 Listing of code examples/mpi/mpf/collectscalar.cxx

```
#include <cstdlib>
#include <complex>
#include <iostream>
#include <vector>
#include <mpl/mpl.hpp>

int main() {

    // MPI Comm world
    const mpl::communicator &comm_world=mpl::environment::comm_world();

    // vector of consecutive floats
    std::vector<float> v;
    if (comm_world.rank()==0)
        for (int i=0; i<comm_world.size(); ++i)
            v.push_back(i);

    // if you scatter, everyone gets a number equal to their rank.
    // rank 0 scatters data to all processes
    float x;
    comm_world.scatter(0, v.data(), x);
    std::cout << "rank " << comm_world.rank() << " got " << x << '\n';

    // wait until all processes have reached this point
    comm_world.barrier();

    // multiply that number, giving twice your rank
    x*=2;

    // rank 0 gathers data from all processes
    comm_world.gather(0, x, v.data());
    if (comm_world.rank()==0) {
        std::cout << "got";
        for (int i=0; i<comm_world.size(); ++i)
            std::cout << " " << i << ":" << v[i];
        std::cout << std::endl;
    }

    // wait until all processes have reached this point
    comm_world.barrier();
```

```
// calculate global sum and pass result to rank 0
if (comm_world.rank()==0) {
    float sum;
    comm_world.reduce(mpl::plus<float>(), 0, x, sum);
    std::cout << "sum = " << sum << '\n';
} else
    comm_world.reduce(mpl::plus<float>(), 0, x);

// wait until all processes have reached this point
comm_world.barrier();

// calculate global sum and pass result to all
{
    float
        xrank = static_cast<float>( comm_world.rank() ),
        xreduce;
    // separate recv buffer
    comm_world.allreduce(mpl::plus<float>(), xrank,xreduce);
    // in place
    comm_world.allreduce(mpl::plus<float>(), xrank);
    if ( comm_world.rank()==comm_world.size()-1 )
        std::cout << "Allreduce got: separate=" << xreduce
                    << ", inplace=" << xrank << std::endl;
}

// calculate global sum and pass result to root
{
    int root = 1;
    float
        xrank = static_cast<float>( comm_world.rank() ),
        xreduce;
    // separate receive buffer
    comm_world.reduce(mpl::plus<float>(), root, xrank,xreduce);
    // in place
    comm_world.reduce(mpl::plus<float>(), root, xrank);
    if ( comm_world.rank()==root )
        std::cout << "Allreduce got: separate=" << xreduce
<< ", inplace=" << xrank << std::endl;
}

return EXIT_SUCCESS;
}
```

#### 3.16.5 Listing of code examples/mpi/mpl/sendarray.cxx

```
#include <cstdlib>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;
```

```
#include <vector>
using std::vector;

#include <mpl/mpl.hpp>

int main() {
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    if (comm_world.size()<2)
        return EXIT_FAILURE;

    /*
     * The compiler knows about arrays so we can send them 'as is'
     */
    double v[2][2][2];

    // Initialize the data
    if (comm_world.rank()==0) {
        double *vt = &(v[0][0][0]);
        for (int i=0; i<8; i++)
            *vt++ = i;

        /*
         * Send and report
         */
        comm_world.send(v, 1); // send to rank 1

        std::cout << "sent: ";
        vt = &(v[0][0][0]);
        for (int i=0; i<8; i++)
            cout << " " << *(vt+i);
        std::cout << '\n';

        // std::cout << "sent: ";
        // for (double &x : v)
        //     std::cout << x << ' ';
        // std::cout << '\n';
    }

    } else if (comm_world.rank()==1) {

        /*
         * Receive data and report
         */
        comm_world.recv(v, 0); // receive from rank 0

        std::cout << "got : ";
        double *vt = &(v[0][0][0]);
        for (int i=0; i<8; i++)
            cout << " " << *(vt+i);
        std::cout << '\n';

    }
    return EXIT_SUCCESS;
}
```

```
}
```

### 3.16.6 Listing of code examples/mpi/c/reduce.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.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.);

    MPI_Finalize();
    return 0;
}
```

### 3.16.7 Listing of code examples/mpi/c/allreduceinplace.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

    int nrandoms = 500000;
    float *myrandoms;
    myrandoms = (float*) malloc(nrandoms*sizeof(float));
    for (int iter=1; iter<=3; iter++) {
        /*
         * We show three different ways of doing the same reduction;
         * this illustrates syntax more than semantics
         */
        if (iter==1) {
            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);
} else if (iter==2) {
    for (int irand=0; irand<nrandoms; irand++)
myrandoms[irand] = (float) rand()/(float)RAND_MAX;
    int root=nprocs-1;
    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);
} else if (iter==2) {
    for (int irand=0; irand<nrandoms; irand++)
myrandoms[irand] = (float) rand()/(float)RAND_MAX;
    int root=nprocs-1;
    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);
}
// the result should be approx nprocs/2:
if (procno==nprocs-1) {
    float sum=0.;
    for (int i=0; i<nrandoms; i++) sum += myrandoms[i];
    sum /= nrandoms*nprocs;
    printf("Result %6.9f compared to .5\n",sum);
}
free(myrandoms);

MPI_Finalize();
return 0;
}
```

### 3.16.8 Listing of code examples/mpi/f/reduceinplace.F90

```
Program ReduceInPlace

use mpi

real :: mynumber,result
integer :: target_proc

#include "globalinit.F90"
```

### 3. MPI topic: Collectives

---

```
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,err)
else
    mynumber = mytid
    call MPI_Reduce(mynumber,result,1,MPI_REAL,MPI_SUM,&
                    target_proc,comm,err)
end if
! the result should be ntids*(ntids-1)/2:
if (mytid.eq.target_proc) then
    write(*,'("Result ",f5.2," compared to n(n-1)/2=",f5.2)') &
        result,ntids*(ntids-1)/2.
end if

call MPI_Finalize(err)

end Program ReduceInPlace
```

#### 3.16.9 Listing of code examples/mpi/p/allreduceinplace.py

```
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

random_number = random.randint(1,nprocs*nprocs)
print("[%d] random=%d" % (procid,random_number))

myrandom = np.empty(1,dtype=np.int)
myrandom[0] = random_number

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

if procid==0:
    print("Python numpy:\n max=%d" % myrandom[0])
```

#### 3.16.10 Listing of code examples/mpi/mpl/collectbuffer.cxx

```
#include <cstdlib>
```

```
#include <complex>
#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;

#include <mpl/mpl.hpp>

int main() {

    // MPI Comm world
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    int nprocs = comm_world.size(), procno = comm_world.rank();
    int iprint = procno==nprocs-1;

    /*
     * Reduce a 2 int buffer
     */
    if (iprint) cout << "Reducing 2p, 2p+1" << endl;

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

    /*
     * Scatter one number to each proc
     */
    if (iprint) cout << "Scattering 0--p" << endl;

    vector<float> v;

    if (comm_world.rank()==0)
        for (int i=0; i<comm_world.size(); ++i)
            v.push_back(i);

    // if you scatter, everyone gets a number equal to their rank.
    // rank 0 scatters data to all processes
    float x;
    comm_world.scatter(0, v.data(), x);

    if (iprint)
        cout << "rank " << procno << " got " << x << '\n';

    /*
     * Scatter two numbers to each proc
     */
}
```

### 3. MPI topic: Collectives

---

```
if (iprint) cout << "Scatter 0--2p" << endl;

vector<float> vrecv(2),vsend(2*nprocs);

if (comm_world.rank()==0)
    for (int i=0; i<2*nprocs; ++i)
        vsend.at(i) = i;

// rank 0 scatters data to all processes
// if you scatter, everyone gets 2p,2p+1
mpl::contiguous_layout<float> twonums(2);
comm_world.scatter
    (0, vsend.data(),twonums, vrecv.data(),twonums );

if (iprint)
    cout << "rank " << procno << " got "
        << vrecv[0] << "," << vrecv[1] << '\n';

return 0;
// multiply that number, giving twice your rank
x*=2;

// rank 0 gathers data from all processes
comm_world.gather(0, x, v.data());
if (comm_world.rank()==0) {
    cout << "got";
    for (int i=0; i<comm_world.size(); ++i)
        cout << " " << i << ":" << v[i];
    cout << endl;
}

// wait until all processes have reached this point
comm_world.barrier();

// calculate global sum and pass result to rank 0
if (comm_world.rank()==0) {
    float sum;
    comm_world.reduce(mpl::plus<float>(), 0, x, sum);
    cout << "sum = " << sum << '\n';
} else
    comm_world.reduce(mpl::plus<float>(), 0, x);

// wait until all processes have reached this point
comm_world.barrier();

return EXIT_SUCCESS;
}
```

**3.16.11 Listing of code examples/mpi/c/usage.c****3.16.12 Listing of code examples/mpi/p/bcast.py**

```
from mpi4py import MPI
import numpy as np

from functools import reduce
import sys

comm = MPI.COMM_WORLD

procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

root = 1
dszie = 10

# first native
if procid==root:
    buffer = [ 5.0 ] * dszie
else:
    buffer = [ 0.0 ] * dszie
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(dszie, dtype=np.float64)
if procid==root:
    for i in range(dszie):
        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.")
```

**3.16.13 Listing of code examples/mpi/c/gather.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

### 3. MPI topic: Collectives

---

```
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"
    int localsize = 10+10*( (float) rand()/(float)RAND_MAX - .5),
        root = nprocs-1;

    int *localsizes=NULL;
    // create local data
    int *localdata = (int*) malloc( localsize*sizeof(int) );
    for (int i=0; i<localsize; i++)
        localdata[i] = procno+1;
    // we assume that each process has a value "localsize"
    // the root process collectes 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);
    if (procno==root) {
        printf("Local sizes: ");
        for (int i=0; i<nprocs; i++)
            printf("%d, ",localsizes[i]);
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

#### 3.16.14 Listing of code examples/mpi/c/transposeblock.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc,char **argv) {

#include "globalinit.c"

/*
 * Allocate matrix and transpose:
 * - one column per rank for regular
 * - one row per rank for transpose
 */
double *regular,*transpose;
regular = (double*) malloc( nprocs*sizeof(double) );
transpose = (double*) malloc( nprocs*sizeof(double) );
```

```
// each process has columnns m*nprocs -- m*(nprocs+1)
for (int ip=0; ip<nprocs; ip++)
    regular[ip] = procno*nprocs + ip;

/*
 * Each proc does a scatter
 */
#ifndef O
    // reference code:
    for (int iproc=0; iproc<nprocs; iproc++) {
        MPI_Scatter( regular,1,MPI_DOUBLE,
        &(transpose[iproc]),1,MPI_DOUBLE,
        iproc,comm);
    }
#else
    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);
#endif

/*
 * Check the result
 */
printf("[%d] :",procno);
for (int ip=0; ip<nprocs; ip++)
    printf(" %5.2f",transpose[ip]);
printf("\n");

MPI_Finalize();

return 0;
}
```

### 3.16.15 Listing of code examples/mpi/c/reducescatter.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

/*
 * Set up an array of which processes you will receive from
 */
int
```

### 3. MPI topic: Collectives

---

```
// 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;  
  
/*  
 * Initialize  
 */  
for (int i=0; i<nprocs; i++) {  
    i_recv_from_proc[i] = 0;  
}  
  
/*  
 * Generate array of "yes/no I recv from proc p",  
 * and condensed array of procs I receive from.  
 */  
nprocs_to_recv_from = 0;  
for (int iproc=0; iproc<nprocs; iproc++)  
    // pick random procs to receive from, not yourself.  
    if ( (float) rand()/(float)RAND_MAX < 2./nprocs && iproc!=procno ) {  
        i_recv_from_proc[iproc] = 1;  
        nprocs_to_recv_from++;  
    }  
procs_to_recv_from = (int*) malloc(nprocs_to_recv_from*sizeof(int));  
int count_procs_to_recv_from = 0;  
for (int iproc=0; iproc<nprocs; iproc++)  
    if ( i_recv_from_proc[iproc] )  
        procs_to_recv_from[count_procs_to_recv_from++] = iproc;  
ASSERT( count_procs_to_recv_from==nprocs_to_recv_from );  
  
/*  
 */  
printf("[%d] receiving from:",procno);  
for (int iproc=0; iproc<nprocs_to_recv_from; iproc++)  
    printf(" %3d",procs_to_recv_from[iproc]);  
printf(".\n");  
  
/*  
 * Now find how many procs will send to you  
 */  
MPI_Reduce_scatter_block  
    (i_recv_from_proc,&nprocs_to_send_to,1,MPI_INT,  
     MPI_SUM,comm);  
  
/*  
 * 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);

    printf("[%d] sending to:",procno);
    for (int iproc=0; iproc<nprocs_to_send_to; iproc++)
        printf(" %3d",procs_to_send_to[iproc]);
    printf(".\n");

    MPI_Finalize();
    return 0;
}

```

### 3.16.16 Listing of code examples/mpi/c/mvp2d.c

### 3.16.17 Listing of code examples/mpi/c/gatherv.c

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

    #include "globalinit.c"
    int localsize = 10+10*( (float) rand()/(float)RAND_MAX - .5),
        root = nprocs-1;

    int *localsizes=NULL,*offsets=NULL,*localdata=NULL,*alldata=NULL;
    // create local data
    localdata = (int*) malloc( localsize*sizeof(int) );
    for (int i=0; i<localsize; i++)
        localdata[i] = procno+1;
    // we assume that each process has an array "localdata"

```

### 3. MPI topic: Collectives

---

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

if (procno==root) {
    printf("Local sizes: ");
    for (int i=0; i<nprocs; i++)
        printf("%d, ",localsizes[i]);
    printf("\n");
}

// the root constructs the offsets array
if (procno==root) {
    int total_data = 0;
    for (int i=0; i<nprocs; i++) {
        offsets[i] = total_data;
        total_data += localsizes[i];
    }
    alldata = (int*) malloc( total_data*sizeof(int) );
}
// everyone contributes their data
MPI_Gatherv(localdata,localsize,MPI_INT,
            alldata,localsizes,offsets,MPI_INT,root,comm);

if (procno==root) {
    int p=0;
    printf("Collected:\n");
    for (int i=0; i<nprocs; i++) {
        int j;
        printf(" %d:",i);
        for (j=0; j<localsizes[i]-1; j++)
printf("%d,",alldata[p++]);
        j=localsizes[i]-1;
        printf("%d;\n",alldata[p++]);
    }
}

MPI_Finalize();
return 0;
}
```

#### 3.16.18 Listing of code examples/mpi/p/gatherv.py

```
import numpy as np
```

```
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

localsize = random.randint(2,10)
print("[%d] local size=%d" % (procid,localsize))
localdata = np.empty(localsize,dtype=np.int)
for i in range(localsize):
    localdata[i] = procid

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

### 3.16.19 Listing of code examples/mpi/c/allgatherv.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

    int my_count = procno+1;
    int *my_array = (int*) malloc(my_count*sizeof(int));
    for (int i=0; i<my_count; i++)
        my_array[i] = procno;
    int *recv_counts = (int*) malloc(nprocs*sizeof(int));
    int *recv_displs = (int*) malloc(nprocs*sizeof(int));

    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 );

if (procno==0) {
    for (int p=0; p<nprocs; p++)
        if (recv_counts[p]!=p+1)
printf("count[%d] should be %d, not %d\n",
    p,p+1,recv_counts[p]);
    int c = 0;
    for (int p=0; p<nprocs; p++)
        for (int q=0; q<=p; q++)
if (global_array[c++]!=p)
    printf("p=%d, q=%d should be %d, not %d\n",
p,q,p,global_array[c-1]);
}

MPI_Finalize();
return 0;
}
```

#### 3.16.20 Listing of code examples/mpi/p/allgatherv.py

```
import numpy as np
import random # random.randint(1,N), random.random()
import sys
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

mycount = procid+1
my_array = np.empty(mycount,dtype=np.float64)

for i in range(mycount):
    my_array[i] = procid
recv_counts = np.empty(nprocs,dtype=np.int)
recv_displs = np.empty(nprocs,dtype=np.int)

my_count = np.empty(1,dtype=np.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] )
```

```

# other syntax:
# comm.Allgatherv( [my_array,mycount,0,MPI.DOUBLE], [global_array,recv_counts,recv_displs,M

if procid==0:
    #print(procid,global_array)
    for p in range(nprocs):
        if recv_counts[p]!=p+1:
            print( "recv count[%d] should be %d, not %d" \
                  % (p,p+1,recv_counts[p]) )
c = 0
for p in range(nprocs):
    for q in range(p+1):
        if global_array[c]!=p:
            print( "p=%d, q=%d should be %d, not %d" \
                  % (p,q,p,global_array[c]) )
        c += 1
print "finished"

```

### 3.16.21 Listing of code examples/mpi/c/scan.c

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

    float myrandom = (float) rand()/(float)RAND_MAX,
          result;
    // 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.);

    MPI_Finalize();
    return 0;
}

```

### 3.16.22 Listing of code examples/mpi/p/scan.py

```

import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD

```

### 3. MPI topic: Collectives

---

```
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

mycontrib = 10+random.randint(1,nprocs)
myfirst = 0
mypartial = comm.scan(mycontrib)
print("[%d] local: %d, partial: %d" % (procid,mycontrib,mypartial))

sbuf = np.empty(1,dtype=np.int)
rbuf = np.empty(1,dtype=np.int)
sbuf[0] = mycontrib
comm.Scan(sbuf,rbuf)

print("[%d] numpy local: %d, partial: %d" % (procid,mycontrib,rbuf[0]))
```

#### 3.16.23 Listing of code examples/mpi/c/exscan.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

    int my_first=0,localsize;
    // localsize = ..... result of local computation ....
    localsize = 10+(int) (procno*(1+ (float) rand()/(float)RAND_MAX ));
    // find myfirst location based on the local sizes
    err = MPI_Exscan(&localsize,&my_first,
                      1,MPI_INT,MPI_SUM,comm); CHK(err);
    printf("[%d] localsize %d, first %d\n",procno,localsize,my_first);

    MPI_Finalize();
    return 0;
}
```

#### 3.16.24 Listing of code examples/mpi/p/exscan.py

```
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
```

```
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

localsize = 10+random.randint(1,nprocs)
myfirst = 0
mypartial = comm.exscan(localsize,0)

print("[%d] local: %d, partial: %d" % (procid,localsize,mypartial))
```

### 3.16.25 Listing of code examples/mpi/c/reductpositive.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

//int reduce_without_zero(int r,int n);
void reduce_without_zero(void *in,void *inout,int *len,MPI_Datatype *type) {
    // r is the already reduced value, n is the new value
    int n = *(int*)in, r = *(int*)inout;
    int m;
    if (n==0) { // new value is zero: keep r
        m = r;
    } else if (r==0) {
        m = n;
    } else if (n<r) { // new value is less but not zero: use n
        m = n;
    } else { // new value is more: use r
        m = r;
    };
#ifdef DEBUG
    printf("combine %d %d : %d\n",r,n,m);
#endif
    // return the new value
    *(int*)inout = m;
}

int main(int argc,char **argv) {

#include "globalinit.c"

    int m,mreduce=2000000000,ndata = 10, data[10] = {2,3,0,5,0,1,8,12,4,0},
        positive_minimum;
    if (nprocs>ndata) {
        printf("Too many procs for this example: at most %d\n",ndata);
        return 1;
    }

    for (int i=0; i<nprocs; i++)
```

```
    if (data[i]<mreduct && data[i]>0)
        mreduct = data[i];

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

    // check that the distributed result is the same as sequential
    if (mreduct!=positive_minimum)
        printf("[%d] Result %d should be %d\n",
               procno,positive_minimum,mreduct);
    else if (procno==0)
        printf("User-defined reduction successful: %d\n",positive_minimum);

    MPI_Finalize();
    return 0;
}
```

#### 3.16.26 Listing of code examples/mpi/p/reductpositive.py

```
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

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

    ndata = 10
    data = np.zeros(10,dtype=np.intc)
```

```
data[:] = [2,3,0,5,0,1,8,12,4,0]

if nprocs>ndata:
    print("Too many procs for this example: at most %d\n" %nndata)
    sys.exit(1)

#
# compute reduction by hand
#
mreduct=2000000000
for i in range(nprocs):
    if data[i]<mreduct and data[i]>0:
        mreduct = data[i]

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

#
# check that the distributed result is the same as sequential
#
if mreduct!=positive_minimum:
    print("[%d] Result %d should be %d\n" % \
          procid,positive_minimum,mreduct)
elif procid==0:
    print("User-defined reduction successful: %d\n" % positive_minimum)
```

### 3.16.27 Listing of code examples/mpic/transposeblock.c

### 3.16.28 Listing of code examples/mpi/c/ibarrierprobe.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc,char **argv) {

#include "globalinit.c"

/*
 * Pick one random process
 * that will do a send
 */
int sender,receiver;
if (procno==0)
    sender = rand()%nprocs;
MPI_Bcast (&sender,1,MPI_INT,0,comm);
```

### 3. MPI topic: Collectives

---

```
int i_do_send = sender==procno;

float data=1.;
MPI_Request send_request;
if (i_do_send) {
    /*
     * Pick a random process to send to,
     * not yourself.
     */
    int receiver = rand()%nprocs;
    while (receiver==procno) receiver = rand()%nprocs;
    printf("[%d] random send performed to %d\n", procno, receiver);
    //MPI_Isend(&data,1,MPI_FLOAT,receiver,0,comm,&send_request);
    MPI_Ssend(&data,1,MPI_FLOAT,receiver,0,comm);
}
/*
 * Everyone posts the non-block barrier
 * and gets a request to test/wait for
 */
MPI_Request barrier_request;
MPI_Ibarrier(comm,&barrier_request);

int step=0;
/*
 * Now everyone repeatedly tests the barrier
 * and probes for incoming message.
 * If the barrier completes, there are no
 * incoming message.
 */
MPI_Barrier(comm);
double tstart = MPI_Wtime();

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!
        int sender = status.MPI_SOURCE;
        MPI_Recv (&data,1,MPI_FLOAT,sender,0,comm,MPI_STATUS_IGNORE);
        printf("[%d] random receive from %d\n", procno, sender);
        }
    }
}
```

```
MPI_Barrier(comm);
double duration = MPI_Wtime()-tstart;
if (procno==0) printf("Probe loop: %e\n",duration);

printf("[%d] concluded after %d steps\n",procno,step);
MPI_Wait(&barrier_request,MPI_STATUS_IGNORE);
/* if (i_do_send) */
/* MPI_Wait(&send_request,MPI_STATUS_IGNORE); */

MPI_Finalize();
return 0;
}
```

### 3.16.29 Listing of code examples/mpi/c/findbarrier.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc,char **argv) {
    MPI_Comm comm;
    int nprocs,procid;

    MPI_Init(&argc,&argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_rank(comm,&procid);

    int mysleep;
    srand(procid*time(NULL));
    mysleep = nprocs * (rand()/(double)RAND_MAX);
    printf("[%d] working for %d seconds\n",procid,mysleep);
    sleep(mysleep);

    printf("[%d] finished, now posting barrier\n",procid);
    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,
```

### 3. MPI topic: Collectives

---

```
    comm, &flag, MPI_STATUS_IGNORE );
printf("[%d] going to work for another second\n", procid);
sleep(1);
global_finish++;
}
} while (1);

MPI_Wait(&final_barrier, MPI_STATUS_IGNORE);
printf("[%d] concluded %d work, total time %d\n",
procid, mysleep, global_finish);

MPI_Finalize();
return 0;
}
```

## Chapter 4

### MPI topic: Point-to-point

#### 4.1 Distributed computing and distributed data

One reason for using MPI is that sometimes you need to work on more data 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];
```

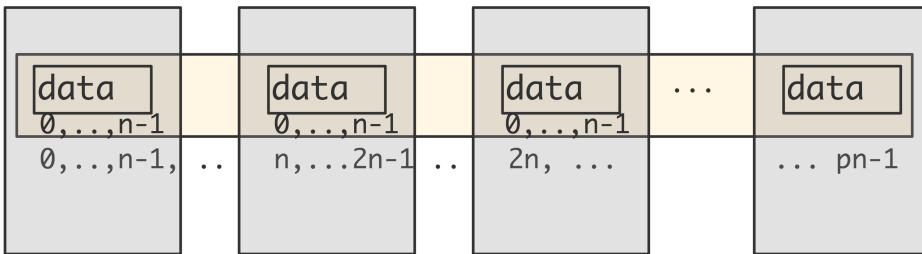


Figure 4.1: Local parts of a distributed array

We sometimes 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 4.1.** Implement a (very simple-minded) Fourier transform: if  $f$  is a function on the interval  $[0, 1]$ , then the  $n$ -th Fourier coefficient is

$$f_n \hat{=} \int_0^1 f(t) e^{-t/\pi} dt$$

which we approximate by

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

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

**Exercise 4.2.** 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?

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 4.3.** Argue that this strategy is not optimal. Can you come up with a better distribution? Load balancing is further discussed in HPSC-??.

**Exercise 4.4.** Implement an inner product routine: let  $x$  be a distributed vector of size  $N$  with elements  $x[i] = i$ , and compute  $x^t x$ . As before, the right value is  $(2N^3 + 3N^2 + N)/6$ .

Use the inner product value to scale to vector so that it has norm 1. Check that your computation is correct.

## 4.2 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 before (see figure 4.1), we give each processor a 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?)

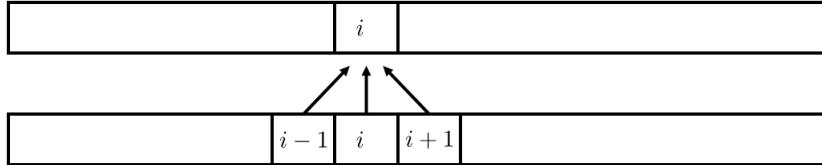


Figure 4.2: Three point averaging

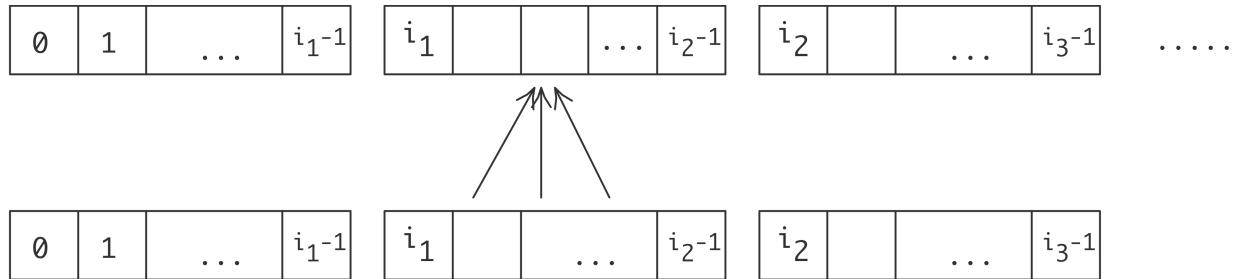


Figure 4.3: Three point averaging in parallel

We often talk about the *owner computes* model of parallel computing: each processor ‘owns’ certain data items, and it computes their value.

Now let’s investigate how processor  $p$  goes about computing  $y_i$  for the  $i$ -values it owns. Let’s assume that processor  $p$  also stores the values  $x_i$  for these same indices. Now, for many values it can compute

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

(figure 4.3). However, there is a problem with computing the first index  $i_p$ :

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

since  $x_{i_p}$  is not stored on processor  $p$ : it is stored on  $p - 1$  (figure 4.4). 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.

### 4.2.1 Send 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 means that process A executes the code

#### 4. MPI topic: Point-to-point

---

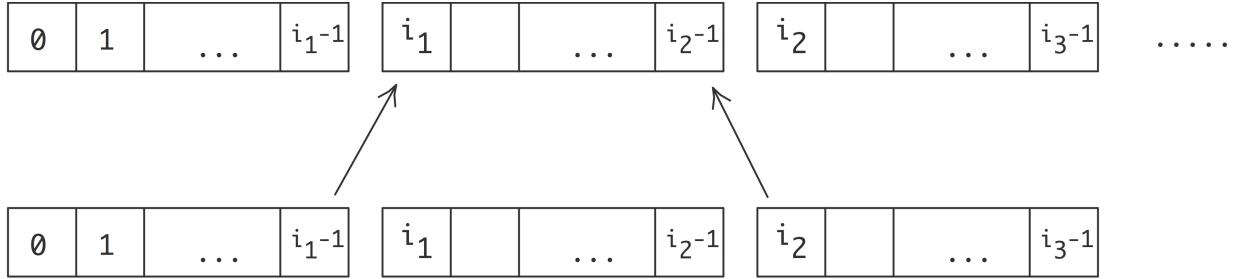


Figure 4.4: Three point averaging in parallel, case of edge points

```
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 ..... );
}
```

##### 4.2.1.1 Send call

The blocking send command is **MPI\_Send** (figure 4.1) . Examples:

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

For the source of this example, see section 4.6.2

The send call has the following elements.

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

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 4.2.2 for discussion.

#### 4.1 MPI\_Send

```
C:  
int MPI_Send(  
    const void* buf, int count, MPI_Datatype datatype,  
    int dest, int tag, MPI_Comm comm)  
  
Semantics:  
IN buf: initial address of send buffer (choice)  
IN count: number of elements in send buffer (non-negative integer)  
IN datatype: datatype of each send buffer element (handle)  
IN dest: rank of destination (integer)  
IN tag: message tag (integer)  
IN comm: communicator (handle)  
  
Fortran:  
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)  
TYPE(*), DIMENSION(..), INTENT(IN) :: buf  
INTEGER, INTENT(IN) :: count, dest, tag  
TYPE(MPI_Datatype), INTENT(IN) :: datatype  
TYPE(MPI_Comm), INTENT(IN) :: comm  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror  
  
Python native:  
MPI.Comm.send(self, obj, int dest, int tag=0)  
Python numpy:  
MPI.Comm.Send(self, buf, int dest, int tag=0)  
  
MPL:  
template<typename T >  
void mpl::communicator::send  
    ( const T scalar&, int dest, tag = tag(0) ) const  
T : scalar type  
    ( const T *buffer, const layout< T > &, int dest, tag = tag(0) ) const  
    ( iterT begin, iterT end, int dest, tag = tag(0) ) const  
begin : begin iterator  
end : end iterator
```

**Remark 5** The structure of the send call 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. However, even then, there will be a copy operation from the sender buffer to the receiver buffer, so there is no actual memory sharing going on.

Next, a message can have a *tag*. Many applications have each sender send only one message 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.

Unless otherwise needed, a tag value of zero is safe to use. Indeed, OO interfaces to MPI typically have this 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 12.1.2.

*MPL note.* 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';
}
```

For the source of this example, see section 4.6.3

Sending a buffer uses a general mechanism that will be discussed later:

```
// sendbuffer.cxx
std::vector<double> v(8);
mpi::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
```

For the source of this example, see section 4.6.4

#### 4.2.1.2 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);
```

## 4.2 MPI\_Recv

```
C:  
int MPI_Recv(  
    void* buf, int count, MPI_Datatype datatype,  
    int source, int tag, MPI_Comm comm, MPI_Status *status)  
  
Semantics:  
OUT buf: initial address of receive buffer (choice)  
IN count: number of elements in receive buffer (non-negative integer)  
IN datatype: datatype of each receive buffer element (handle)  
IN source: rank of source or MPI_ANY_SOURCE (integer)  
IN tag: message tag or MPI_ANY_TAG (integer)  
IN comm: communicator (handle)  
OUT status: status object (Status)  
  
Fortran:  
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)  
TYPE(*), DIMENSION(..) :: buf  
INTEGER, INTENT(IN) :: count, source, tag  
TYPE(MPI_Datatype), INTENT(IN) :: datatype  
TYPE(MPI_Comm), INTENT(IN) :: comm  
TYPE(MPI_Status) :: status  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror  
  
Python native:  
recvbuf = Comm.recv(self, buf=None, int source=ANY_SOURCE, int tag=ANY_TAG,  
                     Status status=None)  
Python numpy:  
Comm.Recv(self, buf, int source=ANY_SOURCE, int tag=ANY_TAG,  
          Status status=None)  
  
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
```

For the source of this example, see section 4.6.2

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

The *receive buffer* has the same buffer/count/data parameters as the send call. However, the *count* argument here indicates the maximum length of a message; the actual length of the received message can be determined from the status object; see section ??, and 4.4.2 for more detail.

Mirroring the target argument of the `MPI_Send` call, `MPI_Recv` has a *message source* argument. This can be either a specific process 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 ??.

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

In the syntax of the `MPI_Recv` command you saw 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 4.4.2 for more about the status object.

**Exercise 4.5.** 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  $\alpha, \beta$ ?

**Exercise 4.6.** 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?

### 4.2.2 Problems with blocking communication

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. For a receive call it is clear that the receiving code will wait until the data has actually come in, but for a send call this is more subtle.

You may be tempted to think that the send call puts the data somewhere in the network, and the sending code can progress, as in figure 4.5, left. But this ideal scenario is not realistic: it assumes that 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.)

#### 4.2.2.1 Deadlock

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

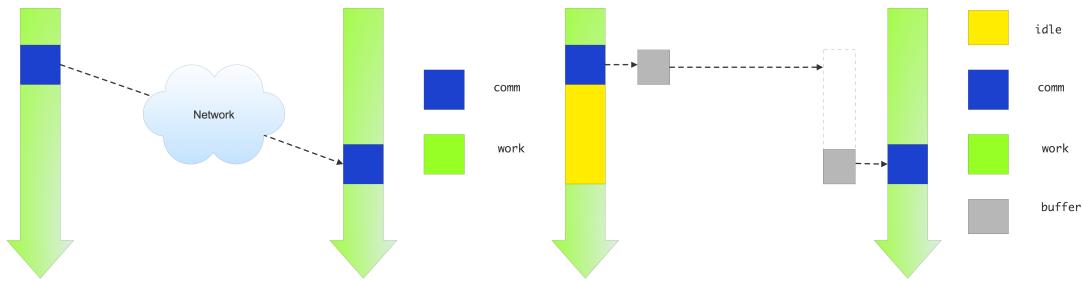


Figure 4.5: Illustration of an ideal (left) and actual (right) send-receive interaction

```
// 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*.

#### 4.2.2.2 Eager limit

If you reverse the send and receive call, you should get deadlock, but in practice that code will often work. The reason is that MPI implementations sometimes send small messages regardless of whether the receive has been posted. This relies on the availability of some amount of available buffer space. The size under which this behaviour is used is sometimes referred to as the *eager limit*.)

The following code is guaranteed to block, since a **MPI\_Recv** always blocks:

```
// recvblock.c
other = 1-procno;
MPI_Recv(&recvbuf, 1, MPI_INT, other, 0, comm, &status);
MPI_Send(&sendbuf, 1, MPI_INT, other, 0, comm);
printf("This statement will not be reached on %d\n", procno);
```

*For the source of this example, see section 4.6.5*

On the other hand, if we put the send call before the receive, code may not block for small messages that fall under 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.

```
// 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) {
```

#### 4. MPI topic: Point-to-point

---

```
    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);
}
```

For the source of this example, see section 4.6.6

```
// 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
```

For the source of this example, see section 4.6.7

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

For the source of this example, see section 4.6.8

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**.

```
// 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");

```

For the source of this example, see section 4.6.9

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);
}

```

The eager limit is implementation-specific. For instance, for *Intel mpi* there is a variable `I_MPI_EAGER_THRESHOLD`, for *mavapich2* it is `MV2_IBA_EAGER_THRESHOLD`, and for *OpenMPI* the `--mca` options `btl_openib_eager_limit` and `btl_openib_rndv_eager_limit`.

#### 4.2.2.3    *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 4.7.** (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 neighbour;
2. Accept the paper from your left neighbour.

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 neighbour.
2. If you are not the leftmost student, turn to your left and accept the paper from your left neighbour.

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

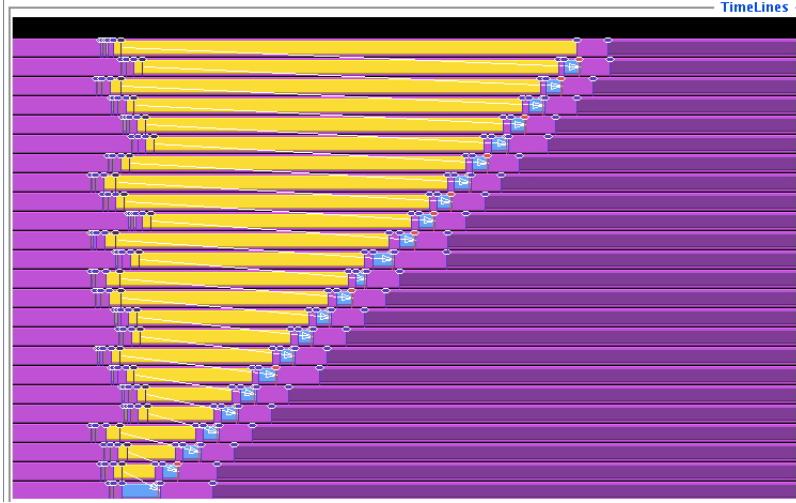


Figure 4.6: Trace of a simple send-recv code

one. This is illustrated in figure 4.6.

**Exercise 4.8.** Implement the above algorithm using `MPI_Send` and `MPI_Recv` calls. Run the code, and use TAU to reproduce the trace output of figure 4.6. If you don't have TAU, can you show this serialization behaviour using timings?

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

**Exercise 4.9.** 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 behaviour?

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

### 4.2.3 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 4.10.** Take the code of exercise 4.8 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.

**Remark 6** All quantities involved here are integers. Is it a good idea to use the integer datatype here?

**Remark 7** There is an `MPI_Scan` call that performs the same computation, but computationally more efficiently. Thus, this exercise only serves to illustrate the principle.

#### 4.2.4 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 up. You would then write

```
|| sendrecv( ....from... ...to... );
```

with the right choice of source and destination. For instance, to send data to your right neighbour:

```
|| 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. MPI allows for the following solution which makes the code slightly more homogeneous:

```
|| MPI_Comm_rank( .... &mytid );
|| if ( /* I am not the first processor */ )
||   predecessor = mytid-1;
|| else
||   predecessor = MPI_PROC_NULL;
|| if ( /* I am not the last processor */ )
||   successor = mytid+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 neighbour; the target value of `MPI_PROC_NULL` (figure 4.4) for the last processor means a ‘send to the null processor’: no actual send is done. The null processor value is also of use with the `MPI_Sendrecv` call; section 4.2.4

Likewise, receive 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.

*MPL note.* 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

## 4. MPI topic: Point-to-point

---

### 4.3 MPI\_Sendrecv

Semantics:

```
MPI_SENDREREV(
    sendbuf, sendcount, sendtype, dest, sendtag,
    recvbuf, recvcount, recvtype, source, recvtag,
    comm, status)
IN sendbuf: initial address of send buffer (choice)
IN sendcount: number of elements in send buffer (non-negative integer)
IN sendtype: type of elements in send buffer (handle)
IN dest: rank of destination (integer)
IN sendtag: send tag (integer)
OUT recvbuf: initial address of receive buffer (choice)
IN recvcount: number of elements in receive buffer (non-negative integer)
IN recvtype: type of elements in receive buffer (handle)
IN source: rank of source or MPI_ANY_SOURCE (integer)
IN recvtag: receive tag or MPI_ANY_TAG (integer)
IN comm: communicator (handle)
OUT status: status object (Status)
```

C:

```
int MPI_Sendrecv(
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    int dest, int sendtag,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int source, int recvtag,
    MPI_Comm comm, MPI_Status *status)
```

Fortran:

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
            recvcount, recvtype, source, recvtag, comm, status, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
                      recvtag
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

```
Sendrecv(self,
          sendbuf, int dest, int sendtag=0,
          recvbuf=None, int source=ANY_SOURCE, int recvtag=ANY_TAG,
          Status status=None)
```

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      Parallel Computing - r428
) const
```

**4.4 MPI\_PROC\_NULL**

```
C:
#include "mpi.h"
MPI_PROC_NULL

Fortran:
#include "mpif.h"
MPI_PROC_NULL

Python:
MPI.PROC_NULL = -1
```

possible routine prototypes, only the versions are available where the send and receive buffer are specified the same way.

```
// sendrecv.cxx
mpl::tag t0(0);
comm_world.sendrecv
( mydata, sendto, t0,
  leftdata, recvfrom, t0 );
```

**Exercise 4.11.** Revisit exercise 4.7 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 4.7.

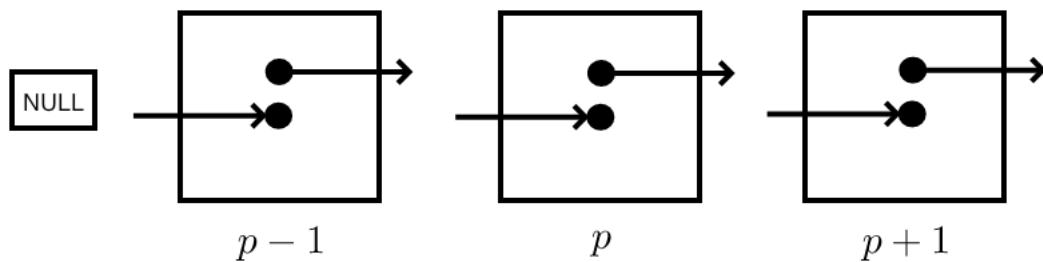


Figure 4.7: An MPI Sendrecv call

For the above three-point combination scheme you need to move data both left right, so you need two **MPI\_Sendrecv** calls; see figure 4.8.

**Exercise 4.12.** Implement the above three-point combination scheme using **MPI\_Sendrecv**; every processor only has a single number to send to its neighbour.

- 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.

#### 4. MPI topic: Point-to-point

---

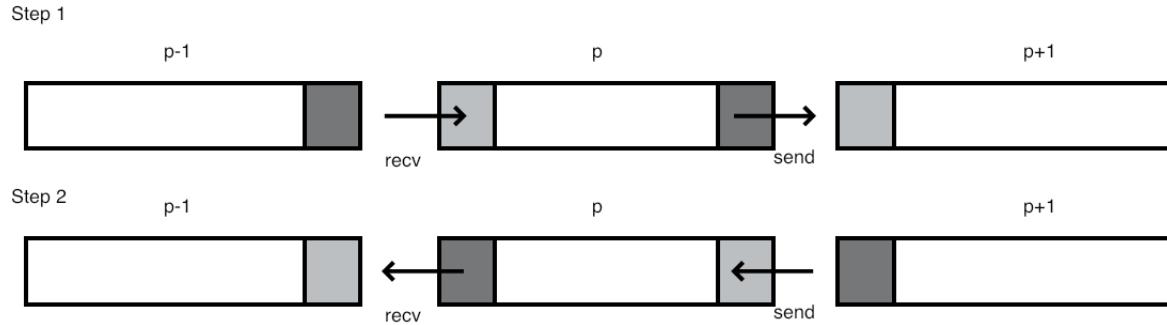


Figure 4.8: Two steps of send/recv to do a three-point combination

- 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.

If the send and receive buffer have the same size, the routine `MPI_Sendrecv_replace` (figure 4.5) will do an in-place replacement.

The following exercise lets you implement a sorting algorithm with the send-receive call<sup>1</sup>.

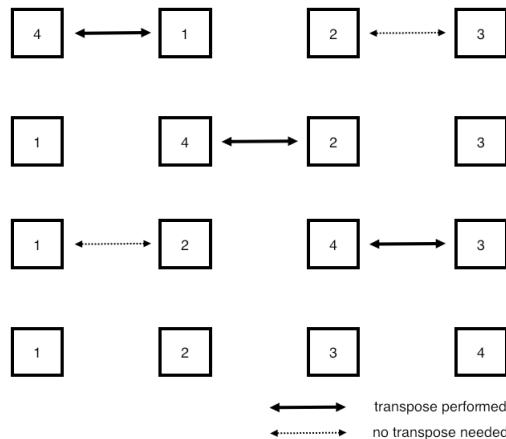


Figure 4.9: Odd-even transposition sort on 4 elements.

**Exercise 4.13.** 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 exchange 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

---

1. There is an `MPI_Compare_and_swap` call. Do not use that.

#### 4.5 MPI\_Sendrecv\_replace

```
C:
int MPI_Sendrecv_replace(
    void *buf, int count, MPI_Datatype datatype,
    int dest, int sendtag, int source, int recvtag,
    MPI_Comm comm, MPI_Status *status)

Fortran:
MPI_SENDREREV_REPLACE (
    BUF, COUNT, DATATYPE,
    DEST, SENDTAG, SOURCE, RECVTAG,
    COMM, STATUS, IERROR)
<type>      BUF(*)
INTEGER :: COUNT, DATATYPE, DEST, SENDTAG
INTEGER :: SOURCE, RECVTAG, COMM
INTEGER   STATUS(MPI_STATUS_SIZE), IERROR

Input/output parameter:
buf : Initial address of send and receive buffer (choice).

Input parameters:
count : Number of elements in send and receive buffer (integer).
datatype : Type of elements to send and receive (handle).
dest : Rank of destination (integer).
sendtag : Send message tag (integer).
source : Rank of source (integer).
recvtag : Receive message tag (integer).
comm : Communicator (handle).

Output parameters:
status : Status object (status).
IERROR : Fortran only: Error status (integer).
```

$2i + 1$  and  $2i + 2$  compare and swap. You need to repeat this  $P/2$  times, where  $P$  is the number of processors; see figure 4.9.

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.

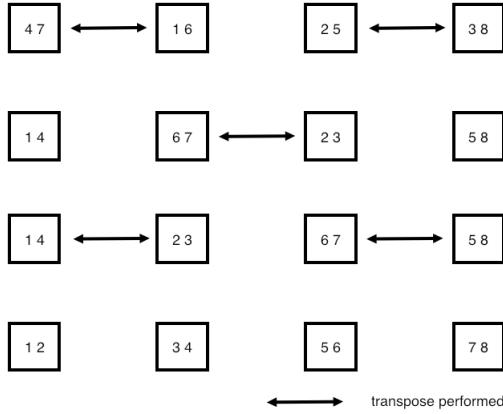


Figure 4.10: Odd-even transposition sort on 4 processes, holding 2 elements each.

**Exercise 4.14.** Extend this exercise to the case where each process hold an equal number of elements, more than 1. Consider figure 4.10 for inspiration. Is it coincidence that the algorithm takes the same number of steps as in the single scalar case?

#### 4.2.5 Message status

In section 4.2.1 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 4.3.1.) Often you specify `MPI_STATUS_IGNORE` for this argument: commonly you know what data is coming in and where it is coming from.

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 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 and allocate enough data when you find out how much data is being sent.

## 4.3 Non-blocking 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 4.2.

We note:

- The data is one-dimensional, and we have a linear ordering of the processors.
- The operation involves neighbouring data points, and we communicate with neighbouring 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 4.11 illustrates such a case, where processors

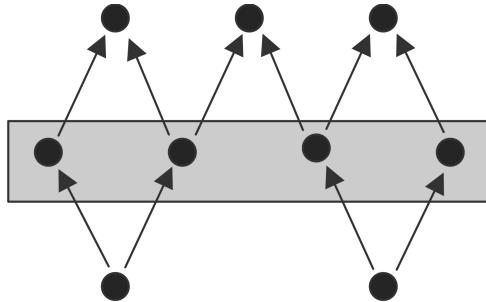


Figure 4.11: Processors with unbalanced send/receive patterns

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 behaviour observed above, or at least be inefficient since processors will be waiting for messages.

### 4.3.1 Non-blocking 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.

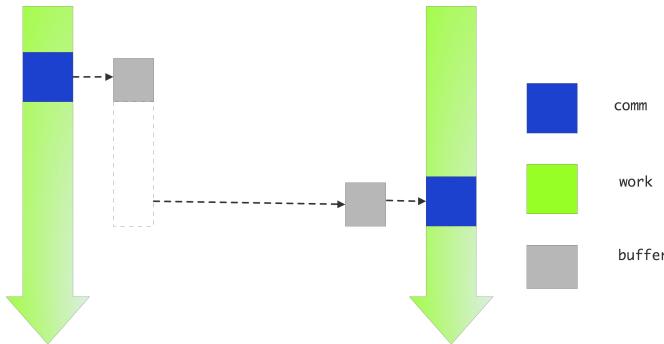


Figure 4.12: Non-blocking send

#### 4.6 MPI\_Isend

```
C:
int MPI_Isend(void *buf,
    int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm, MPI_Request *request)

Fortran:
the request parameter is an integer

Python:
request = MPI.Comm.Isend(self, buf, int dest, int tag=0)

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;
```

By contrast, the non-blocking calls **`MPI_Isend`** (figure 4.6) and **`MPI_Irecv`** (figure 4.7) 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 4.12.

```
// isendandirecv.c
double send_data = 1.;
MPI_Request request;
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);
```

For the source of this example, see section 4.6.10

```
double recv_data;
MPI_Request request;
MPI_Irecv
```

#### 4.7 MPI\_Irecv

```
C:  
int MPI_Irecv(  
    void* buf, int count, MPI_Datatype datatype,  
    int source, int tag, MPI_Comm comm, MPI_Request *request)  
  
Semantics:  
OUT buf: initial address of receive buffer (choice)  
IN count: number of elements in receive buffer (non-negative integer)  
IN datatype: datatype of each receive buffer element (handle)  
IN source: rank of source or MPI_ANY_SOURCE (integer)  
IN tag: message tag or MPI_ANY_TAG (integer)  
IN comm: communicator (handle)  
OUT request: request object (Request)  
  
Fortran:  
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)  
TYPE(*), DIMENSION(..) :: buf  
INTEGER, INTENT(IN) :: count, source, tag  
TYPE(MPI_Datatype), INTENT(IN) :: datatype  
TYPE(MPI_Comm), INTENT(IN) :: comm  
TYPE(MPI_Request), INTENT(out) :: request  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror  
  
Python native:  
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)  
  
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;
```

## 4. MPI topic: Point-to-point

---

### 4.8 MPI\_Request

```
C:  
MPI_Request request ;  
  
Fortran2008:  
Type(MPI_Request) :: request
```

### 4.9 MPI\_Wait

```
Semantics:  
MPI_Wait( request, status)  
  
INOUT request: request object (handle)  
OUT status: status objects (handle)  
  
C:  
int MPI_Wait(  
    MPI_Request *requests,  
    MPI_Status *statuses)  
  
Fortran:  
MPI_Wait( request, status, ierror)  
TYPE(MPI_Request), INTENT(INOUT) :: requests  
TYPE(MPI_Status), INTENT(OUT) :: statuses  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror  
  
Python:  
MPI.Request.Wait(type cls, request, status=None)  
  
Use MPI_STATUS_IGNORE to ignore
```

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

For the source of this example, see section 4.6.10

Issuing the **MPI\_Isend** / **MPI\_Irecv** call is sometimes referred to as *posting* a send/receive.

#### 4.3.2 Request completion: wait calls

From the definition of **MPI\_Isend** / **MPI\_Irecv**, you seen that non-blocking routine yields an **MPI\_Request** (figure 4.8) 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 4.9). 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 );
    }
} else {
    double recv=0.;
    MPI_Request request;
    MPI_Irecv( &recv, 1, MPI_DOUBLE, sender, 0, comm, &request );
    MPI_Wait(&request, MPI_STATUS_IGNORE);
}
```

*For the source of this example, see section 4.6.11*

Note that the request is passed by reference, so that the wait routine can free it.

*MPL note.* The request object, which is in the class `irequest`, is a function result, not a parameter pass by reference. The single wait call is a method of the `irequest` class.

```
double recv_data;
auto recv_request = mpl::irequest
    ( comm_world.irecv( recv_data, sender ) );
recv_request.wait();
```

*For the source of this example, see section 4.6.12*

You can not define the request variable:

```
// DOES NOT COMPILE mpl::irequest recv_request;
```

This means that the normal sequence of first declaring, and then filling in, the request variable is not possible.

The wait call always returns a `status` object; not assigning it means that the destructor is called on it.

To deal with multiple requests, there is a `irequest_pool` that requests can be pushed onto:

```
// 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 ) );
}
```

*For the source of this example, see section 4.6.13*

#### 4.3.2.1 More wait calls

Here we discuss in some detail the various wait calls. These are blocking; for the non-blocking versions see section 4.3.3.2.

#### 4.10 MPI\_Waitall

Semantics:

```
MPI_WAITALL( count, array_of_requests, array_of_statuses)
IN count: lists length (non-negative integer)
INOUT array_of_requests: array of requests (array of handles)
OUT array_of_statuses: array of status objects (array of Status)
```

C:

```
int MPI_Waitall(
    int count, MPI_Request array_of_requests[],
    MPI_Status array_of_statuses[])
```

Fortran:

```
MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
TYPE(MPI_Status) :: array_of_statuses(*)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

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

Use MPI\_STATUSES\_IGNORE to ignore

**4.3.2.1.1 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.

**4.3.2.1.2 Wait for all requests** **MPI\_Waitall** (figure 4.10) 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.

MPI has two types of routines for handling requests; we will start with the **MPI\_Wait...** routines. These calls are blocking: when you issue such a call, your execution will wait until the specified requests have been completed. Typically you use **MPI\_Waitall** to wait for all requests:

```
|| // start non-blocking communication
|| MPI_Isend( ... ); MPI_Irecv( ... );
|| // wait for the Isend/Irecv calls to finish in any order
|| MPI_Waitall( ... );
```

If you don't need the status objects, you can pass **MPI\_STATUSES\_IGNORE**.

**4.11 MPI\_Waitany**

Semantics:

```

int MPI_Waitany(
    int count, MPI_Request array_of_requests[], int *index,
    MPI_Status *status)

IN count: list length (non-negative integer)
INOUT array_of_requests: array of requests (array of handles)
OUT index: index of handle for operation that completed (integer)
OUT status: status object (Status)

C:
MPI_Waitany(count, array_of_requests, index, status, ierror)

Fortran:
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
INTEGER, INTENT(OUT) :: index
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Python:
MPI.Request.Waitany( requests,status=None )
class method, returns index

```

**Exercise 4.15.** Revisit exercise 4.10 and consider replacing the blocking calls by non-blocking ones. How far apart can you put the **MPI\_Isend** / **MPI\_Irecv** calls and the corresponding **MPI\_Waits**?

*Python note.* In python creating the array for the returned requests is somewhat tricky.

```

## irecvloop.py
requests = [ None ] * (2*nprocs)
sendbuffer = np.empty( nprocs, dtype=np.int )
recvbuffer = np.empty( nprocs, dtype=np.int )

for p in range(nprocs):
    left_p = (p-1) % nprocs
    right_p = (p+1) % nprocs
    requests[2*p] = comm.Isend\
        ( sendbuffer[p:p+1], dest=left_p )
    requests[2*p+1] = comm.Irecv\
        ( recvbuffer[p:p+1], source=right_p )
MPI.Request.Waitall(requests)

```

For the source of this example, see section 4.6.14

**4.3.2.1.3 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 4.11) and write:

```

|| for (p=0; p<nrequests; p++) {

```

#### 4. MPI topic: Point-to-point

---

```
    MPI_Irecv(buffer+index, /* ... */, requests+index);
}
for (p=0; p<nrequests; p++) {
    MPI_Waitany(nrequests,request_array,&index,&status);
    // operate on buffer[index]
}
```

Note that this routine takes a single status argument, passed by reference, and not an array of statuses!

*Fortran note.* The `index` parameter is the index in the array of requests, so it uses *1-based indexing*.

```
// irecv_source.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
```

For the source of this example, see section ??

```
// 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
```

For the source of this example, see section ??

```
// 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
```

For the source of this example, see section ??

*MPL note.* Instead of an array of requests, use an `irequest_pool` object, which acts like a vector of requests, meaning that you can `push_back` onto it. The `irequest_pool` class has methods `waitany`, `waitall`, `testany`, `testall`, `waitsome`, `testsome`.

```
auto [success,index] = recv_requests.waitany();
if (success) {
    auto recv_status = recv_requests.get_status(index);
```

For the source of this example, see section 4.6.13

You can not declare a pool of a fixed size and assign elements.

4.3.2.1.4 *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.

```
// irecv_source.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); //MPI_STATUS_IGNORE);
        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); CHK(ierr);
}
```

For the source of this example, see section 4.6.15

```
## irecvsource.py
if procid==nprocs-1:
    receive_buffer = np.empty(nprocs-1, dtype=np.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=np.int)
        mydata[0] = procid
        comm.Send([mydata, MPI_INT], dest=nprocs-1)
```

For the source of this example, see section 4.6.16

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 41.3.

**4.3.2.1.5 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.

Figure 4.13 shows the trace of a non-blocking execution using `MPI_Waitall`.

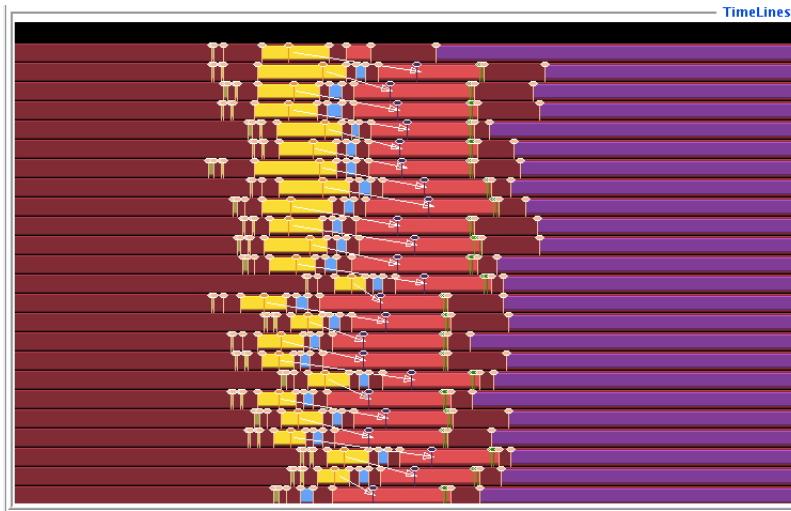


Figure 4.13: A trace of a nonblocking send between neighbouring processors

#### 4.3.2.2 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` and `MPI_Waitsome`.

**Exercise 4.16.** 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. (Hint: use `MPI_PROC_NULL` at the ends.)

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 non-blocking 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 12.5.

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 non-trivial combination of hardware, firmware, and MPI implementation.

**Exercise 4.17.** Take your code of exercise 4.16 and modify it to use latency hiding.

Operations that can be performed without needing data from neighbours should be performed in between the **MPI\_Isend** / **MPI\_Irecv** calls and the corresponding **MPI\_Wait** calls.

**Remark 8** You have now seen various send types: blocking, non-blocking, 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 non-blocking or synchronous message. The **MPI\_Recv** call can match any of the send routines you have seen so far (but not **MPI\_Sendrecv**), and conversely a message sent with **MPI\_Send** can be received by **MPI\_Irecv**.

#### 4.3.2.3 Buffer issues in non-blocking communication

While the use of non-blocking 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 non-blocking 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 non-blocking 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,
        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);

```

For the source of this example, see section 4.6.17

The last example we explicitly noted the possibility of overlapping computation and communication.

### 4.3.3 More about non-blocking communication

#### 4.3.3.1 Asynchronous progress

Above we saw that `Isend/Irecv` calls can overlap communication and computation. However, for this to happen we need for the MPI implementation to make *asynchronous progress*: the message needs to make its way through the network while the application is busy computing. This is not automatic. See section 12.5 for more discussion.

#### 4.3.3.2 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 non-blocking: they test for whether one or more requests have been fulfilled, but otherwise immediately return. This 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
    }
}

```

`MPI_Test` (figure 4.12) `MPI_Testany` (figure 4.13) `MPI_Testall` (figure 4.14)

**Exercise 4.18.** Read section HPSC-?? and give pseudo-code for the distributed sparse matrix-vector product using the above idiom for using `MPI_Test...` calls. Discuss

#### 4.12 MPI\_Test

C:

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

Input Parameters  
request : MPI request (handle)

Output Parameters  
flag : true if operation completed (logical)  
status : status object (Status). May be MPI\_STATUS\_IGNORE.

Python:  
request.Test()

#### 4.13 MPI\_Testany

C:

```
int MPI_Testany(
    int count, MPI_Request array_of_requests[],
    int *index, int *flag, MPI_Status *status)
```

Fortran:

```
MPI_Testany(count, array_of_requests, index, flag, status, ierror)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
INTEGER, INTENT(OUT) :: index
LOGICAL, INTENT(OUT) :: flag
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

#### 4.14 MPI\_Testall

Semantics:

```
MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)
IN countlists length (non-negative integer)
INOUT array_of_requestsarray of requests (array of handles)
OUT flag(logical)
OUT array_of_statusesarray of status objects (array of Status)
```

C:

```
int MPI_Testall(
    int count, MPI_Request array_of_requests[],
    int *flag, MPI_Status array_of_statuses[])
```

Fortran:

```
MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
LOGICAL, INTENT(OUT) :: flag
TYPE(MPI_Status) :: array_of_statuses(*)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

#### 4.15 MPI\_Request\_free

```
C:
#include <mpi.h>
int MPI_Request_free(MPI_Request *request)

Fortran2008:
USE mpi_f08
MPI_Request_free(request, ierror)
TYPE(MPI_Request), INTENT(INOUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran legacy:
INCLUDE 'mpif.h'
MPI_REQUEST_FREE(REQUEST, IERROR)
INTEGER REQUEST, IERROR

Input/output parameter:
request : Communication request (handle).
```

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.

##### 4.3.3.3 More about requests

Every non-blocking 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.

Correspondingly, calls to `MPI_Wait...` or `MPI_Test` free this object, setting the handle to `MPI_REQUEST_NULL`. 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 4.15), 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 non-blocking 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 [15].

You can inspect the status of a request without freeing the request object with `MPI_Request_get_status` (figure 4.16).

## 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

**4.16 MPI\_Request\_get\_status**

```
int MPI_Request_get_status(
    MPI_Request request,
    int *flag,
    MPI_Status *status
);
```

**4.17 MPI\_Probe**

```
int MPI_Probe
( int source, int tag, MPI_Comm comm,
  MPI_Status *status )
int MPI_Iprobe
( int source, int tag, MPI_Comm comm, int *flag,
  MPI_Status *status )

Input parameters:
source : source rank, or MPI_ANY_SOURCE (integer)
tag     : tag value or MPI_ANY_TAG (integer)
comm   : communicator (handle)

Output parameter:
flag    : True if a message with the specified
          source, tag, and communicator is available
status  : message status
```

use a ‘probe’ call.

The routine **MPI\_Probe** (figure 4.17) (and **MPI\_Iprobe**, for which see section 12.5), accepts a message, but does not copy the data. Instead, when probing tells you that there is a message, you can use **MPI\_Get\_count** 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);
}
```

*For the source of this example, see section 4.6.18*

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...

## 4. MPI topic: Point-to-point

---

### 4.18 MPI\_Mprobe

```
int MPI_Mprobe(int source, int tag, MPI_Comm comm,
               MPI_Message *message, MPI_Status *status)
```

Input Parameters:

source - rank of source or MPI\_ANY\_SOURCE (integer)  
tag - message tag or MPI\_ANY\_TAG (integer)  
comm - communicator (handle)

Output Parameters:

message - returned message (handle)  
status - status object (status)

### 4.19 MPI\_Mrecv

```
int MPI_Mrecv(void *buf, int count, MPI_Datatype type,
              MPI_Message *message, MPI_Status *status)
```

Input Parameters:

count - Number of elements to receive (nonnegative integer).  
datatype - Datatype of each send buffer element (handle).  
message - Message (handle).

Output Parameters:

buf - Initial address of receive buffer (choice).  
status - Status object (status).  
IERROR - Fortran only: Error status (integer).

```
MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)
           <type>   BUF(*)
INTEGER     COUNT, DATATYPE, MESSAGE
INTEGER     STATUS(MPI_STATUS_SIZE), IERROR
```

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 not message to receive.

This is solved by **MPI\_Mprobe** (figure 4.18), 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 4.19) call on that message through an object of type **MPI\_Message**.

#### 4.4.2 The Status object and wildcards

With some receive calls you know everything about the message in advance: its source, tag, and size. In other cases you want to leave some options open, and inspect the message for them after it was received. To do this, the receive call has a **MPI\_Status** (figure 4.20) parameter.

This status is a property of the actually received message, so **MPI\_Irecv** does not have a status parameter, but **MPI\_Wait** does.

The **MPI\_Status** object is a structure with the following freely accessible members:

**4.20 MPI\_Status**

```
C:
MPI_Status status;

Fortran:
integer :: status(MPI_STATUS_SIZE)

Fortran2008:
type(MPI_Status) :: recv_status

Python:
MPI.Status() # returns object
```

**4.21 MPI\_SOURCE**

Semantics:  
**MPI\_SOURCE** is the name of an integer field  
 in an **MPI\_STATUS** structure

```
C:
status.MPI_SOURCE // is int

F:
status_object%MPI_SOURCE ! is integer

Python:
status.Get_source() # returns int
```

**4.4.2.1 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;
```

The source of a message can be obtained as the **MPI\_SOURCE** (figure 4.21) member of the status structure.

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.

This code snippet is a simple model for this: all workers processes wait a random amount of time. For efficiency, the manager process accepts message from any source.

```
// anysource.c
if (procno==nprocs-1) {
```

#### 4. MPI topic: Point-to-point

---

```
|| int *recv_buffer;
|| MPI_Status status;

|| recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));

|| /*
||  * Messages can come in in any order, so use MPI_ANY_SOURCE
||  */
|| 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]);
|| }
|| } else {
|| /*
||  * Each rank waits an unpredictable amount of time
||  */
|| 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);
|| }
```

For the source of this example, see section [4.6.19](#)

In Fortran2008 style, the source is a member of the `Status` type.

```
|| // anysource.F90
|| 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
```

For the source of this example, see section [4.6.20](#)

In Fortran90 style, the source is an index in the `Status` array.

```
|| // anysource.F90
|| 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)
```

For the source of this example, see section [4.6.21](#)

MPL note. The `status` object can be queried:

```
|| int source = recv_status.source();
```

**4.22 MPI\_TAG**

```
C:
int status.MPI_TAG;

F:
integer :: MPI_TAG

Python:
status.Get_tag() # returns int
```

**4.23 MPI\_ERROR**

```
C:
int status.MPI_ERROR;

F:
integer :: MPI_ERROR

Python:
status.Get_error() # returns int
```

**4.4.2.2 Tag**

If a processor is expecting more than one message from a single other processor, message tags are used to distinguish between them. In that case, a value of `MPI_ANY_TAG` can be used, and the actual tag of a message can be retrieved as the `MPI_TAG` (figure 4.22) member in the status structure. See the section about `MPI_SOURCE` for how to use this.

*MPL note.* MPL differs from other APIs in its treatment of tags: a tag is not directly an integer, but an object of class `tag`.

```
// sendrecv.cxx
mpl::tag t0(0);
comm_world.sendrecv
( mydata, sendto, t0,
  leftdata, recvfrom, t0 );
```

The `tag` class has a couple of methods such as `mpl::tag::any()`.

**4.4.2.3 Error**

Any errors during the receive operation can be found as the `MPI_ERROR` (figure 4.23) member of the status structure.

**4.4.2.4 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 4.24) :

This may be necessary since the `count` argument to `MPI_Recv` is the buffer size, not an indication of the actually expected number of data items.

#### 4.24 MPI\_Get\_count

```
// C:
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
                  int *count)

! Fortran:
MPI_Get_count( INTEGER status(MPI_STATUS_SIZE), INTEGER datatype,
                INTEGER count, INTEGER ierror)

Python:
status.Get_count( Datatype datatype=BYTE )

MPL:
template<typename T>
int mpl::status::get_count () const

template<typename T>
int mpl::status::get_count (const layout<T> &l) const
```

#### 4.25 MPI\_Get\_elements

Synopsis  
Returns the number of basic elements in a datatype

```
int MPI_Get_elements
      (const MPI_Status *status, MPI_Datatype datatype, int *count)
int MPI_Get_elements_x
      (const MPI_Status *status, MPI_Datatype datatype, MPI_Count *count)

Input Parameters:
status : return status of receive operation (Status)
datatype : datatype used by receive operation (handle)

Output Parameters:
count : number of received basic elements (integer/MPI_Count)
```

#### Remarks.

- Unlike the above, this 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 5.3) this is not the same as the number of elementary datatype elements. For that, use **MPI\_Get\_elements** (figure 4.25) or **MPI\_Get\_elements\_x** which returns the number of basic elements.

*MPL note.* In MPL, the argument type is handled through templating:

```
// recvstatus.cxx
double pi=0;
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';
```

*For the source of this example, see section 4.6.22*

#### 4.4.2.5 Example: receiving from any source

Using the `MPI_ANY_SOURCE` specifier. We retrieve the actual source from the `MPI_Status` object through the `MPI_SOURCE` field.

```
// anysource.c
if (procno==nprocs-1) {
    int *recv_buffer;
    MPI_Status status;

    recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));

    /*
     * Messages can come in in any order, so use MPI_ANY_SOURCE
     */
    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]);
    }
} else {
    /*
     * Each rank waits an unpredictable amount of time
     */
    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);
}
```

For the source of this example, see section 4.6.19

```
## anysource.py
rstatus = MPI.Status()
comm.Recv(rbuf, source=MPI.ANY_SOURCE, status=rstatus)
print("Message came from %d" % rstatus.Get_source())
```

For the source of this example, see section 4.6.23

In sections 41.3 and 4.3.3.2 we explained the manager-worker model, and how it offers an opportunity for inspecting the `MPI_SOURCE` field of the `MPI_Status` object describing the data that was received.

#### 4.4.3 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 4.14.

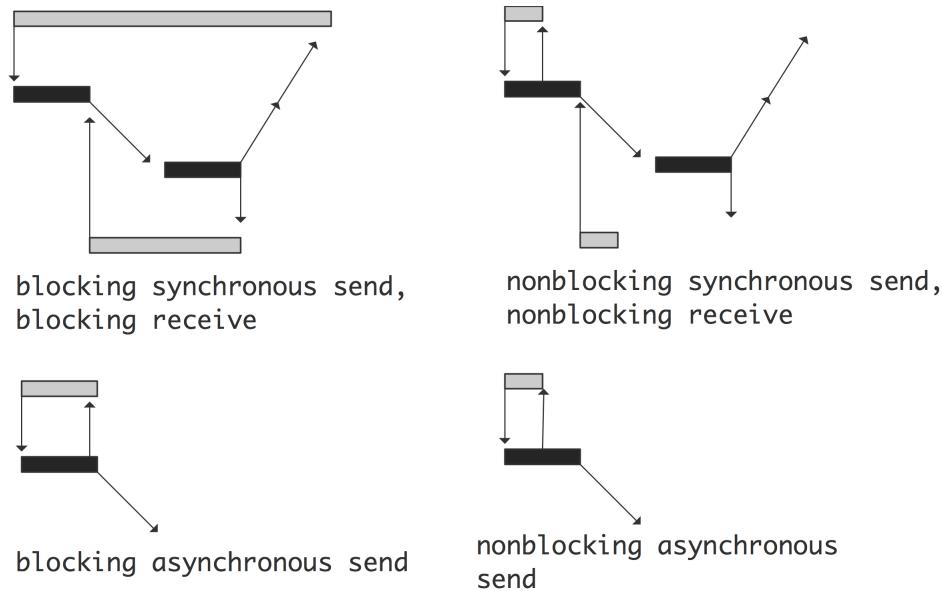


Figure 4.14: Blocking and synchronicity

MPI has a number of routines for synchronous communication, such as `MPI_Ssend`. Driving home the point that non-blocking and asynchronous are different concepts, there is a routine `MPI_Issend`, which is synchronous but non-blocking.

```
// 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");
```

For the source of this example, see section 4.6.9

#### 4.4.4 Persistent communication

An `MPI_Isend` or `MPI_Irecv` call has an `MPI_Request` parameter. This is an object that gets created in the send/recv call, and deleted in the wait call. You can imagine that this carries some overhead, and if the same communication is repeated many times you may want to avoid this overhead by reusing the request object.

To do this, MPI has *persistent communication*:

- You describe the communication with `MPI_Send_init`, which has the same calling sequence as `MPI_Isend`, or `MPI_Recv_init`, which has the same calling sequence as `MPI_Irecv`.
- The actual communication is performed by calling `MPI_Start`, for a single request, or `MPI_Startall` for an array or requests.

**4.26 MPI\_Send\_init**

```
C:
int MPI_Send_init(
    const void* buf, int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm, MPI_Request *request)

Fortran:
MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
INTEGER, INTENT(IN) :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Python:
MPI.Comm.Send_init(self, buf, int dest, int tag=0)

Semantics:
IN buf: initial address of send buffer (choice)
IN count: number of elements sent (non-negative integer)
IN datatype: type of each element (handle)
IN dest: rank of destination (integer)
IN tag: message tag (integer)
IN comm: communicator (handle)
OUT request: communication request (handle)
```

- Completion of the communication is confirmed with **MPI\_Wait** or similar routines as you have seen in the explanation of non-blocking communication.
- The wait call does not release the request object: that is done with **MPI\_Request\_free**.

The calls **MPI\_Send\_init** (figure 4.26) and **MPI\_Recv\_init** (figure 4.27) for creating a persistent communication have the same syntax as those for non-blocking sends and receives. The difference is that they do not start an actual communication, they only create the request object.

Given these request object, a communication (both send and receive) is then started with **MPI\_Start** (figure 4.28) for a single request or **MPI\_Startall** (figure 4.29) for multiple requests, given in an array.

These are equivalent to starting an **MPI\_Isend** or **MPI\_Irecv**; correspondingly, it is necessary to issue an **MPI\_Wait...** call (section 4.3.1) to determine their completion.

After a request object has been used, possibly multiple times, it can be freed; see 4.3.3.3.

In the following example a ping-pong is implemented with persistent communication.

```
// persist.c
if (procno==src) {
    MPI_Send_init(send, s, MPI_DOUBLE, tgt, 0, comm, requests+0);
    MPI_Recv_init(recv, s, MPI_DOUBLE, tgt, 0, comm, requests+1);
    printf("Size %d\n", s);
    t[cnt] = MPI_Wtime();
    for (int n=0; n<NEXPERIMENTS; n++) {
        fill_buffer(send, s, n);
    }
}
```

#### 4.27 MPI\_Recv\_init

```
C:  
int MPI_Recv_init(  
    void* buf, int count, MPI_Datatype datatype,  
    int source, int tag, MPI_Comm comm, MPI_Request *request)  
  
Fortran:  
MPI_Recv_init(buf, count, datatype, source, tag, comm, request,  
ierror)  
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf  
INTEGER, INTENT(IN) :: count, source, tag  
TYPE(MPI_Datatype), INTENT(IN) :: datatype  
TYPE(MPI_Comm), INTENT(IN) :: comm  
TYPE(MPI_Request), INTENT(OUT) :: request  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror  
  
Python:  
MPI.Comm.Recv_init(  
    self, buf, int source=ANY_SOURCE, int tag=ANY_TAG)  
  
Semantics:  
OUT buf: initial address of receive buffer (choice)  
IN count: number of elements received (non-negative integer)  
IN datatype: type of each element (handle)  
IN source: rank of source or MPI_ANY_SOURCE (integer)  
IN tag: message tag or MPI_ANY_TAG (integer)  
IN com: mcommunicator (handle)  
OUT request: communication request (handle)
```

#### 4.28 MPI\_Start

```
C:  
int MPI_Start (MPI_Request request)  
  
Fortran:  
MPI_Start(request, ierror)  
TYPE(MPI_Request), INTENT(INOUT) :: request  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror  
  
MPI_START(REQUEST, IERROR)  
INTEGER REQUESTS, IERROR  
  
Python:  
MPI.Prequest.Start(type cls, request)  
  
Semantics:  
INOUT request : request (handle)
```

**4.29 MPI\_Startall**

C:

```
int MPI_Startall(int count, MPI_Request array_of_requests[])
```

Fortran:

```
MPI_Startall(count, array_of_requests, ierror)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
```

Python:

```
MPI.Prequest.Startall(type cls, requests)
```

Semantics:

```
IN countlist length (non-negative integer)
INOUT array_of_requestsarray of requests (array of handle)
```

```

    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);
}
t[cnt] = MPI_Wtime()-t[cnt];
MPI_Request_free(requests+0); MPI_Request_free(requests+1);
} else if (procno==tgt) {
    for (int n=0; n<NEXPERIMENTS; n++) {
        MPI_Recv(recv,s,MPI_DOUBLE,src,0,comm,MPI_STATUS_IGNORE);
        MPI_Send(recv,s,MPI_DOUBLE,src,0,comm);
    }
}
```

For the source of this example, see section 4.6.24

```

## persist.py
sendbuf = np.ones(size,dtype=np.int)
recvbuf = np.ones(size,dtype=np.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)
```

For the source of this example, see section 4.6.25

As with ordinary send commands, there are the variants `MPI_Bsend_init`, `MPI_Ssend_init`, `MPI_Rsend_init`

#### 4.4.5 Buffered communication

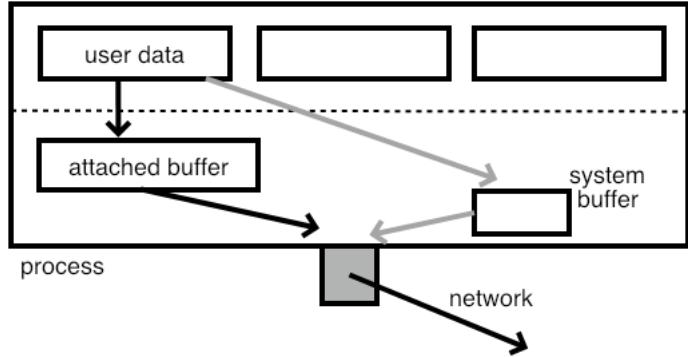


Figure 4.15: User communication routed through an attached buffer

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 4.15. If high-bandwidth memory is available, you could create your buffer there.
2. You use the `MPI_Bsend` call for sending, using otherwise normal send and receive buffers;
3. You detach the buffer when you're done with the buffered sends.

*MPL note.* Attaching a buffer is done through `bsend_buffer` and a support routine `bsend_size` helps in calculating the buffer size:

```
|| int size{ comm_world.bsend_size<decltype(x)>() };
|| mpl::bsend_buffer<> buff(size);
|| comm_world.bsend(x, 1);
```

*MPL note.*

```
|| void mpl::environment::buffer_attach (void *buff, int size);
|| std::pair< void *, int > mpl::environment::buffer_detach ();
```

##### 4.4.5.1 Bufferend send calls

Section 4.3 discusses non-blocking communication, where multiple sends will be under way. Since these sends are under way with possibly no receive having been posted, the send buffers can not be reused. It would be possible to reuse the send buffers if MPI had enough internal buffer space. For this, there is the *buffered send mode*, where you first give MPI internal buffer space; subsequently only a single send buffer is needed. The `MPI_Bsend` (figure 4.30) call is non-blocking.

**4.30 MPI\_Bsend**

```
C:
int MPI_Bsend
  (const void *buf, int count, MPI_Datatype datatype,
   int dest, int tag, MPI_Comm comm)

Input Parameters
buf : initial address of send buffer (choice)
count : number of elements in send buffer (nonnegative integer)
datatype : datatype of each send buffer element (handle)
dest : rank of destination (integer)
tag : message tag (integer)
comm : communicator (handle)
```

**4.31 MPI\_Buffer\_attach**

```
int MPI_Buffer_attach( void *buffer, int size );

Input arguments:
buffer : initial buffer address (choice)
size : buffer size, in bytes (integer)
```

There can be only one buffer per process, attached with **MPI\_Buffer\_attach** (figure 4.31). Its size should be enough for all outstanding **MPI\_Bsend** calls that are simultaneously outstanding, plus **MPI\_BSEND\_OVERHEAD**. You can compute the needed size of the buffer with **MPI\_Pack\_size**; see section 5.5.2.

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 buffer is detached with **MPI\_Buffer\_detach**:

```
|| int MPI_Buffer_detach(
    ||   void *buffer, int *size);
```

This returns the address and size of the buffer; the call blocks until all buffered messages have been delivered.

You can force delivery by

```
|| MPI_Buffer_detach( &b, &n );
|| MPI_Buffer_attach( b, n );
```

The asynchronous version is **MPI\_Ibsend**, the persistent (see section 4.4.4) call is **MPI\_Bsend\_init**.

#### 4.4.5.2 Persistent buffered communication

There is a persistent variant **MPI\_Bsend\_init** (figure 4.32) of buffered sends, as with regular sends (section 4.4.4).

#### 4.32 MPI\_Bsend\_init

```
Synopsis
int MPI_Bsend_init
    (const void *buf, int count, MPI_Datatype datatype,
     int dest, int tag, MPI_Comm comm,
     MPI_Request *request)

Input Parameters
buf : initial address of send buffer (choice)
count : number of elements sent (integer)
datatype : type of each element (handle)
dest : rank of destination (integer)
tag : message tag (integer)
comm : communicator (handle)

Output Parameters
request : communication request (handle)
```

## 4.5 Review questions

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

1. Describe a deadlock scenario involving three processors.
2. True or false: a message sent with **MPI\_Isend** from one processor can be received with an **MPI\_Recv** call on another processor.
3. True or false: a message sent with **MPI\_Send** from one processor can be received with an **MPI\_Irecv** on another processor.
4. Why does the **MPI\_Irecv** call not have an **MPI\_Status** argument?
5. 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?
6. What is the relation between the concepts of ‘origin’, ‘target’, ‘fence’, and ‘window’ in one-sided communication.
7. What are the three routines for one-sided data transfer?
8. 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
```

## 4.6 Sources used in this chapter

### 4.6.1 Listing of code header

### 4.6.2 Listing of code examples/mpi/c/sendandrecv.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

    MPI_Comm comm = MPI_COMM_WORLD;
    int nprocs, procno;

    MPI_Init(&argc,&argv);

    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_rank(comm,&procno);

    /*
     * We set up a single communication between
     * the first and last process
     */
    int sender,receiver;
    sender = 0; receiver = nprocs-1;

    if (procno==sender) {
        double send_data = 1.;
        MPI_Send
            ( /* send buffer/count/type: */ &send_data,1,MPI_DOUBLE,
        /* to: */ receiver, /* tag: */ 0,
        /* communicator: */ comm);
        printf("[%d] Send successfully concluded\n",procno);
    } else if (procno==receiver) {
        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);
        printf("[%d] Receive successfully concluded\n",procno);
    }

    MPI_Finalize();
    return 0;
}
```

### 4.6.3 Listing of code examples/mpi/mpl/sendscalar.cxx

```
#include <cstdlib>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;
#include <mpl/mpl.hpp>

int main() {
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    if (comm_world.size()<2)
        return EXIT_FAILURE;

    // send and receive a single floating point number
    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';
    }
    return EXIT_SUCCESS;
}
```

### 4.6.4 Listing of code examples/mpi/mpl/sendbuffer.cxx

```
#include <cstdlib>
#include <complex>
#include <iostream>
#include <vector>
#include <mpl/mpl.hpp>

int main() {
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    if (comm_world.size()<2)
        return EXIT_FAILURE;

    /*
     * To send a std::vector we declare a contiguous layout
     */
    std::vector<double> v(8);
    mpl::contiguous_layout<double> v_layout(v.size());

    // Initialize the data
    if (comm_world.rank()==0) {
        double init=0;
        for (double &x : v) {
            x=init;
        }
    }
}
```

```

        ++init;
    }

    /*
     * Send and report
     */
    comm_world.send(v.data(), v_layout, 1); // send to rank 1
    std::cout << "sent: ";
    for (double &x : v)
        std::cout << x << ' ';
    std::cout << '\n';

} else if (comm_world.rank() == 1) {

    /*
     * Receive data and report
     */
    comm_world.recv(v.data(), v_layout, 0); // receive from rank 0
    std::cout << "got : ";
    for (double &x : v)
        std::cout << x << ' ';
    std::cout << '\n';

}
return EXIT_SUCCESS;
}

```

#### 4.6.5 Listing of code examples/mpi/c/recvblock.c

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char **argv) {
    int other, recvbuf=2, sendbuf=3;

#include "globalinit.c"
    MPI_Status status;

    other = 1-procno;
    if (procno>1) goto skip;
    MPI_Recv(&recvbuf, 1, MPI_INT, other, 0, comm, &status);
    MPI_Send(&sendbuf, 1, MPI_INT, other, 0, comm);
    printf("This statement will not be reached on %d\n", procno);

skip:
    MPI_Finalize();
    return 0;
}

```

#### 4.6.6 Listing of code examples/mpi/c/sendblock.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

/** This program shows deadlocking behaviour
when two processes exchange data with blocking
send and receive calls.
Under a certain limit MPI_Send may actually not be
blocking; we loop, increasing the message size,
to find roughly the crossover point.

*/
int main(int argc,char **argv) {
    int *recvbuf, *sendbuf;
    MPI_Status status;

#include "globalinit.c"

/* we only use processors 0 and 1 */
int other;
if (procno>1) goto skip;
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);
}

skip:
MPI_Finalize();
return 0;
}
```

#### 4.6.7 Listing of code examples/mpi/f/sendblock.F90

Program SendBlock

#### 4. MPI topic: Point-to-point

---

```
#include "mpif.h"

integer :: other,size,status(MPI_STATUS_SIZE)
integer,dimension(:),allocatable :: sendbuf,recvbuf
#include "globalinit.F90"

if (mytid>1) goto 10
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

10 call MPI_Finalize(err)

end program SendBlock
```

#### 4.6.8 Listing of code examples/mpi/p/sendblock.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

if procid in [0,nprocs-1]:
    other = nprocs-1-procid
    size = 1
    while size<2000000000:
        sendbuf = np.empty(size, dtype=np.int)
        recvbuf = np.empty(size, dtype=np.int)
        comm.Send(sendbuf, dest=other)
        comm.Recv(recvbuf, source=other)
        if procid<other:
            print("Send did not block for",size)
        size *= 10
```

**4.6.9 Listing of code examples/mpi/c/ssendblock.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {
    int other, size, *recvbuf, *sendbuf;
    MPI_Status status;

#include "globalinit.c"

    if (procno>1) goto skip;
    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");
    free(sendbuf); free(recvbuf);

skip:
    MPI_Finalize();
    return 0;
}
```

**4.6.10 Listing of code examples/mpi/c/isendandirecv.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

/*
 * We set up a single communication between
 * the first and last process
 */
int sender,receiver;
sender = 0; receiver = nprocs-1;

if (procno==sender) {
    double send_data = 1.;
    MPI_Request request;
```

```

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);
  printf("[%d] Isend successfully concluded\n",procno);
} else if (procno==receiver) {
  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);
    printf("[%d] Ireceive successfully concluded\n",procno);
}

MPI_Finalize();
return 0;
}

```

#### 4.6.11 Listing of code examples/mpi/c/hangwait.c

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

double
mydata=procno;
int sender = nprocs-1;

if (procno==sender) {
  for (int p=0; p<nprocs-1; p++) {
    double send = 1.;
    MPI_Send( &send,1,MPI_DOUBLE,p,0,comm);
  }
} else {
  double recv=0.;
  MPI_Request request;
  MPI_Irecv( &recv,1,MPI_DOUBLE, sender,0,comm,&request);
  MPI_Wait(&request,MPI_STATUS_IGNORE);
}

```

```
    MPI_Finalize();
    return 0;
}
```

#### 4.6.12 Listing of code examples/mpi/mpf/isendandirecv.cxx

```
#include <cstdlib>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;
#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

    const mpl::communicator &comm_world=mpl::environment::comm_world();
    int
        nprocs = comm_world.size(),
        procno = comm_world.rank();
    if (comm_world.size()<2)
        return EXIT_FAILURE;

    int sender = 0,receiver = nprocs-1;

    if (procno==sender) {
        double send_data = 1.;
        mpl::irequest send_request
            ( comm_world.isend( send_data, receiver ) );
        send_request.wait();
        printf("[%d] Isend successfully concluded\n",procno);
    } else if (procno==receiver) {
        double recv_data;
        auto recv_request = mpl::irequest
            ( comm_world.irecv( recv_data, sender ) );
        recv_request.wait();
        printf("[%d] Ireceive successfully concluded\n",procno);
    }

    return 0;
}
```

#### 4.6.13 Listing of code examples/mpi/mpf/irecvsource.cxx

```
#include <cstdlib>
#include <unistd.h>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;
#include <vector>
```

```

using std::vector;

#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

    const mpl::communicator &comm_world=mpl::environment::comm_world();
    int
        nprocs = comm_world.size(),
        procno = comm_world.rank();
    if (comm_world.size()<2)
        return EXIT_FAILURE;

    // Initialize the random number generator
    srand((int)(procno*(double)RAND_MAX/nprocs));

    if (procno==nprocs-1) {
        mpl::irequest_pool recv_requests;
        vector<int> recv_buffer(nprocs-1);
        for (int p=0; p<nprocs-1; p++) {
            recv_requests.push( comm_world.irecv( recv_buffer[p], p ) );
        }
        printf("Outstanding request #=%d\n",recv_requests.size());
        for (int p=0; p<nprocs-1; p++) {
            auto [success,index] = recv_requests.waitany();
            if (success) {
                auto recv_status = recv_requests.get_status(index);
                int source = recv_status.source();
                if (index!=source)
                    printf("Mismatch index %lu vs source %d\n",index,source);
                printf("Message from %lu: %d\n",index,recv_buffer[index]);
            } else
                break;
        }
    } else {
        float randomfraction = (rand() / (double)RAND_MAX);
        int randomwait = (int) ( 2* nprocs * randomfraction );
        printf("process %d waits for %d\n",procno,randomwait);
        sleep(randomwait);
        comm_world.send( procno, nprocs-1 );
    }

    return 0;
}

```

#### 4.6.14 Listing of code examples/mpi/p/irecvloop.py

```

import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD

```

```
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

requests = [ None ] * (2*nprocs)
sendbuffer = np.empty( nprocs, dtype=np.int )
recvbuffer = np.empty( nprocs, dtype=np.int )

for p in range(nprocs):
    left_p = (p-1) % nprocs
    right_p = (p+1) % nprocs
    requests[2*p] = comm.Isend\
        ( sendbuffer[p:p+1], dest=left_p )
    requests[2*p+1] = comm.Irecv\
        ( recvbuffer[p:p+1], source=right_p )
MPI.Request.Waitall(requests)
```

#### 4.6.15 Listing of code examples/mpi/c/waitforany.c

#### 4.6.16 Listing of code examples/mpi/p/irecvsource.py

```
import numpy as np
import random
import time
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

if procid==nprocs-1:
    receive_buffer = np.empty(nprocs-1,dtype=np.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=np.int)
mydata[0] = procid
comm.Send([mydata,MPI.INT],dest=nprocs-1)
```

#### 4.6.17 Listing of code examples/mpi/c/irecvloop.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

    int recvbuf=2, sendbuf=3, other;
    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,
            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);

skip:
    MPI_Finalize();
    return 0;
}
```

#### 4.6.18 Listing of code examples/mpi/c/probe.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

    // Initialize the random number generator
```

```

srand((int)(procno*(double)RAND_MAX/nprocs));

int sender = 0, receiver = nprocs-1;
if (procno==receiver) {
    MPI_Status status;
    MPI_Probe(sender,0,comm,&status);
    int count;
    MPI_Get_count(&status,MPI_FLOAT,&count);
    printf("Receiving %d floats\n",count);
    float recv_buffer[count];
    MPI_Recv(recv_buffer,count,MPI_FLOAT, sender,0,comm,MPI_STATUS_IGNORE);
} else if (procno==sender) {
    float randomfraction = (rand() / (double)RAND_MAX);
    int buffer_size = (int) ( 10 * nprocs * randomfraction );
    printf("Sending %d floats\n",buffer_size);
    float buffer[buffer_size];
    ierr = MPI_Send(buffer,buffer_size,MPI_FLOAT, receiver,0,comm); CHK(ierr);
}
MPI_Finalize();
return 0;
}

```

#### 4.6.19 Listing of code examples/mpi/c/anysource.c

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

if (procno==nprocs-1) {
    int *recv_buffer;
    MPI_Status status;

    recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));

    /*
     * Messages can come in in any order, so use MPI_ANY_SOURCE
     */
    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]);
    }
} else {

```

```

/*
 * Each rank waits an unpredictable amount of time
 */
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);
}

MPI_Finalize();
return 0;
}

```

#### 4.6.20 Listing of code examples/mpi/f08/anysource.F90

```

Program AnySource

use mpi_f08

implicit none

integer,dimension(:),allocatable :: recv_buffer
Type(MPI_Status)  :: status
real :: randomvalue
integer :: randomint, sender

#include "globalinit.F90"

if (mytid.eq.ntids-1) then
    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
        print *, "Message from",sender
    end do
else
    call random_number(randomvalue)
    randomint = randomvalue*ntids
    call sleep(randomint)
    print *,mytid,"waits for",randomint
    call MPI_Send(p,1,MPI_INTEGER,ntids-1,0,comm)
end if

call MPI_Finalize(err)

end program AnySource

```

**4.6.21 Listing of code examples/mpi/f/anysource.F90**

```
Program AnySource

    implicit none

#include "mpif.h"

    integer,dimension(:),allocatable :: recv_buffer
    integer :: status(MPI_STATUS_SIZE)
    real :: randomvalue
    integer :: randomint, sender

#include "globalinit.F90"

    if (mytid.eq.ntids-1) then
        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)
            print *, "Message from", sender
        end do
    else
        call random_number(randomvalue)
        randomint = randomvalue*ntids
        call sleep(randomint)
        print *, mytid, "waits for", randomint
        call MPI_Send(p,1,MPI_INTEGER,ntids-1,0,comm,err)
    end if

    call MPI_Finalize(err)

end program AnySource
```

**4.6.22 Listing of code examples/mpi/mpf/recvstatus.cxx**

```
#include <cstdlib>
#include <complex>
#include <iostream>
#include <mpl/mpl.hpp>

int main() {
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    if (comm_world.size()<2)
        return EXIT_FAILURE;
    // send and receive a single floating point number
    if (comm_world.rank()==0) {
        double pi=3.14;
        comm_world.send(pi, 1); // send to rank 1
```

```
    std::cout << "sent: " << pi << '\n';
} else if (comm_world.rank()==1) {
    double pi=0;
    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';
}
return EXIT_SUCCESS;
}
```

#### 4.6.23 Listing of code examples/mpi/p/anysource.py

```
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

if procid==nprocs-1:
    rbuf = np.empty(1,dtype=np.float64)
    for p in range(procid):
        rstatus = MPI.Status()
        comm.Recv(rbuf,source=MPI.ANY_SOURCE,status=rstatus)
        print("Message came from %d" % rstatus.Get_source())
else:
    sbuf = np.empty(1,dtype=np.float64)
    sbuf = np.empty(1,dtype=np.float64)
    sbuf[0] = 1.
    comm.Send(sbuf,dest=nprocs-1)
```

#### 4.6.24 Listing of code examples/mpi/c/persist.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

void fill_buffer(double *send,int s,int n) {
    for (int i=0; i<s; i++)
        send[i] = n;
}
int chck_buffer(double *recv,int s,int n) {
    int r = 1;
```

```

        for (int i=0; i<s; i++)
            r = r && recv[i]==n;
        return r;
    }

    int main(int argc,char **argv) {
        #include "globalinit.c"

#define NEXPERIMENTS 10
#define NSIZES 6
        int src = 0,tgt = nprocs-1,maxsize=1000000;
        double t[NSIZES], send[maxsize],recv[maxsize];

        MPI_Request requests[2];

        // First ordinary communication
        for (int cnt=0,s=1; cnt<NSIZES && s<maxsize; s*=10,cnt++) {
            if (procno==src) {
                printf("Size %d\n",s);
                t[cnt] = MPI_Wtime();
                for (int n=0; n<NEXPERIMENTS; n++) {
                    MPI_Isend(send,s,MPI_DOUBLE,tgt,0,comm,requests+0);
                    MPI_Irecv(recv,s,MPI_DOUBLE,tgt,0,comm,requests+1);
                    MPI_Waitall(2,requests,MPI_STATUSES_IGNORE);
                }
                t[cnt] = MPI_Wtime()-t[cnt];
            } else if (procno==tgt) {
                for (int n=0; n<NEXPERIMENTS; n++) {
                    MPI_Recv(recv,s,MPI_DOUBLE,src,0,comm,MPI_STATUS_IGNORE);
                    MPI_Send(recv,s,MPI_DOUBLE,src,0,comm);
                }
            }
            if (procno==src) {
                for (int cnt=0,s=1; cnt<NSIZES; s*=10,cnt++) {
                    t[cnt] /= NEXPERIMENTS;
                    printf("Time for pingpong of size %d: %e\n",s,t[cnt]);
                }
            }
        }

        // Now persistent communication
        for (int cnt=0,s=1; cnt<NSIZES; s*=10,cnt++) {
            if (procno==src) {
                MPI_Send_init(send,s,MPI_DOUBLE,tgt,0,comm,requests+0);
                MPI_Recv_init(recv,s,MPI_DOUBLE,tgt,0,comm,requests+1);
                printf("Size %d\n",s);
                t[cnt] = MPI_Wtime();
                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);
}
t[cnt] = MPI_Wtime()-t[cnt];
MPI_Request_free(requests+0); MPI_Request_free(requests+1);
} else if (procno==tgt) {
    for (int n=0; n<NEXPERIMENTS; n++) {
        MPI_Recv(recv,s,MPI_DOUBLE,src,0,comm,MPI_STATUS_IGNORE);
        MPI_Send(recv,s,MPI_DOUBLE,src,0,comm);
    }
}
if (procno==src) {
    for (int cnt=0,s=1; cnt<NSIZES; s*=10,cnt++) {
        t[cnt] /= NEXPERIMENTS;
        printf("Time for persistent pingpong of size %d: %e\n",s,t[cnt]);
    }
}

MPI_Finalize();
return 0;
}

```

#### 4.6.25 Listing of code examples/mpi/p/persist.py

```

import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

nexperiments = 10
nsizes = 6
times = np.empty(nsizes,dtype=np.float64)
src = 0; tgt = nprocs-1

#
# ordinary communication
#
size = 1
requests = [ None ] * 2
if procid==src:
    print("Ordinary send/recv")
for isize in range(nsizes):
    sendbuf = np.ones(size,dtype=np.int)
    recvbuf = np.ones(size,dtype=np.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)
    size *= 10
if procid==src:
    print("Timings:",times)

#
# ordinary communication
#
size = 1
requests = [ None ] * 2
if procid==src:
    print("Persistent send/recv")
for isize in range(nsizes):
    sendbuf = np.ones(size,dtype=np.int)
    recvbuf = np.ones(size,dtype=np.int)
    if procid==src:
        print("Size:",size)
        requests[0] = comm.Send_init(sendbuf[0:size], dest=tgt)
        requests[1] = comm.Recv_init(recvbuf[0:size], source=tgt)
        times[isize] = MPI.Wtime()
        for n in range(nexperiments):
            MPI.Request.Startall(requests)
            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)
    size *= 10
if procid==src:
    print("Timings:",times)
```

## Chapter 5

### 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 heterogeneous data, or non-contiguous data.

- Communicating the real parts of an array of complex numbers means specifying every other number.
- Communicating a C structure or Fortran type with more than one type of element is not equivalent to sending an array of elements of a single type.

The datatypes you have dealt with so far are known as *elementary datatypes*; irregular objects are known as *derived datatypes*.

#### 5.1 Data type handling

Datatypes such as `MPI_INT` are values of the type `MPI_Datatype`. This type is handled differently in different languages.

In C, the `MPI_Datatype` type is defined through the pre-processor, allowing you to write:

```
// typefloat.c
#ifndef DOUBLE
    MPI_Datatype buffertype = MPI_DOUBLE;
    double send_buffer, recv_buffer;
#else
    MPI_Datatype buffertype = MPI_FLOAT;
    float send_buffer, recv_buffer;
#endif
    send_buffer = 1.1;
    if (procno==processA) {
        MPI_Send(&send_buffer, 1, buffertype,
            processB, 0,
            comm);
```

In Fortran before 2008, datatypes variables are stored in `Integer` variables. With the 2008 standard, datatypes are Fortran derived types:

```
|| Type(Datatype) :: mytype
```

*Python note.* 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.

*MPL note.* MPL routines are templated over the data type. 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.

## 5.2 Elementary data types

MPI has a number of elementary data types, corresponding to the simple data types of programming languages. The names are made to resemble the types of C and Fortran, for instance `MPI_FLOAT` and `MPI_DOUBLE` in C, versus `MPI_REAL` and `MPI_DOUBLE_PRECISION` in Fortran.

### 5.2.1 C/C++

Here we illustrate 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_INT, target, tag, comm);
```

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>
<code>MPI_Aint</code>	<code>MPI_AINT</code>

## 5. MPI topic: Data types

---

There is some, but not complete, support for C99 types.

See section 5.2.4 for `MPI_Aint` and more about byte counting.

### 5.2.2 Fortran

<code>MPI_CHARACTER</code>	Character(Len=1)
<code>MPI_INTEGER</code>	
<code>MPI_INTEGER1</code>	
<code>MPI_INTEGER2</code>	
<code>MPI_INTEGER4</code>	
<code>MPI_INTEGER8</code>	(common compiler extension; not standard)
<code>MPI_INTEGER16</code>	
<code>MPI_REAL</code>	
<code>MPI_DOUBLE_PRECISION</code>	
<code>MPI_REAL2</code>	
<code>MPI_REAL4</code>	
<code>MPI_REAL8</code>	
<code>MPI_COMPLEX</code>	
<code>MPI_DOUBLE_COMPLEX</code>	Complex(Kind=Kind(0.d0))
<code>MPI_LOGICAL</code>	
<code>MPI_PACKED</code>	

Not all these types 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)`.

The C type `MPI_Count` corresponds to an integer of type `MPI_COUNT_KIND`, used most prominently in ‘big data’ routines such as `MPI_Type_size_x` (section 5.4):

```
|| Integer(kind=MPI_COUNT_KIND) :: count
|| call MPI_Type_size_x(my_type, count)
```

Kind `MPI_ADDRESS_KIND` is used for `MPI_Aint` quantities, used in Remote Memory Access (RMA) windows; see section 8.3.1.

The `MPI_OFFSET_KIND` is used to define `MPI_Offset` quantities, used in file I/O; section 9.2.1.

#### 5.2.2.1 Fortran90 kind-defined types

If your Fortran code uses `KIND` to define scalar types with specified precision, these do not in general correspond to any predefined MPI datatypes. Hence the following routines exist to make MPI equivalences of Fortran scalar types: `MPI_Type_create_f90_integer` (figure 5.1) `MPI_Type_create_f90_real` (figure 5.2) `MPI_Type_create_f90_complex` (figure 5.3).

Examples:

**5.1 MPI\_Type\_create\_f90\_integer**

C:

```
int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype);
```

Fortran:

```
MPI_TYPE_CREATE_F90_INTEGER(INTEGER R, INTEGER NEWTYPE, INTEGER IERROR)
```

Input Parameter

r : Precision, in decimal digits (integer).

Output Parameters

newtype : New data type (handle).

IERROR : Fortran only: Error status (integer).

**5.2 MPI\_Type\_create\_f90\_real**

C:

```
int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
```

Fortran:

```
MPI_TYPE_CREATE_F90_REAL (P, R, NEWTYPE, IERROR)
```

Input Parameters

p : Precision, in decimal digits (integer).

r : Decimal exponent range (integer).

Output Parameters

newtype : New data type (handle).

IERROR : Fortran only: Error status (integer).

Either p or r, but not both, may be omitted from calls to  
SELECTED\_REAL\_KIND. Similarly, either argument to  
MPI\_Type\_create\_f90\_real may be set to MPI\_UNDEFINED.

**5.3 MPI\_Type\_create\_f90\_complex**

C:

```
int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
```

Fortran:

```
MPI_TYPE_CREATE_F90_REAL (P, R, NEWTYPE, IERROR)
```

Input Parameters

p : Precision, in decimal digits (integer).

r : Decimal exponent range (integer).

Output Parameters

newtype : New data type (handle).

IERROR : Fortran only: Error status (integer).

Either p or r, but not both, may be omitted from calls to  
SELECTED\_REAL\_KIND. Similarly, either argument to  
MPI\_Type\_create\_f90\_complex may be set to MPI\_UNDEFINED.

```

INTEGER ( KIND = SELECTED_INTEGER_KIND(15) ) , &
DIMENSION(100) :: array INTEGER :: root , integertype , error

CALL MPI_Type_create_f90_integer( 15 , integertype , error )
CALL MPI_Bcast ( array , 100 ,
& integertype , root ,
& MPI_COMM_WORLD , error )

REAL ( KIND = SELECTED_REAL_KIND(15 ,300) ) , &
DIMENSION(100) :: array
CALL MPI_Type_create_f90_real( 15 , 300 , realtype , error )

COMPLEX ( KIND = SELECTED_REAL_KIND(15 ,300) ) , &
DIMENSION(100) :: array
CALL MPI_Type_create_f90_complex( 15 , 300 , complextyppe , error )

```

### 5.2.3 Python

In python, all buffer data comes from *Numpy*.

mpi4py type	NumPy type
MPI.INT	np.intc
MPI.LONG	np.int
MPI.FLOAT	np.float32
MPI.DOUBLE	np.float64

Note that numpy native integers are C longs, and therefore 8 bytes. This is no problem if you send and receive contiguous buffers of integers, but it may trip you up in cases where the actual size of the integer matters, such as in derived types, or window definitions.

Examples:

```

## inttype.py
sizeofint = np.dtype('int').itemsize
print("Size of numpy int: {}".format(sizeofint))
##codesnippet npintc
data = np.empty(2*count,dtype=np.int)
for i in range(2*count):
    data[i] = i
vectortype = MPI.INT.Create_vector(count,1,2)
vectortype.Commit()
comm.Send( [data,1,vectortype], receiver )
elif procno==receiver:
    data = np.empty(count,dtype=np.int)
    comm.Recv( data, sender )
    print(data)

## Send strided buffer as C ints
if procno==sender:
##codesnippet npintc
sizeofint = np.dtype('intc').itemsize
print("Size of C int: {}".format(sizeofint))

```

For the source of this example, see section 5.7.2

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

For the source of this example, see section 5.7.3

#### 5.2.4 Byte addressing type

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 8.1.

Addresses have type `MPI_Aint`. The start of the address range is given in `MPI_BOTTOM`.

Variables of type `MPI_Aint` can be sent as `MPI_AINT`:

```
||MPI_Aint address;
||MPI_Send( address, 1, MPI_AINT, ... );
```

See section 8.5.2 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`.

See also the `MPI_Sizeof` (section 5.4.1) and `MPI_Get_address` routines.

##### 5.2.4.1 Fortran

The equivalent of `MPI_Aint` in Fortran is an integer of kind `MPI_ADDRESS_KIND`:

```
||integer(kind=MPI_ADDRESS_KIND) :: winsize
```

Fortran lacks a `sizeof` operator to query the sizes of datatypes. Since sometimes exact byte counts are necessary, for instance in one-sided communication, Fortran can use the (deprecated) `MPI_Sizeof` (figure 5.4) routine. See section 5.4.1 for details.

Example usage in `MPI_Win_create`:

```
||call MPI_Sizeof(windowdata,window_element_size,ierr)
||window_size = window_element_size*500
||call MPI_Win_create( windowdata,window_size,window_element_size,... )
```

##### 5.2.4.2 Python

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=np.int)
||win = MPI.Win.Create(window_data,intsize,comm=comm)
```

#### 5.4 MPI\_Sizeof

Synopsis:

```
MPI_Sizeof(v, size) - Returns the size, in bytes, of the given type
```

Fortran:

```
MPI_SIZEOF(V, SIZE, IERROR)
<type>      V
INTEGER      SIZE, IERROR
```

Input parameter:

```
V : A Fortran variable of numeric intrinsic type (choice).
```

Output parameter:

```
size : Size of machine representation of that type (integer).
```

```
ierror (Fortran) : Error status (integer); NOTE: NOT OPTIONAL!
```

### 5.3 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 elementary 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.

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 non-contiguous 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 accomodate multiple data types.

And you can combine these mechanisms to get irregularly spaced heterogeneous data, et cetera.

#### 5.3.1 Basic calls

The typical sequence of calls for creating a new datatype is as follows:

```

||| MPI_Datatype newtype;
||| MPI_Type_something( < oldtype specifications >, &newtype );
||| MPI_Type_commit( &newtype );
||| /* code that uses your new type */
||| MPI_Type_free( &newtype );

```

or

```

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.* 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)

```

For the source of this example, see section 5.7.4

*MPL note.* In MPL type creation routines are in the main namespace, templated over the datatypes.

```

// vector.cxx
vector<double>
source(stride*count),
target(count);
if (procno==sender) {
    mpl::strided_vector_layout<double> newvectortype(count,1,stride);
    comm_world.send
        (source.data(),newvectortype,the_other);
}

```

For the source of this example, see section 5.7.5

The commit call is part of the type creation, and freeing is done in the destructor.

### 5.3.1.1 Datatype objects

MPI derived data types are stored in variables of type **MPI\_Datatype** (figure 5.5) ; section 5.3.1.1.

### 5.3.1.2 Create calls

The **MPI\_Datatype** variable gets its value by a call to one of the following routines:

### 5.5 MPI\_Datatype

```
C:
MPI_Datatype datatype ;
```

Fortran:

```
Type(MPI_Datatype) datatype
```

### 5.6 MPI\_Type\_commit

```
C:
int MPI_Type_commit(MPI_Datatype *datatype)
```

Fortran:

```
MPI_Type_commit(datatype, ierror)
TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- **`MPI_Type_contiguous`** for contiguous blocks of data; section 5.3.2;
- **`MPI_Type_vector`** for regularly strided data; section 5.3.3;
- **`MPI_Type_create_subarray`** for subsets out higher dimensional block; section 5.3.4;
- **`MPI_Type_create_struct`** for heterogeneous irregular data; section 5.3.6;
- **`MPI_Type_indexed`** and **`MPI_Type_hindexed`** for irregularly strided data; section 5.3.5.

These calls take an existing type, whether elementary or also derived, and produce a new type.

#### 5.3.1.3 Commit and free

It is necessary to call **`MPI_Type_commit`** (figure 5.6) 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 5.7). (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.

#### 5.3.2 Contiguous type

The simplest derived type is the ‘contiguous’ type, constructed with **`MPI_Type_contiguous`** (figure 5.8).

A contiguous type describes an array of items of an elementary 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 5.1.

```
// contiguous.c
MPI_Datatype newvectortype;
if (procno==sender) {
    MPI_Type_contiguous(count,MPI_DOUBLE,&newvectortype);
```

**5.7 MPI\_Type\_free**

```
int MPI_Type_free (MPI_Datatype *datatype)
```

**5.8 MPI\_Type\_contiguous**

Semantics:

```
MPI_TYPE_CONTIGUOUS
  (count, oldtype, newtype)
IN count: replication count (non-negative integer)
IN oldtype: old datatype (handle)
OUT newtype: new datatype (handle)
```

C:

```
int MPI_Type_contiguous
  (int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Fortran:

```
MPI_Type_contiguous
  (count, oldtype, newtype, ierror)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: oldtype
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

```
Create_contiguous(self, int count)
```

```

  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);
}
```

For the source of this example, see section 5.7.6

```

// contiguous.F90
integer :: newvectortype
if (mytid==sender) then
  call MPI_Type_contiguous(count, MPI_DOUBLE_PRECISION, 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_DOUBLE_PRECISION, sender, 0, comm, &
    recv_status, err)
  call MPI_Get_count(recv_status, MPI_DOUBLE_PRECISION, recv_count, err)
  !ASSERT(count==recv_count);
end if
```



Figure 5.1: A contiguous datatype is built up out of elements of a constituent type

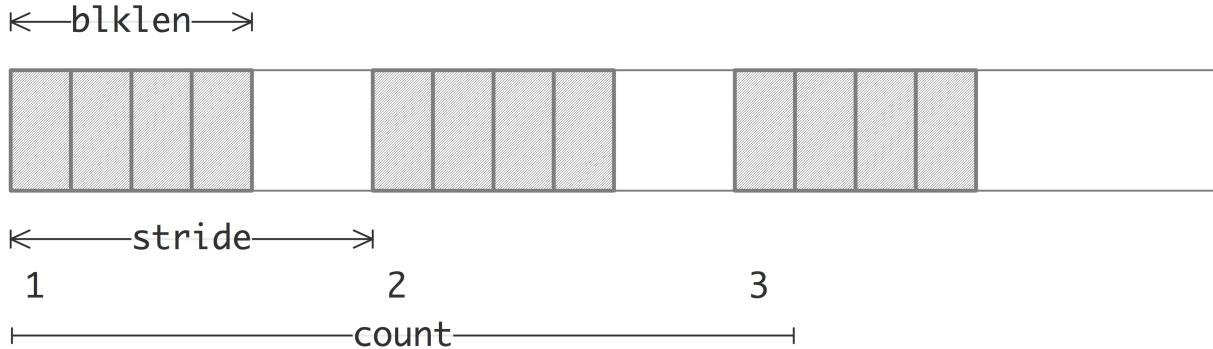


Figure 5.2: A vector datatype is built up out of strided blocks of elements of a constituent type

For the source of this example, see section [5.7.7](#)

```
## 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)
```

For the source of this example, see section [5.7.8](#)

*MPL note.* The MPL interface makes extensive use of *contiguous\_layout*, as it is the only way to declare a non-scalar buffer; see section [3.2.4](#).

### 5.3.3 Vector type

The simplest non-contiguous datatype is the ‘vector’ type, constructed with **`MPI_Type_vector`** (figure 5.9).

A vector type describes a series of blocks, all of equal size, spaced with a constant stride. This is illustrated in figure 5.2.

The vector datatype gives the first non-trivial 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

### 5.9 MPI\_Type\_vector

Semantics:

```
MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)
IN count: number of blocks (non-negative integer)
IN blocklength: number of elements in each block (non-negative integer)
IN stride: number of elements between start of each block (integer)
IN oldtype: old datatype (handle)
OUT newtype: new datatype (handle)
```

C:

```
int MPI_Type_vector
    (int count, int blocklength, int stride,
     MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Fortran:

```
MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
INTEGER, INTENT(IN) :: count, blocklength, stride
TYPE(MPI_Datatype), INTENT(IN) :: oldtype
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

```
MPI.Datatype.Create_vector(self, int count, int blocklength, int stride)
```

elements, either as a contiguous datatype, or as a contiguous buffer of an elementary type; see figure 5.3. In this case, 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.

```
// 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);
}
```

For the source of this example, see section 5.7.9

We illustrate Fortran2008:

```
// vector.F90
Type(MPI_Datatype) :: newvectortype
```

## 5. MPI topic: Data types

---

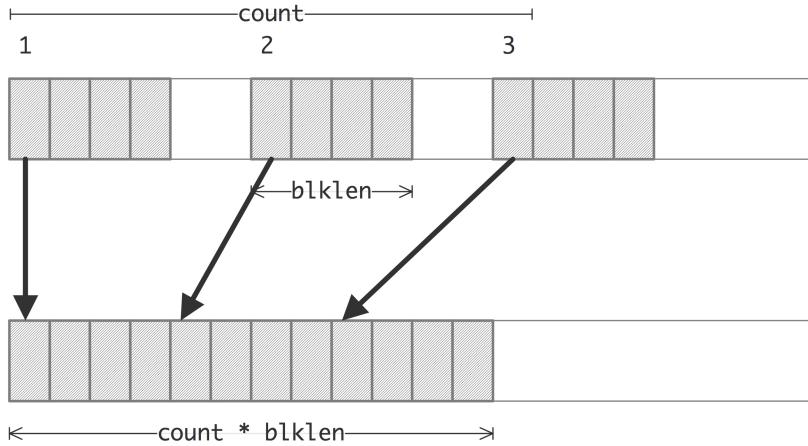


Figure 5.3: Sending a vector datatype and receiving it as elementary or contiguous

```

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

```

For the source of this example, see section [5.7.10](#)

In legacy mode, Fortran code stays the same except that the type is declared as `Integer`:

```

// vector.F90
integer :: newvectortype
integer :: recv_status(MPI_STATUS_SIZE),recv_count
call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
    newvectortype,err)
call MPI_Type_commit(newvectortype,err)

```

For the source of this example, see section [5.7.11](#)

**Python note.** The vector creation routine is a method of the `MPI.Datatype` class. For the general discussion, see section [5.3.1](#).

```

## 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)

```

For the source of this example, see section 5.7.4

**MPL note.** MPL uses the `strided_vector_layout` class:

```

|| // vector.cxx
|| vector<double>
||   source(stride*count),
||   target(count);
|| if (procno==sender) {
||   mpl::strided_vector_layout<double> newvectortype(count,1,stride);
||   comm_world.send
||     (source.data(),newvectortype,the_other);

```

For the source of this example, see section 5.7.5

It is possible to send containers by iterators

```

|| // 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

```

For the source of this example, see section 5.7.12

Non-contiguous 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);

```

Figure 5.4 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 5.1.** 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 (see section HPSC-??). 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 );

```

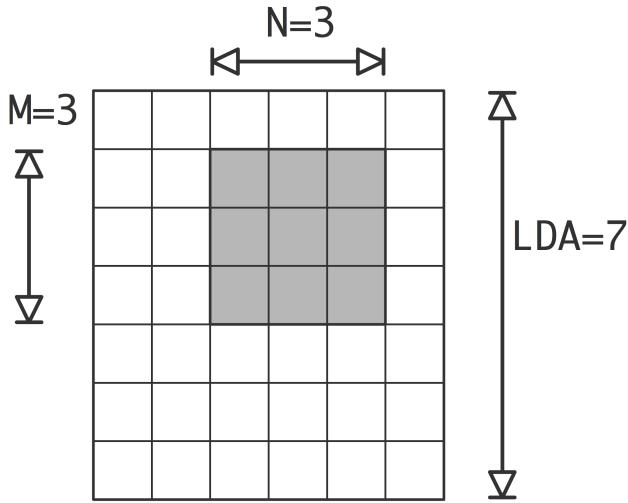


Figure 5.4: Memory layout of a row and column of a matrix in column-major storage

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, ... );
```

You can make this marginally more efficient (and harder to read) by replacing the index expression by  $mat + 1$ .

**Exercise 5.2.** 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 5.3.** Allocate a matrix on processor zero, using Fortran column-major storage. Using  $P$  sendrecv calls, distribute the rows of this matrix among the processors.

**Exercise 5.4.** 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  $x[i] = i$ .

**Exercise 5.5.** 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.

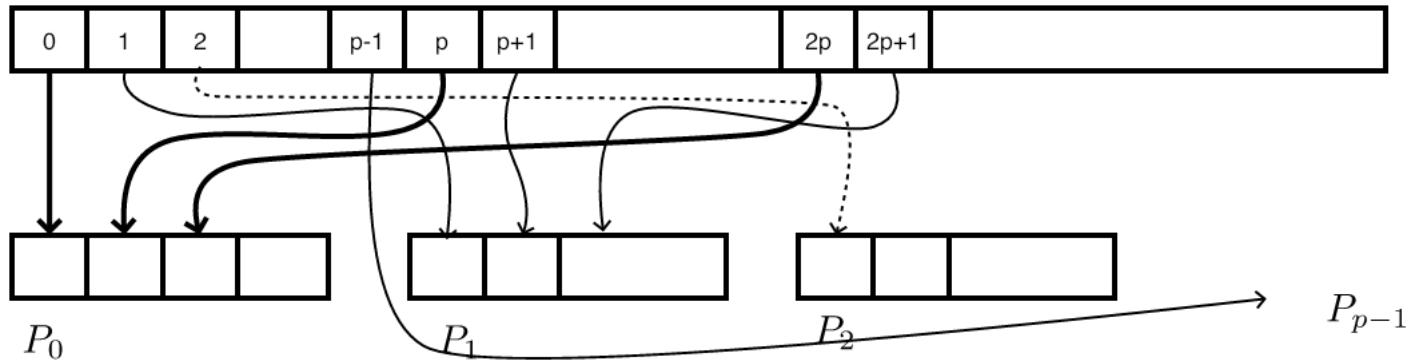


Figure 5.5: Send strided data from process zero to all others

### 5.3.4 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 5.10) This describes the dimensionality and extent of the array, and the starting point (the ‘upper left corner’) and extent of the subarray.

*MPL note.* 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 5.6.

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.

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
```

For the source of this example, see section 5.7.13

### 5.10 MPI\_Type\_create\_subarray

```
Semantics:
MPI_TYPE_CREATE_SUBARRAY(
    ndims, array_of_sizes, array_of_subsizes,
    array_of_starts, order, oldtype, newtype)
IN ndims: number of array dimensions (positive integer)
IN array_of_sizes: number of elements of type oldtype in each dimension
    of the full array (array of positive integers)
IN array_of_subsizes: number of elements of type oldtype in each
    dimension of the subarray (array of positive integers)
IN array_of_starts: starting coordinates of the subarray in each
    dimension (array of non-negative integers)
IN order: array storage order flag (state)
IN oldtype: array element datatype (handle)
OUT newtype: new datatype (handle)

C:
int MPI_Type_create_subarray(
    int ndims, const int array_of_sizes[],
    const int array_of_subsizes[], const int array_of_starts[],
    int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

Fortran:
MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
    array_of_starts, order, oldtype, newtype, ierror)
INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),
    array_of_subsizes(ndims), array_of_starts(ndims), order
TYPE(MPI_Datatype), INTENT(IN) :: oldtype
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Python:
MPI.Datatype.Create_subarray
    (self, sizes, subsizes, starts, int order=ORDER_C)
```

at least, since *MPI 3*.

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  $4 \times 4$  array in C order with the numbers  $0 \cdots 15$ , and send the  $[0, 1] \times [0 \cdots 4]$  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,&rowtype);
MPI_Type_create_subarray
(2,sizes,subsizes,starts,
 MPI_ORDER_FORTRAN,MPI_DOUBLE,&coltype);
```

For the source of this example, see section [5.7.14](#)

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
```

### 5.3.5 Indexed type

The indexed datatype, constructed with `MPI_Type_indexed` (figure 5.11) 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));
MPI_Datatype newvectortype;
if (procno==sender) {
    MPI_Type_indexed(count,blocklengths,displacements,MPI_INT,&newvectortype);
```

### 5.11 MPI\_Type\_indexed

Semantics:

```
count [in] number of blocks --
      also number of entries in indices and blocklens
blocklens [in] number of elements in each block
      (array of nonnegative integers)
indices [in] displacement of each block in multiples of old_type
      (array of integers)
old_type [in] old datatype (handle)
newtype [out] new datatype (handle)
```

C:

```
int MPI_Type_indexed(int count,
                     const int array_of_blocklengths[],
                     const int array_of_displacements[],
                     MPI_Datatype oldtype, MPI_Datatype
                     *newtype)
```

Fortran:

```
MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
                  oldtype, newtype, ierror)
INTEGER, INTENT(IN) :: count, array_of_blocklengths(count),
array_of_displacements(count)
TYPE(MPI_Datatype), INTENT(IN) :: oldtype
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

```
MPI.Datatype.Create_vector(self, blocklengths,displacements )
```

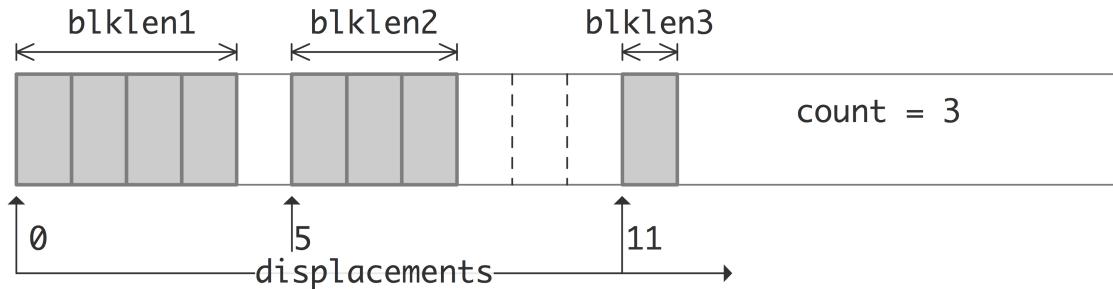


Figure 5.6: The elements of an MPI Indexed datatype

```

    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 the source of this example, see section [5.7.15](#)

For Fortran we show the legacy syntax for once:

```

// indexed.F90
integer :: newvectortype;
ALLOCATE(indices(count))
ALLOCATE(blocklengths(count))
ALLOCATE(source(totalcount))
ALLOCATE(targt(count))
if (mytid==sender) then
    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(targt,count,MPI_INT, sender,0,comm,&
        recv_status,err)
    call MPI_Get_count(recv_status,MPI_INT,recv_count,err)
    !     ASSERT(recv_count==count);
end if

```

For the source of this example, see section [5.7.16](#)

```

## indexed.py
displacements = np.empty(count,dtype=np.int)
blocklengths = np.empty(count,dtype=np.int)
source = np.empty(totalcount,dtype=np.float64)

```

## 5. MPI topic: Data types

---

```
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)
```

For the source of this example, see section [5.7.17](#)

**MPL note.** 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<intif (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);
}
```

For the source of this example, see section [5.7.18](#)

For the case where all block lengths are the same, use `indexed_block_layout`:

```
// indexedblock.cxx
const int count = 5;
mpl::contiguous_layout<int>
    fiveints(count);
mpl::indexed_block_layout<int
```

For the source of this example, see section [5.7.19](#)

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 5.12) 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

**5.12 MPI\_Type\_create\_hindexed\_block**

```
int MPI_Type_create_hindexed_block
    (int count, int blocklength,
     const MPI_Aint array_of_displacements[],
     MPI_Datatype oldtype, MPI_Datatype *newtype)

Input Parameters:
count : length of array of displacements (integer)
blocklength : size of block (integer)
array_of_displacements : array of displacements (array of integer)
oldtype : old datatype (handle)

Output Parameter:
newtype : new datatype (handle)
```

**5.13 MPI\_Get\_address**

```
C:
int MPI_Get_address
    (void *location,
     MPI_Aint *address
    );

Input Parameters:
location : location in caller memory (choice)

Output parameters:
address : address of location (address)
```

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 5.13) .

In C++ you can use this to send an `std::vector<T>`, that is, a vector object from the *C++ standard library*, if the component type is a pointer.

### 5.3.6 Struct type

The structure type, created with **MPI\_Type\_create\_struct** (figure 5.14) , 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,

```
|| 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 5.7.

**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.

### 5.14 MPI\_Type\_create\_struct

```
C:
int MPI_Type_create_struct(
    int count, int blocklengths[], MPI_Aint displacements[],
    MPI_Datatype types[], MPI_Datatype *newtype);
```

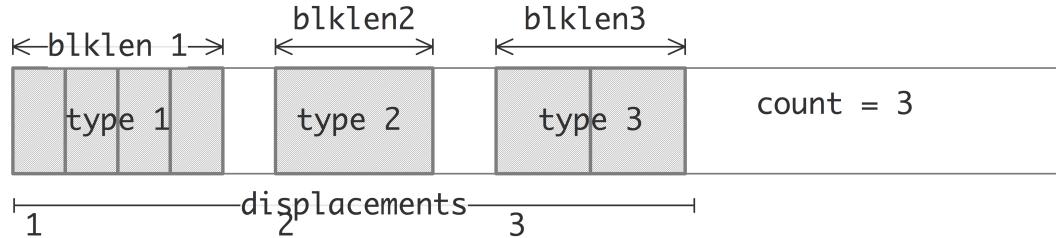


Figure 5.7: The elements of an MPI Struct datatype

**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 *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];

/*
 * 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);

```

For the source of this example, see section [5.7.20](#)

```

// 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)

```

## 5. MPI topic: Data types

---

```
|| end if  
|| call MPI_Type_free(newstructuretype)
```

For the source of this example, see section [5.7.21](#)

Note the displacement calculations in this example, which involve some not so elegant pointer arithmetic. (Alternatively, you could use `MPI_Get_address`, which is the only way address calculations can be done in Fortran.)

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 memory<sup>1</sup>.

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.* 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

```
|| int i; float x; double z;  
|| heterogeneous_layout l( i,x,z );  
|| comm.send( l /* sort of */, receiver );
```

More complicated data than scalars takes more work:

```
// struct.cxx  
char c;  
vector<double> x(2);  
int i;  
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 );  
}
```

For the source of this example, see section [5.7.22](#)

---

1. Homework question: what does the language standard say about this?

### 5.15 MPI\_Type\_match\_size

Synopsis:

```
int MPI_Type_match_size
    (int typeclass, int size, MPI_Datatype *datatype)
```

Input Parameters

typeclass : generic type specifier (integer)  
size : size, in bytes, of representation (integer)

Output Parameters

datatype : datatype with correct type, size (handle)

Notes

typeclass is one of:  
 - MPI\_TYPECLASS\_REAL,  
 - MPI\_TYPECLASS\_INTEGER and  
 - MPI\_TYPECLASS\_COMPLEX.

Here, the *absolute* indicates the lack of an implicit buffer: the layout is absolute rather than a relative description. Note also the *make\_absolute*.

## 5.4 Type size

### 5.4.1 Matching MPI and language type sizes

The size of a datatype is not always statically known, for instance if the Fortran KIND keyword is used. The translation of datatypes in the source language can be translated to MPI types with [MPI\\_Type\\_match\\_size](#) (figure 5.15) where the *typeclass* argument is one of `MPI_TYPECLASS_REAL`, `MPI_TYPECLASS_INTEGER`, `MPI_TYPECLASS_COMPLEX`.

```
// 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);
MPI_Type_size(mpi_x10,&s10);
```

For the source of this example, see section 5.7.23

In Fortran, the size of the datatype in the language can be obtained with [MPI\\_Szef](#) (note the non-optional error parameter!). This routine is deprecated in MPI-4: use of `storage_size` and/or `c_sizeof` is recommended.

```
// matchkind.F90
call MPI_Szef(x10,s10,ierr)
call MPI_Type_match_size(MPI_TYPECLASS_REAL,s10,mpi_x10)
```

### 5.16 MPI\_Type\_size

Semantics:

```
int MPI_Type_size(
    MPI_Datatype datatype,
    int *size
);

datatype: [in] datatype to get information on (handle)
size: [out] datatype size in bytes
```

```
|| call MPI_Type_size(mpi_x10,s10)
|| print *, "10 positions supported, MPI type size is",s10
```

For the source of this example, see section [5.7.23](#)

The space that MPI takes for a structure type can be queried in a variety of ways. First of all **MPI\_Type\_size** (figure 5.16) 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) );
MPI_Type_free(&newtype);
```

For the source of this example, see section [5.7.24](#)

#### 5.4.2 Extent and true extent

On the other hand, the *datatype extent*, measured with **MPI\_Type\_get\_extent** (figure 5.17) is strictly the distance from the first to the last data item of the type, that is, with counting the gaps in the type.

```
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);
```

For the source of this example, see section [5.7.25](#)

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;
MPI_Type_extent(newstructuretype,&typesize);
assert( typesize==size_of_struct );
```

For the source of this example, see section [5.7.20](#)

See section [5.3.6](#) for the code defining the structure type.

**5.17 MPI\_Type\_get\_true\_extent**

Semantics:

```
MPI_Type_get_extent (datatype,lb,extent)
MPI_Type_get_true_extent (datatype,true_lb,true_extent)
MPI_Type_get_true_extent_x (datatype,true_lb,true_extent)
```

Input argument:

datatype: Data type for which information is wanted (handle).

Output arguments:

```
lb / true_lb: (True) lower bound of data type (integer).
extent / true_extent: (True) extent of data type (integer).
```

C:

```
int MPI_Type_get_extent (
    MPI_Datatype datatype,
    MPI_Aint *true_lb, MPI_Aint *true_extent)
int MPI_Type_get_true_extent (
    MPI_Datatype datatype,
    MPI_Aint *true_lb, MPI_Aint *true_extent)
int MPI_Type_get_true_extent_x(
    MPI_Datatype datatype,
    MPI_Count *true_lb, MPI_Count *true_extent)
```

Fortran

```
MPI_TYPE_GET_TRUE_EXTENT (DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
    INTEGER      DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
MPI_TYPE_GET_TRUE_EXTENT_X (DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
    INTEGER      DATATYPE, IERROR
    INTEGER(KIND=MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT
```

**Remark 9** Routine ***MPI\_Type\_get\_extent*** replaces deprecated functions ***MPI\_Type\_extent***, ***MPI\_Type\_lb***, ***MPI\_Type\_ub***.

The subarray *datatype* need not start at the first element of the buffer, so the extent is an overstatement of how much data is involved. The routine ***MPI\_Type\_get\_true\_extent*** (figure 5.17) returns the lower bound, indicating where the data starts, and the extent from that point.

```
// trueextent.c
int sender = 0, receiver = 1, the_other = 1-procno,
count = 4;
int sizes[2] = {4,6}, subsizes[2] = {2,3}, starts[2] = {1,2};
MPI_Datatype subarraytype;
if (procno==sender) {
    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_extent(subarraytype,&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) *
        ( (starts[0]+subsizes[0]-1)*sizes[1] + starts[1]+subsizes[1] )
        - comp_lb;
//      ASSERT(extent==true_lb+extent);
ASSERT(true_lb==comp_lb);
ASSERT(true_extent==comp_extent);

MPI_Send(source,1,subarraytype,the_other,0,comm);
MPI_Type_free(&subarraytype);

```

For the source of this example, see section [5.7.26](#)

There is also a ‘big data’ routine [\*\*MPI\\_Type\\_get\\_true\\_extent\\_x\*\*](#).

#### 5.4.3 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.

Consider for instance figure [5.5](#) and exercise [5.4](#). There, 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!

The technicality on which the solution hinges is that you can ‘resize’ a type to give it a different extent, while not affecting how much data there actually is in it.

Let’s consider an example where each process makes a buffer of integers that will be interleaved in a gather call:

```

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 );
```

```
0 1879048192 1100361260 3 3 0 6 0 0 9 1 198654
```

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 );
```

*For the source of this example, see section 5.7.27*

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
```

#### 5.4.3.1 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);
```

*For the source of this example, see section 5.7.28*

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:

## 5. MPI topic: Data types

---

```

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);

```

For the source of this example, see section [5.7.28](#)

### 5.4.3.2 Example: transpose

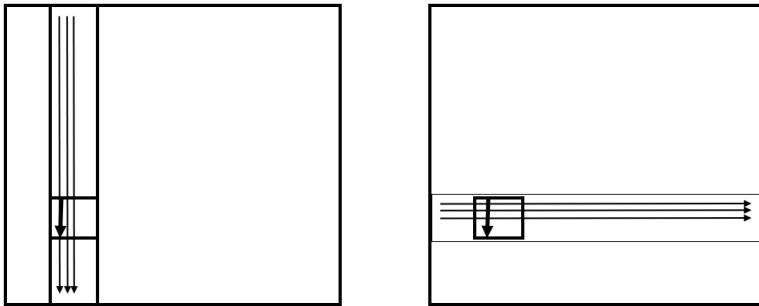


Figure 5.8: Transposing a 1D partitioned array

Transposing data is an important part of such operations as the FFT. We develop this in steps. Refer to figure [5.8](#).

The source data is easily 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.c
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 5.7.** 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?

## 5.5 More about data

### 5.5.1 Big data types

The `size` parameter in MPI send and receive calls is of type integer, meaning that it's maximally  $2^{31} - 1$ . These day computers are big enough that this is a limitation. Derived types offer some way out: 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 that return that type of count. (The alternative would be to break backwards compatibility and use `MPI_Count` parameter in all existing routines.)

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);
```

For the source of this example, see section 5.7.29

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);
```

For the source of this example, see section 5.7.29

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);

```

For the source of this example, see section [5.7.29](#)

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 [4.25](#)) that return a count of this type:

```

MPI_Count recv_count;
MPI_Get_elements_x(&recv_status,MPI_FLOAT,&recv_count);

```

For the source of this example, see section [5.7.29](#)

**Remark 10** 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;
;

```

For the source of this example, see section ??

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.

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 ??) and **MPI\_Type\_get\_true\_extent\_x** (figure [5.17](#)).

*Python note.* Since python has unlimited size integers there is no explicit need for the ‘x’ variants of routines. Internally, `MPI.Status.Get_count` is implemented in terms of `MPI_Get_count_x`.

### 5.5.2 Packing

One of the reasons for derived datatypes is dealing with non-contiguous 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`** data elements can be added to a buffer one at a time. The `position` parameter is updated each time by the packing routine.

```
// int MPI_Pack(
    void *inbuf, int incount, MPI_Datatype datatype,
    void *outbuf, int outcount, int *position,
    MPI_Comm comm);
```

Conversely, **`MPI_Unpack`** retrieves one element from the buffer at a time. You need to specify the MPI datatype.

```
// int MPI_Unpack(
    void *inbuf, int insize, int *position,
    void *outbuf, int outcount, MPI_Datatype datatype,
    MPI_Comm comm);
```

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.

```
// pack.c
if (procno==sender) {
    MPI_Pack(&nsends, 1, MPI_INT, buffer, buflen, &position, comm);
    for (int i=0; i<nsends; i++) {
        double value = rand()/(double) RAND_MAX;
        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);
    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);
    }
    MPI_Unpack(buffer, buflen, &position, &irecv_value, 1, MPI_INT, comm);
    ASSERT(irecv_value==nsends);
}
```

### 5.18 MPI\_Pack\_size

C:

```
int MPI_Pack_size  
(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)
```

Input parameters:

incount : Count argument to packing call (integer).  
datatype : Datatype argument to packing call (handle).  
comm : Communicator argument to packing call (handle).

Output parameters:

size : Upper bound on size of packed message, in bytes (integer).

Fortran:

```
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
```

input parameters:

INTEGER :: INCOUNT, DATATYPE, COMM

INTEGER :: SIZE, IERROR

*For the source of this example, see section 5.7.30*

You can precompute the size of the required buffer with **MPI\_Pack\_size** (figure 5.18) Add one time **MPI\_BSEND\_OVERHEAD**.

**Exercise 5.8.** 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.

## 5.6 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?

## 5.7 Sources used in this chapter

### 5.7.1 Listing of code header

### 5.7.2 Listing of code examples/mpi/p/inttype.py

```
#####
##### This program file is part of the book and course
##### "Parallel Computing"
##### by Victor Eijkhout, copyright 2020
#####
##### inttype.py : illustrating integer types
#####
##### from mpi4py import MPI
import numpy as np
import sys

comm = MPI.COMM_WORLD

nprocs = comm.Get_size()
procno = comm.Get_rank()

if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

## set up sender and receiver
sender = 0; receiver = nprocs-1
## how many elements to each process?
count = 6

## Send a contiguous buffer as numpy ints
if procno==sender:
    data = np.empty(count,dtype=np.int)
    for i in range(count):
        data[i] = i
    comm.Send( data, receiver )
elif procno==receiver:
    data = np.empty(count,dtype=np.int)
    comm.Recv( data, sender )
    print(data)

## Send a strided buffer as numpy ints
## this is wrong because numpy ints are not C ints
if procno==sender:
    sizeofint = np.dtype('int').itemsize
    print("Size of numpy int: {}".format(sizeofint))
```

```
data = np.empty(2*count,dtype=np.int)
for i in range(2*count):
    data[i] = i
vectortype = MPI.INT.Create_vector(count,1,2)
vectortype.Commit()
comm.Send( [data,1,vectortype], receiver )
elif procno==receiver:
    data = np.empty(count,dtype=np.int)
    comm.Recv( data, sender )
    print(data)

## Send strided buffer as C ints
if procno==sender:
    sizeofint = np.dtype('intc').itemsize
    print("Size of C int: {}".format(sizeofint))
    data = np.empty(2*count,dtype=np.intc)
    for i in range(2*count):
        data[i] = i
    vectortype = MPI.INT.Create_vector(count,1,2)
    vectortype.Commit()
    comm.Send( [data,1,vectortype], receiver )
elif procno==receiver:
    data = np.empty(count,dtype=np.intc)
    comm.Recv( data, sender )
    print(data)
```

### 5.7.3 Listing of code examples/mpi/p/allgatherv.py

```
import numpy as np
import random # random.randint(1,N), random.random()
import sys
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

mycount = procid+1
my_array = np.empty(mycount,dtype=np.float64)

for i in range(mycount):
    my_array[i] = procid
recv_counts = np.empty(nprocs,dtype=np.int)
recv_displs = np.empty(nprocs,dtype=np.int)

my_count = np.empty(1,dtype=np.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] )

# other syntax:
# comm.Allgatherv( [my_array,mycount,0,MPI.DOUBLE], [global_array,recv_counts,recv_displs,MPI.DOUBLE] )

if procid==0:
    #print(procid,global_array)
    for p in range(nprocs):
        if recv_counts[p]!=p+1:
            print( "recv count[%d] should be %d, not %d" \
                  % (p,p+1,recv_counts[p]) )
c = 0
for p in range(nprocs):
    for q in range(p+1):
        if global_array[c]!=p:
            print( "p=%d, q=%d should be %d, not %d" \
                  % (p,q,p,global_array[c]) )
    c += 1
print "finished"

```

#### 5.7.4 Listing of code examples/mpi/p/vector.py

```

import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

sender = 0; receiver = 1; the_other = 1-procid
count = 5; stride = 2

source = np.empty(stride*count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)

for i in range(stride*count):
    source[i] = i+.5

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)

if procid==sender:
    print("finished")
if procid==receiver:
    for i in range(count):
        if target[i]!=source[stride*i]:
            print("error in location %d: %e s/b %e" % (i,target[i],source[stride*i]))
```

### 5.7.5 Listing of code examples/mpi/mpf/vector.cxx

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <vector>
using std::vector;
#include <cassert>

#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

    const mpl::communicator &comm_world = mpl::environment::comm_world();
    int nprocs,procno;
    // compute communicator rank and size
    nprocs = comm_world.size();
    procno = comm_world.rank();

    if (nprocs<2) {
        printf("This program needs at least two processes\n");
        return -1;
    }
    int sender = 0, receiver = 1, the_other = 1-procno,
        count = 5,stride=2;
    vector<double>
        source(stride*count),
        target(count);

    for (int i=0; i<stride*count; i++)
        source[i] = i+.5;

    if (procno==sender) {
        mpl::strided_vector_layout<double> newvectortype(count,1,stride);
        comm_world.send
            (source.data(),newvectortype,the_other);
    } else if (procno==receiver) {
        int recv_count;
        mpl::status recv_status;
        recv_status = comm_world.recv
            (target.data(),mpl::contiguous_layout<double>(count),
             the_other);
```

```

    recv_count = recv_status.get_count<double>();
    assert(recv_count==count);
}

if (procno==receiver) {
    for (int i=0; i<count; i++)
        if (target[i]!=source[stride*i])
printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
    }

if (procno==0)
printf("Finished\n");

return 0;
}

```

### 5.7.6 Listing of code examples/mpi/c/contiguous.c

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
}
int sender = 0, receiver = 1, count = 5;
double *source,*target;
source = (double*) malloc(count*sizeof(double));
target = (double*) malloc(count*sizeof(double));

for (int i=0; i<count; i++)
    source[i] = i+.5;

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);
}

```

```
    }

    if (procno==receiver) {
        for (int i=0; i<count; i++)
            if (target[i]!=source[i])
                printf("location %d %e s/b %e\n",i,target[i],source[i]);
    }

    if (procno==0)
        printf("Finished\n");

    MPI_Finalize();
    return 0;
}
```

### 5.7.7 Listing of code examples/mpi/f/contiguous.F90

Program Contiguous

```
use mpi
implicit none

integer :: sender = 0, receiver = 1, count = 5
double precision, dimension(:),allocatable :: source,target

integer :: newvectortype
integer :: recv_status(MPI_STATUS_SIZE),recv_count

#include "globalinit.F90"

if (ntids<2) then
    print *, "This program needs at least two processes"
    stop
end if

ALLOCATE(source(count))
ALLOCATE(target(count))

do i=1,count
    source(i) = i+.5;
end do

if (mytid==sender) then
    call MPI_Type_contiguous(count,MPI_DOUBLE_PRECISION,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_DOUBLE_PRECISION,receiver,0,comm,&
        recv_status,err)
    call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count,err)
    !ASSERT(count==recv_count);
```

```
    end if

    if (mytid==receiver) then
        ! for (i=0; i<count; i++)
        !     if (target[i]!=source[i])
        !         printf("location %d %e s/b %e\n",i,target[i],source[i]);
    end if

    call MPI_Finalize(err)

end Program Contiguous
```

### 5.7.8 Listing of code examples/mpi/p/contiguous.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

sender = 0; receiver = 1; the_other = 1-procid
count = 5

source = np.empty(count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)

for i in range(count):
    source[i] = i+.5

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)

if procid==sender:
    print("finished")
if procid==receiver:
    for i in range(count):
        if target[i]!=source[i]:
            print("error in location %d: %e s/b %e" % (i,target[i],source[i]))
```

### 5.7.9 Listing of code examples/mpi/c/vector.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

    if (nprocs<2) {
        printf("This program needs at least two processes\n");
        return -1;
    }
    int sender = 0, receiver = 1, the_other = 1-procno,
        count = 5,stride=2;
    double *source,*target;
    source = (double*) malloc(stride*count*sizeof(double));
    target = (double*) malloc(count*sizeof(double));

    for (int i=0; i<stride*count; i++)
        source[i] = i+.5;

    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);
    }

    if (procno==receiver) {
        for (int i=0; i<count; i++)
            if (target[i]!=source[stride*i])
                printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
    }

    if (procno==0)
        printf("Finished\n");

    MPI_Finalize();
    return 0;
}
```

### 5.7.10 Listing of code examples/mpi/f08/vector.F90

Program Vector

```

use mpi_f08
implicit none

double precision, dimension(:),allocatable :: source,target
integer :: sender = 0,receiver = 1, count = 5, stride = 2

Type(MPI_Datatype) :: newvectortype
integer :: recv_count
Type(MPI_Status) :: recv_status

Type(MPI_Comm) :: comm;
integer :: mytid,ntids,i,p,err;

call MPI_Init()
comm = MPI_COMM_WORLD
call MPI_Comm_rank(comm,mytid)
call MPI_Comm_size(comm,ntids)
call MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN)

if (ntids<2) then
    print *, "This program needs at least two processes"
    stop
end if

ALLOCATE(source(stride*count))
ALLOCATE(target(stride*count))

do i=1,stride*count
    source(i) = i+.5;
end do

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

if (mytid==receiver) then

```

```
! for (i=0; i<count; i++)
!   if (target[i]!=source[stride*i])
!     printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
end if

call MPI_Finalize(err)

end Program Vector
```

### 5.7.11 Listing of code examples/mpi/f/vector.F90

```
Program Vector

use mpi
implicit none

double precision, dimension(:),allocatable :: source,target
integer :: sender = 0,receiver = 1, count = 5, stride = 2

integer :: newvectortype
integer :: recv_status(MPI_STATUS_SIZE),recv_count

#include "globalinit.F90"

if (ntids<2) then
  print *, "This program needs at least two processes"
  stop
end if

ALLOCATE(source(stride*count))
ALLOCATE(target(stride*count))

do i=1,stride*count
  source(i) = i+.5;
end do

if (mytid==sender) then
  call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
    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_DOUBLE_PRECISION, sender, 0, comm,&
    recv_status,err)
  call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count,err)
end if

if (mytid==receiver) then
  ! for (i=0; i<count; i++)
  !   if (target[i]!=source[stride*i])
  !     printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
```

```
    end if

    call MPI_Finalize(err)

end Program Vector
```

### 5.7.12 Listing of code examples/mpi/mpl/sendrange.cxx

```
#include <cstdlib>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;

#include <mpl/mpl.hpp>

int main() {
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    if (comm_world.size()<2)
        return EXIT_FAILURE;

    vector<double> v(15);

    if (comm_world.rank()==0) {

        // initialize
        for ( auto &x : v ) x = 1.41;

        /*
         * Send and report
         */
        comm_world.send(v.begin(), v.end(), 1); // send to rank 1

    } else if (comm_world.rank()==1) {

        /*
         * Receive data and report
         */
        comm_world.recv(v.begin(), v.end(), 0); // receive from rank 0

        cout << "Got:";
        for ( auto x : v )
            cout << " " << x;
        cout << endl;
    }
    return EXIT_SUCCESS;
}
```

**5.7.13 Listing of code examples/mpi/f08/section.F90**

```
Program F90Section
use mpi_f08
implicit none

integer :: i,j, nprocs,procno
integer,parameter :: siz=20
real,dimension(siz,siz) :: matrix = [ ((j+(i-1)*siz,i=1,siz),j=1,siz) ]
real,dimension(2,2) :: submatrix
Type(MPI_Comm) :: comm

call MPI_Init()
comm = MPI_COMM_WORLD

call MPI_Comm_size(comm,nprocs)
call MPI_Comm_rank(comm,procno)
if (nprocs<2) then
    print *, "This example really needs 2 processors"
    call MPI_Abort(comm,0)
end if
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

call MPI_Finalize()

end Program F90Section
```

**5.7.14 Listing of code code/mpi/c/row2col.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

#define ERR(c,m) if (c) { printf("Error: %s\n",m); MPI_Abort(MPI_COMM_WORLD,c); }

int main(int argc,char **argv) {

#define MOD(i,n) (i+n)%n
#define ABS(x) ((x)<0 ? (-x) : (x))
#define MAX(x,y) ((x)>(y) ? (x) : (y))

#define MALLOC( t,v,n ) \
```

## 5. MPI topic: Data types

---

```
t *v = (t*) malloc ( (n)*sizeof(t) ); \
if (!v) { printf("Allocation failed in line %d\n", __LINE__); MPI_Abort(comm,0); }

MPI_Comm comm;
int procno=-1,nprocs,ierr;
MPI_Init(&argc,&argv);
comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm,&procno);
MPI_Comm_size(comm,&nprocs);
MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN);

if (nprocs==1) {
    printf("This program needs at least 2 procs\n");
    MPI_Abort(comm,0);
}
int sender = 0, receiver = nprocs-1;
#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_Request req;

if (procno==sender) {
/*
 * Write lexicographic test data
 */
double data[SIZE][SIZE];
for (int i=0; i<SIZE; i++)
    for (int j=0; j<SIZE; j++)
data[i][j] = j+i*SIZE;
/*
 * Make a datatype that enumerates the storage in C order
 */
MPI_Datatype rowtype;
ierr =
MPI_Type_create_subarray
(2,sizes,subsizes,starts,
 MPI_ORDER_C,MPI_DOUBLE,&rowtype);
ERR(ierr,"creating rowtype");
MPI_Type_commit(&rowtype);
MPI_Send(data,1,rowtype, receiver,0,comm);
MPI_Type_free(&rowtype);

/*
 * Make a datatype that enumerates the storage in F order
 */
MPI_Datatype coltype;
ierr =
MPI_Type_create_subarray
(2,sizes,subsizes,starts,
```

```
    MPI_ORDER_FORTRAN,MPI_DOUBLE,&coltype);
ERR(ierr,"creating rowtype");
MPI_Type_commit(&coltype);
MPI_Send(data,1,coltype, receiver,0,comm);
MPI_Type_free(&coltype);

} else if (procno==receiver) {
int linearssize = SIZE * SIZE/2;
double linedata[ linearssize ];

/*
 * Receive msg in C order:
 */
MPI_Recv(linedata,linearssize,MPI_DOUBLE, sender,0,comm, MPI_STATUS_IGNORE);
printf("Received C order:\n");
for (int i=0; i<SIZE/2; i++) {
    for (int j=0; j<SIZE; j++)
printf(" %5.3f",linedata[j+i*SIZE]);
    printf("\n");
}

/*
 * Receive msg in F order:
 */
MPI_Recv(linedata,linearssize,MPI_DOUBLE, sender,0,comm, MPI_STATUS_IGNORE);
printf("Received F order:\n");
for (int j=0; j<SIZE; j++) {
    for (int i=0; i<SIZE/2; i++)
printf(" %5.3f",linedata[i+j*SIZE/2]);
    printf("\n");
}

/*
 * MPI_Finalize(); */
return 0;
}
```

### 5.7.15 Listing of code examples/mpi/c/indexed.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
```

```

    }
    int sender = 0, receiver = 1, the_other = 1-procno,
        count = 5, totalcount = 15, targetbuffersize = 2*totalcount;
    int *source,*target;
    int *displacements,*blocklengths;

    displacements = (int*) malloc(count*sizeof(int));
    blocklengths = (int*) malloc(count*sizeof(int));
    source = (int*) malloc(totalcount*sizeof(int));
    target = (int*) malloc(targetbuffersize*sizeof(int));

    displacements[0] = 2; displacements[1] = 3; displacements[2] = 5;
    displacements[3] = 7; displacements[4] = 11;
    for (int i=0; i<count; ++i)
        blocklengths[i] = 1;
    for (int i=0; i<totalcount; ++i)
        source[i] = i;

    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);
    }

    if (procno==receiver) {
        int i=3,val=7;
        if (target[i]!=val)
            printf("location %d %d s/b %d\n",i,target[i],val);
        i=4; val=11;
        if (target[i]!=val)
            printf("location %d %d s/b %d\n",i,target[i],val);
    }
    if (procno==0)
        printf("Finished\n");

    MPI_Finalize();
    return 0;
}

```

### 5.7.16 Listing of code examples/mpi/f/indexed.F90

Program Indexed

```
use mpi
implicit none

integer :: newvectortype;
integer,dimension(:),allocatable :: indices,blocklengths,&
    source,targt
integer :: sender = 0, receiver = 1, count = 5,totalcount = 15
integer :: recv_status(MPI_STATUS_SIZE),recv_count

#include "globalinit.F90"

if (ntids<2) then
    print *, "This program needs at least two processes"
    stop
end if

ALLOCATE(indices(count))
ALLOCATE(blocklengths(count))
ALLOCATE(source(totalcount))
ALLOCATE(targt(count))

indices(0) = 2; indices(1) = 3; indices(2) = 5;
indices(3) = 7; indices(4) = 11;
do i=1,count
    blocklengths(i) = 1
end do
do i=1,totalcount
    source(i) = i
end do

if (mytid==sender) then
    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(targt,count,MPI_INT,receiver,0,comm,&
        recv_status,err)
    call MPI_Get_count(recv_status,MPI_INT,recv_count,err)
    ! ASSERT(recv_count==count);
end if

! if (mytid==receiver) {
!     int i=3,val=7;
!     if (targt(i)!=val)
!         printf("location %d %d s/b %d\n",i,targt(i),val);
!     i=4; val=11;
!     if (targt(i)!=val)
!         printf("location %d %d s/b %d\n",i,targt(i),val);
! }

call MPI_Finalize(err)
```

```
end Program Indexed
```

### 5.7.17 Listing of code examples/mpi/p/indexed.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

sender = 0; receiver = 1; the_other = 1-procid
count = 5; totalcount = 15

displacements = np.empty(count,dtype=np.int)
blocklengths = np.empty(count,dtype=np.int)
source = np.empty(totalcount,dtype=np.float64)
target = np.empty(count,dtype=np.float64)

idcs = [2,3,5,7,11]
for i in range(len(idcs)):
    displacements[i] = idcs[i]
    blocklengths[i] = 1
for i in range(totalcount):
    source[i] = i+.5

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)

if procid==sender:
    print("finished")
if procid==receiver:
    target_loc = 0
    for block in range(count):
        for element in range(blocklengths[block]):
            source_loc = displacements[block]+element
            if target[target_loc]!=source[source_loc]:
                print("error in src/tar location %d/%d: %e s/b %e" \
                      % (source_loc,target_loc,target[target_loc],source[source_loc]) )
            target_loc += 1
```

**5.7.18 Listing of code examples/mpi/mpl/indexed.cxx**

```
#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;

#include <cassert>

#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

    // MPI Comm world
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    int nprocs = comm_world.size(), procno = comm_world.rank();

    int sender = 0, receiver = 1, the_other = 1-procno,
        totalcount = 15, targetbuffersize = 2*totalcount;

    vector<int>
        source_buffer(totalcount),
        target_buffer(targetbuffersize);
    for (int i=0; i<totalcount; ++i)
        source_buffer[i] = i;

    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);
    }

    if (procno==receiver) {
        int i=3,val=7;
        if (target_buffer[i]!=val)
            printf("Error: location %d %d s/b %d\n",i,target_buffer[i],val);
        i=4; val=11;
        if (target_buffer[i]!=val)
            printf("Error: location %d %d s/b %d\n",i,target_buffer[i],val);
        printf("Finished. Correctly sent indexed primes.\n");
    }
}
```

```
    return 0;
}
```

### 5.7.19 Listing of code examples/mpi/mpl/indexedblock.cxx

```
#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;

#include <cassert>

#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

    // MPI Comm world
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    int nprocs = comm_world.size(), procno = comm_world.rank();

    int sender = 0, receiver = 1, the_other = 1-procno,
        totalcount = 15, targetbuffersize = 2*totalcount;

    vector<int>
        source_buffer(totalcount),
        target_buffer(targetbuffersize);
    for (int i=0; i<totalcount; ++i)
        source_buffer[i] = i;

    const int count = 5;
    mpl::contiguous_layout<int>
        fiveints(count);
    mpl::indexed_block_layout<int>
        indexed_where( 1, {2,3,5,7,11} );
    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);
    }

    if (procno==receiver) {
        int i=3,val=7;
        if (target_buffer[i]!=val)
            printf("Error: location %d s/b %d\n",i,target_buffer[i],val);
        i=4; val=11;
        if (target_buffer[i]!=val)
```

```
    printf("Error: location %d s/b %d\n",i,target_buffer[i],val);
    printf("Finished. Correctly sent indexed primes.\n");
}

return 0;
}
```

### 5.7.20 Listing of code examples/mpi/c/struct.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
}
int sender = 0, receiver = 1, the_other = 1-procno;
struct object {
    char c;
    double x[2];
    int i;
};

size_t size_of_struct = sizeof(struct object);
if (procno==sender)
    printf("Structure has size %ld, naive size %ld\n",
    size_of_struct,
    sizeof(char)+2*sizeof(double)+sizeof(int));
struct object myobject;
if (procno==sender) {
    myobject.c = 'x';
    myobject.x[0] = 2.7; myobject.x[1] = 1.5;
    myobject.i = 37;
}

MPI_Datatype newstructuretype;
int structlen = 3;
int blocklengths[structlen]; MPI_Datatype types[structlen];
MPI_Aint displacements[structlen];

/*
 * where are the components relative to the structure?
 */
MPI_Aint current_displacement=0;
```

## 5. MPI topic: Data types

---

```
// 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_Aint typesize;
    MPI_Type_extent(newstructuretype,&typesize);
    assert( typesize==size_of_struct );
    printf("Type extent: %ld bytes; displacements: %ld %ld %ld\n",
    typesize,displacements[0],displacements[1],displacements[2]);
}
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);

/* if (procno==sender) */
/*     printf("char x=%ld, l=%ld; double x=%ld, l=%ld, int x=%ld, l=%ld\n", */
/*            char_extent,char_lb,double_extent,double_lb,int_extent,int_lb); */

if (procno==receiver) {
    printf("Char '%c' double0=%e double1=%e int=%d\n",
    myobject.c,myobject.x[0],myobject.x[1],myobject.i);
    ASSERT(myobject.x[1]==1.5);
    ASSERT(myobject.i==37);
}

if (procno==0)
    printf("Finished\n");

MPI_Finalize();
return 0;
}

#ifndef _WIN32
blocklengths[0] = 1; types[0] = MPI_CHAR;
displacements[0] = (size_t)&(myobject.c) - (size_t)&myobject;
blocklengths[1] = 2; types[1] = MPI_DOUBLE;
displacements[1] = (size_t)&(myobject.x[0]) - (size_t)&myobject;
blocklengths[2] = 1; types[2] = MPI_INT;
displacements[2] = (size_t)&(myobject.i) - (size_t)&myobject;
```

```
MPI_Aint char_extent,char_lb;
MPI_Type_get_extent(MPI_CHAR,&char_lb,&char_extent);
/* if (procno==0) */
/*   printf("CHAR lb=%ld xt=%ld disp=%ld\n",char_lb,char_extent,current_displacement); */
MPI_Aint double_extent,double_lb;
MPI_Type_get_extent(MPI_DOUBLE,&double_lb,&double_extent);
/* if (procno==0) */
/*   printf("DOUBLE lb=%ld xt=%ld disp=%ld\n",double_lb,double_extent,current_displacement); */
MPI_Aint int_extent,int_lb;
MPI_Type_get_extent(MPI_INT,&int_lb,&int_extent);
/* if (procno==0) */
/*   printf("INT lb=%ld xt=%ld disp=%ld\n",int_lb,int_extent,current_displacement); */
#endif
```

### 5.7.21 Listing of code examples/mpi/f/struct.F90

```
Program StructType

use mpi_f08
implicit none

type(MPI_Comm) :: comm;
integer :: nprocs,procno;

!!
!! The Type that we are going to send
!!
Type object
    character :: c
    real*8,dimension(2) :: x
    integer :: i
end type object

#define MPI_Aint integer(kind=MPI_ADDRESS_KIND)

!!
!! local data
!!
integer :: sender=0,receiver=1
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

!!
!! Initial setup
!!
```

## 5. MPI topic: Data types

---

```
call MPI_Init()
comm = MPI_COMM_WORLD;
call MPI_Comm_rank(comm,procno)
call MPI_Comm_size(comm,nprocs)
call MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN)

if (nprocs<2) then
    print *, "This program needs at least two processes"
    call MPI_Abort(comm,1)
end if

!!
!! Fill in meaningful data only on the sender
!!
if (procno==sender) then
    myobject%c = 'x'
    myobject%x(0) = 2.7; myobject%x(1) = 1.5
    myobject%i = 37

    print '("Set: char=",a1,x,", double0=",x,f9.5,x,"double1=",x,f9.5,x,", int=",i5)', &
           myobject%c,myobject%x(0),myobject%x(1),myobject%i
else
    myobject%c = ' '
    myobject%x(0) = 0.0; myobject%x(1) = 0.0
    myobject%i = 0
end if

!!
!! Where are the components relative to the structure?
!!

!! 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
    print '("Displacements:",3(1x,i0))',displacements
end if

!!
!! Create the structure type
```

```
!!  
call MPI_Type_create_struct(structlen,blocklengths,displacements,types,newstructuretype)  
call MPI_Type_commit(newstructuretype)  
  
!!  
!! Send and receive call, both using the structure type  
!!  
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)  
  
!!  
!! Print out what we sent and received  
!!  
if (procno==sender) then  
    print '("Sent: char=",a1,x,", double0=",x,f9.5,x,"double1=",x,f9.5,x,", int=",i5)', &  
        myobject%c,myobject%x(0),myobject%x(1),myobject%i  
else if (procno==receiver) then  
    print '("Received: char=",a1,x,", double0=",x,f9.5,x,"double1=",x,f9.5,x,", int=",i5)', &  
        myobject%c,myobject%x(0),myobject%x(1),myobject%i  
end if  
  
if (procno==0) then  
    print *, "Finished"  
end if  
  
call MPI_Finalize()  
  
end Program StructType
```

### 5.7.22 Listing of code examples/mpi/mpl/struct.cxx

```
#include <stdlib.h>  
#include <stdio.h>  
#include <string.h>  
#include <vector>  
using std::vector;  
#include <cassert>  
  
#include <mpl/mpl.hpp>  
  
int main(int argc,char **argv) {  
  
    const mpl::communicator &comm_world = mpl::environment::comm_world();  
    int nprocs,procno;  
    // compute communicator rank and size  
    nprocs = comm_world.size();  
    procno = comm_world.rank();
```

```

if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
}
int sender = 0, receiver = 1, the_other = 1-procno;
char c;
vector<double> x(2);
int i;
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 );
}

if (procno==receiver) {
    printf("Char '%c' double0=%e double1=%e int=%d\n",
    c,x[0],x[1],i);
    assert(x[1]==1.5);
    assert(i==37);
}

if (procno==0)
    printf("Finished\n");

return 0;
}

```

### 5.7.23 Listing of code examples/mpi/c/typematch.c

```

int main() {

    MPI_Init(0,0);

    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);

```

```
    MPI_Type_size(MPI_X10, &s10);

    printf("%d, %d\n", s5, s10);

    MPI_Finalize();

    return 0;
}
```

### 5.7.24 Listing of code examples/mpi/c/typesize.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char **argv) {

#include "globalinit.c"

    int size, count, stride, bs;
    MPI_Aint lb, asize;

    MPI_Datatype newtype;

    count = 3; bs = 2; stride = 5;
    MPI_Type_vector(count, bs, stride, MPI_DOUBLE, &newtype);
    MPI_Type_commit(&newtype);
    MPI_Type_size(newtype, &size);
    ASSERT( size==(count*bs)*sizeof(double) );
    MPI_Type_free(&newtype);

    printf("count=%d, stride=%d, bs=%d, size=%d\n", count, stride, bs, size);

    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);

    printf("count=%d, stride=%d, bs=%d: lb=%ld, extent=%ld\n", count, stride, bs, lb, asize);

    if (procno==0)
        printf("Finished\n");

    MPI_Finalize();
    return 0;
}
```

**5.7.25 Listing of code examples/mpi/c/typeextent.c**

**5.7.26 Listing of code examples/mpi/c/trueextent.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
}
int sender = 0, receiver = 1, the_other = 1-procno,
    count = 4;
int sizes[2] = {4,6},subsizes[2] = {2,3},starts[2] = {1,2};
double *source,*target;
source = (double*) malloc(sizes[0]*sizes[1]*sizeof(double));
target = (double*) malloc(subsizes[0]*subsizes[1]*sizeof(double));

for (int i=0; i<sizes[0]*sizes[1]; i++)
    source[i] = i+.5;

MPI_Datatype subarraytype;
if (procno==sender) {
    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_extent(subarraytype,&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) *
        ( (starts[0]+subsizes[0]-1)*sizes[1] + starts[1]+subsizes[1] )
        - comp_lb;
    printf("Found lb=%ld, extent=%ld\n",true_lb,true_extent);
    printf("Computing lb=%ld extent=%ld\n",comp_lb,comp_extent);
//    ASSERT(extent==true_lb+extent);
    ASSERT(true_lb==comp_lb);
    ASSERT(true_extent==comp_extent);

    MPI_Send(source,1,subarraytype,the_other,0,comm);
    MPI_Type_free(&subarraytype);
```

```
    } 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);
    }

    if (procno==receiver) {
        printf("received:");
        for (int i=0; i<count; i++)
            printf(" %6.3f", target[i]);
        printf("\n");
        int icnt = 0;
        for (int i=starts[0]; i<starts[0]+subsizes[0]; i++) {
            for (int j=starts[1]; j<starts[1]+subsizes[1]; j++) {
                printf("%d,%d\n", i, j);
                ASSERT(icnt<count, "icnt too hight");
                int isrc = i*sizes[1]+j;
                if (source[isrc]!=target[icnt])
                    printf("target location (%d,%d)->%d %e s/b %e\n", i, j, icnt, target[icnt], source[isrc]);
                icnt++;
            }
        }
    }

    if (procno==0)
        printf("Finished\n");

    /* MPI_Finalize(); */
    return 0;
}
```

### 5.7.27 Listing of code examples/mpi/c/interleavegather

### 5.7.28 Listing of code examples/mpi/c/stridestretch.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char **argv) {
    #include "globalinit.c"

    if (nprocs<2) {
        printf("This program needs at least two processes\n");
```

## 5. MPI topic: Data types

---

```
        return -1;
    }
int sender = 0, receiver = 1, the_other = 1-procno,
    count = 5,stride=2;
double *source,*target;
source = (double*) malloc(stride*count*sizeof(double));
target = (double*) malloc(stride*count*sizeof(double));

for (int i=0; i<stride*count; i++)
    source[i] = i+.5;

if (procno==sender) {
    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);
    MPI_Type_free(&oneblock);
    MPI_Type_free(&paddedblock);
} else if (procno==receiver) {
    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);
    /* MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm, */
    /*          &recv_status); */
    int recv_count;
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    ASSERT(recv_count==count);
}

if (procno==receiver) {
    for (int i=0; i<count; i++)
        if (target[i*stride]!=source[i*stride])
printf("location %d %e s/b %e\n",i,target[i*stride],source[stride*i]);
}

if (procno==0)
```

```
    printf("Finished\n");

    MPI_Finalize();
    return 0;
}
```

### 5.7.29 Listing of code examples/mpi/c/vectorx.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
}
int sender = 0, receiver = 1, the_other = 1-procno,
count = 5,stride=2;

if (procno==sender) printf("size of size_t = %d\n",sizeof(size_t));

float *source=NULL,*target=NULL;
int mediumsize = 1<<30;
int nblocks = 8;
size_t datasize = (size_t)mediumsize * nblocks * sizeof(float);

if (procno==sender)
    printf("datasize = %lld bytes =%7.3f giga-bytes = %7.3f gfloats\n",
datasize,datasize*1.e-9,datasize*1.e-9/sizeof(float));

if (procno==sender) {
    source = (float*) malloc(datasize);
    if (source) {
        printf("Source allocated\n");
    } else {
        printf("Could not allocate source data\n"); MPI_Abort(comm,1);
    }
    long int idx = 0;
    for (int iblock=0; iblock<nblocks; iblock++) {
        for (int element=0; element<mediumsize; element++) {
source[idx] = idx+.5; idx++;
if (idx<0) { printf("source index wrap\n"); MPI_Abort(comm,0); }
        }
    }
}

if (procno==receiver) {
    target = (float*) malloc(datasize);
```

## 5. MPI topic: Data types

---

```
if (target) {
    printf("Target allocated\n");
} else {
    printf("Could not allocate target data\n"); MPI_Abort(comm,1);
}

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);
} else if (procno==receiver) {
    MPI_Status recv_status;
    MPI_Recv(target,nblocks,blocktype,receiver,0,comm,
              &recv_status);
    MPI_Count recv_count;
    MPI_Get_elements_x(&recv_status,MPI_FLOAT,&recv_count);
    printf("Received %7.3f medium size elements\n",recv_count * 1e-9);
}
MPI_Type_free(&blocktype);

if (0 && procno==receiver) {
    for (int i=0; i<count; i++)
        if (target[i]!=source[stride*i])
printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
}

if (procno==0)
    printf("Finished\n");

if (procno==sender)
    free(source);
if (procno==receiver)
    free(target);

MPI_Finalize();
return 0;
}
```

### 5.7.30 Listing of code examples/mpi/pack/pack.c

# Chapter 6

## MPI topic: Communicators

### 6.1 Communicator basics

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. This group is fixed: it can neither be extended with additional processes, nor can it contract, for instance to eliminate crashed processes. But some flexibility in process handling may still be needed. For instance, there are circumstances where you want one subset of processes to operate independently of another subset. Examples are:

- If processors are organized in a  $p \times q$  grid, you may want to do broadcasts inside a row or column.
- For an application that includes a producer and a consumer part, it makes sense to split the processors accordingly.

In this section we will see mechanisms for defining new communicators that are subsets of `MPI_COMM_WORLD`. Chapter 7 discusses dynamic process management, which, while not extending `MPI_COMM_WORLD` does extend the set of available processes.

#### 6.1.1 Communicator types

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. Routines that construct communicators can give this as result if an error occurs.

If you don't want to write `MPI_COMM_WORLD` repeatedly, you can assign that value to a variable of type `MPI_Comm` (figure 6.1).

Examples:

```
// C:  
#include <mpi.h>  
MPI_Comm comm = MPI_COMM_WORLD;
```

## 6. MPI topic: Communicators

---

### 6.1 MPI\_Comm

```
C:  
MPI_Comm commvariable; // datatype is defined in mpi.h  
  
Fortran with 2008 support:  
Type(MPI_Comm) :: commvariable  
  
Fortran legacy interface:  
Integer :: commvariable  
  
Python:  
MPI.Comm : class of communicators  
MPI.Intracomm  
MPI.Intercomm : subclasses of the MPI.Comm class.  
MPI.Comm.Is_inter()  
MPI.Comm.Is_intra() : convenience functions, not part of the MPI standard.
```

```
|| ! 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.

```
|| comm = MPI.COMM_WORLD
```

MPL note.

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

## 6.2 Making new 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 for instance 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 really distinct from each other.

In order to make such subsets of processes, MPI has the mechanism of taking a subset of `MPI_COMM_WORLD` 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. 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 `MPI_Comm_dup` (section 6.3), `MPI_Comm_split` (section 6.4), `MPI_Comm_create` (section 6.5), `MPI_Intercomm_create` (section 6.6), `MPI_Comm_spawn` (section 7.1);
- 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`.

### 6.2.1 Scenario: distributed linear algebra

For *scalability* reasons, matrices should often be distributed in a 2D manner, that is, each process receives a subblock that is not a block of 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.

### 6.2.2 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.

### 6.2.3 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.

### 6.2.4 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 11.1.

### 6.2.5 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 7) which start in a `MPI_COMM_WORLD` of their own. However, another communicator will be created that spawns the old and new worlds so that you can communicate with the new processes.

### 6.3 Duplicating communicators

With `MPI_Comm_dup` you can make an exact duplicate of a communicator. This 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.

In section 12.7 it was explained that MPI messages are non-overtaking. This may lead to confusing situations, witness the following. 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();
};
```

*For the source of this example, see section 6.8.2*

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` (figure 6.2) 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;
```

## 6.2 MPI\_Comm\_dup

Semantics:

`MPI_COMM_DUP (comm, newcomm)`  
 IN `comm`: communicator (handle)  
 OUT `newcomm`: copy of `comm` (handle)

C:

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

F:

```
MPI_Comm_dup(comm, newcomm, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Comm), INTENT(OUT) :: newcomm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Py:

```
newcomm = oldcomm.Dup(Info info=None)
```

```
    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();
};
```

For the source of this example, see section 6.8.3

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 communication_start(self):
        sendbuf = np.empty(1, dtype=np.int); sendbuf[0] = 37
        recvbuf = np.empty(1, dtype=np.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)
```

## 6. MPI topic: Communicators

---

### 6.3 MPI\_Comm\_split

Semantics:  
MPI\_COMM\_SPLIT(comm, color, key, newcomm)  
IN comm: communicator (handle)  
IN color: control of subset assignment (integer)  
IN key: control of rank assignment (integer)  
OUT newcomm: new communicator (handle)

C:

```
int MPI_Comm_split(
    MPI_Comm comm, int color, int key,
    MPI_Comm *newcomm)
```

F:

```
MPI_Comm_split(comm, color, key, newcomm, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: color, key
TYPE(MPI_Comm), INTENT(OUT) :: newcomm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
```

Py:

```
newcomm = comm.Split(int color=0, int key=0)
```

```
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()
```

For the source of this example, see section 6.8.4

## 6.4 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 6.3) uses a ‘colour’ to define these subcommunicators: all processes in the old communicator with the same colour 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. 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.

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;
```

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, /* tag: */ 0, 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*.

The `MPI_Comm_split` routine has a ‘key’ parameter, which controls how the processes in the new communicator are ordered. By supplying the rank from the original communicator you let them be arranged in the same order.

*Python note.* In Python, the ‘key’ argument is optional:

```
## commsplit.py
mydata = procid

# communicator modulo 2
color = procid%2
mod2comm = comm.Split(color)
new_procid = mod2comm.Get_rank()

# communicator modulo 4 recursively
color = new_procid%2
mod4comm = mod2comm.Split(color)
new_procid = mod4comm.Get_rank()
```

For the source of this example, see section 6.8.5

*MPL note.* In MPL, splitting a communicator is done as one of the overloads of the communicator constructor;

```
// commsplit.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();
```

For the source of this example, see section 6.8.6

The `communicator::split` identifier itself is an otherwise empty subclass of `communicator`.

There is also a routine `MPI_Comm_split_type` which uses a type rather than a key to split the communicator. We will see this in action in section 11.1.

One application of communicator splitting is setting up a processor grid, with the possibility of using MPI solely within one row or column; see figure 6.1.

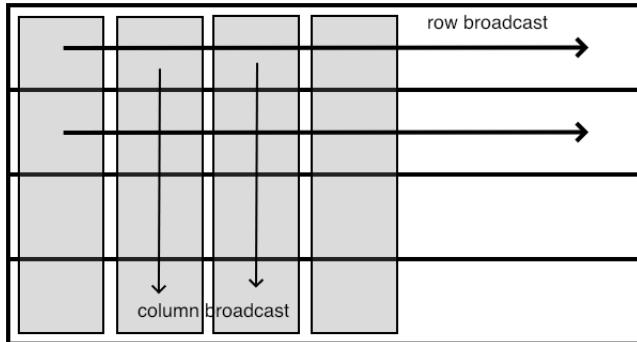


Figure 6.1: Row and column broadcasts in subcommunicators

**Exercise 6.1.** 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  $2 \times 3$  processor grid you should find:

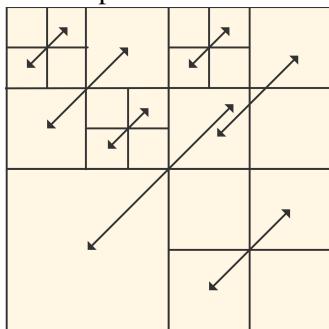
Global ranks:			Ranks in row:			Ranks in column:		
0	1	2	0	1	2	0	0	0
3	4	5	0	1	2	1	1	1

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.

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  $2 \times 2$  block form.

**Exercise 6.2.** Implement a recursive algorithm for matrix transposition:



- 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.

## 6.4 MPI\_Group

```
C:
MPI_Group groupvariable; // datatype is defined in mpi.h

Fortran with 2008 support:
Type(MPI_Group) :: groupvariable

Fortran legacy interface:
Integer :: groupvariable

Python:
MPI.Group is a class
```

## 6.5 MPI\_Comm\_group

Synopsis  
`int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)`

Input Parameters:  
`comm` : Communicator (handle)

Output Parameters  
`group` : Group in communicator (handle)

## 6.5 Communicators and groups

You saw in section 6.4 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` (figure 6.4) objects.

Using groups, it takes three steps to create a new communicator:

1. Access the `MPI_Group` of a communicator object using `MPI_Comm_group` (figure 6.5) .
2. Use various routines, discussed next, to form a new group.
3. Make a new communicator object from the group with `MPI_Group`, using `MPI_Comm_create` (figure 6.6)

Creating a new communicator from a group is collective on the old communicator. There is also a routine `MPI_Comm_create_group` that only needs to be called on the group that constitutes the new communicator.

### 6.5.1 Process groups

Groups are manipulated with `MPI_Group_incl`, `MPI_Group_excl`, `MPI_Group_difference` and a few more.

You can name your communicators with `MPI_Comm_set_name`, which could improve the quality of error messages when they arise.

```
|| MPI_Comm_group (comm, group, ierr)
|| MPI_Comm_create (MPI_Comm comm, MPI_Group group, MPI_Comm newcomm, ierr)

|| MPI_Group_union(group1, group2, newgroup, ierr)
|| MPI_Group_intersection(group1, group2, newgroup, ierr)
|| MPI_Group_difference(group1, group2, newgroup, ierr)
```

### 6.6 MPI\_Comm\_create

Synopsis

```
MPI_Comm_create( MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm )
```

Input parameters:

comm : Communicator (handle).

group : Group, which is a subset of the group of comm (handle).

Output parameters:

newcomm : New communicator (handle).

C:

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
```

Fortran90:

```
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
INTEGER      COMM, GROUP, NEWCOMM, IERROR
```

Fortran2008:

```
MPI_Comm_create(comm, group, newcomm, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Group), INTENT(IN) :: group
TYPE(MPI_Comm), INTENT(OUT) :: newcomm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

|| **MPI\_Group\_incl**(group, n, ranks, newgroup, ierr)

|| **MPI\_Group\_excl**(group, n, ranks, newgroup, ierr)

|| **MPI\_Group\_size**(group, size, ierr)

|| **MPI\_Group\_rank**(group, rank, ierr)

## 6.6 Inter-communicators

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, postprocessor) that need to stream data pairwise. In another example, dynamically spawned processes (section 7.1) 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*.

A call to `MPI_Intercomm_create` (figure 6.7) involves the following communicators:

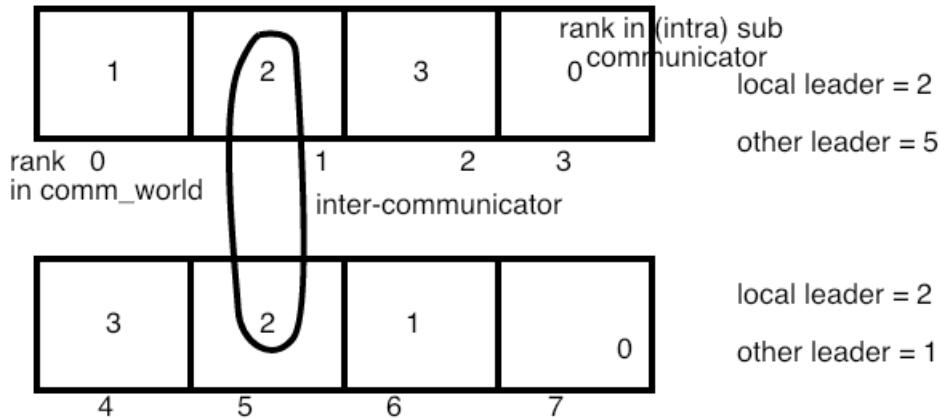


Figure 6.2: Illustration of ranks in an inter-communicator setup

- 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.

### 6.6.1 Inter-communicator 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) {
        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)
    }
}

```

### 6.7 MPI\_Intercomm\_create

Synopsis:

```
int MPI_Intercomm_create
    (MPI_Comm local_comm, int local_leader,
     MPI_Comm peer_comm, int remote_leader,
     int tag, MPI_Comm *newintercomm
    );
```

Input parameters:

```
local_comm : Local (intra)communicator
local_leader : Rank in local_comm of leader (often 0)
peer_comm : Communicator used to communicate between a designated process in
            the other communicator. Significant only at the process in local_comm
            with rank local_leader.
remote_leader : Rank in peer_comm of remote leader (often 0)
tag : Message tag to use in constructing intercommunicator; if multiple
      MPI_IntercommCreates are being made, they should use different tags
      (more precisely, ensure that the local and remote leaders are using
      different tags for each MPI_intercomm_create).
```

Output Parameter:

comm\_out : Created intercommunicator

```
    fprintf(stderr,
            "Got inter communication from unexpected %d; s/b %d\n",
            inter_source, local_number_of_other_leader);
}
```

For the source of this example, see section 6.8.7

#### 6.6.2 Inter-communicator 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!

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.

Inter-communicators can be used if two groups of process work asynchronously with respect to each other; another application is fault tolerance (section 12.4).

```
if (color==0) { // sending group: the local leader sends
if (i_am_local_leader)
    root = MPI_ROOT;
```

### 6.8 MPI\_Comm\_test\_inter

```

MPI_COMM_TEST_INTER(comm, flag)
IN comm : communicator (handle)
OUT flag : (logical)

int MPI_Comm_test_inter(MPI_Comm comm, int *flag)

MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
INTEGER COMM, IERROR
LOGICAL FLAG

||| 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);

```

For the source of this example, see section 6.8.7

#### 6.6.3 Inter-communicator querying

Some of the operations you have seen before for *intra-communicators* behave differently with inter-communicator:

- **MPI\_Comm\_size** returns the size of the local group, not the size of the inter-communicator.
- **MPI\_Comm\_rank** returns the rank in the local group.
- **MPI\_Comm\_group** returns the local group.

**MPI\_Comm\_get\_parent**: the other leader (see chapter 7).

Test whether a communicator is intra or inter: **MPI\_Comm\_test\_inter** (figure 6.8) .

**MPI\_Comm\_compare** works for inter-communicators.

**MPI\_Comm\_remote\_size** (figure 6.9) **MPI\_Comm\_remote\_group** (figure 6.10)

Virtual topologies (chapter 10) 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 6.11) .

## 6. MPI topic: Communicators

---

### 6.9 MPI\_Comm\_remote\_size

Semantics:

```
MPI_COMM_REMOTE_SIZE(comm, size)
IN comm: inter-communicator (handle)
OUT size: number of processes in the remote group of comm (integer)
```

C:

```
int MPI_Comm_remote_size(MPI_Comm comm, int *size)
```

Fortran:

```
MPI_Comm_remote_size(comm, size, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

```
Intercomm.Get_remote_size(self)
```

### 6.10 MPI\_Comm\_remote\_group

Semantics:

```
MPI_COMM_REMOTE_GROUP(comm, group)
IN comm: inter-communicator (handle)
OUT group: group of processes in the remote group of comm
```

C:

```
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
```

Fortran:

```
MPI_Comm_remote_group(comm, group, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Group), INTENT(OUT) :: group
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

```
Intercomm.Get_remote_group(self)
```

### 6.11 MPI\_Intercomm\_merge

Synopsis:

```
int MPI_Intercomm_merge
    (MPI_Comm intercomm, int high,
     MPI_Comm *newintraComm)
```

Input Parameters:

intercomm : Intercommunicator (handle)

high : Used to order the groups within comm (logical) when creating the new communicator. This is a boolean value; the group that sets high true has its processes ordered after the group that sets this value to false. If all processes in the intercommunicator provide the same value, the choice of which group is ordered first is arbitrary.

Output Parameters:

newintraComm : Created intracommunicator (handle)

## 6.7 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  $p' < q'$ .

## 6.8 Sources used in this chapter

### 6.8.1 Listing of code header

### 6.8.2 Listing of code examples/mpi/c/commdupwrong.cxx

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

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();
};

int main(int argc, char **argv) {
    #include "globalinit.c"

    int other = 1 - procno, sdata=5, rdata;
    MPI_Request request[2];
    MPI_Status status[2];

    if (procno>1) { MPI_Finalize(); return 0; }

    library my_library(comm);
    MPI_Isend(&sdata, 1, MPI_INT, other, 1, comm, &(request[0]));
    my_library.communication_start();
    MPI_Irecv(&rdata, 1, MPI_INT, other, MPI_ANY_TAG, comm, &(request[1]));
    MPI_Waitall(2, request, status);
    my_library.communication_end();

    if (status[1].MPI_TAG==2)
        printf("wrong!\n");

    MPI_Finalize();
    return 0;
}

int library::communication_start() {
```

```
int sdata=6,rdata;
MPI_Isend(&sdata,1,MPI_INT,other,2,comm,&(request[0]));
MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,
          comm,&(request[1]));
return 0;
}

int library::communication_end() {
    MPI_Status status[2];
    MPI_Waitall(2,request,status);
    return 0;
}
```

### 6.8.3 Listing of code examples/mpi/c/commdupright.cxx

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

class library {
private:
    MPI_Comm comm;
    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();
};

int main(int argc,char **argv) {

#include "globalinit.c"

    int other = 1-procno, sdata=5,rdata;
    MPI_Request request[2];
    MPI_Status status[2];

    if (procno>1) { MPI_Finalize(); return 0; }

    library my_library(comm);
    MPI_Isend(&sdata,1,MPI_INT,other,1,comm,&(request[0]));
```

## 6. MPI topic: Communicators

---

```
my_library.communication_start();
MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,
          comm,&(request[1]));
MPI_Waitall(2,request,status);
my_library.communication_end();

if (status[1].MPI_TAG==2)
    printf("wrong!\n");

MPI_Finalize();
return 0;
}

int library::communication_start() {
    int sdata=6,rdata, ierr;
    MPI_Isend(&sdata,1,MPI_INT,other,2,comm,&(request[0]));
    MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,comm,&(request[1]));
    return 0;
}

int library::communication_end() {
    MPI_Status status[2];
    int ierr;
    ierr = MPI_Waitall(2,request,status); CHK(ierr);
    return 0;
}
```

### 6.8.4 Listing of code examples/mpi/p/commdup.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

other = nprocs-procid-1
my_requests = [ None ] * 2
my_status = [ MPI.Status() ] * 2
sendbuffer = np.empty(1,dtype=np.int)
recvbuffer = np.empty(1,dtype=np.int)

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 communication_start(self):
    sendbuf = np.empty(1,dtype=np.int); sendbuf[0] = 37
    recvbuf = np.empty(1,dtype=np.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()

if my_status[1].Get_tag()==2:
    print("Caught wrong message!")
```

### 6.8.5 Listing of code examples/mpi/p/commsplit.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<4:
    print( "Need 4 procs at least")
    sys.exit(1)

mydata = procid

# communicator modulo 2
color = procid%2
mod2comm = comm.Split(color)
new_procid = mod2comm.Get_rank()

# communicator modulo 4 recursively
color = new_procid%2
mod4comm = mod2comm.Split(color)
new_procid = mod4comm.Get_rank()

if mydata/4!=new_procid:
    print("Error",procid,new_procid,mydata/4)

if procid==0:
    print("Finished")
```

### 6.8.6 Listing of code examples/mpi/mpl/commsplit.cxx

```
#include <iostream>
using std::cout;
using std::endl;
#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

    const mpl::communicator &comm_world=mpl::environment::comm_world();
    auto procno = comm_world.rank();
    auto nprocs = comm_world.size();

    // 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();

    int mod4ranks[nprocs];
    comm_world.gather( 0, procno4,mod4ranks );
    if (procno==0) {
        cout << "Ranks mod 4:";
        for (int ip=0; ip<nprocs; ip++)
            cout << " " << mod4ranks[ip];
        cout << endl;
    }

    if (procno/4!=procno4)
        printf("Error %d %d\n",procno,procno4);

    if (procno==0)
        printf("Finished\n");

    return 0;
}
```

### 6.8.7 Listing of code examples/mpi/c/intercomm.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

#ifndef DEBUG
#define DEBUG 0
#endif
```

```
int main(int argc,char **argv) {

#include "globalinit.c"

if (nprocs<4) {
    fprintf(stderr,"This program needs at least four processes\n");
    return -1;
}
if (nprocs%2>0) {
    fprintf(stderr,"This program needs an even number of processes\n");
    return -1;
}

int color,colors=2;
MPI_Comm split_half_comm;

int mydata = procno;
// create sub communicator first & second half
color = procno<nprocs/2 ? 0 : 1;
int key=procno;
if (color==0)
    // first half rotated
    key = ( procno+1 ) % (nprocs/2);
else
    // second half numbered backwards
    key = nprocs-procno;
MPI_Comm_split(MPI_COMM_WORLD,color,key,&split_half_comm);
int sub_procno,sub_nprocs;
MPI_Comm_rank(split_half_comm,&sub_procno);
MPI_Comm_size(split_half_comm,&sub_nprocs);
if (DEBUG) fprintf(stderr,"%d key=%d local rank: %d\n",procno,key,sub_procno);

int
local_leader_in_inter_comm
= color==0 ? 2 : (sub_nprocs-2)
,
local_number_of_other_leader
= color==1 ? 2 : (sub_nprocs-2)
;

if (local_leader_in_inter_comm<0 || local_leader_in_inter_comm>=sub_nprocs) {
    fprintf(stderr,
    "[%d] invalid local member: %d\n",
    procno,local_leader_in_inter_comm);
    MPI_Abort(2,comm);
}
int
global_rank_of_other_leader =
1 + ( procno<nprocs/2 ? nprocs/2 : 0 )
;

int
```

## 6. MPI topic: Communicators

---

```
i_am_local_leader = sub_procno==local_leader_in_inter_comm,
inter_tag = 314;
if (i_am_local_leader)
    fprintf(stderr, "[%d] creating intercomm with %d\n",
procno,global_rank_of_other_leader);
MPI_Comm intercomm;
MPI_Intercomm_create
    /* local_comm:          */ split_half_comm,
    /* local_leader:        */ local_leader_in_inter_comm,
    /* peer_comm:           */ MPI_COMM_WORLD,
    /* remote_peer_rank:   */ global_rank_of_other_leader,
    /* tag:                 */ inter_tag,
    /* newintercomm:         */ &intercomm );
if (DEBUG) fprintf(stderr, "[%d] intercomm created.\n",procno);

if (i_am_local_leader) {
    int inter_rank,inter_size;
    MPI_Comm_size(intercomm,&inter_size);
    MPI_Comm_rank(intercomm,&inter_rank);
    if (DEBUG) fprintf(stderr, "[%d] inter rank/size: %d/%d\n",procno,inter_rank,inter_size)
}

double interdata=0.;
if (i_am_local_leader) {
    if (color==0) {
        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);
    }
}

int root; int bcast_data = procno;
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);
fprintf(stderr, "[%d] bcast data: %d\n", procno,bcast_data);

if (procno==0)
    fprintf(stderr, "Finished\n");

MPI_Finalize();
return 0;
}
```

# Chapter 7

## 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.

### 7.1 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 6.6) is established between this new group and the existing one. The first routine is `MPI_Comm_spawn` (figure 7.1), 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 12.1.2), 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. 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.

Here is an example of a work manager. First we query how much space we have for new processes:

```
// spawnmanager.c
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
MPI_Comm_rank(MPI_COMM_WORLD, &manager_rank);

err = MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
                      (void*)&universe_sizep, &flag);
universe_size = *universe_sizep;
```

## 7.1 MPI\_Comm\_spawn

Semantics:

```
MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm,
                intercomm, array_of_errcodes)

IN command: name of program to be spawned
              (string, significant only at root)
IN argv: arguments to command
              (array of strings, significant only at root)
IN maxprocs: maximum number of processes to start
              (integer, significant only at root)
IN info: a set of key-value pairs telling the runtime system where and
          how to start the processes (handle, significant only at root)
IN root: rank of process in which previous arguments are examined
          (integer)
IN comm: intracommunicator containing group of spawning processes
          (handle)
OUT intercomm: intercommunicator between original group and the
               newly spawned group (handle)
OUT array_of_errcodes: one code per process (array of integer)
```

C:

```
int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
                   MPI_Info info, int root, MPI_Comm comm,
                   MPI_Comm *intercomm, int array_of_errcodes[])
```

Fortran:

```
MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
               array_of_errcodes, ierror)
CHARACTER(LEN=*) , INTENT(IN) :: command, argv(*)
INTEGER, INTENT(IN) :: maxprocs, root
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Comm), INTENT(OUT) :: intercomm
INTEGER :: array_of_errcodes(*)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

```
MPI.Intracomm.Spawn(self,
                     command, args=None, int maxprocs=1, Info info=INFO_NULL,
                     int root=0, errcodes=None)
returns an intracommunicator
```

## 7. MPI topic: Process management

---

For the source of this example, see section [7.3.2](#)

Then we actually spawn them:

```
int nworkers = universe_size-world_size;
const char *worker_program = "spawnworker";
int errorcodes[nworkers];
MPI_Comm inter_to_workers; /* intercommunicator */
MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, nworkers,
                MPI_INFO_NULL, 0, MPI_COMM_WORLD, &inter_to_workers,
                errorcodes);
```

For the source of this example, see section [7.3.2](#)

```
## 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

itercomm = comm.Spawn("./spawn_worker.py", maxprocs=nworkers)
```

For the source of this example, see section [7.3.3](#)

You could start up a single copy of this program with

```
mpirun -np 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:

```
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
```

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);
```

## 7.2 MPI\_Open\_port

```
C:
#include <mpi.h>
int MPI_Open_port(MPI_Info info, char *port_name)

Input parameters:
info : Options on how to establish an address (handle). No options currently supported.

Output parameters:
port_name : Newly established port (string).

||| MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
||| MPI_Comm_get_parent(&parent);
||| MPI_Comm_remote_size(parent, &remotesize);
||| if (workerno==0) {
|||   printf("Worker deduces %d workers and %d parents\n",nworkers,remotesize);
||| }
```

For the source of this example, see section 7.3.4

```
||| ## spawnworker.py
||| parentcomm = comm.Get_parent()
||| nparents = parentcomm.Get_remote_size()
```

For the source of this example, see section 7.3.5

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`. The number of spawning processes can be found through `MPI_Comm_remote_size` on the parent communicator (see section 6.6.3).

### 7.1.1 MPMD

Instead of spawning a single executable, you can spawn multiple with `MPI_Comm_spawn_multiple`.

## 7.2 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.

### 7.2.1 Server calls

The server calls `MPI_Open_port` (figure 7.2), 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`.

### 7.3 MPI\_Comm\_accept

Synopsis:

```
int MPI_Comm_accept
  (const char *port_name, MPI_Info info, int root,
   MPI_Comm comm, MPI_Comm *newcomm)
```

Input parameters:

port\_name : Port name (string, used only on root).  
info : Options given by root for the accept (handle, used only on root). No options currently supported.  
root : Rank in comm of root node (integer).  
comm : Intracommunicator over which call is collective (handle).

Output parameters:

newcomm : Intercommunicator with client as remote group (handle)

### 7.4 MPI\_Comm\_connect

Synopsis

```
int MPI_Comm_connect
  (const char *port_name, MPI_Info info, int root,
   MPI_Comm comm, MPI_Comm * newcomm)
```

Input Parameters

port\_name : network address (string, used only on root)  
info : implementation-dependent information (handle, used only on root)  
root : rank in comm of root node (integer)  
comm : intracommunicator over which call is collective (handle)

Output Parameters

newcomm : intercommunicator with server as remote group (handle)

The server then needs to call **MPI\_Comm\_accept** (figure 7.3) prior to the client doing a connect call. This is collective over the calling communicator. It returns an intercommunicator that allows communication with the client.

The port can be closed with **MPI\_Close\_port**.

## 7.2.2 Client calls

After the server has generated a port name, the client needs to connect to it with **MPI\_Comm\_connect** (figure 7.4), again specifying the port through a character buffer.

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**

The connect call is collective over its communicator.

### 7.5 MPI\_Publish\_name

Synopsis:

```
MPI_Publish_name(service_name, info, port_name)
```

Input parameters:

service\_name : a service name to associate with the port (string)

info : implementation-specific information (handle)

port\_name : a port name (string)

C:

```
int MPI_Publish_name  
    (char *service_name, MPI_Info info, char *port_name)
```

Fortran77:

```
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)  
INTEGER INFO, IERROR  
CHARACTER* (*) SERVICE_NAME, PORT_NAME
```

#### 7.2.3 Published service names

[MPI\\_Publish\\_name](#) (figure 7.5)

[MPI\\_Unpublish\\_name](#)

Unpublishing a non-existing or already unpublished service gives an error code of [MPI\\_ERR\\_SERVICE](#).

[MPI\\_Comm\\_join](#)

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.

### 7.3 Sources used in this chapter

#### 7.3.1 Listing of code header

#### 7.3.2 Listing of code examples/mpi/c/spawnmanager.c

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{

#define ASSERT(p) if (!(p)) {printf("Assertion failed for proc %d at line %d\n",procno,__LINE__)
#define ASSERTm(p,m) if (!(p)) {printf("Message<<%s>> for proc %d at line %d\n",m,procno,__LINE__)

    MPI_Comm comm;
    int procno=-1,nprocs,err;
    MPI_Init(&argc,&argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_rank(comm,&procno);
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN);

    /*
     * To investigate process placement, get host name
     */
{
    int namelen = MPI_MAX_PROCESSOR_NAME;
    char procname[namelen];
    MPI_Get_processor_name(procname,&namelen);
    printf("[%d] manager process runs on <<%s>>\n",procno,procname);
}

    int world_size,manager_rank, universe_size, *universe_sizep, flag;

    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &manager_rank);

    err = MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
                           (void*)&universe_sizep, &flag);
    if (err==MPI_ERR_KEYVAL) {
        printf("MPI_ERR_KEYVAL on proc %d\n",procno);
        MPI_Abort(comm,0); }

    if (!flag) {
        if (manager_rank==0) {
            printf("This MPI does not support UNIVERSE_SIZE.\nHow many processes total?");
            scanf("%d", &universe_size);
        }
        MPI_Bcast (&universe_size,1,MPI_INTEGER,0,MPI_COMM_WORLD);
    } else {
        universe_size = *universe_sizep;
```

```
    if (manager_rank==0)
        printf("Universe size deduced as %d\n",universe_size);
    }
ASSERTm(universe_size>world_size,"No room to start workers");

int nworkers = universe_size-world_size;

/*
 * Now spawn the workers. Note that there is a run-time determination
 * of what type of worker to spawn, and presumably this calculation must
 * be done at run time and cannot be calculated before starting
 * the program. If everything is known when the application is
 * first started, it is generally better to start them all at once
 * in a single MPI_COMM_WORLD.
 */

if (manager_rank==0)
    printf("Now spawning %d workers\n",nworkers);
const char *worker_program = "spawnworker";
int errorcodes[nworkers];
MPI_Comm inter_to_workers;           /* intercommunicator */
MPI_Spawn(worker_program, MPI_ARGV_NULL, nworkers,
MPI_INFO_NULL, 0, MPI_COMM_WORLD, &inter_to_workers,
errorcodes);
for (int ie=0; ie<nworkers; ie++)
    if (errorcodes[ie]!=0)
        printf("Error %d in spawning worker %d\n",errorcodes[ie],ie);

/*
 * Parallel code here. The communicator "inter_to_workers" can be used
 * to communicate with the spawned processes, which have ranks 0,..
 * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
 * "inter_to_workers".
 */

MPI_Finalize();
return 0;
}
```

### 7.3.3 Listing of code examples/mpi/p/spawnmanager.py

```
import numpy as np
import random # random.randint(1,N), random.random()
import sys

from mpi4py import MPI

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

```

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

itercomm = comm.Spawn("./spawn_worker.py", maxprocs=nworkers)

```

### 7.3.4 Listing of code examples/mpi/c/spawnworker.c

```

#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
#define ASSERT(p) if (!(p)) {printf("Assertion failed for proc %d at line %d\n",procno,__LINE__)
#define ASSERTm(p,m) if (!(p)) {printf("Message<<%s>> for proc %d at line %d\n",m,procno,__LINE__)

    MPI_Comm comm;
    int procno=-1,nprocs,err;
    MPI_Init(&argc,&argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_rank(comm,&procno);
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN);

    int remotesize,nworkers,workerno;
    MPI_Comm parent;

    MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
    MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
    MPI_Comm_get_parent (&parent);
    ASSERTm(parent!=MPI_COMM_NULL,"No parent!");

    /*
     * To investigate process placement, get host name
     */
    {
        int namelen = MPI_MAX_PROCESSOR_NAME;
        char procname[namelen];
        MPI_Get_processor_name(procname,&namelen);
        printf("[%d] worker process runs on <<%s>>\n",workerno,procname);
    }
}

```

```
}

MPI_Comm_remote_size(parent, &remotesize);
if (workerno==0) {
    printf("Worker deduces %d workers and %d parents\n",nworkers,remotesize);
}
// ASSERT(nworkers==size-1,"nworkers mismatch. probably misunderstanding");

/*
 * Parallel code here.
 * The manager is represented as the process with rank 0 in (the remote
 * group of) MPI_COMM_PARENT. If the workers need to communicate among
 * themselves, they can use MPI_COMM_WORLD.
 */

char hostname[256]; int namelen = 256;
MPI_Get_processor_name(hostname,&namelen);
printf("worker %d running on %s\n",workerno,hostname);

MPI_Finalize();
return 0;
}
```

### 7.3.5 Listing of code examples/mpi/p/spawnworker.py

```
import numpy as np
import random # random.randint(1,N), random.random()

from mpi4py import MPI

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

if procid==0:
    print("#workers:",nprocs)

parentcomm = comm.Get_parent()
nparents = parentcomm.Get_remote_size()

print("#parents=",nparents)
```

## Chapter 8

### 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, also known as Remote Direct Memory Access (RDMA) or Remote Memory Access (RMA) operations, 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 user-space 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.

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. The MPI RMA model makes it possible to lock a window which makes programming slightly more cumbersome, but the implementation more efficient.

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

### 8.1 MPI\_Win

```
C:
MPI_Win win ;
```

Fortran:

```
Type(MPI_Win) :: win
```

small transfers, which are aggregated behind the scenes. 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 strange deadlocks.

## 8.1 Windows

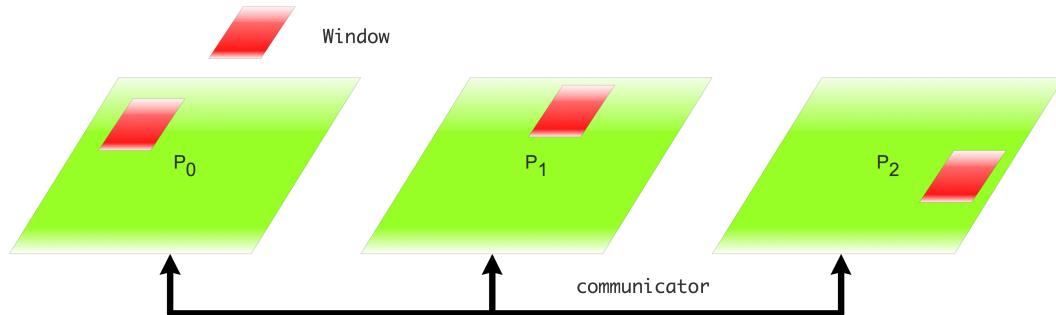


Figure 8.1: Collective definition of a window for one-sided data access

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](#) (figure 8.1). 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 8.1.
- 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 8.2.
- There can be memory associated with a window, so it needs to be freed explicitly.

The typical calls involved are:

```
|| MPI_Info info;
|| MPI_Win window;
|| MPI_Win_allocate( /* size info */, info, comm, &memory, &window );
// do put and get calls
|| MPI_Win_free( &window );
```

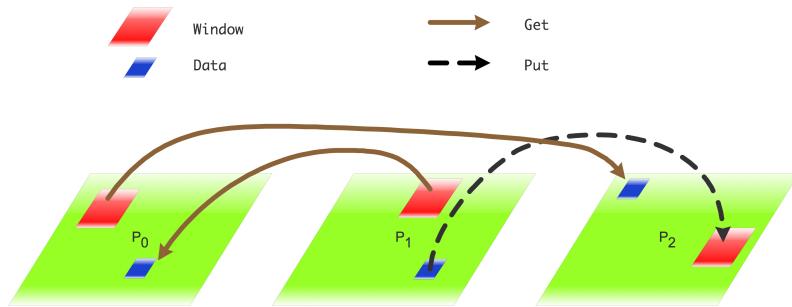


Figure 8.2: Put and get between process memory and windows

### 8.1.1 Window creation and allocation

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 8.2). This buffer can be an ordinary array, or it can be created with `MPI_Alloc_mem`.
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 8.3) to create the data and the window in one call.
  - If a communicator is on a shared memory (see section 11.1) you can create a window in that shared memory with `MPI_Win_allocate_shared`. This will be useful for *MPI shared memory*; see chapter 11.
3. Finally, you can create a window with `MPI_Win_create_dynamic` which postpones the allocation; see section 8.5.2.

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 is easily done with the `sizeof` operator;

```
|| // putfencealloc.c
|| MPI_Win the_window;
|| int *window_data;
```

## 8.2 MPI\_Win\_create

```
C:
int MPI_Win_create
    (void *base, MPI_Aint size, int disp_unit,
     MPI_Info info, MPI_Comm comm, MPI_Win *win)

Fortran:
MPI_Win_create(base, size, disp_unit, info, comm, win, ierror)
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
INTEGER, INTENT(IN) :: disp_unit
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Win), INTENT(OUT) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Python:
MPI.Win.Create
    (memory, int disp_unit=1,
     Info info=INFO_NULL, Intracomm comm=COMM_SELF)
```

## 8.3 MPI\_Win\_allocate

Semantics:

```
MPI_WIN_ALLOCATE(size, disp_unit, info, comm, baseptr, win)
```

Input parameters:

- size: size of local window in bytes (non-negative integer)
- disp\_unit local unit size for displacements, in bytes (positive integer)
- info: info argument (handle)
- comm: intra-communicator (handle)

Output parameters:

- baseptr: address of local allocated window segment (choice)
- win: window object returned by the call (handle)

C:

```
int MPI_Win_allocate
    (MPI_Aint size, int disp_unit, MPI_Info info,
     MPI_Comm comm, void *baseptr, MPI_Win *win)
```

Fortran:

```
MPI_Win_allocate
    (size, disp_unit, info, comm, baseptr, win, ierror)
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
INTEGER, INTENT(IN) :: disp_unit
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(C_PTR), INTENT(OUT) :: baseptr
TYPE(MPI_Win), INTENT(OUT) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

#### 8.4 MPI\_Alloc\_mem

```
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
```

```
|| MPI_Win_allocate(2*sizeof(int), sizeof(int),
|| MPI_INFO_NULL, comm,
|| &window_data, &the_window);
```

*For the source of this example, see section 8.9.2*

for doing this calculation in Fortran, see section 12.3.1.

*Python note.* For computing the displacement in bytes, here is a good way for finding the size of `numpy` datatypes:

```
## putfence.py
intsize = np.dtype('int').itemsize
window_data = np.zeros(2, dtype=np.int)
win = MPI.Win.Create(window_data, intsize, comm=comm)
```

*For the source of this example, see section 8.9.3*

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 prototype; 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 8.4) performs only the allocation part of `MPI_Win_allocate`, after which you need to `MPI_Win_create`:

This memory is freed with

```
|| MPI_Free_mem()
```

These calls reduce to `malloc` and `free` if there is no special memory area; SGI is an example where such memory does exist.

There will be more discussion of window memory in section 8.5.1.

*Python note.* Unlike in C, the python window allocate call does not return a pointer to the buffer memory. Should you need this, 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.attrs
|| 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:

### 8.5 MPI\_Win\_fence

Semantics:

```
MPI_WIN_FENCE(assert, win)
IN assert: program assertion (integer)
IN win: window object (handle)
```

C:

```
int MPI_Win_fence(int assert, MPI_Win win)
```

F:

```
MPI_Win_fence(assert, win, ierror)
INTEGER, INTENT(IN) :: assert
TYPE(MPI_Win), INTENT(IN) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

```
win.Fence(self, int assertion=0)
```

```
|| mem = win.tomemory() # type(mem) is MPI.memory, similar to
||   memoryview, but quite limited in functionality
|| base = mem.address
|| size = mem.nbytes
```

## 8.2 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 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 8.5). This operation is collective on the communicator of the window. It is comparable to **`MPI_Wait`** calls for non-blocking communication.

The use of fences is somewhat complicated. The interval between two fences is known as an *epoch*. You can give various hints to the system about this epoch versus the ones before and after through the `assert` parameter.

```
|| MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
|| MPI_Get /* operands */, win;
|| MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
```

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:

## 8. MPI topic: One-sided communication

---

```
|| 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.

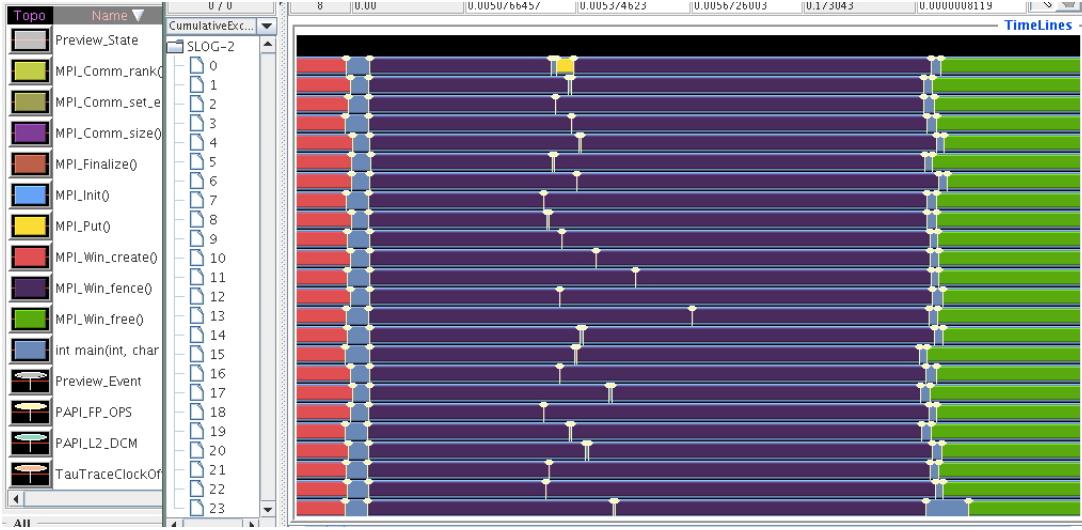


Figure 8.3: A trace of a one-sided communication epoch where process zero only originates a one-sided transfer

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.

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 8.6.

### 8.3 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 8.3.8.) 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 Put/Get/Accumulate routines have an `MPI_Op` argument that can be any of the usual operators, but no user-defined ones (see section 3.10.1).

## 8.6 MPI\_Put

```
C:  
int MPI_Put(  
    const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,  
    int target_rank,  
    MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,  
    MPI_Win win)  
  
Semantics:  
IN origin_addr: initial address of origin buffer (choice)  
IN origin_count: number of entries in origin buffer (non-negative integer)  
IN origin_datatype: datatype of each entry in origin buffer (handle)  
IN target_rank: rank of target (non-negative integer)  
IN target_disp: displacement from start of window to target buffer (non-negative integer)  
IN target_count: number of entries in target buffer (non-negative integer)  
IN target_datatype: datatype of each entry in target buffer (handle)  
IN win: window object used for communication (handle)  
  
Fortran:  
MPI_Put(origin_addr, origin_count, origin_datatype,  
        target_rank, target_disp, target_count, target_datatype, win, ierror)  
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr  
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count  
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype  
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp  
TYPE(MPI_Win), INTENT(IN) :: win  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror  
  
Python:  
win.Put(self, origin, int target_rank, target=None)
```

### 8.3.1 Put

The **MPI\_Put** (figure 8.6) 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.

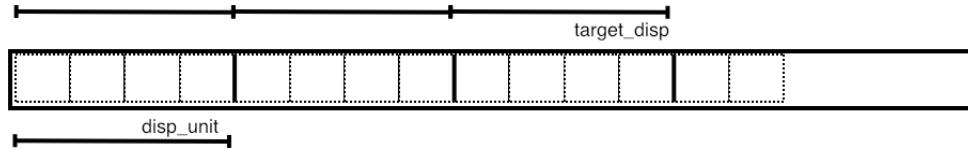
The description of the data on the origin is the usual trio of buffer/count/datatype. However, the description of the data on the target is more complicated. It has a count and a datatype, but instead of an address it has a *displacement* with respect to the start of the window on the target. This displacement 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

$$\text{window\_base} + \text{target\_disp} \times \text{disp\_unit}.$$

## 8. MPI topic: One-sided communication

---



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),
     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);
```

For the source of this example, see section 8.9.4

Fortran note. 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)
```

For the source of this example, see section ??

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
```

**Exercise 8.1.** Revisit exercise 4.7 and solve it using MPI\_Put.

**Exercise 8.2.** Write code where process 0 randomly writes in the window on 1 or 2.

### 8.3.2 Get

The MPI\_Get (figure 8.7) call is very similar.

Example:

## 8.7 MPI\_Get

C:

```
int MPI_Get(
    const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
    int target_rank,
    MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,
    MPI_Win win)
```

Semantics:

```
IN origin_addr: initial address of origin buffer (choice)
IN origin_count: number of entries in origin buffer (non-negative integer)
IN origin_datatype: datatype of each entry in origin buffer (handle)
IN target_rank: rank of target (non-negative integer)
IN target_disp: displacement from start of window to target buffer (non-negative integer)
IN target_count: number of entries in target buffer (non-negative integer)
IN target_datatype: datatype of each entry in target buffer (handle)
IN win: window object used for communication (handle)
```

Fortran:

```
MPI_Get(origin_addr, origin_count, origin_datatype,
        target_rank, target_disp, target_count, target_datatype, win, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
TYPE(MPI_Win), INTENT(IN) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

```
win.Get(self, origin, int target_rank, target=None)
```

## 8. MPI topic: One-sided communication

---

```
// getfence.c
MPI_Win_create(&other_number, 2*sizeof(int), sizeof(int),
                MPI_INFO_NULL, comm, &the_window);
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);
```

For the source of this example, see section 8.9.5

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()
```

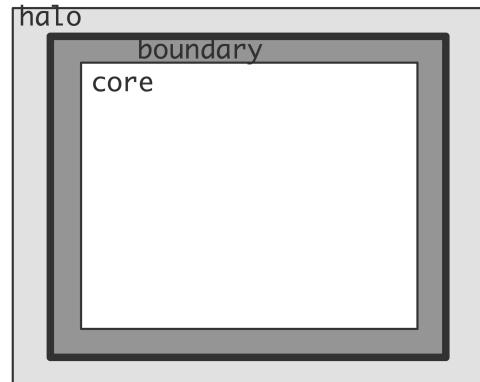
For the source of this example, see section 8.9.6

### 8.3.3 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 NOPRECEDE and NOSUCCEED assertions. The 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 < toneighbors; i++)
    |||     MPI_Put( ... );
    ||| MPI_Win_fence( MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
}

```

For much more about assertions, see section 8.6 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 NOPRECEDE and NOSUCCEED assertions still hold, but the Get operation implies that instead of NOSTORE in the second fence, we use NOPUT in the first.

### 8.3.4 Accumulate

A third one-sided routine is **MPI\_Accumulate** (figure 8.8) 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 gives 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 8.3.** 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.

### Exercise 8.4.

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?

### 8.8 MPI\_Accumulate

```
C:
int MPI_Accumulate
  (const void *origin_addr, int origin_count,MPI_Datatype origin_datatype,
   int target_rank,MPI_Aint target_disp, int target_count,MPI_Datatype target_datatype,
   MPI_Op op, MPI_Win win)
int MPI_Raccumulate
  (const void *origin_addr, int origin_count,MPI_Datatype origin_datatype,
   int target_rank,MPI_Aint target_disp, int target_count,MPI_Datatype target_datatype,
   MPI_Op op, MPI_Win win,MPI_Request *request)

Input Parameters

origin_addr : Initial address of buffer (choice).
origin_count : Number of entries in buffer (nonnegative integer).
origin_datatype : Data type of each buffer entry (handle).
target_rank : Rank of target (nonnegative integer).
target_disp : Displacement from start of window to beginning of target buffer (nonnegative
target_count : Number of entries in target buffer (nonnegative integer).
target_datatype : Data type of each entry in target buffer (handle).
op : Reduce operation (handle).
win : Window object (handle).

Output Parameter

MPI_Raccumulate: RMA request
IERROR (Fortran only): Error status (integer).

Fortran:

MPI_ACCUMULATE
  (ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
   TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE,
   OP, WIN, IERROR)
<type> ORIGIN_ADDR(*)
INTEGER(KIND=MPI_ADDRESS_KIND) :: TARGET_DISP
INTEGER :: ORIGIN_COUNT, ORIGIN_DATATYPE,
            TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE,
            OP, WIN, IERROR
MPI_RACCUMULATE
  (ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
   TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE,
   OP, WIN, REQUEST, IERROR)
<type> ORIGIN_ADDR(*)
INTEGER(KIND=MPI_ADDRESS_KIND) :: TARGET_DISP
INTEGER :: ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE,
            OP, WIN, REQUEST, IERROR

Python:
MPI.Win.Accumulate(self, origin, int target_rank, target=None, Op op=SUM)
```

## 8.9 MPI\_Rput

```
C:
int MPI_Rput(
    const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
    int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,
    MPI_Win win, MPI_Request *request)

Semantics:
IN origin_addr: initial address of origin buffer (choice)
IN origin_count: number of entries in origin buffer (non-negative integer)
IN origin_datatype: datatype of each entry in origin buffer (handle)
IN target_rank: rank of target (non-negative integer)
IN target_disp: displacement from start of window to target buffer (non-negative integer)
IN target_count: number of entries in target buffer (non-negative integer)
IN target_datatype: datatype of each entry in target buffer (handle)
IN win: window object used for communication (handle)
OUT request: RMA request (handle)
```

### 8.3.5 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 8.5.3.

### 8.3.6 Request-based operations

Analogous to `MPI_Isend` there are request based one-sided operations: `MPI_Rput` (figure 8.9) 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.

### 8.3.7 More active target synchronization

The ‘fence’ mechanism (section 8.2) uses a global synchronization on the communicator of the window. 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. 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`:

## 8. MPI topic: One-sided communication

---

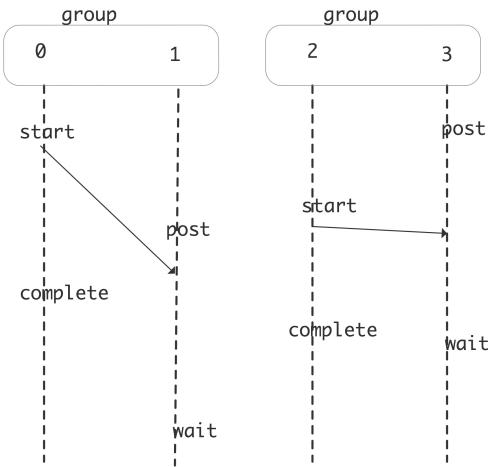


Figure 8.4: Window locking calls in fine-grained active target synchronization

```

|| int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
|| int MPI_Win_wait(MPI_Win win)

```

In other words, this turns your window into the *target* for a remote access.

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
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);
}

```

For the source of this example, see section 8.9.7

Both pairs of operations declare a *group of processors*; see section 6.5.1 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.

### 8.3.8 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 inconsistent views of the data are called a *race condition*.

In MPI 3 some atomic routines have been added. Both `MPI_Fetch_and_op` (figure 8.10) and `MPI_Get_accumulate` (figure 8.11) 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 8.5.** Redo exercise 8.4 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?

**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;
```

### 8.10 MPI\_Fetch\_and\_op

Semantics:

```
MPI_FETCH_AND_OP(origin_addr, result_addr, datatype, target_rank,
                  target_disp, op, win)
IN origin_addr: initial address of buffer (choice)
OUT result_addr: initial address of result buffer (choice)
IN datatype: datatype of the entry in origin, result, and target buffers
(handle)
IN target_rank: rank of target (non-negative integer)
IN target_disp: displacement from start of window to beginning of target
buffer (non-negative integer)
IN op: reduce operation (handle)
IN win: window object (handle)
```

C:

```
int MPI_Fetch_and_op
      (const void *origin_addr, void *result_addr,
       MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,
       MPI_Op op, MPI_Win win)
```

Fortran:

```
MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,
                  target_disp, op, win, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(IN) :: target_rank
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Win), INTENT(IN) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**8.11 MPI\_Get\_accumulate**

```
C:
int MPI_Get_accumulate
  (const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
   void *result_addr, int result_count, MPI_Datatype result_datatype,
   int target_rank,
   MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,
   MPI_Op op, MPI_Win win)

Input Parameters
origin_addr : initial address of buffer (choice)
origin_count : number of entries in buffer (nonnegative integer)
origin_datatype : datatype of each buffer entry (handle)

result_addr : initial address of result buffer (choice)
result_count : number of entries in result buffer (non-negative integer)
result_datatype : datatype of each entry in result buffer (handle)
target_rank : rank of target (nonnegative integer)
target_disp : displacement from start of window to beginning
  of target buffer (nonnegative integer)
target_count : number of entries in target buffer (nonnegative integer)
target_datatype : datatype of each entry in target buffer (handle)
op : predefined reduce operation (handle)
win : window object (handle)

    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);
}
```

For the source of this example, see section ??

```
## 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)
```

### 8.12 MPI\_Compare\_and\_swap

```
C:
int MPI_Compare_and_swap
    (const void *origin_addr, const void *compare_addr,
     void *result_addr, MPI_Datatype datatype,
     int target_rank, MPI_Aint target_disp,
     MPI_Win win)

Input Parameters

origin_addr : initial address of buffer (choice)
compare_addr : initial address of compare buffer (choice)
result_addr : initial address of result buffer (choice)
datatype : datatype of the entry in origin, result, and target buffers (handle)
target_rank : rank of target (nonnegative integer)
target_disp : displacement from start of window to beginning
              of target buffer (non-negative integer)
win : window object (handle)

||      win.Unlock( repository )
```

*For the source of this example, see section ??*

Finally, **MPI\_Compare\_and\_swap** (figure 8.12) swaps the origin and target data if the target data equals some comparison value.

#### 8.3.8.1 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:

```
// countdownop.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);
```

*For the source of this example, see section 8.9.8*

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) {
    my_counter_values[ n_my_counter_values++ ] = counter_value;
    total_decrement++;
    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);

```

For the source of this example, see section 8.9.9

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 8.6.** 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) {
    my_counter_values[n_my_counter_values++] = counter_value;
    total_decrement++;
    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);

```

For the source of this example, see section 8.9.10

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**:

### 8.13 MPI\_Win\_lock

C:

```
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
```

Input Parameters:

lock\_type - Indicates whether other processes may access the target window at the same time (if MPI\_LOCK\_SHARED) or not (MPI\_LOCK\_EXCLUSIVE)  
 rank - rank of locked window (nonnegative integer)  
 assert - Used to optimize this call; zero may be used as a default. (integer)  
 win - window object (handle)

Python:

```
MPI.Win.Lock(self,
             int rank, int lock_type=LOCK_EXCLUSIVE, int assertion=0)
```

```
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;
}
```

For the source of this example, see section 8.9.8

Now, if there are multiple accesses, each retrieves the counter value and updates it in one atomic, that is, indivisible, action.

## 8.4 Passive target synchronization

In *passive target synchronization* only the origin is actively involved: the target makes no calls whatsoever. 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 8.13) :

```
if (rank == 0) {
    MPI_Win_lock (MPI_LOCK_EXCLUSIVE, 1, 0, win);
```

### 8.14 MPI\_Win\_unlock

C:

```
Py:
MPI.Win.Unlock(self, int rank)
MPI.Win.Unlock_all(self)
```

```
|| MPI_Put (outbuf, n, MPI_INT, 1, 0, n, MPI_INT, win);
|| MPI_Win_unlock (1, win);
|| }
```

#### 8.4.1 Lock types

A lock is needed for an origin to acquire the capability to access a target. You can either acquire a lock on a specific rank with **MPI\_Win\_lock**, or on all ranks (of a communicator) with **MPI\_Win\_lock\_all**. Unlike **MPI\_Win\_fence**, this is not a collective call.

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.

To unlock a window, use **MPI\_Win\_unlock** (figure 8.14), respectively **MPI\_Win\_unlock\_all** (figure 8.14).

**Exercise 8.7.** 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 ranks but one update a window:

```
int one=1;
MPI_Fetch_and_op(&one, &readout,
                  MPI_INT, repo, zero_disp, MPI_SUM,
                  the_win);
```

- while the remaining rank spins until the others have performed their update.

Use an atomic operation for the latter rank to read out the shared value.

**Exercise 8.8.** As exercise 8.7, 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**.

#### 8.4.2 Lock all

To lock the windows of all processes in the group of the windows, use **MPI\_Win\_lock\_all** (figure 8.15).

### 8.15 `MPI_Win_lock_all`

C:

```
int MPI_Win_lock( int assert, MPI_Win win)
```

Input Parameters:

assert - Used to optimize this call; zero may be used as a default. (integer)  
win - window object (handle)

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` (figure 8.14).

The expected use of a ‘lock/unlock all’ is that they surround an extended epoch with get/put and flush calls.

#### 8.4.3 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 a Put or Get call. For instance, after a 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*.
- The window data. While this may be publicly accessible, it is not necessarily always consistent with internal copies.
- The remote data. Even a successful Put does not guarantee that the other process has received the data. A successful transfer is a *remote completion*.

As observed, RMA operations are non-blocking, 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 11** 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

##### 8.4.3.1 Local completion

The call `MPI_Win_flush_local` (figure 8.16) ensure that all operations with a given target is completed

### 8.16 MPI\_Win\_flush\_local

Synopsis:

`MPI_WIN_FLUSH_LOCAL(rank, win)`

Input arguments:

rank: rank of target window (non-negative integer)  
win : window object (handle)

C:

```
int MPI_Win_flush_local(int rank, MPI_Win win)
```

Fortran:

```
MPI_Win_flush_local(rank, win, ierror)
INTEGER, INTENT(IN) :: rank
TYPE(MPI_Win), INTENT(IN) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
INTEGER RANK, WIN, IERROR
```

Synopsis:

`MPI_WIN_FLUSH_LOCAL_ALL(win)`

Input arguments:

win : window object (handle)

C:

```
int MPI_Win_flush_local_all(MPI_Win win)
```

Fortran:

```
MPI_Win_flush_local_all(win, ierror)
TYPE(MPI_Win), INTENT(IN) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
INTEGER WIN, IERROR
```

### 8.17 MPI\_Win\_flush

Synopsis

`MPI_WIN_FLUSH(rank, win)`

Input arguments:

`rank` : rank of target window (non-negative integer)

`win` : window object (handle)

C:

```
int MPI_Win_flush(int rank, MPI_Win win)
```

Fortran:

```
MPI_Win_flush(rank, win, ierror)
INTEGER, INTENT(IN) :: rank
TYPE(MPI_Win), INTENT(IN) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_WIN_FLUSH(RANK, WIN, IERROR)
INTEGER RANK, WIN, IERROR
```

Synopsis:

`MPI_WIN_FLUSH_ALL(win)`

Input arguments:

`win` : window object (handle)

C:

```
int MPI_Win_flush_all(MPI_Win win)
```

Fortran:

```
MPI_Win_flush_all(win, ierror)
TYPE(MPI_Win), INTENT(IN) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_WIN_FLUSH_ALL(WIN, IERROR)
INTEGER WIN, IERROR
```

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 8.4.2).

#### 8.4.3.2 Remote completion

The calls `MPI_Win_flush` (figure 8.17) 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`.

#### 8.4.3.3 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.

**8.18 MPI\_Win\_create\_dynamic**

```

int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)

Input Parameters
info : info argument (handle)
comm : communicator (handle)

Output Parameters
win : window object returned by the call (handle)

```

**8.5 More about window memory****8.5.1 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 8.1.1). 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**.
- 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.

(Window models can be queried as attributes; see section 8.5.4.)

**8.5.2 Dynamically attached memory**

In section 8.1.1 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 8.18) and attach memory to the window with **MPI\_Win\_attach** (figure 8.19).

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));

```

*For the source of this example, see section 8.9.11*

(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,

### 8.19 MPI\_Win\_attach

Semantics:

```
MPI_Win_attach(win, base, size)
```

Input Parameters:

```
win : window object (handle)
base : initial address of memory to be attached
size : size of memory to be attached in bytes
```

C:

```
int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)
```

Fortran:

```
MPI_Win_attach(win, base, size, ierror)
TYPE(MPI_Win), INTENT(IN) :: win
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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_LONG,data_proc,comm);
```

For the source of this example, see section [8.9.11](#)

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);
```

For the source of this example, see section [8.9.11](#)

Notes.

- The attached memory can be released with **`MPI_Win_detach`** (figure [8.20](#)).
- 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.

#### 8.5.3 Window usage hints

The following keys can be passed as info argument:

## 8.20 MPI\_Win\_detach

Semantics:

```
MPI_Win_detach(win, base)
```

Input parameters:

win : window object (handle)

base : initial address of memory to be detached

C:

```
int MPI_Win_detach(MPI_Win win, const void *base)
```

Fortran:

```
MPI_Win_detach(win, base, ierror)
TYPE(MPI_Win), INTENT(IN) :: win
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- *no\_locks*: if set to true, passive target synchronization (section 8.4) 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`.

### 8.5.4 Window information

The `MPI_Info` parameter can be used to pass implementation-dependent information; see section 12.1.1.

A number of attributes are stored with a window when it is created.

Obtaining a pointer to the start of the window area:

```
|| void *base;
|| MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag)
```

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),
```

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`;

## 8. MPI topic: One-sided communication

---

- `MPI_WIN_FLAVOR_ALLOCATE` if the window was created with `MPI_Win_allocate`;
- `MPI_WIN_FLAVOR_DYNAMIC` if the window was created 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 created with `MPI_Win_allocate_shared`;

The window model:

```
|| int *memory_model;
|| MPI_Win_get_attr(win, MPI_WIN_MODEL, &memory_model, &flag);
```

with possible values:

- `MPI_WIN_SEPARATE`,
- `MPI_WIN_UNIFIED`,

Get the group of processes associated with a window:

```
int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
MPI_Win_get_group(win, group, ierror)
TYPE(MPI_Win), INTENT(IN) :: win
TYPE(MPI_Group), INTENT(OUT) :: group
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

int MPI_Win_set_info(MPI_Win win, MPI_Info info)
MPI_Win_set_info(win, info, ierror)
TYPE(MPI_Win), INTENT(IN) :: win
TYPE(MPI_Info), INTENT(IN) :: info
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)
MPI_Win_get_info(win, info_used, ierror)
TYPE(MPI_Win), INTENT(IN) :: win
TYPE(MPI_Info), INTENT(OUT) :: info_used
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

## 8.6 Assertions

The `MPI_Win_fence` call, as well `MPI_Win_start` and such, 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.

## 8.7 Implementation

You may wonder how one-sided communication is realized<sup>1</sup>. 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.

---

1. For more on this subject, see [16].

**Exercise 8.9.** 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 8.10.** 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.

## 8.8 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;
}
```

## 8.9 Sources used in this chapter

### 8.9.1 Listing of code header

### 8.9.2 Listing of code examples/mpi/c/examples/putfencealloc.c

### 8.9.3 Listing of code examples/mpi/p/putfence.py

```
import numpy as np
import random # random.randint(1,N), random.random()
import sys

from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

intsize = np.dtype('int').itemsize
window_data = np.zeros(2,dtype=np.int)
win = MPI.Win.Create(window_data,intsize,comm=comm)

my_number = np.empty(1,dtype=np.int)
src = 0; tgt = nprocs-1
if procid==src:
    my_number[0] = 37
else:
    my_number[0] = 1
win.Fence()
if procid==src:
    # put data in the second element of the window
    win.Put(my_number,tgt,target=1)
win.Fence()

if procid==tgt:
    print("Window after put:",window_data)
```

### 8.9.4 Listing of code examples/mpi/c/putfence.c

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

```
#include <unistd.h>

int main(int argc,char **argv) {

#include "globalinit.c"

MPI_Win the_window;
int my_number=0, window_data[2], other = nprocs-1;
if (procno==0)
    my_number = 37;

MPI_Win_create
(&window_data,2*sizeof(int),sizeof(int),
 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);
if (procno==other)
    printf("I got the following: %d\n",window_data[1]);
MPI_Win_free(&the_window);

MPI_Finalize();
return 0;
}
```

### 8.9.5 Listing of code examples/mpi/c/getfence.c

```
#include <stdlib.h>
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc,char **argv) {

#include "globalinit.c"

{
    MPI_Win the_window;
    int my_number, other_number[2], other = nprocs-1;
    if (procno==other)
        other_number[1] = 27;
    MPI_Win_create(&other_number,2*sizeof(int),sizeof(int),
                  MPI_INFO_NULL,comm,&the_window);
    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);
if (procno==0)
    printf("I got the following: %d\n",my_number);
MPI_Win_free(&the_window);
}

MPI_Finalize();
return 0;
}
```

### 8.9.6 Listing of code examples/mpi/p/getfence.py

```
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

other = nprocs-1-procid
mydata = random.random()

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()

# see what you got
if procid==0 or procid==nprocs-1:
    print("[%d] getting %e" % (procid,win_mem[0]))

win.Free()
```

### 8.9.7 Listing of code examples/mpi/c/postwaitwin.c

```
#include <stdlib.h>
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc,char **argv) {

#include "globalinit.c"

{
    MPI_Win the_window;
    MPI_Group all_group,two_group;
    int my_number = 37, other_number,
        twotids[2],origin,target;

    MPI_Win_create(&other_number,1,sizeof(int),
                   MPI_INFO_NULL,comm,&the_window);
    if (procno>0 && procno<nprocs-1) goto skip;

    origin = 0; target = nprocs-1;
    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);
    }
    if (procno==target)
        printf("Got the following: %d\n",other_number);

    MPI_Group_free(&all_group);
    MPI_Group_free(&two_group);
skip:
    MPI_Win_free(&the_window);
}

MPI_Finalize();
return 0;
}
```

**8.9.8 Listing of code examples/mpi/c/countdownnop.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#include <mpi.h>
#include "gather_sort_print.h"

int main(int argc, char **argv) {

    int nprocs, procno;
    MPI_Init(0,0);
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm, &nprocs);
    MPI_Comm_rank(comm, &procno);

    if (nprocs<2) {
        printf("Need at least 2 procs\n");
        MPI_Abort(comm,0);
    }

    // first set a unique random seed
    srand(procno*time(0));

    {
        /*
         * Create a window.
         * We only need a nonzero size on the last process,
         * which we label the 'counter_process';
         * everyone else makes a window of size zero.
         */
        MPI_Win the_window;
        int counter_process = nprocs-1;
        int window_data, check_data;
        if (procno==counter_process) {
            window_data = 2*nprocs-1;
            check_data = window_data;
            MPI_Win_create(&window_data,sizeof(int),sizeof(int),
                           MPI_INFO_NULL,comm,&the_window);
        } else {
            MPI_Win_create(&window_data,0,sizeof(int),
                           MPI_INFO_NULL,comm,&the_window);
        }
        /*
         * Initialize the window
         * - PROCWRITES is approx the number of writes we want each process to do
         * - COLLISION is approx how many processes will collide on a write
         */
        #ifndef COLLISION
        #define COLLISION 2
        #endif
        #ifndef PROCWRITES
        #define PROCWRITES 40
```

```

#endif
    int counter_init = nprocs * PROCWRITES;
    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);

    /*
     * Allocate an array (grossly over-dimensioned)
     * for the counter values that belong to me
     */
    int *my_counter_values = (int*) malloc( counter_init * sizeof(int) );
    if (!my_counter_values) {
        printf("[%d] could not allocate counter values\n",procno);
        MPI_Abort(comm,0);
    }
    int n_my_counter_values = 0;

    /*
     * Loop:
     * - at random times update the counter on the counter process
     * - and read out the counter to see if we stop
     */
    int total_decrement = 0;
    int nsteps = PROCWRITES / COLLISION;
    if (procno==0)
        printf("Doing %d steps, counter starting: %d\n      probably %d-way collision on each s",
               nsteps,counter_init,COLLISION);
    for (int step=0; step<nsteps ; step++) {

        /*
         * Basic probability of a write is 1/P,
         * so each step only one proc will write.
         * Increase chance of collision by upping
         * the value of COLLISION.
         */
        float randomfraction = (rand() / (double)RAND_MAX);
        int i_am_available = randomfraction < ( COLLISION * 1./nprocs );

        /*
         * Exercise:
         * - atomically read and decrement the counter
         */
        MPI_Win_fence(0,the_window);
        int
            counter_value;
        if (i_am_available) {
            int
                decrement = -1;
            total_decrement++;
            MPI_Fetch_and_op

```

## 8. MPI topic: One-sided communication

---

```
( /* operate with data from origin: */ &decrement,
  /* retrieve data from target: */ &counter_value,
  MPI_INT, counter_process, 0, MPI_SUM,
  the_window);

#ifdef DEBUG
    printf("[%d] updating in step %d; retrieved %d\n", procno, step, counter_value);
#endif
}

MPI_Win_fence(0, the_window);
if (i_am_available) {
    my_counter_values[n_my_counter_values++] = counter_value;
}
}

/*
 * What counter values were actually obtained?
 */
gather_sort_print( my_counter_values, n_my_counter_values, comm );

/*
 * We do a correctness test by computing what the
 * window_data is supposed to be
 */
{
    MPI_Win_fence(0, the_window);
    int counter_value;
    MPI_Get( /* origin data to set: */ &counter_value, 1, MPI_INT,
            /* window data to get: */ counter_process, 0, 1, MPI_INT,
            the_window);
    MPI_Win_fence(0, the_window);
    MPI_Allreduce(MPI_IN_PLACE, &total_decrement, 1, MPI_INT, MPI_SUM, comm);
    if (procno==counter_process) {
        if (counter_init-total_decrement==counter_value)
            printf("[%d] initial counter %d decreased by %d correctly giving %d\n",
                   procno, counter_init, total_decrement, counter_value);
        else
            printf("[%d] initial counter %d decreased by %d, giving %d s/b %d\n",
                   procno, counter_init, total_decrement, counter_value, counter_init-total_decrement);
    }
    MPI_Win_free(&the_window);
}

MPI_Finalize();
return 0;
}
```

### 8.9.9 Listing of code examples/mpi/c/countdownput.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
```

```
#include "time.h"

#include <mpi.h>
#include "gather_sort_print.h"

int main(int argc,char **argv) {

    int nprocs,procno;
    MPI_Init(0,0);
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_rank(comm,&procno);

    if (nprocs<2) {
        printf("Need at least 2 procs\n");
        MPI_Abort(comm,0);
    }

    // first set a unique random seed
    srand(procno*time(0));

    {
        /*
         * Create a window.
         * We only need a nonzero size on the last process,
         * which we label the 'counter_process';
         * everyone else makes a window of size zero.
         */
        MPI_Win the_window;
        int counter_process = nprocs-1;
        int window_data,check_data;
        if (procno==counter_process) {
            window_data = 2*nprocs-1;
            check_data = window_data;
            MPI_Win_create(&window_data,sizeof(int),sizeof(int),
                           MPI_INFO_NULL,comm,&the_window);
        } else {
            MPI_Win_create(&window_data,0,sizeof(int),
                           MPI_INFO_NULL,comm,&the_window);
        }
        /*
         * Initialize the window
         * - PROCWRITES is approx the number of writes we want each process to do
         * - COLLISION is approx how many processes will collide on a write
         */
#define COLLISION
#define PROCWRITES 10
#define nprocs 4
#define procno 1
#define MPI_Win_fence MPI_Win_fence
        int counter_init = nprocs * PROCWRITES;
        MPI_Win_fence(0,the_window);
    }
}
```

## 8. MPI topic: One-sided communication

---

```
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);

/*
 * Allocate an array (grossly over-dimensioned)
 * for the counter values that belong to me
 */
int *my_counter_values = (int*) malloc( counter_init * sizeof(int) );
if (!my_counter_values) {
    printf("[%d] could not allocate counter values\n",procno);
    MPI_Abort(comm,0);
}
int n_my_counter_values = 0;

/*
 * Loop forever:
 * - at random times update the counter on the counter process
 * - and read out the counter to see if we stop
 */
int total_decrement = 0;
int nsteps = PROCWRITES / COLLISION;
if (procno==0)
    printf("Doing %d steps, %d writes per proc,\n.. probably %d-way collision on each ste
for (int step=0; step<nsteps ; step++) {

/*
 * Basic probability of a write is 1/P,
 * so each step only one proc will write.
 * Increase chance of collision by upping
 * the value of COLLISION.
 */
float randomfraction = (rand() / (double)RAND_MAX);
int i_am_available = randomfraction < ( COLLISION * .8/nprocs );

/*
 * Exercise:
 * - decrement the counter by Get, compute new value, Put
 */
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) {
#endif DEBUG
    printf("[%d] obtaining value %d in step %d\n",
           procno,counter_value,step);
#endif
}
```

```
my_counter_values[ n_my_counter_values++ ] = counter_value;
total_decrement++;
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);
}

/*
 * What counter values were actually obtained?
 */
gather_sort_print( my_counter_values,n_my_counter_values, comm );

/*
 * We do a correctness test by computing what the
 * window_data is supposed to be
 */
{
  MPI_Win_fence(0,the_window);
  int counter_value;
  MPI_Get( /* origin data to set: */ &counter_value,1,MPI_INT,
           /* window data to get: */ counter_process,0,1,MPI_INT,
           the_window);
  MPI_Win_fence(0,the_window);
  MPI_Allreduce(MPI_IN_PLACE,&total_decrement,1,MPI_INT,MPI_SUM,comm);
  if (procno==counter_process) {
    if (counter_init-total_decrement==counter_value)
      printf("[%d] initial counter %d decreased by %d correctly giving %d\n",
             procno,counter_init,total_decrement,counter_value);
    else
      printf("[%d] initial counter %d decreased by %d, giving %d s/b %d\n",
             procno,counter_init,total_decrement,counter_value,counter_init-total_decre
      }
  }
  MPI_Win_free(&the_window);
}

MPI_Finalize();
return 0;
}
```

### 8.9.10 Listing of code examples/mpi/c/countdownacc.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#include <mpi.h>
```

## 8. MPI topic: One-sided communication

---

```
#include "gather_sort_print.h"

int main(int argc,char **argv) {

#include "globalinit.c"
    if (nprocs<2) {
        printf("Need at least 2 procs\n");
        MPI_Abort(comm,0);
    }

    // first set a unique random seed
    srand(procno*time(0));

    {
    /*
     * Create a window.
     * We only need a nonzero size on the last process,
     * which we label the 'counter_process';
     * everyone else makes a window of size zero.
     */
    MPI_Win the_window;
    int counter_process = nprocs-1;
    int window_data,check_data;
    if (procno==counter_process) {
        window_data = 2*nprocs-1;
        check_data = window_data;
        MPI_Win_create(&window_data,sizeof(int),sizeof(int),
                       MPI_INFO_NULL,comm,&the_window);
    } else {
        MPI_Win_create(&window_data,0,sizeof(int),
                       MPI_INFO_NULL,comm,&the_window);
    }
    /*
     * Initialize the window
     * - PROCWRITES is approx the number of writes we want each process to do
     * - COLLISION is approx how many processes will collide on a write
     */
#ifndef COLLISION
#define COLLISION 2
#endif
#ifndef PROCWRITES
#define PROCWRITES 40
#endif
        int counter_init = nprocs * PROCWRITES;
        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);

    /*
     * Allocate an array (grossly over-dimensioned)
```

```
* for the counter values that belong to me
*/
int *my_counter_values = (int*) malloc( counter_init * sizeof(int) );
if (!my_counter_values) {
    printf("[%d] could not allocate counter values\n",procno);
    MPI_Abort(comm,0);
}
int n_my_counter_values = 0;

/*
 * Loop:
 * - at random times update the counter on the counter process
 * - and read out the counter to see if we stop
 */
int total_decrement = 0;
int nsteps = PROCWRITES / COLLISION;
if (procno==0)
    printf("Doing %d steps, counter starting: %d\n probably %d-way collision on each s
           nsteps,counter_init,COLLISION);
for (int step=0; step<nsteps ; step++) {

/*
 * Basic probability of a write is 1/P,
 * so each step only one proc will write.
 * Increase chance of collision by upping
 * the value of COLLISION.
 */
float randomfraction = (rand() / (double)RAND_MAX);
int i_am_available = randomfraction < ( COLLISION * 1./nprocs );

/*
 * Exercise:
 * - decrement the counter by Get, compute new value, Accumulate
 */
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) {
#endif DEBUG
    printf("[%d] updating in step %d\n",procno,step);
#endif
    my_counter_values[n_my_counter_values++] = counter_value;
    total_decrement++;
    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);
    }

    /*
     * What counter values were actually obtained?
     */
gather_sort_print( my_counter_values,n_my_counter_values, comm );

    /*
     * We do a correctness test by computing what the
     * window_data is supposed to be
     */
{
    MPI_Win_fence(0,the_window);
    int counter_value;
    MPI_Get( /* origin data to set: */ &counter_value,1,MPI_INT,
             /* window data to get: */ counter_process,0,1,MPI_INT,
             the_window);
    MPI_Win_fence(0,the_window);
    MPI_Allreduce(MPI_IN_PLACE,&total_decrement,1,MPI_INT,MPI_SUM,comm);
    if (procno==counter_process) {
        if (counter_init-total_decrement==counter_value)
            printf("[%d] initial counter %d decreased by %d correctly giving %d\n",
                   procno,counter_init,total_decrement,counter_value);
        else
            printf("[%d] initial counter %d decreased by %d, giving %d s/b %d\n",
                   procno,counter_init,total_decrement,counter_value,counter_init-total_decre
            )
    }
    MPI_Win_free(&the_window);
}

MPI_Finalize();
return 0;
}

```

### 8.9.11 Listing of code examples/mpi/c/windynamic.c

```

#include <stdlib.h>
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc,char **argv) {

#include "globalinit.c"

{
    MPI_Win the_window;
    int origin=0, data_proc = nprocs-1;
    int *retrieve=NULL,*window_buffer=NULL;

```

```
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));

if (procno==data_proc) {
    window_buffer[0] = 1;
    window_buffer[1] = 27;
}
if (procno==origin) {
    retrieve = (int*) malloc( sizeof(int) );
}

MPI_Aint data_address;
if (procno==data_proc) {
    MPI_Get_address(window_buffer,&data_address);
}
MPI_Bcast (&data_address,1,MPI_LONG,data_proc,comm);

MPI_Win_fence(0,the_window);
if (procno==origin) {
    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);
}
MPI_Win_fence(0,the_window);

if (procno==origin)
    printf("I got the following: %d\n",retrieve[0]);
MPI_Win_free(&the_window);
}

MPI_Finalize();
return 0;
}
```

## Chapter 9

### 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.

- 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 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 an 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 well-defined locations where to access the data. In fact, MPI-IO uses 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`.)

### 9.1 MPI\_File

```
C:  
MPI_File file ;  
  
Fortran:  
Type(MPI_File) file  
  
Python:  
MPI.File
```

### 9.2 MPI\_File\_open

```
Semantics:  
MPI_FILE_OPEN(comm, filename, amode, info, fh)  
IN comm: communicator (handle)  
IN filename: name of file to open (string)  
IN amode: file access mode (integer)  
IN info: info object (handle)  
OUT fh: new file handle (handle)  
  
C:  
int MPI_File_open  
(MPI_Comm comm, char *filename, int amode,  
 MPI_Info info, MPI_File *fh)  
  
Fortran:  
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)  
CHARACTER*(*) FILENAME  
INTEGER COMM, AMODE, INFO, FH, IERROR  
  
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.

## 9.1 File handling

MPI has its own file handle: `MPI_File` (figure 9.1).

You open a file with `MPI_File_open` (figure 9.2). 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.

*Python note.* 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.

You can delete a file with `MPI_File_delete`.

Buffers can be flushed with `MPI_File_sync`, which is a collective call.

## 9.2 File reading and writing

The basic file operations, in between the open and close calls, are the POSIX-like, non-collective, calls

- `MPI_File_seek` (figure 9.3) . 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 9.4) . 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` (figure 9.4).
- `MPI_File_read` (figure 9.5) 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 9.5).

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`.

### 9.3 MPI\_File\_seek

```
MPI_File_seek - Updates individual file pointers (noncollective)

C:
#include <mpi.h>
int MPI_File_seek(MPI_File fh, MPI_Offset offset,int whence)

Fortran 2008:
USE mpi_f08
MPI_File_seek(fh, offset, whence, ierror)
  TYPE(MPI_File), INTENT(IN) :: fh
  INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
  INTEGER, INTENT(IN) :: whence
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran 90:
USE MPI
! or the older form: INCLUDE 'mpif.h'
MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
  INTEGER    FH, WHENCE, IERROR
  INTEGER(KIND=MPI_OFFSET_KIND)    OFFSET

Input parameters:
fh      : File handle (handle).
offset : File offset (integer).
whence : Update mode (integer).

Output parameters:
IERROR : Fortran only: Error status (integer)
```

#### 9.4 MPI\_File\_write

Synopsis:

```
write at current file pointer
MPI_File_write: non-collective
MPI_File_write_all : collective
```

C Syntax

```
#include <mpi.h>
int MPI_File_write
int MPI_File_write_all(MPI_File fh, const void *buf,
                      int count, MPI_Datatype datatype,
                      MPI_Status *status)
```

Input parameters:

```
buf : Initial address of buffer (choice).
count : Number of elements in buffer (integer).
datatype : Data type of each buffer element (handle).
```

Output parameters:

```
status : Status object (status).
IERROR : Fortran only: Error status (integer).
```

USE mpi\_f08

```
MPI_File_write
MPI_File_write_all(fh, buf, count, datatype, status, ierror)
  TYPE(MPI_File), INTENT(IN) :: fh
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER, INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

USE MPI

```
! or the older form: INCLUDE 'mpif.h'
MPI_FILE_WRITE(FH, BUF, COUNT,
               DATATYPE, STATUS, IERROR)
<type>    BUF(*)
INTEGER     FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

## 9.5 MPI\_File\_read

Synopsis:

Reads a file starting at the location specified by the individual file pointer (blocking, noncollective).

C Syntax

```
#include <mpi.h>
int MPI_File_read(MPI_File fh, void *buf,
                  int count, MPI_Datatype datatype, MPI_Status *status)

USE mpi_f08
MPI_File_read(fh, buf, count, datatype, status, ierror)
  TYPE(MPI_File), INTENT(IN) :: fh
  TYPE(*), DIMENSION(..) :: buf
  INTEGER, INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

USE MPI
! or the older form: INCLUDE 'mpif.h'
MPI_FILE_READ(FH, BUF, COUNT,
              DATATYPE, STATUS, IERROR)
  <type>    BUF(*)
  INTEGER     FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

Input parameters:

fh : File handle (handle).  
count : Number of elements in buffer (integer).  
datatype : Data type of each buffer element (handle).

Output parameters:

buf : Initial address of buffer (integer).  
status : Status object (status).  
IERROR : Fortran only: Error status (integer).

### 9.6 MPI\_File\_iwrite

Synopsis  
 nonblocking write using individual file pointer  
 MPI\_File\_iwrite: non-collective

C syntax:  
`#ifdef HAVE_MPI_GREQUEST  
">#include "mpiu_greq.h"  
#endif  
 int MPI_File_iwrite(MPI_File fh,  
 ROMIO_CONST void *buf, int count, MPI_Datatype datatype,  
 MPI_Request * request)`

Input parameters:  
`fh : file handle  
 buf : Initial address of buffer (choice).  
 count : Number of elements in buffer (integer).  
 datatype : Data type of each buffer element (handle).`

Output parameters:  
`request : request object (handle)  
 status : Status object (status).  
 IERROR : Fortran only: Error status (integer).`

- **MPI\_File\_write\_at**: combine write and seekl The collective variant is **MPI\_File\_write\_at\_all**
- 

Writing to and reading from a parallel file is rather similar to sending a receiving:

- The process uses an elementary 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.

Just like there are blocking and non-blocking sends, there are also non-blocking writes and reads: **MPI\_File\_iwrite** (figure 9.6) , **MPI\_File\_iread** operations, that output an **MPI\_Request** object, which can then be tested with **MPI\_Wait** or **MPI\_Test**.

Having a non-blocking version of **MPI\_File\_write\_all** is tricky because there is no collective version of **MPI\_Wait** or **MPI\_Test**. Therefore, there are two routines: **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.

**Remark 12** So what is [https://www.mpich.org/static/docs/latest/www3/MPI\\_File\\_iwrite\\_all.html](https://www.mpich.org/static/docs/latest/www3/MPI_File_iwrite_all.html) ?

### 9.7 MPI\_File\_write\_at

```

MPI_File_write_at(fh, offset, buf, count, datatype)

Semantics:
Input Parameters
fh : File handle (handle).
offset : File offset (integer).
buf : Initial address of buffer (choice).
count : Number of elements in buffer (integer).
datatype : Data type of each buffer element (handle).

Output Parameters:
status : Status object (status).

C:
int MPI_File_write_at
    (MPI_File fh, MPI_Offset offset, const void *buf,
     int count, MPI_Datatype datatype, MPI_Status *status)

Fortran:
MPI_FILE_WRITE_AT
    (FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
<type>      BUF(*)
INTEGER :: FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
INTEGER(KIND=MPI_OFFSET_KIND) :: OFFSET

Python:
MPI.File.Write_at(self, Offset offset, buf, Status status=None)

```

#### 9.2.1 Individual file pointers, contiguous writes

After the collective open call, each rank holds an *individual file pointer* each rank can individually position the pointer somewhere in the shared file. Let's explore this modality.

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 9.7) gives this location in absolute terms with a parameter of type `MPI_Offset`, which counts bytes.

Figure 9.1: Writing at an offset



**Exercise 9.1.** Create a buffer of length `nwords=3` on each process, and write these buffers

### 9.8 MPI\_File\_set\_view

```
Semantics:
MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)
INOUT fh: file handle (handle)
IN disp: displacement (integer)
IN etype: elementary datatype (handle)
IN filetype: filetype (handle)
IN datarep: data representation (string)
IN info: info object (handle)

C:
int MPI_File_set_view
(MPI_File fh,
 MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype,
 char *datarep, MPI_Info info)

Fortran:
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
INTEGER FH, ETYP, FILETYPE, INFO, IERROR
CHARACTER*(*) DATAREP
INTEGER(KIND=MPI_OFFSET_KIND) DISP

Python:
mpifile = MPI.File.Open( .... )
mpifile.Set_view
(self,
 Offset disp=0, Datatype etype=None, Datatype filetype=None,
 datarep=None, Info info=INFO_NULL)
```

as a sequence to one file with [MPI\\_File\\_write\\_at](#).

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 9.2.** Rewrite the code of exercise 9.1 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 9.3.** 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  $p, p + W, p + 2W, \dots$ . Use a loop where each iteration

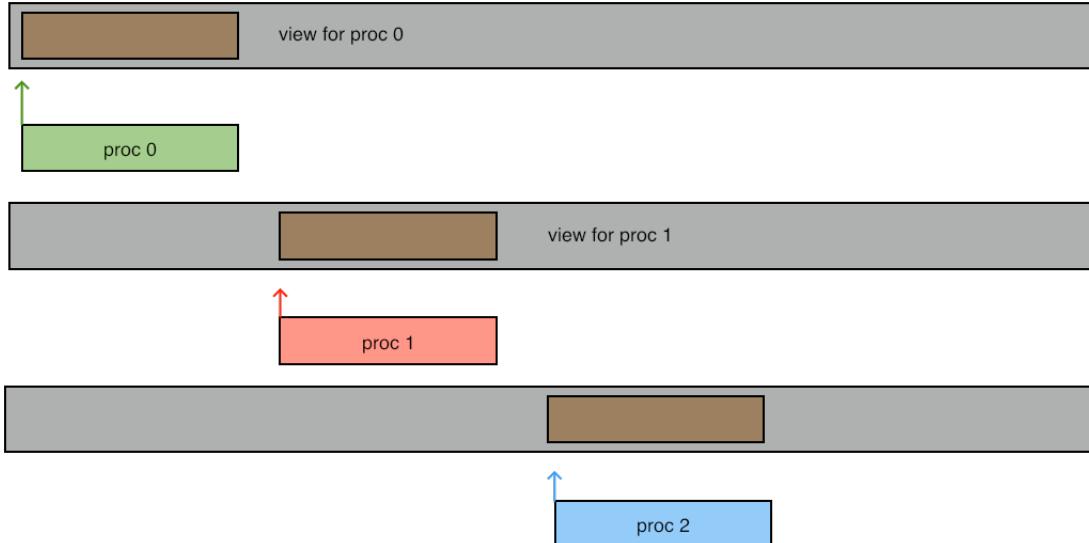
1. writes a single number with [MPI\\_File\\_write](#), and
2. advanced the file pointer with [MPI\\_File\\_seek](#) with a *whence* parameter of [MPI\\_SEEK\\_CUR](#).

#### 9.2.2 File views

The previous mode of writing is enough for writing simple contiguous blocks in the file. However, you can also access non-contiguous areas in the file. For this you use [MPI\\_File\\_set\\_view](#) (figure 9.8). This call is collective, even if not all processes access the file.

- The `disp` displacement parameter 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 12.1.1.3 for more.

Figure 9.2: Writing at a view



**Exercise 9.4.** Write a file in the same way as in exercise 9.1, 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.* 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.

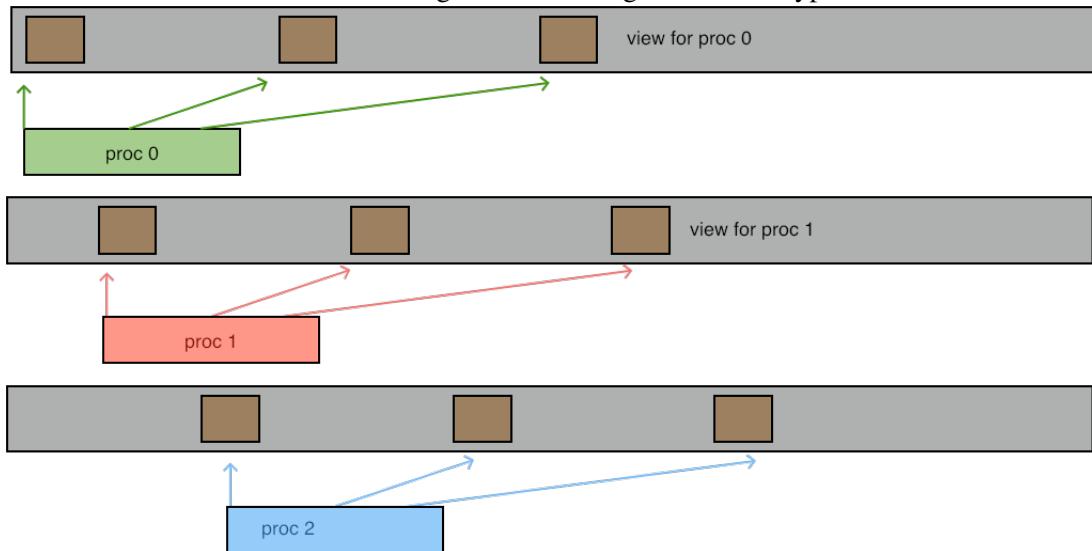
More: `MPI_File_set_size` `MPI_File_get_size` `MPI_File_pallocate` `MPI_File_get_view`

### 9.2.3 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`. Non-blocking variants are `MPI_File_iread_shared` and `MPI_File_iwrite_shared`.

Figure 9.3: Writing at a derived type



- Collective access are done with `MPI_File_read_ordered` and `MPI_File_write_ordered`, which execute the operations in order ascending by rank.

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.

### 9.3 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`.

### 9.4 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.

## 9.5 Review questions

**Exercise 9.5.** T/F? After your *SLURM* job ends, you can copy from the login node the files you've written to `\tmp`.

**Exercise 9.6.** 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 9.7.** 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?

**9.6 Sources used in this chapter**

**9.6.1 Listing of code header**

## Chapter 10

### 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 neighbours. 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 10.1.
- `MPI_GRAPH`: this value describes the graph topology that was defined in *MPI 1*; section 10.2.4. 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 10.2.

These values can be discovered with the routine `MPI_Topo_test` (figure 10.1).

#### 10.1 Cartesian grid topology

A *Cartesian grid* is a structure, typically in 2 or 3 dimensions, of points that have two neighbours 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 neighbours  $(k \pm 1, m, n), (k, m \pm 1, n), (k, m, n \pm 1)$ ; the exception are the edge points. A grid where edge processors are connected through *wraparound connections* is called a *periodic grid*.

The most common use of Cartesian coordinates is to find the rank of process by referring to it in grid terms. For instance, one could ask ‘what are my neighbours offset by  $(1, 0, 0), (-1, 0, 0), (0, 1, 0)$  et cetera’.

### 10.1 MPI\_Topo\_test

```
int MPI_Topo_test(MPI_Comm comm, int *status)

status:
MPI_UNDEFINED
MPI_CART
MPI_GRAPH
MPI_DIST_GRAPH
```

While the Cartesian topology interface is fairly easy to use, as opposed to the more complicated general graph topology below, it is not actually sufficient for all Cartesian graph uses. Notably, in a so-called *star stencil*, such as the *nine-point stencil*, there are diagonal connections, which can not be described in a single step. Instead, it is necessary to take a separate step along each coordinate dimension. In higher dimensions this is of course fairly awkward.

Thus, even for Cartesian structures, it may be advisable to use the general graph topology interface.

#### 10.1.1 Cartesian routines

The cartesian topology is specified by giving `MPI_Cart_create` the sizes of the processor grid along each axis, and whether the grid is periodic along that axis.

```
int MPI_Cart_create(
    MPI_Comm comm_old, int ndims, int *dims, int *periods,
    int reorder, MPI_Comm *comm_cart)
```

Each point in this new communicator has a coordinate and a rank. They can be queried with `MPI_Cart_coords` and `MPI_Cart_rank` respectively.

```
int MPI_Cart_coords(
    MPI_Comm comm, int rank, int maxdims,
    int *coords);
int MPI_Cart_rank(
    MPI_Comm comm, int *coords,
    int *rank);
```

Note that these routines can give the coordinates for any rank, not just for the current process.

```
// cart.c
MPI_Comm comm2d;
ndim = 2; periodic[0] = periodic[1] = 0;
dimensions[0] = idim; dimensions[1] = jdim;
MPI_Cart_create(comm, ndim, dimensions, periodic, 1, &comm2d);
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);
```

For the source of this example, see section 10.3.2

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.

Strangely enough you can only shift in one direction, you can not specify a shift vector.

```
|| int MPI_Cart_shift(MPI_Comm comm, int direction, int displ, int *source,
                      int *dest)
```

If you specify a processor outside the grid the result is `MPI_PROC_NULL`.

```
char mychar = 65+procno;
MPI_Cart_shift(comm2d, 0, +1, &rank_2d, &rank_right);
MPI_Cart_shift(comm2d, 0, -1, &rank_2d, &rank_left);
MPI_Cart_shift(comm2d, 1, +1, &rank_2d, &rank_up);
MPI_Cart_shift(comm2d, 1, -1, &rank_2d, &rank_down);
int irequest = 0; MPI_Request *requests = malloc(8*sizeof(MPI_Request));
MPI_Isend(&mychar, 1, MPI_CHAR, rank_right, 0, comm, requests+irequest++);
MPI_Isend(&mychar, 1, MPI_CHAR, rank_left, 0, comm, requests+irequest++);
MPI_Isend(&mychar, 1, MPI_CHAR, rank_up, 0, comm, requests+irequest++);
MPI_Isend(&mychar, 1, MPI_CHAR, rank_down, 0, comm, requests+irequest++);
MPI_Irecv( indata+idata++, 1, MPI_CHAR, rank_right, 0, comm, requests+irequest
    ++);
MPI_Irecv( indata+idata++, 1, MPI_CHAR, rank_left, 0, comm, requests+irequest
    ++);
MPI_Irecv( indata+idata++, 1, MPI_CHAR, rank_up, 0, comm, requests+irequest
    ++);
MPI_Irecv( indata+idata++, 1, MPI_CHAR, rank_down, 0, comm, requests+irequest
    ++);
```

For the source of this example, see section 10.3.2

## 10.2 Distributed graph topology

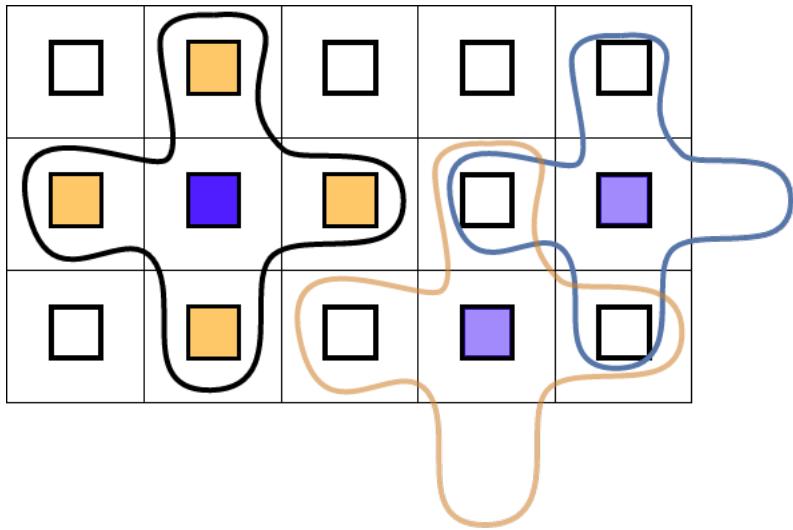


Figure 10.1: Illustration of a distributed graph topology where each node has four neighbours

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 neighbour processes. The number of neighbours 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 neighbours; processes on the edge with three, and processes in the corners with two.

Such a topology is illustrated in figure 10.1.

MPI's notion of *graph topology*, and the *neighbourhood 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 ranks, 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 neighbourhood. However, scheduling would then lead to deadlock or serialization.
- The normal way of dealing with graph problems is through non-blocking 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 neighbour 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 it, 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.

### 10.2.1 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 neighbour structure.

**10.2 MPI\_Dist\_graph\_create**

```

int MPI_Dist_graph_create
    (MPI_Comm comm_old, int n, const int sources[],
     const int degrees[], const int destinations[], const int weights[],
     MPI_Info info, int reorder,
     MPI_Comm *comm_dist_graph)

Input Parameters:
comm_old : input communicator (handle)
n : number of source nodes for which this process specifies edges (non-negative integer)
sources : array containing the n source nodes for which this process specifies edges (array)
degrees : array specifying the number of destinations for each source node in the source no
destinations : destination nodes for the source nodes in the source
node array (array of
non-negative
integers)
weights : weights for source to destination edges (array of
non-negative integers or MPI_UNWEIGHTED)
info : hints on optimization and interpretation of weights (handle)
reorder : the process may be reordered (true) or not (false) (logical)

Output Parameters:
comm_dist_graph : communicator with distributed graph topology added (handle)

Python:
MPI.Comm.Create_dist_graph
    (self, sources, degrees, destinations, weights=None, Info info=INFO_NULL, bool reorder=
returns graph communicator

```

The routine **MPI\_Dist\_graph\_create\_adjacent** assumes that a process knows both who it is sending it, 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 10.2), 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 neighbours 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.<sup>1</sup>. Consequently, some amount of processing – including communication – is needed to build the converse information, the ranks that will be sending to a process.

Figure 10.1 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 neighbour nodes; strictly speaking the ones that this process

---

1. I disagree with this design decision. Specifying your sources is usually easier than specifying your destinations.

### 10.3 MPI\_Neighbor\_allgather

Synopsis

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

Input Parameters:
  sendbuf : starting address of the send buffer (choice)
  sendcount : number of elements sent to each neighbor (non-negative integer)
  sendtype : data type of send buffer elements (handle)
  recvcount : number of elements received from each neighbor (non-negative integer)
  recvtype : data type of receive buffer elements (handle)
  comm : communicator (handle)

Output Parameters
  recvbuf : starting address of the receive buffer (choice)
```

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 ranks 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`.

*Python note.* 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.

#### 10.2.2 Neighbour collectives

We can now use the graph topology to perform a gather or allgather `MPI_Neighbor_allgather` (figure 10.3) that combines only the processes directly connected to the calling process.

The neighbour collectives have the same argument list as the regular collectives, but they apply to a graph communicator.

**Exercise 10.1.** Revisit exercise 4.7 and solve it using `MPI_Dist_graph_create`. Use figure 10.2 for inspiration.  
Use a degree value of 1.

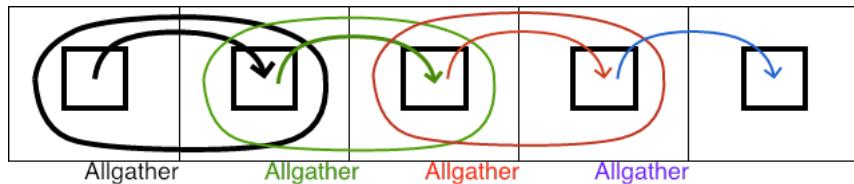


Figure 10.2: Solving the right-send exercise with neighbourhood collectives

The previous exercise can be done with a degree value of:

- 1, reflecting that each process communicates with just 1 other; or
- 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 10.2.3.

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`.

Non-blocking: `MPI_Ineighbor_allgather`, `MPI_Ineighbor_allgatherv`, `MPI_Ineighbor_alltoall`, `MPI_Ineighbor_alltoallv`, `MPI_Ineighbor_alltoallw`.

For unclear reasons there is no `MPI_Neighbor_allreduce`.

### 10.2.3 Query

There are two routines for querying the neighbors of a process: `MPI_Dist_graph_neighbors_count` and `MPI_Dist_graph_neighbors`.

While this information seems derivable from the graph construction, that is not entirely true for two reasons.

1. With the non-adjoint 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.

### 10.2.4 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 10.2) instead.

Other legacy routines: `MPI_Graph_neighbors`, `MPI_Graph_neighbors_count`, `MPI_Graph_get`, `MPI_Graphdims_get`

### 10.3 Sources used in this chapter

#### 10.3.1 Listing of code header

#### 10.3.2 Listing of code examples/mpi/c/cart.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

if (nprocs<4) {
    printf("This program needs at least four processes\n");
    return -1;
}

// 
int idim,jdim;
int ndim,periodic[2],dimensions[2],coord_2d[2],rank_2d;
for (idim=(int)(sqrt(1.*nprocs)); idim>=2; idim--) {
    jdim = nprocs/idim;
    if (idim*jdim==nprocs) goto found;
}
printf("No prime numbers please\n"); return -1;
MPI_Comm comm2d;
found:

ndim = 2; periodic[0] = periodic[1] = 0;
dimensions[0] = idim; dimensions[1] = jdim;
MPI_Cart_create(comm,ndim,dimensions,periodic,1,&comm2d);
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);

int rank_left,rank_right,rank_up,rank_down;
char indata[4]; int idata=0,sdata=0;
for (int i=0; i<4; i++)
    indata[i] = 32;
char mychar = 65+procno;
MPI_Cart_shift(comm2d,0,+1,&rank_2d,&rank_right);
MPI_Cart_shift(comm2d,0,-1,&rank_2d,&rank_left);
MPI_Cart_shift(comm2d,1,+1,&rank_2d,&rank_up);
MPI_Cart_shift(comm2d,1,-1,&rank_2d,&rank_down);
int irequest = 0; MPI_Request *requests = malloc(8*sizeof(MPI_Request));
MPI_Isend(&mychar,1,MPI_CHAR,rank_right, 0,comm, requests+irequest++);
MPI_Isend(&mychar,1,MPI_CHAR,rank_left,   0,comm, requests+irequest++);
MPI_Isend(&mychar,1,MPI_CHAR,rank_up,     0,comm, requests+irequest++);
```

```
MPI_Isend(&mychar,1,MPI_CHAR,rank_down, 0,comm, requests+irequest++);
MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_right, 0,comm, requests+irequest++);
MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_left, 0,comm, requests+irequest++);
MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_up,    0,comm, requests+irequest++);
MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_down, 0,comm, requests+irequest++);
MPI_Waitall(irequest,requests,MPI_STATUSES_IGNORE);
printf("[%d] %s\n",procno,indata);
/* for (int i=0; i<4; i++) */
/*   sdata += indata[i]; */
/* printf("[%d] %d,%d,%d,%d sum=%d\n",procno,indata[0],indata[1],indata[2],indata[3],sdat
if (procno==0)
  printf("Finished\n");

MPI_Finalize();
return 0;
}
```

# Chapter 11

## 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 8) 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.

### 11.1 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` (figure 11.1).

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.
- `MPI_COMM_TYPE_HW_GUIDED` (MPI-4): split using an `info` value from `MPI_Get_hw_resource_types`

In the following example, `CORES_PER_NODE` is a platform-dependent constant:

### 11.1 MPI\_Comm\_split\_type

```
C:
int MPI_Comm_split_type(
    MPI_Comm comm, int split_type, int key,
    MPI_Info info, MPI_Comm *newcomm)

Fortran:
MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: split_type, key
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Comm), INTENT(OUT) :: newcomm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Python:
MPI.Comm.Split_type(
    self, int split_type, int key=0, Info info=INFO_NULL)

||| // 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);
```

For the source of this example, see section 11.3.2

## 11.2 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`**, and
2. We need to get pointers to where a process' area is in this shared space; this is done with **`MPI_Win_shared_query`**.

### 11.2.1 Pointers to a shared window

The first step is to create a window (in the sense of one-sided MPI; section 8.1) on the processes on one node. Using the **`MPI_Win_allocate_shared`** (figure 11.2) call presumably will put the memory close to the socket on which the process runs.

```
// sharedbulk.c
MPI_Aint window_size; double *window_data; MPI_Win node_window;
if (onnode_procid==0)
    window_size = sizeof(double);
else window_size = 0;
MPI_Win_allocate_shared
```

### 11.2 MPI\_Win\_allocate\_shared

Semantics:

```
MPI_WIN_ALLOCATE_SHARED(size, disp_unit, info, comm, baseptr, win)
```

Input parameters:

size: size of local window in bytes (non-negative integer)  
disp\_unit local unit size for displacements, in bytes (positive integer)  
info: info argument (handle)  
comm: intra-communicator (handle)

Output parameters:

baseptr: address of local allocated window segment (choice)  
win: window object returned by the call (handle)

C:

```
int MPI_Win_allocate_shared  
(MPI_Aint size, int disp_unit, MPI_Info info,  
 MPI_Comm comm, void *baseptr, MPI_Win *win)
```

Fortran:

```
MPI_Win_allocate_shared  
(size, disp_unit, info, comm, baseptr, win, ierror)  
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR  
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size  
INTEGER, INTENT(IN) :: disp_unit  
TYPE(MPI_Info), INTENT(IN) :: info  
TYPE(MPI_Comm), INTENT(IN) :: comm  
TYPE(C_PTR), INTENT(OUT) :: baseptr  
TYPE(MPI_Win), INTENT(OUT) :: win  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
    ( window_size, sizeof(double), MPI_INFO_NULL,
      nodecomm,
      &window_data, &node_window );
```

For the source of this example, see section [11.3.3](#)

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.

```
// 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);
```

For the source of this example, see section [11.3.4](#)

Let's now consider a scenario where you spawn two MPI ranks per node, and the node has 100G of memory. Using the above option to allow for non-contiguous window allocation, you hope that the windows of the two ranks are placed 50G apart. However, if you print out the addresses, you will find that that they are placed considerably closer together. For a small windows that distance may be as little as 4K, the size of a *small page*.

The reason for this mismatch is that an address that you obtain with 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*.

### 11.2.2 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 [11.3](#)) .

```
MPI_Aint window_size0; int window_unit; double *win0_addr;
MPI_Win_shared_query( node_window, 0,
                      &window_size0, &window_unit, &win0_addr );
```

For the source of this example, see section [11.3.5](#)

### 11.2.3 Heat equation example

As an example, which consider the 1D heat equation. On each process we create a local area of three point:

### 11.3 MPI\_Win\_shared\_query

Semantics:

```
MPI_WIN_SHARED_QUERY(win, rank, size, disp_unit, baseptr)
```

Input arguments:

```
win: shared memory window object (handle)
rank: rank in the group of window win (non-negative integer)
      or MPI_PROC_NULL
```

Output arguments:

```
size: size of the window segment (non-negative integer)
disp_unit: local unit size for displacements,
           in bytes (positive integer)
baseptr: address for load/store access to window segment (choice)
```

C:

```
int MPI_Win_shared_query
      (MPI_Win win, int rank, MPI_Aint *size, int *disp_unit,
       void *baseptr)
```

Fortran:

```
MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror)
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
TYPE(MPI_Win), INTENT(IN) :: win
INTEGER, INTENT(IN) :: rank
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
INTEGER, INTENT(OUT) :: disp_unit
TYPE(C_PTR), INTENT(OUT) :: baseptr
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
// sharedshared.c
MPI_Win_allocate_shared(3,sizeof(int),info,sharedcomm,&shared_baseptr,&
shared_window);
```

For the source of this example, see section [11.3.6](#)

#### 11.2.4 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 11.1.** 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.

### 11.3 Sources used in this chapter

#### 11.3.1 Listing of code header

#### 11.3.2 Listing of code examples/mpi/c/commsplittype.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

#ifndef CORES_PER_NODE
#define CORES_PER_NODE 16
#endif

int main(int argc,char **argv) {

#include "globalinit.c"

    if (nprocs<3) {
        printf("This program needs at least three processes\n");
        return -1;
    }

    if (procno==0)
        printf("There are %d ranks total\n",nprocs);

    int new_procno,new_nprocs;
    MPI_Comm sharedcomm;

    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);

    ASSERT(new_procno<CORES_PER_NODE);

    if (new_procno==0) {
        char procname[MPI_MAX_PROCESSOR_NAME]; int namlen;
        MPI_Get_processor_name(procname,&namlen);
        printf("I am processor %d in a shared group of %d, running on %s\n",
        new_procno,new_nprocs,procname);
    }
    if (procno==0)
        printf("Finished\n");

    MPI_Finalize();
    return 0;
}
```

### 11.3.3 Listing of code examples/mpi/c/sharedbulk.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc,char **argv) {
    MPI_Comm comm;
    int nprocs,procid;

    MPI_Init(&argc,&argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_rank(comm,&procid);

    /*
     * Find the subcommunicator on the node,
     * and get the procid on the node.
     */
    MPI_Comm nodecomm; int onnode_procid;
    MPI_Comm_split_type
        (comm,MPI_COMM_TYPE_SHARED,procid,MPI_INFO_NULL,
         &nodecomm);
    MPI_Comm_rank(nodecomm,&onnode_procid);

    /*
     * Find the subcommunicators of
     * identical 'onnode_procid' processes;
     * the procid on that communicator is the node ID
     */
    MPI_Comm crosscomm; int nodeid;
    MPI_Comm_split
        (comm,onnode_procid,procid,&crosscomm);
    MPI_Comm_rank(crosscomm,&nodeid);
    printf("[%d] %dx%d\n",procid,nodeid,onnode_procid);

    /*
     * Create data on global process zero,
     * and broadcast it to the zero processes on other nodes
     */
    double shared_data = 0;
    if (procid==0) shared_data = 3.14;
    if (onnode_procid==0)
        MPI_Bcast(&shared_data,1,MPI_DOUBLE,0,crosscomm);
    printf("[%d] Head nodes should have shared data: %e\n",procid,shared_data);

    /*
     * Create window on the node communicator;
     * it only has nonzero size on the first process
     */
    MPI_Aint window_size; double *window_data; MPI_Win node_window;
    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);

/*
 * Put data on process zero of the node window
 * We use a Put call rather than a straight copy:
 * the Fence calls enforce coherence
 */
MPI_Win_fence(0,node_window);
if (onnode_procid==0) {
    MPI_Aint disp = 0;
    MPI_Put( &shared_data,1,MPI_DOUBLE,0,disp,1,MPI_DOUBLE,node_window);
}
MPI_Win_fence(0,node_window);

/*
 * Now get on each process the address of the window of process zero.
 */
MPI_Aint window_size0; int window_unit; double *win0_addr;
MPI_Win_shared_query( node_window,0,
&window_size0,&window_unit, &win0_addr );

/*
 * Check that we can indeed get at the data in the shared memory
 */
printf("[%d,%d] data at shared window: %e\n",nodeid,onnode_procid,*win0_addr);

/*
 * cleanup
 */
MPI_Win_free(&node_window);
MPI_Finalize();
return 0;
}

```

#### 11.3.4 Listing of code examples/mpi/c/numa.c

```

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc,char **argv) {

#include "globalinit.c"

/*

```

```
* Find the subcommunicator on the node,
* and get the procid on the node.
*/
MPI_Comm nodecomm;
int onnode_procno, onnode_nprocs;
MPI_Comm_split_type
    (comm,MPI_COMM_TYPE_SHARED,procno,MPI_INFO_NULL,
     &nodecomm);
MPI_Comm_size(nodecomm,&onnode_nprocs);
if (onnode_nprocs<2) {
    printf("This example needs at least two ranks per node\n");
    MPI_Abort(comm,0);
}
MPI_Comm_rank(nodecomm,&onnode_procno);

for (int strategy=0; strategy<2; strategy++) {
/*
 * Create window on the node communicator;
 * one item on each process
 */
MPI_Aint window_size; double *window_data; MPI_Win node_window;
window_size = sizeof(double);
MPI_Info window_info;
MPI_Info_create(&window_info);
if (strategy==0) {
    if (procno==0)
        printf("Strategy 0 : default behavior of shared window allocation\n");
    MPI_Info_set(window_info,"alloc_shared_noncontig","false");
} else {
    if (procno==0)
        printf("Strategy 1 : allow non-contiguous shared window allocation\n");
    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);

/*
 * Now process zero checks on window placement
 */
if (onnode_procno==0) {
    MPI_Aint window_size0; int window0_unit; double *win0_addr;
    MPI_Win_shared_query( node_window,0,
                          &window_size0,&window0_unit, &win0_addr );
    size_t dist1,distp;
    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);
    }
}
```

```
    if (p==1)
        dist1 = distp;
    else {
        if (distp%dist1!=0)
            printf("!!!! not a multiple of distance 0--1 !!!!\n");
    }
}
MPI_Win_free(&node_window);
}

/*
 * cleanup
 */
MPI_Finalize();
return 0;
}
```

### 11.3.5 Listing of code code/mpi/shared.c

### 11.3.6 Listing of code examples/mpi/c/sharedshared.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

#ifndef CORES_PER_NODE
#define CORES_PER_NODE 16
#endif

int main(int argc,char **argv) {

#include "globalinit.c"

if (nprocs<3) {
    printf("This program needs at least three processes\n");
    return -1;
}

if (procno==0)
    printf("There are %d ranks total\n",nprocs);

int new_procno,new_nprocs;
MPI_Comm sharedcomm;

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);

ASSERT(new_procno<CORES_PER_NODE);

if (new_nprocs!=nprocs) {
    printf("This example can only run on shared memory\n");
    MPI_Abort(comm, 0);
}

MPI_Win shared_window; int *shared_baseptr;
MPI_Win_allocate_shared(3,sizeof(int),info,sharedcomm,&shared_baseptr,&shared_window);

{
    MPI_Aint check_size; int check_unit; int *check_baseptr;
    MPI_Win_shared_query
        (shared_window,new_procno,
         &check_size,&check_unit,&check_baseptr);
    printf("[%d;%d] size=%ld\n",procno,new_procno,check_size);
}

int *left_ptr,*right_ptr;
int left_proc = new_procno>0 ? new_procno-1 : MPI_PROC_NULL,
    right_proc = new_procno<new_nprocs-1 ? new_procno+1 : MPI_PROC_NULL;
MPI_Win_shared_query(shared_window,left_proc,NULL,NULL,&left_ptr);
MPI_Win_shared_query(shared_window,right_proc,NULL,NULL,&right_ptr);

if (procno==0)
    printf("Finished\n");

MPI_Finalize();
return 0;
}
```

# Chapter 12

## MPI leftover topics

### 12.1 Contextual information, attributes, etc.

#### 12.1.1 Info objects

Certain MPI routines can accept `MPI_Info` (figure 12.1) objects. These contain key-value pairs that can offer system or implementation dependent information.

Create an info object with `MPI_Info_create` (figure 12.2) and delete it with `MPI_Info_free` (figure 12.3).

Keys are then set with `MPI_Info_set` (figure 12.4), and they can be queried with `MPI_Info_get` (figure 12.5). 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 12.6).

There is a straightforward duplication of info objects: `MPI_Info_dup` (figure 12.7).

You can also query the number of keys in an info object with `MPI_Info_get_nkeys` (figure 12.8), after which the keys can be queried in succession with `MPI_Info_get_nthkey`.

##### 12.1.1.1 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.

**12.1 MPI\_Info**

C  
MPI\_Info info ;  
  
Fortran:  
Type(MPI\_Info) info  
  
Python:

**12.2 MPI\_Info\_create**

MPI\_INFO\_CREATE(info)  
OUT info info object created (handle)  
  
C:  
int MPI\_Info\_create(MPI\_Info \*info)  
  
Fortran legacy:  
MPI\_INFO\_CREATE(INFO, IERROR)  
INTEGER INFO, IERROR

**12.3 MPI\_Info\_free**

MPI\_INFO\_FREE(info)  
INOUT infoinfo object (handle)  
int MPI\_Info\_free(MPI\_Info \*info)  
MPI\_INFO\_FREE(INFO, IERROR)  
INTEGER INFO, IERROR

**12.4 MPI\_Info\_set**

MPI\_INFO\_SET(info, key, value)  
INOUT infoinfo object (handle)  
IN keykey (string)  
IN valuevalue (string)  
int MPI\_Info\_set(MPI\_Info info, char \*key, char \*value)  
MPI\_INFO\_SET(INFO, KEY, VALUE, IERROR)  
INTEGER INFO, IERROR  
CHARACTER(\*) KEY, VALUE

**12.5 MPI\_Info\_get**

MPI\_INFO\_GET(info, key, valuelen, value, flag)  
IN infoinfo object (handle)  
IN keykey (string)  
IN valuelenlength of value arg (integer)  
OUT valuevalue (string)  
OUT flagtrue if key defined, false if not (boolean)  
int MPI\_Info\_get(MPI\_Info info, char \*key, int valuelen, char \*value,  
int \*flag)  
MPI\_INFO\_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)  
INTEGER INFO, VALUELEN, IERROR  
CHARACTER(\*) KEY, VALUE  
LOGICAL FLAG

### 12.6 MPI\_Info\_delete

```
MPI_INFO_DELETE(info, key)
INOUT infoinfo object (handle)
IN keykey (string)
int MPI_Info_delete(MPI_Info info, char *key)
MPI_INFO_DELETE(INFO, KEY, IERROR)
INTEGER INFO, IERROR
CHARACTER(*) KEY
```

### 12.7 MPI\_Info\_dup

```
MPI_INFO_DUP(info, newinfo)
IN infoinfo object (handle)
OUT newinfoinfo object (handle)
int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
MPI_INFO_DUP(INFO, NEWINFO, IERROR)
INTEGER INFO, NEWINFO, IERROR
```

#### 12.1.1.2 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](#) would cause the info to be copied; to attach new information to the copy there is [MPI\\_Comm\\_dup\\_with\\_info](#).

#### 12.1.1.3 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 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
- *nb\_proc*: integer number of processes expected to access a file simultaneously

**12.8 MPI\_Info\_get\_nkeys**

```

MPI_INFO_GET_NKEYS(info, nkeys)
IN infoinfo object (handle)
OUT nkeysnumber of defined keys (integer)
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
INTEGER INFO, NKEYS, IERROR

```

**12.9 MPI\_Comm\_get\_attr**

```

int MPI_Comm_get_attr(
    MPI_Comm comm, int keyval, void *attribute_val, int *flag)

Python:
MPI.Comm.Get_attr(self, int keyval)

return codes:
MPI_SUCCESS : No error; MPI routine completed successfully.
MPI_ERR_COMM : Invalid communicator.
MPI_ERR_KEYVAL : Invalid keyval

```

- *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.

**12.1.2 Attributes**

Some runtime (or installation dependent) values are available as attributes through **MPI\_Comm\_get\_attr** (figure 12.9). (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;

```

For the source of this example, see section 12.13.2

```

## tags.py
tag_upperbound = comm.Get_attr(MPI.TAG_UB)
if procid==0:
    print("Determined tag upperbound: {}".format(tag_upperbound))

```

For the source of this example, see section 12.13.3

Attributes are:

### 12.10 MPI\_Get\_version

Semantics:

```
MPI_GET_VERSION( version, subversion )
OUT version version number (integer)
OUT subversion subversion number (integer)
```

C:

```
int MPI_Get_version(int *version, int *subversion)
```

Fortran:

```
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
INTEGER VERSION, SUBVERSION, IERROR
```

- `MPI_TAG_UB` Upper bound for *tag value*. Note that `MPI_TAG_UB` is the key, not the actual upper bound!
- `MPI_HOST` Host process rank, if such exists, `MPI_PROC_NULL`, otherwise.
- `MPI_IO` rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter.
- `MPI_WTIME_IS_GLOBAL` Boolean variable that indicates whether clocks are synchronized.

Also:

- `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 7.1.  
Python: `mpi4py.MPI.UNIVERSE_SIZE`.
- `MPI_APPNUM`: if MPI is used in MPMD mode, or if `MPI_Comm_spawn_multiple` is used, this attribute reports the how-manyeth program we are in.

#### 12.1.3 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.

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.

#### 12.1.4 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 12.10) .

## 12.2 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 abort 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 abort. That is, if one incarnation of your executable aborts, the MPI runtime will kill all others.
- Deadlocks and other hanging executions: there are various scenarios where your processes individually do not abort, 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 non-blocking calls. In another scenario, through an error in program logic, one process will be waiting for more messages (including non-blocking ones) than are sent to it.

### 12.2.1 Error codes

- `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_INTERN` An internal error in MPI has been detected.
- `MPI_ERR_INFO`: invalid info object.
- `MPI_ERR_OTHER`: an error occurred; use `MPI_Error_string` to retrieve further information about this error.
- `MPI_ERR_PORT`: invalid port; this applies to `MPI_Comm_connect` and such.
- `MPI_ERR_SERVICE`: invalid service in `MPI_Unpublish_name`.
- `MPI_SUCCESS`: no error; MPI routine completed successfully.

### 12.2.2 Error handling

The MPI library has a general mechanism for dealing with errors that it detects. The default behaviour, where the full run is aborted, is equivalent to your code having the following call `MPI_Comm_set_errhandler`:

```
|| MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_ARE_FATAL);
```

**Remark 13** The routine `MPI_Errhandler_set` is deprecated.

Another simple possibility is to specify

```
|| MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
```

which gives you the opportunity to write code that handles the error return value. The values `MPI_ERRORS_ARE_FATAL` and `MPI_ERRORS_RETURN` are of type `MPI_Errhandler`.

In most cases where an MPI error occurs a complete abort is the sensible thing, since there are few ways to recover. Alternatively, you could compare the return code to `MPI_SUCCESS` and print out debugging information:

```

    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 12.11), which is then installed with `MPI_Comm_set_errhandler`. You can retrieve the error handler with `MPI_Comm_get_errhandler`.

*MPL note.* MPL does not allow for access to the wrapped communicators. However, for `MPI_COMM_WORLD`, the routine `MPI_Comm_set_errhandler` can be called directly.

### 12.3 Fortran issues

MPI is typically written in C, what if you program *Fortran*?

See section 5.2.2.1 for MPI types corresponding to *Fortran90 types*.

### 12.11 MPI\_Comm\_create\_errhandler

```

Synopsis
MPI_Comm_create_errhandler( errhandler_fn, err_handler )
Input argument:
errhandler_fn

Output argument:
err_handler

C:
int MPI_Comm_create_errhandler
( MPI_Comm_errhandler_function *errhandler_fn,
  MPI_Errhandler *err_handler )
typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *, ...)

Fortran:
Subroutine MPI_Comm_errhandler_function(comm, error_code) BIND(C)
TYPE(MPI_Comm) :: comm
INTEGER :: error_code

```

#### 12.3.1 Assumed-shape arrays

Use of other than contiguous data, for instance `A(1:N:2)`, was a problem in MPI calls, especially non-blocking 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 prototype 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`.

## 12.4 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 12.2) 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 7.1 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.

## 12.5 Asynchronous 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; for instance, *Intel MPI* has a number of `I_MPI_ASYNC_PROGRESS...` variables.
- Absent such dedicated resources, the application can force MPI to make progress by occasional calls to a *polling* routine such as `MPI_Iprobe`.

**Remark 14** 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.

*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>  
<https://software.intel.com/en-us/mpi-developer-reference-linux-environment-variables>

## 12.6 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.

**12.12 MPI\_Wtime**

```
C:
double MPI_Wtime(void);

Fortran:
DOUBLE PRECISION MPI_WTIME()

Python:
MPI.Wtime()
```

**12.13 MPI\_Wtick**

```
C:
double MPI_Wtick(void);

Fortran:
DOUBLE PRECISION MPI_WTICK()

Python:
MPI.Wtick()
```

**12.6.1 Timing**

MPI has a *wall clock* timer: **`MPI_Wtime`** (figure 12.12) 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 12.13) .

*MPL note.* 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;

```

#### 12.6.1.1 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*`MPI_Barrier`:

```

|| MPI_Barrier(comm)
|| t = MPI_Wtime();
|| // something happens here
|| MPI_Barrier(comm)
|| t = MPI_Wtime()-t;

```

**Exercise 12.1.** This scheme also has some overhead associated with it. How would you measure that?

#### 12.6.1.2 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.

#### 12.6.2 Simple profiling

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 12.1.

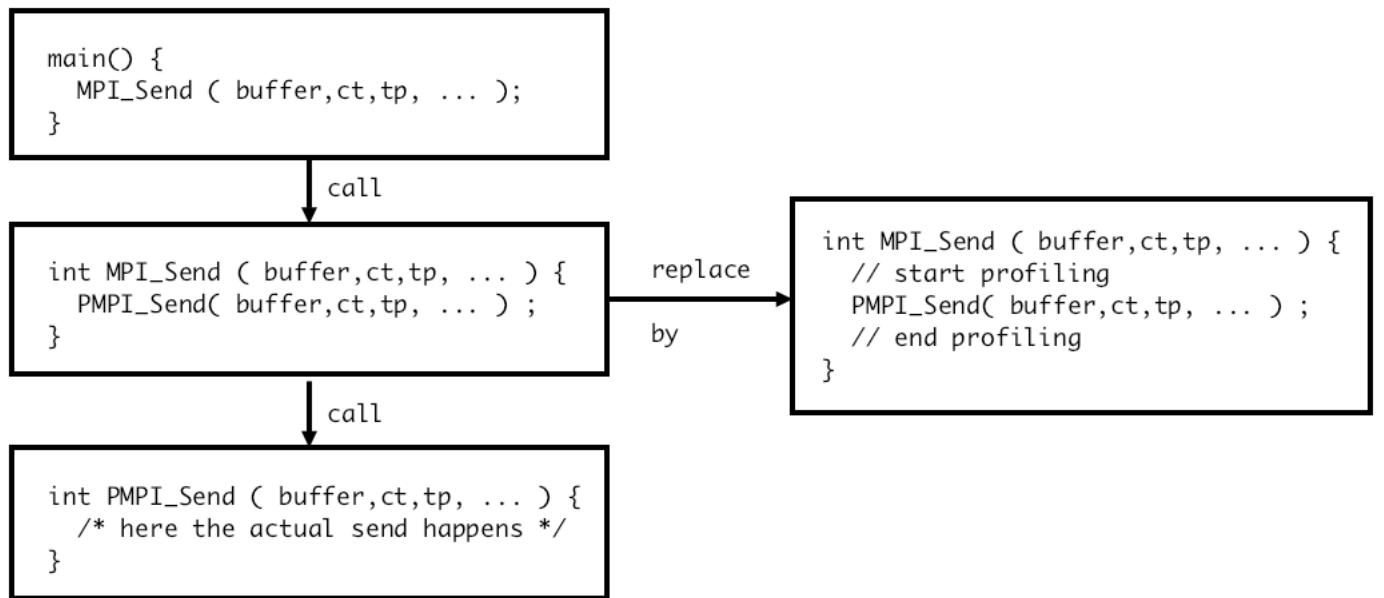


Figure 12.1: Calling hierarchy of MPI and PMPI routines

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 12.2, normally only the PMPI routines show up in the stack trace.

### 12.6.3 Tools interface

Recent versions of MPI have a standardized way of reading out performance variables: the *tools interface* which improves on the old interface described in section 12.6.2.

The realization of the tools interface is installation-dependent, you first need to query how much of the tools interface is provided.

```
// mpit.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);
```

For the source of this example, see section 12.13.4

```
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++) {
```

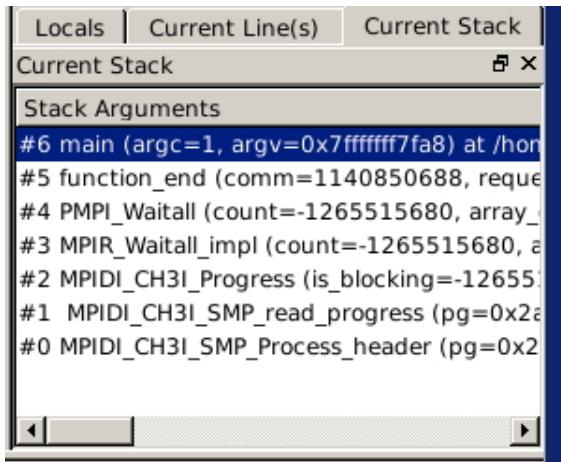


Figure 12.2: A stack trace, showing the `PMPI` calls.

```

    ||| 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);
    |||
}

```

For the source of this example, see section [12.13.4](#)

#### 12.6.4 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 [4.2.2.2](#)), 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 non-blocking** The issue of *blocking* versus *non-blocking* communication is something of a red herring. While non-blocking communication allows *latency hiding*, we can not consider it an alternative to blocking sends, since replacing non-blocking by blocking calls will usually give *deadlock*.

Still, even if you use non-blocking communication for the mere avoidance of deadlock or serialization (section [4.2.2.3](#)), 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 non-blocking, 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 12.5, 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 4.4.4.

**Buffering** MPI uses internal buffers, and the copying from user data to these buffers may affect performance. For instance, derived types (section 5.3) can typically not be streamed straight through the network (this requires special hardware support [12]) so they are first copied. Somewhat surprisingly, we find that *buffered communication* (section 4.4.5) does not help. Perhaps MPI implementors have not optimized this mode since it is so rarely used.

This issue is extensively investigated in [4].

**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 10.2) MPI is allowed to reorder processes, which could be used to support such activities. It can also serve for better message sequencing when *neighbourhood 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.

## 12.7 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 HPSC-??.

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 ‘non-overtaking’: 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.

## 12.8 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 non-blocking routines, which are explained below in section 4.3.1):

```
if ( receiving ) {
    MPI_Irecv() // post non-blocking 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 12.2.** Give pseudo-code for a scheme where you synchronize the two processes through the exchange of a blocking zero-size message.

## 12.9 Shell interaction

MPI programs are not run directly from the shell, but are started through an *ssh tunnel*. We briefly discuss ramifications of this.

### 12.9.1 Standard input

Letting MPI processes interact with the environment is not entirely straightforward. For instance, *shell input redirection* as in

```
mpirun -np 2 mpiprogram < someinput
```

may not work.

Instead, use a script `programscript` that has one parameter:

```
#!/bin/bash
mpirunprogram < $1
```

and run this in parallel:

```
mpirun -np 2 programscript someinput
```

### 12.9.2 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 `mpirun`.

```
// outerr.c
fprintf(stdout, "This goes to std out\n");
fprintf(stderr, "This goes to std err\n");
```

For the source of this example, see section [12.13.5](#)

### 12.9.3 Process status

The return code of `MPI_Abort` is returned as the *processes status* of `mpirun`. Running

```
// abort.c
if (procno==nprocs-1)
    MPI_Abort(comm, 37);
```

For the source of this example, see section [12.13.6](#)

as

```
mpirun -np 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>>
```

#### 12.9.4 Multiple program start

The sort of script of section 12.9.1 can also be used to implement *MPMD* runs: we let the script start one of a number of programs. For this, we use the fact that the MPI rank is known in the environment as `PMI_RANK`. Use a script `mpmdscript`:

```
#!/bin/bash
if [ ${PMI_RANK} -eq 0 ] ; then
    ./programmaster
else
    ./programworker
fi
```

which is then run in parallel:

```
mpirun -np 25 mpmdscript
```

### 12.10 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.

### 12.11 Leftover topics

#### 12.11.1 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 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
- MPI\_STATUS\_SIZE (Fortran only)
- MPI\_ADDRESS\_KIND (Fortran only)
- MPI\_COUNT\_KIND (Fortran only)
- MPI\_INTEGER\_KIND (Fortran only)
- MPI\_OFFSET\_KIND (Fortran only)
- MPI\_SUBARRAYS\_SUPPORTED (Fortran only)
- MPI\_ASYNC\_PROTECTS\_NONBLOCKING (Fortran only)

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
- 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>)

### 12.11.2 32-bit size issues

The `size` parameter in MPI routines is defined as an `int`, meaning that it is limited to 32-bit quantities. There are ways around this, such as sending a number of `MPI_Type_contiguous` blocks that add up to more than  $2^{31}$ .

### 12.11.3 Python issues

#### 12.11.3.1 Byte calculations

The `MPI_Win_create` routine needs a displacement in bytes. Here is a good way for finding the size of `numpy` datatypes:

```
|| numpy.dtype('i').itemsize
```

#### 12.11.3.2 Arrays of objects

Objects of type `MPI.Status` or `MPI.Request` often need to be created in an array, for instance when looping through a number of `Irecv` calls. In that case the following idiom may come in handy:

```
|| requests = [ None ] * nprocs
|| for p in range(nprocs):
||     requests[p] = comm.Irecv( ... )
```

### 12.11.4 Cancelling messages

In section 41.3 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);
    }
}
```

For the source of this example, see section [12.13.7](#)

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 *non-blocking collectives* or one-sided transfers.

### 12.11.5 Constants

MPI constants such as `MPI_COMM_WORLD` or `MPI_INT` are not necessarily statically defined, such as by a `#define` statement: the best you can say is that they have a value after `MPI_Init` or `MPI_Init_thread`. That means you can not transfer a compiled MPI file between platforms, or even between compilers on one platform. However, a working MPI source on one MPI implementation will also work on another.

## 12.12 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 [7] by some of the original authors.

### 12.13 Sources used in this chapter

#### 12.13.1 Listing of code header

#### 12.13.2 Listing of code examples/mpi/c/tags.c

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

int main(int argc,char **argv) {

#ifndef FREQUENCY
#define FREQUENCY -1
#endif

/*
 * Standard initialization
 */
MPI_Comm comm = MPI_COMM_WORLD;
int nprocs, procid;
MPI_Init(&argc,&argv);
MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN);
MPI_Comm_size(comm,&nprocs);
MPI_Comm_rank(comm,&procid);
int ierr;

if (nprocs<2) {
    printf("This test needs at least 2 processes, not %d\n",nprocs);
    MPI_Abort(comm,0);
}
int sender = 0, receiver = nprocs-1;
if (procid==0) {
    printf("Running on comm world of %d procs; communicating between %d--%d\n",
nprocs,sender,receiver);
}

int tag_upperbound;
void *v; int flag=1;
ierr = MPI_Comm_get_attr(comm,MPI_TAG_UB,&v,&flag);
tag_upperbound = *(int*)v;
if (ierr!=MPI_SUCCESS) {
    printf("Error getting attribute: return code=%d\n",ierr);
    if (ierr==MPI_ERR_COMM)
        printf("invalid communicator\n");
    if (ierr==MPI_ERR_KEYVAL)
        printf("errorneous keyval\n");
    MPI_Abort(comm,0);
}
if (!flag) {
    printf("Could not get keyval\n");
    MPI_Abort(comm,0);
}
```

```
    } else {
        if (procid==sender)
            printf("Determined tag upperbound: %d\n",tag_upperbound);
    }

    MPI_Finalize();
    return 0;
}
```

### 12.13.3 Listing of code examples/mpi/p/tags.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<4:
    print( "Need 4 procs at least")
    sys.exit(1)

tag_upperbound = comm.Get_attr(MPI.TAG_UB)
if procid==0:
    print("Determined tag upperbound: {}".format(tag_upperbound))
```

### 12.13.4 Listing of code code/mpi/c/mpit.c

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

int main(int argc,char **argv) {
    int procid,nprocs;

    int tlevel;
    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);

    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&procid);

    if (procid==0)
        printf("#pvars: %d\n",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++) {
    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);
}

MPI_T_finalize();
MPI_Finalize();

return 0;
}
```

### 12.13.5 Listing of code examples/mpi/c/outerr.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

    fprintf(stdout,"This goes to std out\n");
    fprintf(stderr,"This goes to std err\n");

    return 0;
}
```

### 12.13.6 Listing of code examples/mpi/c/abort.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

    if (procno==nprocs-1)
        MPI_Abort(comm,37);

    MPI_Finalize();
    return 0;
}
```

### 12.13.7 Listing of code examples/mpi/c/cancel.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

int main(int argc,char **argv) {
    int first_tid,dummy[11];

#include "globalinit.c"

    // Initialize the random number generator
    srand((int)(procno*(double)RAND_MAX/nprocs));

    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);
        }
    }

    MPI_Finalize();
    return 0;
}
```

# Chapter 13

## MPI Reference

This section gives reference information and illustrative examples of the use of MPI. While the code snippets given here should be enough, full programs can be found in the repository for this book <https://bitbucket.org/VictorEijkhout/parallel-computing-book>.

### 13.1 Leftover topics

#### 13.1.1 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 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
MPI_STATUS_SIZE (Fortran only)
MPI_ADDRESS_KIND (Fortran only)
```

```
|| MPI_COUNT_KIND (Fortran only)
|| MPI_INTEGER_KIND (Fortran only)
|| MPI_OFFSET_KIND (Fortran only)
|| MPI_SUBARRAYS_SUPPORTED (Fortran only)
|| MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)
```

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:

```
C type: const int (or unnamed enum)
Fortran type: INTEGER

|| MPI_PROC_NULL
|| 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>)

**13.2 Sources used in this chapter**

**13.2.1 Listing of code header**

## **PART II**

### **OPENMP**

## Chapter 14

### Getting started with OpenMP

This chapter explains the basic concepts of OpenMP, and helps you get started on running your first OpenMP program.

#### 14.1 The OpenMP model

We start by establishing a mental picture of the hardware and software that OpenMP targets.

##### 14.1.1 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* or *motherboard*, and getting the most out of the available parallelism available there.

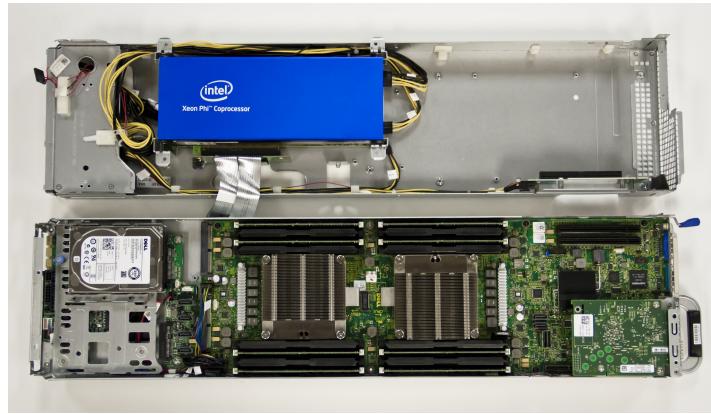


Figure 14.1: A node with two sockets and a co-processor

Figure 14.1 pictures a typical design of a node: within one enclosure you find two sockets: single processor chips. Your personal laptop of computer will probably have one socket, most supercomputers have nodes

with two or four sockets (the picture is of a *Stampede node* with two sockets)<sup>1</sup>, although the recent *Intel Knight's Landing* is again a single-socket design.

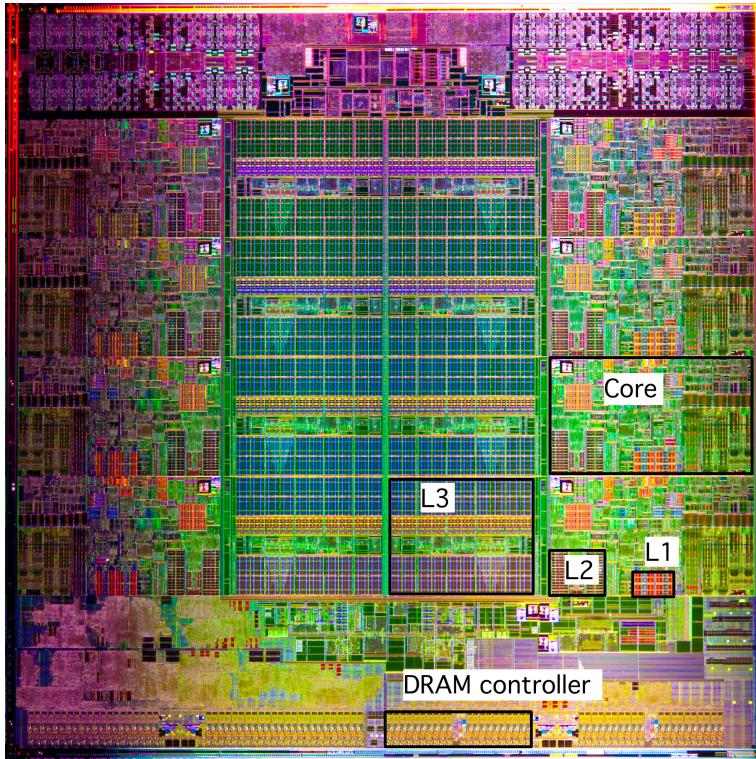


Figure 14.2: Structure of an Intel Sandybridge eight-core socket

To see where OpenMP operates we need to dig into the sockets. Figure 14.2 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 14.1 you saw four memory banks attached to each of the two sockets; all of the sixteen cores have access to all that memory.)

To summarize the structure of the architecture that OpenMP targets:

- A node has up to four sockets;
- each socket has up to 60 cores;
- each core is an independent processing unit, with access to all the memory on the node.

### 14.1.2 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 computer 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*

1. In that picture you also see a co-processor: OpenMP is increasingly targeting those too.

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 14.3 gives a simple picture of this: a thread forks into a team of threads, and these threads themselves can fork again.

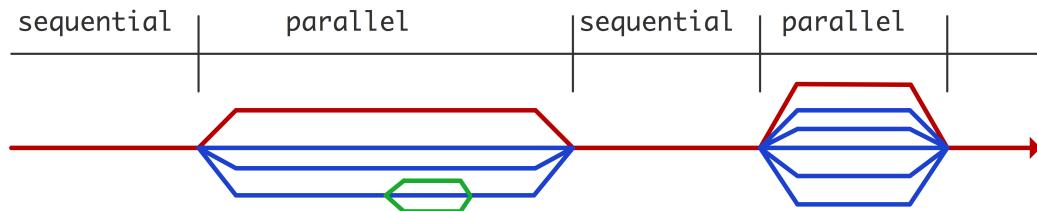


Figure 14.3: Thread creation and deletion during parallel execution

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 parallelism 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. This is known as as *dynamic mode*. 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.

### 14.1.3 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: 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  $2\times$  or  $4\times$  the number of cores, depending on the hardware.

### 14.1.4 About 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
}
```

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.

Fortran has simpler rules, since it does not have blocks inside blocks.

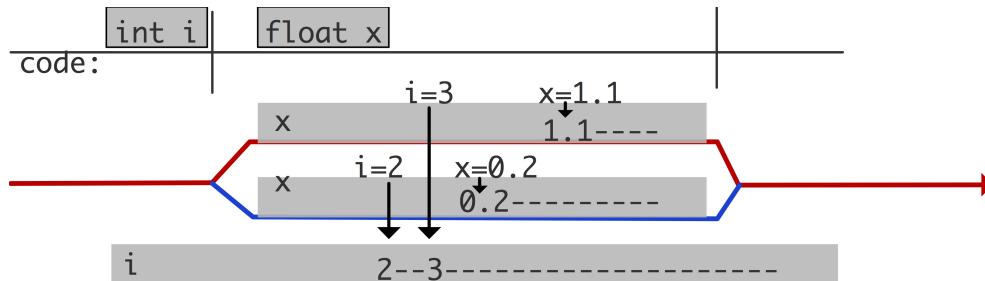


Figure 14.4: Locality of variables in threads

In OpenMP the situation is a bit more tricky because of the threads. When a team of threads is created they can all see the data of the master thread. However, they can also create data of their own. This is illustrated in figure 14.5. We will go into the details later.

## 14.2 Compiling and running an OpenMP program

### 14.2.1 Compiling

Your file or Fortran module needs to contain

```
|| #include "omp.h"
```

in C, and

```
|| use omp_lib
```

or

```
|| #include "omp_lib.h"
```

for Fortran.

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 -openmp
```

If you have separate compile and link stages, you need that option in both.

When you use the openmp compiler option, a *cpp* variable `_OPENMP` will be defined. Thus, you can have conditional compilation by writing

```
|| #ifdef _OPENMP
    ...
|| #else
    ...
|| #endif
```

### 14.2.2 Running an OpenMP program

You run an OpenMP program by invoking it the regular way (for instance `./a.out`), but its behaviour 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. See section 25.1 for a list of all environment variables.

## 14.3 Your first OpenMP program

In this section you will see just enough of OpenMP to write a first program and to explore its behaviour. For this we need to introduce a couple of OpenMP language constructs. They will all be discussed in much greater detail in later chapters.

### 14.3.1 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)
||     parallel statement;

|| #pragma omp somedirective clause(value,othervalue)
|| {
||     parallel statement 1;
||     parallel 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.

The sentinel in Fortran looks like a comment:

```
|| !$omp directive clause(value)
||     statements
|| !$omp end directive
```

The difference with the C directive is that Fortran can not have a block, so there is 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 do &
|| %OMP    copyin(x), copyout(y)
```

The directives are case-insensitive. In *Fortran fixed-form source files*, `C$omp` and `*$omp` are allowed too.

### 14.3.2 Parallel regions

The simplest way to create parallelism in OpenMP is to use 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 *Single Program Multiple Data (SPMD)* model: all threads execute the same segment of code.

```

||| #pragma omp parallel
||{
|||   // this is executed by a team of threads
||}

```

We will go into much more detail in section 15.

### 14.3.3 An actual OpenMP program!

**Exercise 14.1.** Write a program that contains the following lines:

```

||| printf("There are %d processors\n",omp_get_num_procs());
||| #pragma omp parallel
|||   printf("There are %d threads\n",
|||         /* !!!! something missing here !!!! */ );

```

The first print statement tells you the number of available cores in the hardware.

Your assignment is to supply the missing function that reports the number of threads used. Compile and run the program. Experiment with the OMP\_NUM\_THREADS environment variable. What do you notice about the number of lines printed?

**Exercise 14.2.** Extend the program from exercise 14.1. Make a complete program based on these lines:

```

||| int tsum=0;
||| #pragma omp parallel
|||   tsum += /* the thread number */
||| printf("Sum is %d\n",tsum);

```

Compile and run again. (In fact, run your program a number of times.) Do you see something unexpected? Can you think of an explanation?

### 14.3.4 Code and execution structure

Here are a couple of important concepts:

#### Definition 1

**structured block** 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.

**construct** 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.

**region of code** 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.

## 14.4 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. First of all, 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:

```
|| {
||   int x;
||   #pragma omp parallel
||   {
||     double x;
||   }
|| }
```

There is an important difference: each thread in the team gets its own instance of the enclosed variable.

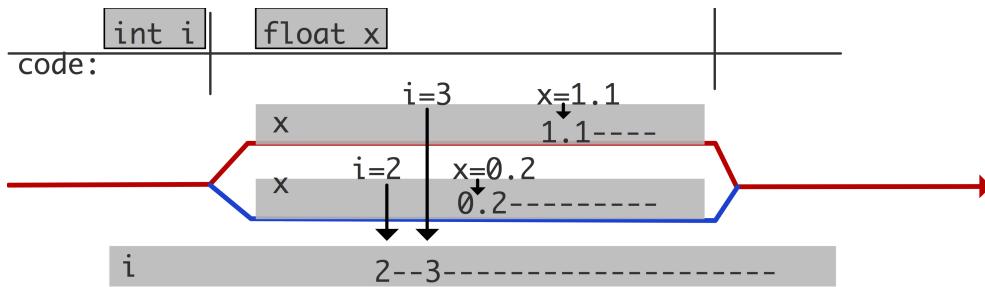


Figure 14.5: Locality of variables in threads

This is illustrated in figure 14.5.

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, you not guarantee an order on the updates.

We will discuss all this in detail in section 18.

## 14.5 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 section 15. The division of the work over the threads is controlled by *work sharing construct* (section 17).
2. Alternatively, you can use tasks and indicating one parallel activity at a time. You will see this in section 21.

Note that OpenMP only indicates how much parallelism is present; whether independent activities are in fact executed in parallel is a runtime decision. The factors influencing this are discussed in section 14.5.

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 25.1 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 clause on a parallel region.

If you specify a greater amount of parallelism than the hardware supports, the runtime system will probably ignore your specification and choose a lower value. 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`.

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. The variable `OMP_NESTED` controls whether the inner region will create a team of more than one thread.

**14.6 Sources used in this chapter**

**14.6.1 Listing of code header**

## Chapter 15

### OpenMP topic: Parallel regions

The simplest way to create parallelism in OpenMP is to use 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. There is also a function `omp_get_num_threads` to find out the total number of threads. Both these functions give a number relative to the current team; recall from figure 14.3 that new teams can be created recursively.

For instance, if you program computes

```
|| 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;
```

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:

```
|| // hello.c
|| #pragma omp parallel
|| {
||   int t = omp_get_thread_num();
||   printf("Hello world from %d!\n", t);
|| }
```

*For the source of this example, see section 15.3.2*

or in Fortran

```
// hellocount.F90
!$omp parallel
  nthreads = omp_get_num_threads()
  mythread = omp_get_thread_num()
  write(*,'("Hello from",i3," out of",i3)') mythread,nthreads
!$omp end parallel
```

*For the source of this example, see section 15.3.3*

or in C++

```
// hello.hxx
#pragma omp parallel
{
  int t = omp_get_thread_num();
  stringstream proctext;
  proctext << "Hello world from " << t << endl;
  cerr << proctext.str();
}
```

*For the source of this example, see section 15.3.4*

(Note the use of `stringstream`: without that the output lines from the various threads may get mixed up.)

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 15** *In future versions of OpenMP, the master thread will be called the primary thread. In 5.1 the master construct will be deprecated, and masked (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 15.1.** Make a full program based on this fragment. Insert different print statements before, inside, and after the parallel region. Run this example. How many times is each print statement executed?

You see that the `parallel` directive

- Is preceded by a special marker: a `#pragma omp` for C/C++, and the `!$OMP sentinel` for Fortran;

- Is followed by a single statement or a block in C/C++, or followed by a block in Fortran which is delimited by an `!$omp end` directive.

Directives look like *cpp directives*, but they are actually handled by the compiler, not the preprocessor.

**Exercise 15.2.** Take the ‘hello world’ program above, and modify it so that you get multiple messages to your screen, saying

```
Hello from thread 0 out of 4!  
Hello from thread 1 out of 4!
```

and so on. (The messages may very well appear out of sequence.)

What happens if you set your number of threads larger than the available cores on your computer?

**Exercise 15.3.** What happens if you call `omp_get_thread_num` and `omp_get_num_threads` outside a parallel region?

|| **omp\_get\_thread\_limit**

`OMP_WAIT_POLICY` values: ACTIVE, PASSIVE

## 15.1 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  
||   {  
||     ...  
||   }  
|| }
```

By default, the nested parallel region will have only one thread. To allow nested thread creation, set

```
OMP_NESTED=true  
or  
omp_set_nested(1)
```

**Exercise 15.4.** 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`.

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.

```
OMP_MAX_ACTIVE_LEVELS=123

omp_set_max_active_levels( n )
n = omp_get_max_active_levels()

OMP_THREAD_LIMIT=123

n = omp_get_thread_limit()

omp_set_max_active_levels
omp_get_max_active_levels
omp_get_level
omp_get_active_level
```

```
omp_get_ancestor_thread_num  
omp_get_team_size(level)
```

## 15.2 Cancel parallel construct

```
|| !$omp cancel construct [if (expr)]
```

where construct is *parallel*, **sections**, **do** or *taskgroup*

### 15.3 Sources used in this chapter

#### 15.3.1 Listing of code header

#### 15.3.2 Listing of code examples/omp/c/hello.c

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc,char **argv) {

#pragma omp parallel
{
    int t = omp_get_thread_num();
    printf("Hello world from %d!\n",t);
}

return 0;
}
```

#### 15.3.3 Listing of code examples/omp/f/hellocount.F90

```
use omp_lib
integer :: nthreads,mythread

 !$omp parallel
    nthreads = omp_get_num_threads()
    mythread = omp_get_thread_num()
    write(*,'(Hello from',i3," out of",i3)') mythread,nthreads
 !$omp end parallel

end Program Hello
```

#### 15.3.4 Listing of code examples/omp/cxx/hello.cxx

```
#include <iostream>
using std::cerr;
using std::cout;
using std::endl;
#include <sstream>
using std::stringstream;

#include <omp.h>

int main(int argc,char **argv) {

#pragma omp parallel
```

```
{  
    int t = omp_get_thread_num();  
    stringstream proctext;  
    proctext << "Hello world from " << t << endl;  
    cerr << proctext.str();  
}  
  
return 0;  
}
```

# Chapter 16

## OpenMP topic: Loop parallelism

### 16.1 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 17 for the full list): constructs that take an amount of work and distribute it over the available threads in a parallel region.

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 (i=low; i<high; i++)
        // do something with i
}
```

A more natural option is to use the `parallel for` pragma:

```
#pragma omp parallel
#pragma omp for
for (i=0; i<N; i++) {
    // do something with i
}
```

This has several advantages. For one, you don’t have to calculate the loop bounds for the threads yourself, but you can also tell OpenMP to assign the loop iterations according to different schedules (section 16.2).

Figure 16.1 shows the execution on four threads of

```
#pragma omp parallel
{
    code1();
#pragma omp for
    for (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.

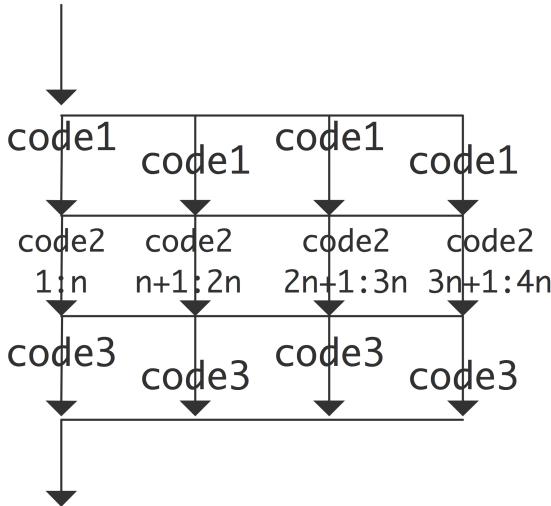


Figure 16.1: Execution of parallel code inside and outside a loop

Note that the `parallel do` and `parallel for` pragmas do not create a team of threads: 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. It is also possible to have a combined `omp parallel for` or `omp parallel do` directive.

If your parallel region only contains a loop, you can combine the pragmas for the parallel region and distribution of the loop iterations:

```

#pragma omp parallel for
for (i=0; ....

```

**Exercise 16.1.** Compute  $\pi$  by *numerical integration*. We use the fact that  $\pi$  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.

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 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.

Use different numbers of cores and compute the speedup you attain over the sequential computation. Is there a performance difference between the OpenMP code with 1 thread and the sequential code?

**Remark 16** 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 HPSC-?? for more explanation.

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.
- The `continue (C)` or `cycle (F)` statement is allowed.
- The index update has to be an increment (or decrement) by a fixed amount.
- The loop index variable is automatically private, and not changes to it inside the loop are allowed.

## 16.2 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 16.2 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.

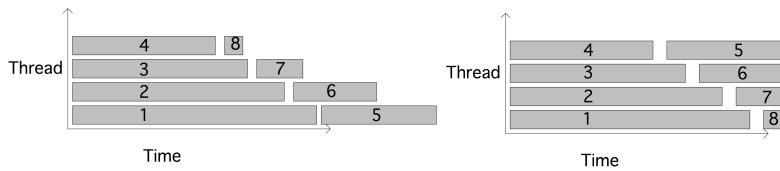


Figure 16.2: Illustration static round-robin scheduling versus dynamic

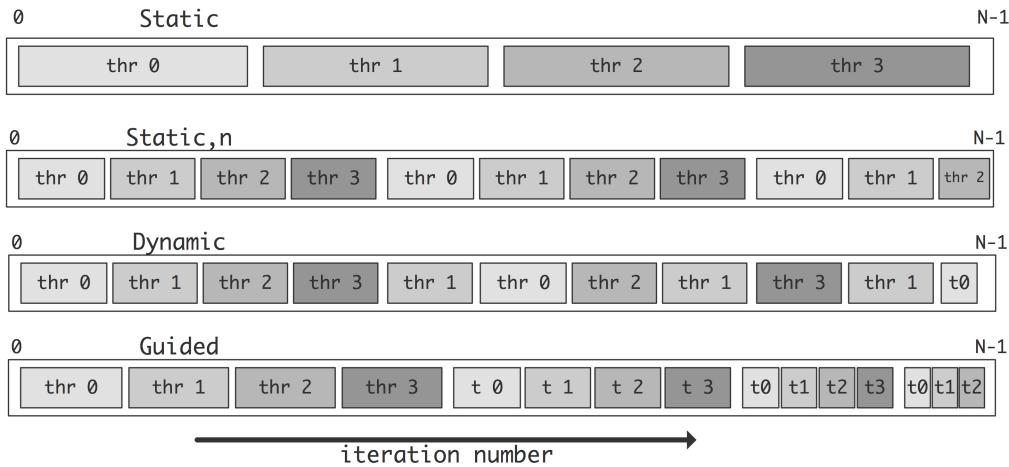


Figure 16.3: Illustration of the scheduling strategies of loop iterations

The default static schedule is to assign one consecutive block of iterations to each thread. If you want different sized blocks you can defined a chunk size:

```
|| #pragma omp for schedule(static[,chunk])
```

(where the square brackets indicate an optional argument). With static scheduling, the compiler will split up the loop iterations at compile time, so, provided the iterations take roughly the same amount of time, this is the most efficient at runtime.

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 16.2.** Why is a chunk size of 1 typically a bad idea? (Hint: think about cache lines, and read HPSC-??.)

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 16.3.

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 16.3.** We continue with exercise 16.1. We add ‘adaptive integration’: where needed, the program refines the step size<sup>1</sup>. This means that the iterations no longer take a predictable amount of time.

```

|| for (i=0; i<nsteps; i
||   ++
||   double
||     x = i*h, x2 = (i+1)*
||     h,
||     y = sqrt(1-x*x), y2
||     = sqrt(1-x2*x2),
||     slope = (y-y2)/h;
||     if (slope>15) slope
||       = 15;
||     int
||     samples = 1+(int)
||     slope, is;
||   }
||   for (is=0; is<
samples; is++) {
||     double
||       hs = h/samples,
||       xs = x+ is*hs,
||       ys = sqrt(1-xs*
||         xs);
||       quarterpi += hs
||       *ys;
||       nsamples++;
||     }
||   pi = 4*quarterpi;

```

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, using up to 8 cores. However, it is possible to get completely linear speedup. For this you need to adjust 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 16.4.** 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`.

```
|| 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>	modifier default
static	<code>static[,n]</code>	<code>static[,n]</code>	$N/nthreads$
dynamic	<code>dynamic[,n]</code>	<code>dynamic[,n]</code>	1
guided	<code>guided[,n]</code>	<code>guided[,n]</code>	

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`

**dynamic** Value: 2. The modifier parameter is the `chunk` size; default 1. 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`

**static** Value: 1. The modifier parameter is the `chunk` size. Set by using value `omp_sched_static`

## 16.3 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 section 19.

## 16.4 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 ( i=0; i<N; i++ )
||   for ( 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 ( i=0; i<N; i++ )
||   for ( j=0; j<N; j++ )
||     A[i][j] = B[i][j] + C[i][j]
```

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 ( i=0; i<N; i++ ) {
||   y[i] = 0. ;
||   for ( j=0; j<N; j++ )
||     y[i] += A[i][j] * x[j]
|| }
```

can not be collapsed.

**Exercise 16.5.** Can you rewrite the preceding code example so that it can be collapsed? Do timing tests to see if you can notice the improvement from collapsing.

## 16.5 Ordered iterations

Iterations in a parallel loop that are execution 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.

## 16.6 nowait

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 (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 (i=0; i<N; i++) {
|||    x[i] = ...
|||
|||#pragma omp for schedule(static)
|||    for (i=0; i<N; i++) {
|||        y[i] = ... x[i] ...
|||
||}

```

## 16.7 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:

```

||while ( a[i] !=0 && i<iMax ) {
||    i++;
||    // now i is the first index for which \n{a[i]} is zero.

```

We replace the while loop by a for loop that examines all locations:

```
|| result = -1;
|| #pragma omp parallel for
|| for (i=0; i<imax; i++) {
||   if (a[i]!=0 && result<0) result = i;
|| }
```

**Exercise 16.6.** Show that this code has a race condition.

You can fix the race condition by making the condition into a critical section; section 20.2.1. 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 (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.

**16.8      Sources used in this chapter**

**16.8.1    Listing of code header**

## Chapter 17

### 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. The work sharing constructs are:

- `for` (for C) or `do` (for Fortran). The threads divide up the loop iterations among themselves; see section 16.1.
- `sections` The threads divide a fixed number of sections between themselves; see section 17.1.
- `single` The section is executed by a single thread; section 17.2.
- `task` See section 21.
- `workshare` Can parallelize Fortran array syntax; section 17.3.

#### 17.1 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)
```

You could then write

```

|| double y1,y2;
|| #pragma omp sections
{
|| #pragma omp section
||     y1 = f(x)
|| #pragma omp section
||     y2 = g(x)
}
y = y1+y2;

```

Instead of using two temporaries, you could also use a critical section; see section 20.2.1. However, the best solution is have a reduction clause on the `sections` directive:

```

|| y = f(x) + g(x)

```

You could then write

```

|| y = 0;
|| #pragma omp sections reduction(+:y)
{
|| #pragma omp section
||     y += f(x)
|| #pragma omp section
||     y += g(x)
}

```

## 17.2 Single/master

The `single` and `master` pragma limit 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 23.3).

**Exercise 17.1.** What is the difference between this approach and how the same computation would be parallelized in MPI?

The `master` directive, also enforces execution on a single thread, specifically the master thread of the team, but it does not have the synchronization through the implicit barrier.

**Exercise 17.2.** 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.

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 17.3.** 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?

### 17.3 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.

The `workshare` directive exists only in Fortran. It can be used to parallelize the implied loops in *array syntax*, as well as *forall* loops.

**17.4 Sources used in this chapter**

**17.4.1 Listing of code header**

## Chapter 18

### 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.

#### 18.1 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. See [HPSC-??](#) for an explanation of the issues involved; see [20.2.1](#) for a solution in OpenMP.

Sometimes this global update is what you want; in other cases the variable is intended only for intermediate results in a computation. In that case there are various ways of creating data that is local to a thread, and therefore invisible to other threads.

#### 18.2 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.

Example:

```
|| int x = 5;
|| #pragma omp parallel
|| {
||     int x; x = 3;
||     printf("local: x is %d\n", x);
|| }
```

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.

The Fortran language does not have this concept of scope, so you have to use a `private` clause:

```
|| !$OMP parallel private(x)
```

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 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); // also dangerous
```

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 megabyte; you can control it with the environment variable `OMP_STACKSIZE`. Its values can be literal or with suffixes:

```
123 456k 567K 678m 789M 246g 357G
```

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.

### 18.3 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.

## 18.4 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 redefine the variables 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.

## 18.5 Default

- Loop variables in an `omp for` are private;
- Local variables in the parallel region are private.

You can alter this default behaviour 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 behaviour.
- 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.

## 18.6     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: in

```
// alloc3.c
int array[nthreads];
{
    int t = 2;
    array += t;
    array[0] = t;
}
```

*For the source of this example, see section 18.9.2*

each thread gets a private copy of the array, properly initialized.

On the other hand, in

```
// alloc1.c
int *array = (int*) malloc(nthreads*sizeof(int));
#pragma omp parallel firstprivate(array)
{
    int t = omp_get_thread_num();
    array += t;
    array[0] = t;
}
```

*For the source of this example, see section 18.9.3*

each thread gets a private pointer, but all pointers point to the same object.

## 18.7 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.

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 (i=0; i<N; i++) {
||     tmp = .....
||     x[i] = .... tmp ....
|| }
.... tmp ....
```

## 18.8 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.

### 18.8.1 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
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

static int tp;

int main(int argc, char **argv) {

#pragma omp threadprivate(tp)

#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);

    return 0;
}
```

For the source of this example, see section [18.9.4](#)

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();
    ...
}
```

## 18.8.2 Thread private example

The typical application for thread-private variables is in *random number generation*. 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 18.1.** 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.

*Fortran note.* Named common blocks can be made thread-private with the syntax

```
|| $!OMP threadprivate( /blockname/ )
```

Threadprivate variables require `OMP_DYNAMIC` to be switched off.

## 18.9 Sources used in this chapter

### 18.9.1 Listing of code header

### 18.9.2 Listing of code examples/omp/c/alloc2.c

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc,char **argv) {

    int nthreads;
#pragma omp parallel
#pragma omp master
    nthreads = omp_get_num_threads();

    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;
}

printf("Array result:\n");
for (int i=0; i<nthreads; i++)
    printf("%d:%d, ",i,array[i]);
printf("\n");

return 0;
}
```

### 18.9.3 Listing of code examples/omp/c/alloc1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc,char **argv) {

    int nthreads;
#pragma omp parallel
#pragma omp master
    nthreads = omp_get_num_threads();

    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();
    array += t;
    array[0] = t;
}

printf("Array result:\n");
for (int i=0; i<nthreads; i++)
    printf("%d:%d, ", i, array[i]);
printf("\n");

return 0;
}
```

#### 18.9.4 Listing of code examples/omp/c/threadprivate.c

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

static int tp;

int main(int argc, char **argv) {

#pragma omp threadprivate(tp)

#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);

    return 0;
}
```

## Chapter 19

### OpenMP topic: Reductions

Parallel tasks often produce some quantity that needs to be summed or otherwise combined. In section 15 you saw an example, and it was stated that the solution given there was not very good.

The problem in that example was the *race condition* involving the `result` variable. The simplest solution 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 19.1.** Can you think of a small modification of this code, that still uses a critical section, that is more efficient? Time both codes.

The easiest way to effect a reduction is of course to use the `reduction` clause. Adding this to an `omp for` or an `omp sections` construct 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 loop, the local results are combined, again using the reduction operator, into the global variable.

This 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. See HPSC-?? for more explanation.

If your code can not be easily structure 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 phenomemon called *false sharing*. Even though the threads write to separate variables, those variables are likely to be on the same *cacheline* (see HPSC-?? 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;
}
```

## 19.1 Built-in reduction operators

Arithmetic reductions:  $+$ ,  $*$ ,  $-$ ,  $\max$ ,  $\min$

Logical operator reductions in C:  $\&$   $\&\&$   $|$   $||$   $\wedge$

Logical operator reductions in Fortran:  $.and.$   $.or.$   $.eqv.$   $.neqv.$   $.iand.$   $.ior.$   $.ieor.$

**Exercise 19.2.** The maximum and minimum reductions were not added to OpenMP until version 3.1. Write a parallel loop that computes the maximum and minimum values in an array. Discuss the various options. Do timings to evaluate the speedup that is attained and to find the best option.

## 19.2 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.

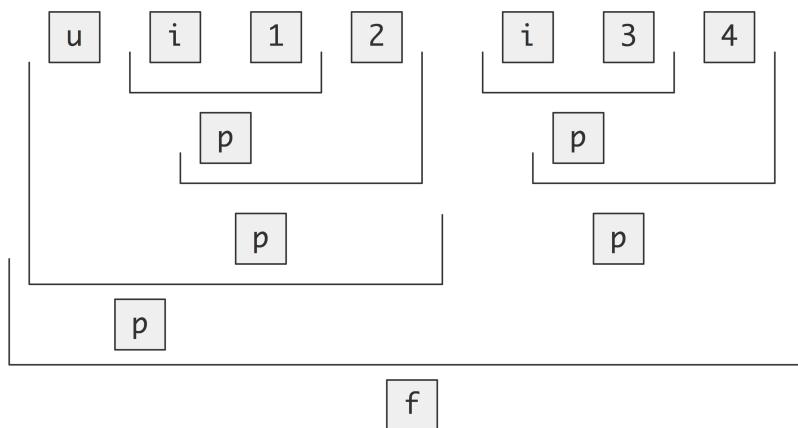


Figure 19.1: Reduction of four items on two threads, taking into account initial values.

Figure 19.1 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 19.3.** Write a program to test the fact that the partial results are initialized to the unit of the reduction operator.

### 19.3 User-defined reductions

With *user-defined reductions*, the programmer specifies the function that does the elementwise comparison. 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.

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]);
```

For the source of this example, see section 19.5.2

**Exercise 19.4.** Write a reduction routine that operates on an array of non-negative integers, finding the smallest nonzero one. If the array has size zero, or entirely consists of zeros, return `-1`.

Support for *C++ iterators*

```
|| #pragma omp declare reduction (merge : std::vector<int>
| : omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()))
```

## 19.4 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 HPSC-?. OpenMP ignores this issue: it is the programmer's job to ensure proper rounding behaviour.

## 19.5 Sources used in this chapter

### 19.5.1 Listing of code header

#### 19.5.2 Listing of code examples/omp/c/ireduct.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <omp.h>

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;
    }
    printf("combine %d %d : %d\n",r,n,m);
    return m;
}

int main(int argc,char **argv) {
    int m,ndata = 4, data[4] = {2,-3,0,5};

    #pragma omp declare reduction \
        (rwz:int:omp_out=mymax(omp_out,omp_in)) \
        initializer(omp_priv=INT_MIN)

    m = INT_MIN;
    for (int idata=0; idata<ndata; idata++)
        m = mymax(m,data[idata]);
    if (m!=5)
        printf("Sequential: wrong reduced value: %d, s/b %d\n",m,2);
    else
        printf("Sequential case succeeded\n");

    m = INT_MIN;
    #pragma omp parallel for reduction(rwz:m)
    for (int idata=0; idata<ndata; idata++)
        m = mymax(m,data[idata]);

    if (m!=5)
        printf("Parallel: wrong reduced value: %d, s/b %d\n",m,2);
    else
        printf("Finished\n");

    return 0;
}
```

}

## Chapter 20

### 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 [20.2.1](#).
- `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 [20.1](#).
- `ordered`: section [16.5](#).
- `locks`: section [20.3](#).
- `flush`: section [23.3](#).
- `nowait`: section [16.6](#).

#### 20.1 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];
}
```

Apart from the barrier directive, which inserts an explicit barrier, OpenMP has *implicit barriers* after a load sharing construct. 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.

### 20.1.1 Implicit barriers

At the end of a parallel region the team of threads is dissolved and only the master thread continues. Therefore, there is an *implicit barrier at the end of a parallel region*.

There is some *barrier behaviour* associated with `omp for` loops and other *worksharing constructs* (see section 17.3). For instance, there is an *implicit barrier* at the end of the loop. This barrier behaviour can be cancelled 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.

## 20.2 Mutual exclusion

Sometimes it is necessary to let only one thread execute a piece of code. Such a piece of code is called a *critical section*, and OpenMP has several mechanisms for realizing this.

The most common use of critical sections is to update a variable. Since updating involves reading the old value, and writing back the new, this has the possibility for a *race condition*: another thread reads the current value before the first can update it; the second thread updates to the wrong value.

Critical sections are an easy way to turn an existing code into a correct parallel code. However, there are disadvantages to this, and sometimes a more drastic rewrite is called for.

### 20.2.1 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.

The typical application of a critical section is to update a variable:

```
|| #pragma omp parallel
|| {
||     int mytid = omp_get_thread_num();
||     double tmp = some_function(mytid);
||     #pragma omp critical
||         sum += tmp;
|| }
```

**Exercise 20.1.** 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.

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.

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.

The problem with `critical` sections being mutually exclusive can be mitigated by naming them:

```
|| #pragma omp critical (optional_name_in_parens)
```

## 20.3 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.

One simple example of the use of locks is generation of a *histogram*. A histogram consists of a number of bins, that get updated depending on some data. Here is the basic structure of such a code:

```
|| int count[100];
|| float x = some_function();
|| int ix = (int)x;
|| if (ix>=100)
||     error();
|| else
||     count[ix]++;
||
```

It would be possible to guard the last line:

```
|| #pragma omp critical
||     count[ix]++;
||
```

but that is unnecessarily restrictive. If there are enough bins in the histogram, and if the `some_function` takes enough time, there are unlikely to be conflicting writes. The solution then is to create an array of locks, with one lock for each `count` location.

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

```
|| omp_test_lock();
```

Unsetting a lock needs to be done by the thread that set it.

Lock operations implicitly have a `flush`.

**Exercise 20.2.** 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 */
```

### 20.3.1 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*

*lock—)*

## 20.4 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];
}

```

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 21):

```

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.

**20.5 Sources used in this chapter**

**20.5.1 Listing of code header**

## Chapter 21

### OpenMP topic: Tasks

Tasks are a mechanism that OpenMP uses under the cover: if you specify something as being parallel, 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.

Let’s look at a simple example using the `task` directive.

Code	Execution
<code>x = f();</code>	the variable <code>x</code> gets a value
<code>#pragma omp task</code> <code>{ y = g(x); }</code>	a task is created with the current value of <code>x</code>
<code>z = h();</code>	the variable <code>z</code> gets a value

The thread that executes this code segment creates a task, which will later be executed, probably by a different thread. The exact timing of the execution of the task is up to a *task scheduler*, which operates invisible to the user.

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
<code>p = head_of_list();</code>	one thread traverses the list
<code>while (!end_of_list(p)) {</code>	
<code>    #pragma omp task</code>	a task is created,
<code>    process(p);</code>	one for each element
<code>    p = next_element(p);</code>	the generating thread goes on without waiting
<code>}</code>	the tasks are executed while more are being generated.

The way tasks and threads interact is different from the worksharing constructs you’ve seen so far. Typically, one thread will generate the tasks, adding them to a queue, from which all threads can take and execute them. This leads to the following idiom:

```
|| #pragma omp parallel
|| #pragma omp single
{
|| ...
|| #pragma omp task
|| { ... }
```

```
|| } ...
```

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)

With tasks it becomes possible to parallelize processes that did not fit the earlier OpenMP constructs. For instance, if a certain operation needs to be applied to all elements of a linked list, you can have one thread go down the list, generating a task for each element of the list.

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 21.1.** 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. Start with this code:

```
int factor=0;
#pragma omp parallel
#pragma omp single
for (int f=2; f<4000; f++) {
    // see if 'f' is a factor
    if (N%f==0) { // found factor!
        factor = f;
    }
    if (factor>0)
        break;
}
if (factor>0)
    printf("Found a factor: %d\n", 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.

## 21.1 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 another. 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 when which possibility applies.

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. In the first example:

```
|| 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
```

the variable `count` is declared outside the parallel region and is therefore shared. When the print 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:

```
|| #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
```

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.

## 21.2 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.

In order to have a guarantee that a task is finished, you need 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 $x$ gets a value
<code>#pragma omp task { y1 = g1(x); }</code>	two tasks are created with the current value of $x$
<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 $z$ 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.

Another aspect of the distinction between generating tasks and executing them: usually the tasks are generated by one thread, but executed by many threads. Thus, the typical idiom is:

```
|| #pragma omp parallel
|| #pragma omp single
{
    // code that generates tasks
}
```

This makes it possible to execute loops in parallel that do not have the right kind of iteration structure for a `omp parallel for`. As an example, you could traverse and process a linked list:

```
|| #pragma omp parallel
|| #pragma omp single
{
    while (!tail(p)) {
        p = p->next();
        #pragma omp task
        process(p)
    }
    #pragma omp taskwait
}
```

One task traverses the linked list creating an independent task for each element in the list. These tasks are then executed in parallel; their assignment to threads is done by the task scheduler.

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, followed by a structured block, ensures completion of all tasks created in the block, even if recursively created.

3. Each OpenMP task can have a `depend` clause, indicating what *data dependency* of the task. By indicating what data is produced or absorbed by the tasks, the scheduler can construct the dependency graph for you.

Another mechanism for dealing with tasks is the `taskgroup`: a task group is a code block that can contain task directives; all these tasks need to be finished before any statement after the block is executed.

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.

### 21.3 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)
```

**Exercise 21.2.** 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 indicate 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)

```

## 21.4 More

### 21.4.1 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
|| ;

```

### 21.4.2 Task cancelling

It is possible (in *OpenMP version 4*) 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.

The directive `cancel` takes an argument of the surrounding construct (`parallel`, `for`, `sections`, `taskgroup`) in which the tasks are cancelled.

**Exercise 21.3.** Modify the prime finding example.

## 21.5 Examples

### 21.5.1 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];
```

*For the source of this example, see section 21.6.2*

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];
}
```

*For the source of this example, see section 21.6.3*

### 21.5.2 Binomial coefficients

**Exercise 21.4.** An array of binomial coefficients can be computed as follows:

```
// binomial11.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];
```

*For the source of this example, see section ??*

Putting a single task group around the double loop, and use depend clauses to make the execution satisfy the proper dependencies.

### 21.5.3 Tree traversal

OpenMP tasks are a great way of handling trees.

#### 21.5.3.1 Post-order traversal

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:

$$S = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}$$

where the two inverses  $A_{11}^{-1}, A_{22}^{-1}$  can be computed independently and recursively.

#### 21.5.3.2 Pre-order traversal

If a property needs to propagate from the root to all subtrees and nodes, you can use *pre-order tree traversal*:

```
Update node value  $s$ 
for all children c do
    update c with the new value  $s$ 
```

## 21.6 Sources used in this chapter

### 21.6.1 Listing of code header

### 21.6.2 Listing of code examples/omp/c/taskgroup0.c

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#include <string.h>

int main(int argc,char **argv) {
    int N = 1000;
    if (argc>1) {
        if (!strcmp(argv[1],"-h")) {
            printf("usage: %s [nnn]\n",argv[0]);
            return 0;
        }
        N = atoi(argv[1]);
        if (N>99) {
            printf("Sorry, this overflows: setting N=99\n");
            N = 99;
        }
    }
    long int *fibo_values = (long int*)malloc(N*sizeof(long int));
    fibo_values[0] = 1; fibo_values[1] = 1;
    {
        for (int i=2; i<N; i++)
        {
            fibo_values[i] = fibo_values[i-1]+fibo_values[i-2];
        }
    }
    printf("F(%d) = %ld\n",N,fibo_values[N-1]);
    return 0;
}
```

### 21.6.3 Listing of code examples/omp/c/taskgroup2.c

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#include <string.h>

int main(int argc,char **argv) {
    int N = 1000;
    if (argc>1) {
        if (!strcmp(argv[1],"-h")) {
```

```
    printf("usage: %s [nnn]\n", argv[0]);
    return 0;
}
N = atoi(argv[1]);
if (N>99) {
    printf("Sorry, this overflows: setting N=99\n");
    N = 99;
}
}

long int *fibo_values = (long int*)malloc(N*sizeof(long int));

fibo_values[0] = 1; fibo_values[1] = 1;
#pragma omp parallel
#pragma omp single
#pragma omp taskgroup
{
    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];
    }
}
printf("F(%d) = %ld\n",N,fibo_values[N-1]);

return 0;
}
```

## Chapter 22

### OpenMP topic: Affinity

#### 22.1 OpenMP thread affinity control

The matter of thread affinity becomes important on *multi-socket nodes*; see the example in section 22.2.

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.

##### 22.1.1 Thread binding

The variable `OMP_PLACES` defines a series of places to which the threads are assigned.

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 `proc_bind` clause rather than the environment variable, set to `spread` for the initial team, and to `master` for the recursively created team.

### 22.1.2 Effects of thread binding

Let's consider two example program. First we consider the program for computing  $\pi$ , 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

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.

### 22.1.3 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 placed 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.

#### 22.1.4 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 `proc_bind` 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++;
        }
    }
}
```

*For the source of this example, see section 22.4.2*

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.

## 22.2 First-touch

The affinity issue shows up in the *first-touch* phenomenon. Memory allocated with `malloc` and like routines is not actually allocated; 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 master 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.

**Exercise 22.1.** Finish the following fragment and run it with first all the cores of one socket, then all cores of both sockets. (If you know how to do explicit placement, you can also try fewer cores.)

```
for (int i=0; i<nlocal+2; i++)
    in[i] = 1.;
for (int i=0; i<nlocal; i++)
    out[i] = 0.;

for (int step=0; step<nsteps; step++) {
#pragma omp parallel for schedule(static)
    for (int i=0; i<nlocal; i++) {
        out[i] = (in[i]+in[i+1]+in[i+2])/3.;
    }
#pragma omp parallel for schedule(static)
    for (int i=0; i<nlocal; i++)
```

```
    in[i+1] = out[i];
    in[0] = 0; in[nlocal+1] = 1;
}
```

**Exercise 22.2.** How do the OpenMP dynamic schedules relate to this?

C++ valarray does initialization, so it will allocate memory on thread 0.

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` [13]. 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.

## 22.3 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: `pbing` 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
```

**22.4 Sources used in this chapter****22.4.1 Listing of code header****22.4.2 Listing of code examples/omp/c/sharing.c**

```
#include <stdio.h>
#include <omp.h>

int main() {

    int i,j;
    int reps = 1000;
    int N = 8*10000;

    double start, stop, delta;
    double a;

#pragma omp parallel
    a = 0;

    start = omp_get_wtime();
#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++;
        }
    }
    stop = omp_get_wtime();
    delta = ((double)(stop - start))/reps;
    printf("run time = %fusec\n", 1.0e6*delta);

    printf("sum = %.1f\n", a);

    return 0;
}
```

## Chapter 23

### OpenMP topic: Memory model

#### 23.1 Thread synchronization

Let's do a *producer-consumer* model<sup>1</sup>. 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)
```

and the consumer does:

---

1. This example is from Intel's excellent OMP course by Tim Mattson

```

||| #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;
|||
}

```

## 23.2 Data races

OpenMP, being based on shared memory, has a potential for *race conditions*. These happen when two threads access the same data item. The problem with race conditions is that programmer convenience runs counter to efficient execution. For this reason, OpenMP simply does not allow some things that would be desirable.

For a simple example:

```

// race.c
#pragma omp parallel for shared(counter)
for (int i=0; i<count; i++)
    counter++;
printf("Counter should be %d, is %d\n",
       count, counter);

```

*For the source of this example, see section 23.4.2*

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.

As an illustration of a possible problem:

```

c = d = 0;
#pragma omp sections
{

```

```
|| #pragma omp section
  { a = 1; c = b; }
|| #pragma omp section
  { b = 1; d = a; }
}
```

Under any reasonable interpretation of parallel execution, the possible values for `c`, `d` 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 HPSC-??.)

However, without synchronization, threads are allowed to maintain a value for a variable locally that is not the same as the stored value. In this example, that means that the thread executing the first section need not write its value of `a` to memory, and likewise `b` in the second thread, so 0, 0 is in fact a possible outcome.

In order to resolve multiple accesses:

1. Thread one reads the variable.
2. Thread one flushes the variable.
3. Thread two flushes the variable.
4. Thread two reads the variable.

### 23.3 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.

## 23.4 Sources used in this chapter

### 23.4.1 Listing of code header

#### 23.4.2 Listing of code examples/omp/c/race.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <omp.h>

int main(int argc, char **argv) {

    int count = 100000;
    int counter = 0;

#pragma omp parallel for shared(counter)
    for (int i=0; i<count; i++)
        counter++;
    printf("Counter should be %d, is %d\n",
count,counter);

    return 0;
}
```

## Chapter 24

### OpenMP topic: SIMD processing

You can declare a loop to be executable with *vector instructions* with `simd`

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.

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:

```
// tools.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);
||}

```

For the source of this example, see section [24.1.2](#)

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          4 <- y1*y2
vmulpd    %xmm1, %xmm1, %xmm5          5 <- x1*x2
vbroadcastsd .L_2i10floatpacket.0(%rip), %zmm21
movl      $3, %eax
vbroadcastsd .L_2i10floatpacket.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_2i10floatpacket.1(%rip){1to8}, %zmm0, %zmm3{ %k1 } #25.26 c15
vmovaps   %zmm4, %zmm26
vmovapd   %zmm2, %zmm7{ %k3 }{ z }
vpcmpgtq %zmm2, %zmm21, %k2{ %k3 }
vscalefpd .L_2i10floatpacket.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_2i10floatpacket.2(%rip){1to8}, %zmm16, %zmm17{ %k3 }{ z } #25.26 c28
vmulpd    {rn-sae}, %zmm7, %zmm20, %zmm23{ %k3 }{ z } #25.26 c29
vscalefpd .L_2i10floatpacket.2(%rip){1to8}, %zmm20, %zmm22{ %k3 }{ z } #25.26 c30
vfnmadd231pd {rn-sae}, %zmm17, %zmm19, %zmm18{ %k3 } #25.26 c33 stall 1
vfnmadd231pd {rn-sae}, %zmm22, %zmm23, %zmm21{ %k3 } #25.26 c35
vfmadd231pd {rn-sae}, %zmm19, %zmm18, %zmm19{ %k3 } #25.26 c39 stall 1
vfmadd231pd {rn-sae}, %zmm23, %zmm21, %zmm23{ %k3 } #25.26 c41
vfmadd213pd {rn-sae}, %zmm17, %zmm17, %zmm18{ %k3 } #25.26 c45 stall 1
vfnmadd231pd {rn-sae}, %zmm19, %zmm19, %zmm3{ %k3 } #25.26 c47
vfmadd213pd {rn-sae}, %zmm22, %zmm22, %zmm21{ %k3 } #25.26 c51 stall 1
vfnmadd231pd {rn-sae}, %zmm23, %zmm23, %zmm7{ %k3 } #25.26 c53
vfmadd213pd %zmm19, %zmm18, %zmm3{ %k3 } #25.26 c57 stall 1
vfmadd213pd %zmm23, %zmm21, %zmm7{ %k3 } #25.26 c59
vscalefpd .L_2i10floatpacket.3(%rip){1to8}, %zmm3, %zmm3{ %k1 } #25.26 c63 st
vscalefpd .L_2i10floatpacket.3(%rip){1to8}, %zmm7, %zmm7{ %k2 } #25.26 c65
vfixupimmpd $112, .L_2i10floatpacket.4(%rip){1to8}, %zmm0, %zmm3{ %k3 } #25.26 c66
vfixupimmpd $112, .L_2i10floatpacket.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));
|| }
```

## 24.1 Sources used in this chapter

### 24.1.1 Listing of code header

### 24.1.2 Listing of code code/omp/c/simd/tools.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#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);
}
```

# Chapter 25

## OpenMP remaining topics

### 25.1 Runtime functions and 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:

- `omp_set_num_threads`
- `omp_get_num_threads`
- `omp_get_max_threads`
- `omp_get_thread_num`
- `omp_get_num_procs`
- `omp_in_parallel`
- `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
- `OMP_DISPLAY_ENV` Show OpenMP version and environment variables

- `OMP_DEFAULT_DEVICE` Set the device used in target regions
- `OMP_DYNAMIC` Dynamic adjustment of threads
- `OMP_MAX_ACTIVE_LEVELS` Set the maximum number of nested parallel regions
- `OMP_MAX_TASK_PRIORITY` Set the maximum task priority value
- `OMP_NESTED` Nested parallel regions
- `OMP_NUM_THREADS` Specifies the number of threads to use
- `OMP_PROC_BIND` Whether threads may be moved between CPUs
- `OMP_PLACES` Specifies on which CPUs the threads should be placed
- `OMP_STACKSIZE` Set default thread stack size
- `OMP_SCHEDULE` How threads are scheduled
- `OMP_THREAD_LIMIT` Set the maximum number of threads
- `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 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`.

Non-obvious 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 (see for instance section 15).
- `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; the space for private variables.
- `OMP_WAIT_POLICY` determines the behaviour 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.

## 25.2 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 25.1.** 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.

## 25.3 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*.

## 25.4 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 HPSC-??). 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 22.2.

**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 22.1.1 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 [22.1.2](#).)

**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.

## 25.5 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 a`: place data
- `update`: make data consistent between host and device

**25.6 Sources used in this chapter**

**25.6.1 Listing of code header**

# **Chapter 26**

## **OpenMP Review**

### **26.1 Concepts review**

#### **26.1.1 Basic concepts**

- process / thread / thread team
- threads / cores / tasks
- directives / library functions / environment variables

#### **26.1.4 Data scope**

- shared vs private, C vs F
- loop variables and reduction variables
- default declaration
- firstprivate, lastprivate

#### **26.1.2 Parallel regions**

execution by a team

#### **26.1.5 Synchronization**

- barriers, implied and explicit
- nowait
- critical sections
- locks, difference with critical

#### **26.1.3 Work sharing**

- loop / sections / single / workshare
- implied barrier
- loop scheduling, reduction
- sections
- single vs master
- (F) workshare

#### **26.1.6 Tasks**

- generation vs execution
- dependencies

## 26.2 Review questions

### 26.2.1 Directives

What do the following program output?

```
int main() {
    printf("procs %d\n",
        omp_get_num_procs());
    printf("threads %d\n",
        omp_get_num_threads());
    printf("num %d\n",
        omp_get_thread_num());
    return 0;
}
```

```
int main() {
#pragma omp parallel
{
    printf("procs %d\n",
        omp_get_num_procs());
    printf("threads %d\n",
        omp_get_num_threads());
    printf("num %d\n",
        omp_get_thread_num());
}
return 0;
}
```

```
Program main
use omp_lib
print *, "Procs:", &
omp_get_num_procs()
print *, "Threads:", &
omp_get_num_threads()
print *, "Num:", &
omp_get_thread_num()
End Program
```

```
Program main
use omp_lib
!$OMP parallel
print *, "Procs:", &
omp_get_num_procs()
print *, "Threads:", &
omp_get_num_threads()
print *, "Num:", &
omp_get_thread_num()
!$OMP end parallel
End Program
```

### 26.2.2 Parallelism

Can the following loops be parallelized? If so, how? (Assume that all arrays are already filled in, and that there are no out-of-bounds errors.)

```
// variant #1
for (i=0; i<N; i++) {
    x[i] = a[i]+b[i+1];
    a[i] = 2*x[i] + c[i+1];
}
```

```
// variant #3
for (i=1; i<N; i++) {
    x[i] = a[i]+b[i+1];
    a[i] = 2*x[i-1] + c[i+1];
}
```

```
// variant #2
for (i=0; i<N; i++) {
    x[i] = a[i]+b[i+1];
    a[i] = 2*x[i+1] + c[i+1];
}
```

```
// variant #4
for (i=1; i<N; i++) {
    x[i] = a[i]+b[i+1];
    a[i+1] = 2*x[i-1] + c[i+1];
}
```

```
! variant #1
do i=1,N
    x(i) = a(i)+b(i+1)
    a(i) = 2*x(i) + c(i+1)
end do
```

```
! variant #3
do i=2,N
    x(i) = a(i)+b(i+1)
    a(i) = 2*x(i-1) + c(i+1)
end do
```

```
! variant #2
do i=1,N
    x(i) = a(i)+b(i+1)
    a(i) = 2*x(i+1) + c(i+1)
end do
```

```
! variant #3
do i=2,N
    x(i) = a(i)+b(i+1)
    a(i+1) = 2*x(i-1) + c(i+1)
end do
```

### 26.2.3 Data and synchronization

#### 26.2.3.1

What is the output of the following fragments? Assume that there are four threads.

```
// variant #1
int nt;
#pragma omp parallel
{
    nt = omp_get_thread_num();
    printf("thread number: %d\n", nt);
}
```

```
// variant #2
int nt;
#pragma omp parallel private(nt)
{
    nt = omp_get_thread_num();
    printf("thread number: %d\n", nt);
}
```

```
// variant #3
int nt;
#pragma omp parallel
{
    #pragma omp single
    {
        nt = omp_get_thread_num();
        printf("thread number: %d\n",
               nt);
    }
}
```

```
! variant #1
integer nt
!$OMP parallel
    nt = omp_get_thread_num()
    print *, "thread number:", nt
!$OMP end parallel
```

```
! variant #2
integer nt
!$OMP parallel private(nt)
    nt = omp_get_thread_num()
    print *, "thread number:", nt
!$OMP end parallel
```

```
// variant #4
int nt;
#pragma omp parallel
{
    #pragma omp master
    {
        nt = omp_get_thread_num();
        printf("thread number: %d\n",
               nt);
    }
}
```

```
// variant #5
int nt;
#pragma omp parallel
{
    #pragma omp critical
    {
        nt = omp_get_thread_num();
        printf("thread number: %d\n",
               nt);
    }
}
```

```
! variant #3
integer nt
!$OMP parallel
!$OMP single
    nt = omp_get_thread_num()
    print *, "thread number:", nt
!$OMP end single
!$OMP end parallel
```

```

! variant #4
integer nt
!$OMP parallel
!$OMP master
    nt = omp_get_thread_num()
    print *, "thread number:", nt
!$OMP end master
!$OMP end parallel

```

```

! variant #5
integer nt
!$OMP parallel
!$OMP critical
    nt = omp_get_thread_num()
    print *, "thread number:", nt
!$OMP end critical
!$OMP end parallel

```

### 26.2.3.2

The following is an attempt to parallelize a serial code. Assume that all variables and arrays are defined. What errors and potential problems do you see in this code? How would you fix them?

```

#pragma omp parallel
{
    x = f();
    #pragma omp for
    for (i=0; i<N; i++)
        y[i] = g(x, i);
    z = h(y);
}

```

```

!$OMP parallel
    x = f()
    !$OMP do
        do i=1,N
            y(i) = g(x, i)
        end do
    !$OMP end do
    z = h(y)
    !$OMP end parallel

```

## 26.2.3.3

Assume two threads. What does the following program output?

```
int a;
#pragma omp parallel private(a) {
    ...
    a = 0;
    #pragma omp for
    for (int i = 0; i < 10; i++)
    {
        #pragma omp atomic
        a++; }
    #pragma omp single
    printf("a=%e\n", a);
}
```

## 26.2.4 Reductions

## 26.2.4.1

Is the following code correct? Is it efficient? If not, can you improve it?

```
#pragma omp parallel shared(r)
{
    int x;
    x = f(omp_get_thread_num());
#pragma omp critical
    r += f(x);
}
```

## 26.2.4.2

Compare two fragments:

```
// variant 1
#pragma omp parallel reduction(+:s)
#pragma omp for
for (i=0; i<N; i++)
    s += f(i);
```

```
// variant 2
#pragma omp parallel
#pragma omp for reduction(+:s)
for (i=0; i<N; i++)
    s += f(i);
```

```
! variant 1
!$OMP parallel reduction(+:s)
!$OMP do
do i=1,N
    s += f(i);
```

```
end do
!$OMP end do
!$OMP end parallel
```

```
! variant 2
!$OMP parallel
!$OMP do reduction(+:s)
  do i=1,N
```

```
s += f(i);
end do
!$OMP end do
!$OMP end parallel
```

Do they compute the same thing?

### 26.2.5 Barriers

Are the following two code fragments well defined?

```
#pragma omp parallel
{
#pragma omp for
for (mytid=0; mytid<nthreads;
     mytid++)
    x[mytid] = some_calculation();
#pragma omp for
for (mytid=0; mytid<nthreads-1;
     mytid++)
    y[mytid] = x[mytid]+x[mytid+1];
}
```

```
#pragma omp parallel
{
#pragma omp for
for (mytid=0; mytid<nthreads;
     mytid++)
    x[mytid] = some_calculation();
#pragma omp for nowait
for (mytid=0; mytid<nthreads-1;
     mytid++)
    y[mytid] = x[mytid]+x[mytid+1];
}
```

### 26.2.6 Data scope

The following program is supposed to initialize as many rows of the array as there are threads.

```
int main() {
    int i, icount, iarray[100][100];
    icount = -1;
#pragma omp parallel private(i)
    {
#pragma omp critical
        { icount++; }
        for (i=0; i<100; i++)
            iarray[icount][i] = 1;
    }
    return 0;
}
```

```
Program main
integer :: i, icount, iarray
(100,100)
icount = 0
!$OMP parallel private(i)
!$OMP critical
icount = icount + 1
!$OMP end critical
do i=1,100
    iarray(icount,i) = 1
end do
!$OMP end parallel
End program
```

Describe the behaviour of the program, with argumentation,

- as given;
- if you add a clause `private(icount)` to the `parallel` directive;
- if you add a clause `firstprivate(icount)`.

What do you think of this solution:

```
#pragma omp parallel private(i)
shared(icount)
{
#pragma omp critical
{ icount++;
```

```
for (i=0; i<100; i++)
    iarray[icount][i] = 1;
}
return 0;
```

}

```
!$OMP parallel private(i) shared(
    icount)
!$OMP critical
    icount = icount+1
do i=1,100
    iarray(icount,i) = 1
end do
!$OMP critical
!$OMP end parallel
```

### 26.2.7 Tasks

Fix two things in the following example:

```
#pragma omp parallel
#pragma omp single
{
    int x,y,z;
#pragma omp task
    x = f();
#pragma omp task
    y = g();
#pragma omp task
    z = h();
    printf("sum=%d\n",x+y+z);
}
```

```
integer :: x,y,z
!$OMP parallel
!$OMP single

!$OMP task
    x = f()
!$OMP end task

!$OMP task
    y = g()
!$OMP end task

!$OMP task
    z = h()
!$OMP end task

print *, "sum=",x+y+z
!$OMP end single
!$OMP end parallel
```

### 26.2.8 Scheduling

Compare these two fragments. Do they compute the same result? What can you say about their efficiency?

```
#pragma omp parallel
#pragma omp single
{
    for (i=0; i<N; i++) {
        #pragma omp task
        x[i] = f(i)
    }
    #pragma omp taskwait
}
```

```
#pragma omp parallel
#pragma omp for schedule(dynamic)
{
    for (i=0; i<N; i++) {
        x[i] = f(i)
    }
}
```

How would you make the second loop more efficient? Can you do something similar for the first loop?

**26.3 Sources used in this chapter**

**26.3.1 Listing of code header**

## **PART III**

### **PETSC**

## Chapter 27

### PETSc basics

#### 27.1 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 Partial Differential Equations (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 17** *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://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/singleindex.html>. Each man page comes with links to related routines, as well as (usually) example codes for that routine.*

#### 27.1.1 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.

### 27.1.2 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 27.1.3) 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*.

### 27.1.3 Design philosophy

PETSc has an object-oriented design, even though it is written in C. There are classes of objects, such `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)
```

### 27.1.4 Language support

#### 27.1.4.1 C/C++

PETSc is implemented in C, so there is a natural interface to C. There is no separate C++ interface.

#### 27.1.4.2 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`.

### 27.1.4.3 Python

A `python` interface was written by Lisandro Dalcin, and requires separate installation, based on already defined `PETSC_DIR` and `PETSC_ARCH` variables. This can be downloaded at <https://bitbucket.org/petsc/petsc4py/src/master/>, with documentation at <https://www.mcs.anl.gov/petsc/petsc4py-current/docs/>.

### 27.1.5 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://www.mcs.anl.gov/petsc/documentation/index.html>.

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.

## 27.2 Basics of running a PETSc program

### 27.2.1 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:

```
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. If you want to write your own compiler rule, use

```
include ${PETSC_DIR}/lib/petsc/conf/variables
%.o : %.c
    ${CC} -c $^ ${PETSC_CC_INCLUDES}
program : program.o
    ${CLINKER} -o $@ $^ ${PETSC_LIB}
```

(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
```

and copying the results into your compilation script.

The build process assumes that variables `PETSC_DIR` and `PETSC_ARCH` have been set. These depend on your local installation. Usually there will be one installation with debug settings and one with production settings. Develop your code with the former: it will do memory and bound checking. Then recompile and run your code with the optimized production installation.

*TACC note.* On TACC clusters, a petsc installation is loaded by commands such as

```
module load petsc/3.11
```

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.

### 27.2.2 Running

PETSc programs use MPI for parallelism, so they are started like any other MPI program:

### 27.1 **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.
```

mpirun -np 5 -machinefile mf \
 your\_petsc\_program option1 option2 option3

TACC note. On TACC clusters, use ibrun.

#### 27.2.3 Initialization and finalization

PETSc has an call that initializes both PETSc and MPI, so normally you would replace **MPI\_Init** by **PetscInitialize** (figure 27.1). 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 31.3.

```
// init.c
ierr = PetscInitialize(&argc, &argv, (char*)0, help); CHKERRQ(ierr);
int flag;
MPI_Initialized(&flag);
if (flag)
  printf("MPI was initialized by PETSc\n");
else
  printf("MPI not yet initialized\n");
```

For the source of this example, see section 27.4.2

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.

- The Fortran version has no arguments for commandline options; it also doesn't take a help string.
- If no help string is passed, give `PETSC_NULL_CHARACTER` as argument.

- 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"
```

For the source of this example, see section ??

*Python note.* 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,&ntrids);
```

*Python note.*

```
comm = PETSc.COMM_WORLD
nprocs = comm.getSize(self)
procno = comm.getRank(self)
```

The corresponding call to replace `MPI_Finalize` is `PetscFinalize`.

## 27.3 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
```

### 27.3.1 Debug

For any set of options, you will typically make two installations: one with `-with-debugging=yes` and once `no`. See section 31.1.1 for more detail.

### 27.3.2 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.

### 27.3.3 Variants

- Scalars: the option `-with-scalar-type` has values `real`, `complex`; `-with-precision` has values `single`, `double`, `__float128`, `__fp16`.

### 27.3.4 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
```

**Remark 18** 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.

## 27.4 Sources used in this chapter

### 27.4.1 Listing of code header

#### 27.4.2 Listing of code examples/petsc/c/init.c

```
#include <stdlib.h>
#include <stdio.h>

#include <petscsys.h>

int main(int argc,char **argv)
{
    PetscErrorCode ierr;

    char help[] = "\nInit example.\n\n";
    ierr = PetscInitialize(&argc,&argv,(char*)0,help); CHKERRQ(ierr);
    int flag;
    MPI_Initialized(&flag);
    if (flag)
        printf("MPI was initialized by PETSc\n");
    else
        printf("MPI not yet initialized\n");
    ierr = PetscFinalize(); CHKERRQ(ierr);
    return 0;
}
```

## Chapter 28

### PETSc objects

#### 28.1 Distributed objects

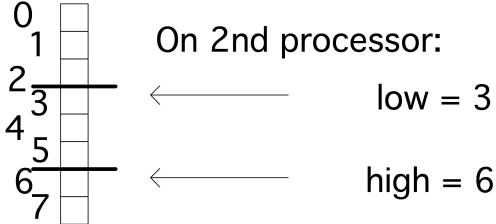
PETSc is based on the SPMD model, and all its objects act like they exist in parallel, spread out over all the processes. Therefore, prior to discussing specific objects in detail, we briefly discuss how PETSc treats distributed objects.

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  $0 \dots 99$ , process 1 gets  $1 \dots 199$ , et cetera. This simple scheme suffices for many cases, but PETSc has facilities for more sophisticated load balancing.

##### 28.1.1 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.

## 28.1 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);
```

For the source of this example, see section 28.9.2

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 28.1) 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.

## 28.2 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`.

### 28.2.1 Integers

Integers in PETSc are likewise of a size determined at installation time: `PetscInt` can be 32 or 64 bits. Furthermore, there is a `PetscErrorCode` type for catching the return code of PETSc routines.

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.

Many other packages do not support 64-bit integers.

### 28.2.2 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`.

### 28.2.3 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`.

## 28.3 Vec: Vectors

Vectors are objects with a linear index. The elements of a vector are floating point numbers or complex numbers (see section 28.2), but not integers: for that see section 28.6.1.

## 28.2 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()
```

## 28.3 VecDestroy

```
Synopsis
#include "petscvec.h"
PetscErrorCode VecDestroy(Vec *v)

Collective on Vec

Input Parameters:
v -the vector
```

### 28.3.1 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 28.2)

*Python note.* 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`.

The corresponding routine **VecDestroy** (figure 28.3) deallocates data and zeros the pointer.

The vector type needs to be set with **VecSetType** (figure 28.4).

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.

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);
```

**28.4 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

**28.5 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

for multiple vectors. For the latter, there is a joint destroy call **VecDestroyVecs**:

|| **VecDestroyVecs** (PetscInt n, Vec \*\*vecs);

(which is different in Fortran).

**28.3.2 Vector layout**

Next in the creation process the vector size is set with **VecSetSizes** (figure 28.5). 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** (**PETSc.DECIDE** in python).

The size is queried with **VecGetSize** (figure 28.6) for the global size and **VecGetLocalSize** (figure 28.6) for the local size.

Each processor gets a contiguous part of the vector. Use **VecGetOwnershipRange** (figure 28.7) 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

## 28.6 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)
```

## 28.7 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.
```

## 28.8 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)
```

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  
|||   (MPI_Comm comm,PetscInt bs,  
|||     PetscInt n,PetscInt N,PetscScalar *array,Vec *vv);
```

As you will see in section 28.4.1, you can also create vectors based on the layout of a matrix, using `MatCreateVecs`.

### 28.3.3 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), 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 28.8) routine can be used to display vectors on screen as ascii output,

```
||| // fftsine.c  
||| ierr = VecView(signal,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);  
||| ierr = MatMult(transform,signal,frequencies); CHKERRQ(ierr);  
||| ierr = VecView(frequencies,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
```

*For the source of this example, see section 28.9.3*

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;  
||| ierr = VecNorm(y,NORM_2,&lambda); CHKERRQ(ierr);  
||| ierr = VecScale(y,1./lambda); CHKERRQ(ierr);
```

## 28.9 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

## 28.10 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

## 28.11 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

### 28.12 **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`

**Exercise 28.1.** Use the routines **VecDot** (figure 28.9) , **VecScale** (figure 28.10) and **VecNorm** (figure 28.11) to compute the inner product of vectors  $x, y$ , scale the vector  $x$ , and check its norm:

$$\begin{aligned} p &\leftarrow x^t y \\ x &\leftarrow x/p \\ n &\leftarrow \|x\|_2 \end{aligned}$$

#### 28.3.3.1 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.

#### 28.3.4 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 28.12) for setting elements that uses global numbering; any process can set any elements in the vector. There is also a routine **VecSetValues** (figure 28.13) 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.

### 28.13 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
```

### 28.14 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);

call VecSetValue(x,i,v,INSERT_VALUES,ierr)
ii(1) = 1; ii(2) = 2; vv(1) = 2.7; vv(2) = 3.1
call VecSetValues(x,2,ii,vv,INSERT_VALUES,ierr)
```

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 28.14) and **VecAssemblyEnd**.

```
if (myrank==0) then
do vecidx=0,globalsize-1
  vecelt = vecidx
  call VecSetValue(vector,vecidx,vecelt,INSERT_VALUES,ierr)
end do
end if
call VecAssemblyBegin(vector,ierr)
call VecAssemblyEnd(vector,ierr)
```

For the source of this example, see section 28.9.4

**28.15 VecGetArray**

```
C:  
#include "petscvec.h"  
PetscErrorCode VecGetArray(Vec x,PetscScalar **a)  
PetscErrorCode VecGetArrayRead(Vec x,const PetscScalar **a)  
  
Input Parameter  
x : the vector  
  
Output Parameter  
a : location to put pointer to the array  
  
PetscErrorCode VecRestoreArray(Vec x,PetscScalar **a)  
PetscErrorCode VecRestoreArrayRead(Vec x,const PetscScalar **a)  
  
Input Parameters  
x : the vector  
a : location of pointer to array obtained from VecGetArray()  
  
Fortran90:  
#include <petsc/finclude/petscvec.h>  
use petscvec  
VecGetArrayF90(Vec x,{Scalar, pointer :: xx_v(:)},integer ierr)  
(there is a Fortran77 version)  
VecRestoreArrayF90(Vec x,{Scalar, pointer :: xx_v(:)},integer ierr)  
  
Python:  
PETSc.Vec.getArray(self, readonly=False)  
?? PETSc.Vec.resetArray(self, force=False)
```

(If you know the MPI library, you'll recognize that the first call corresponds to posting non-blocking 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.

#### 28.3.4.1 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 28.15) for general access or `VecGetArrayRead` (figure 28.15) for read-only.

PETSc insists that you properly release this pointer again with `VecRestoreArray` (figure 28.16) or `VecRestoreArrayRead` (figure 28.16).

**28.16 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
```

**28.17 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
```

Note that in a distributed running context you can only get the array of local elements. Accessing the elements from another process requires explicit communication; see section 28.6.2.

```
PetscScalar *in_array,*out_array;
VecGetArrayRead(in,&in_array);
VecGetArray(out,&out_array);
VecGetLocalSize(in,&localsize);
for (int i=0; i<localsize; i++)
    out_array[i] = 2*in_array[i];
VecRestoreArrayRead(in,&in_array);
VecRestoreArray(out,&out_array);
```

There are some variants to the **VecGetArray** operation:

- **VecReplaceArray** (figure 28.17) 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 28.17) 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,

### 28.18 MatCreate

```
C:  
PetscErrorCode MatCreate(MPI_Comm comm,Mat *v);  
  
Python:  
mat = PETSc.Mat()  
mat.create()  
# or:  
mat = PETSc.Mat().create()
```

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. 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)
```

For the source of this example, see section [28.9.4](#)

## 28.4 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.

### 28.4.1 Matrix creation

Creating a matrix also starts by specifying a communicator on which the matrix lives collectively: **MatCreate** (figure 28.18)

Set the matrix type with **MatSetType** (figure 28.19). 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.

**28.19 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,
```

**28.20 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
```

**28.21 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
```

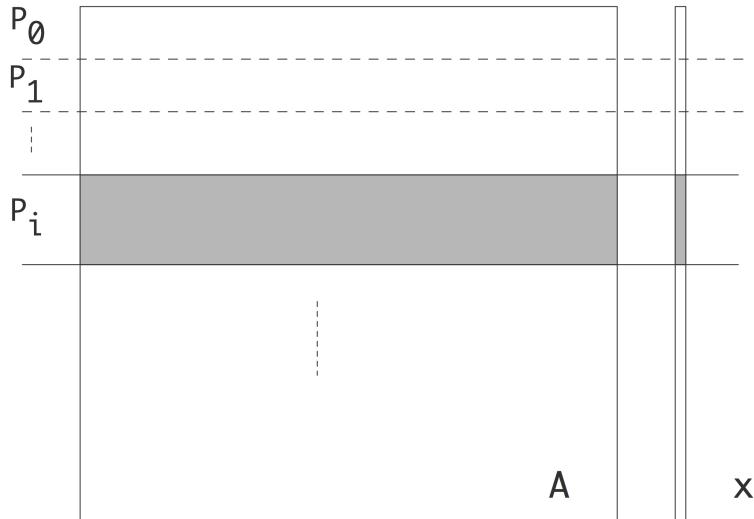


Figure 28.1: Matrix partitioning by block rows

## 28.22 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
```

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 28.20) and subsequently query them with [MatSizes](#) (figure 28.21). 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 28.22) can be used. (Sometimes this is even required; see section 28.4.8.)

### 28.4.2 Nonzero structure

In case of a dense matrix, once you have specified the size and the number of MPI ranks, it is simple to determine how much space PETSc needs to allocate for the matrix. For a sparse matrix this is more

**28.23 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)
```

complicated, since the 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 28.23). In the case of a tridiagonal matrix you would specify that each row has three elements:

```
|| MatSeqAIJSetPreallocation(A, 3, NULL);
```

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,NULL,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 28.2, 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 28.23), 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:

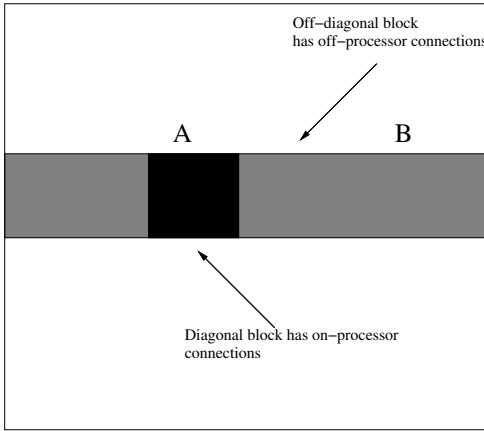


Figure 28.2: The diagonal and off-diagonal parts of a matrix

### 28.24 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  
  
|| MatMPIAIJSetPreallocation(A, 3, NULL, 2, NULL);
```

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 NULL arguments by an array that lists for each row the number of nonzeros in that row.

#### 28.4.3 Matrix elements

You can set a single matrix element with **MatSetValue** (figure 28.24) or a block of them, where you supply a set of  $i$  and  $j$  indices, using **MatSetValues**.

**28.25 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 28.25) and **MatAssemblyEnd** (figure 28.25) 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)
```

**28.4.3.1 Element access**

If you absolutely need access to the matrix elements, there are routines such as **MatGetRow** (figure 28.26). With this, any rank 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 28.4.5.)

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 `NULL`.

PETSc insists that you properly release the row again with **MatRestoreRow** (figure 28.26).

It is also possible to retrieve the full Compressed Row Storage (CRS) contents of the local matrix with

**28.26 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
```

**28.27 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
```

|| [MatGetArray](#)  
|| [MatRestoreArray](#)

**28.4.4 Matrix operations****28.4.4.1 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 28.27) and the transpose product [MatMultTranspose](#) (figure 28.27). (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 28.28) computes  $z \leftarrow Ax + y$ .

## 28.28 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.

### 28.4.4.2 Matrix-matrix operations

There is a number of matrix-matrix routines such as `MatMatMult`.

## 28.4.5 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 rank.

## 28.4.6 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 30.1) only need the matrix-times-vector (and perhaps the matrix-transpose-times-vector) product.

PETSc supports this mode of working. The routine `MatCreateShell` (figure 28.29) declares the argument to be a matrix given in operator form.

### 28.4.6.1 Shell operations

The next step is then to add the custom multiplication routine, which will be invoked by `MatMult: MatShellSetOperation` (figure 28.30)

The routine that implements the actual product should have the same prototype as `MatMult`, accepting a matrix and two vectors. The key to realizing your own product routine lies in the 'context' argument to

**28.29 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
```

**28.30 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.
```

**28.31 MatShellSetContext**

Synopsis

```
#include "petscmat.h"
PetscErrorCode MatShellSetContext(Mat mat,void *ctx)

Input Parameters
mat - the shell matrix
ctx - the context
```

**28.32 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
```

the create routine. With **MatShellSetContext** (figure 28.31) you pass a pointer to some structure that contains all contextual information you need. In your multiplication routine you then retrieve this with **MatShellGetContext** (figure 28.32).

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**.)

#### 28.4.6.2 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 prototype 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.

#### 28.4.7 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.  $3 \times 3$  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 `MATMPIBJ` format, and use so-called *field-split preconditioners*; see section ??.

#### 28.4.8 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 27.3.4.

FFT libraries may use padding, so vectors should be created with `MatCreateVecsFFTW`, not with an independent `VecSetSizes`.

### 28.5 DMDA: distributed arrays

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` (figure 28.33) 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.

Once you have a grid, can create its associated matrix:

```
|| DMSetUp (grid);
|| DMCreateMatrix (grid, &A)
```

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`, which returns an `DMDALocalInfo` object.

### 28.33 **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
```

#### 28.5.1 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 communicatin and computation use **DMLocalToGlobalBegin** and **DMLocalToGlobalEnd** .)

#### 28.5.2 Matrices of a distributed array

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;
||| ierr = DMDAGetCorners(grid, &i_first, NULL, NULL, &i_local, NULL, NULL); CHKERRQ(ierr);
||| for (PetscInt i_index=i_first; i_index<i_first+i_local; i_index++) {
```

*For the source of this example, see section 28.9.5*

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++; }
||| ierr = MatSetValuesStencil(A, 1, &row, ncols, col, v, INSERT_VALUES); CHKERRQ(ierr);
```

*For the source of this example, see section 28.9.5*

## 28.6 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.

### 28.34 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

#### 28.6.1 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) {
    ierr = ISCreateStride(comm,Nglobal/2,0,2,&oddeven); CHKERRQ(ierr);
} else {
    ierr = ISCreateStride(comm,Nglobal/2,1,2,&oddeven); CHKERRQ(ierr);
}
```

*For the source of this example, see section 28.9.6*

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**.

#### 28.6.2 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 28.34) 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, `NULL` (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, `NULL` 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;
||| ierr = VecScatterCreate
|||   (in, oddeven, out, NULL, &separate); CHKERRQ(ierr);
||| ierr = VecScatterBegin
|||   (separate, in, out, INSERT_VALUES, SCATTER_FORWARD); CHKERRQ(ierr);
||| ierr = VecScatterEnd
|||   (separate, in, out, INSERT_VALUES, SCATTER_FORWARD); CHKERRQ(ierr);
```

For the source of this example, see section 28.9.6

Note that the index set is applied to the input vector, since it describes the components to be moved. The output vector uses `NULL` since these components are placed in sequence.

**Exercise 28.2.** Modify this example so that the components are still separated odd/even, but now placed in descending order on each process.

**Exercise 28.3.** 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  $n_{procs}$ ?

#### 28.6.2.1 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.

### 28.7 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.

### 28.8 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 native to 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, NULL, &isrows);
|| MatCreateSubMatrix(A, isrows, isrows, MAT_INITIAL_MATRIX, &perA);
```

Other scenario:

```
|| MatPartitioningSetAdjacency(part, A);
|| MatPartitioningSetType(part, MATPARTITIONINGHIERARCH);
|| MatPartitioningHierarchicalSetNcoarseparts(part, 2);
|| MatPartitioningHierarchicalSetNfineparts(part, 2);
```

**28.9 Sources used in this chapter****28.9.1 Listing of code header****28.9.2 Listing of code examples/petsc/c/split.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

static char help[] = "\nOwnership example.\n\n";

#include <petscvec.h>

int main(int argc,char **argv)
{
    Vec           x,y;                  /* vectors */
    int nprocs,procid;
    PetscErrorCode ierr;

    PetscInitialize(&argc,&argv,(char*)0,help);
    MPI_Comm comm = PETSC_COMM_WORLD;
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_rank(comm,&procid);

    PetscInt N,n;
    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);

    ierr = PetscFinalize();
    return 0;
}
```

**28.9.3 Listing of code examples/petsc/c/fftsine.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

static char help[] = "\nFFT example.\n\n";

#include <petscmat.h>

int main(int argc,char **argv)
{
```

```

Vec           x,y;                  /* vectors */
int nprocs,procid;
PetscErrorCode ierr;

PetscInitialize(&argc,&argv,(char*)0,help);
MPI_Comm comm = PETSC_COMM_WORLD;
MPI_Comm_size(comm,&nprocs);
MPI_Comm_rank(comm,&procid);

PetscInt Nlocal = 10, Nglobal = Nlocal*nprocs;
PetscPrintf(comm,"FFT examples on N=%d n=%d\n",Nglobal,Nlocal);

Mat transform;
int dimensionality=1;
PetscInt dimensions[dimensionality]; dimensions[0] = Nglobal;
PetscPrintf(comm,"Creating fft D=%d, dim=%d\n",dimensionality,dimensions[0]);
ierr = MatCreateFFT(comm,dimensionality,dimensions,MATFFTW,&transform); CHKERRQ(ierr);
{
    PetscInt fft_i,fft_j;
    ierr = MatGetSize(transform,&fft_i,&fft_j); CHKERRQ(ierr);
    PetscPrintf(comm,"FFT global size %d x %d\n",fft_i,fft_j);
}
Vec signal,frequencies;
ierr = MatCreateVecsFFTW(transform,&frequencies,&signal,PETSC_NULL); CHKERRQ(ierr);
ierr = PetscObjectSetName((PetscObject)signal,"signal"); CHKERRQ(ierr);
ierr = PetscObjectSetName((PetscObject)frequencies,"frequencies"); CHKERRQ(ierr);
ierr = VecAssemblyBegin(signal); CHKERRQ(ierr);
ierr = VecAssemblyEnd(signal); CHKERRQ(ierr);
{
    PetscInt nlocal,nglobal;
    ierr = VecGetLocalSize(signal,&nlocal); CHKERRQ(ierr);
    ierr = VecGetSize(signal,&nglobal); CHKERRQ(ierr);
    ierr = PetscPrintf(comm,"Signal local=%d global=%d\n",nlocal,nglobal); CHKERRQ(ierr);
}

PetscInt myfirst,mylast;
ierr = VecGetOwnershipRange(signal,&myfirst,&mylast); CHKERRQ(ierr);
printf("Setting %d -- %d\n",myfirst,mylast);
// for (PetscInt vecindex=myfirst; vecindex<mylast; vecindex++) {
for (PetscInt vecindex=0; vecindex<Nglobal; vecindex++) {
    PetscScalar
        pi = 4. * atan(1.0),
        h = 1./Nglobal,
        phi = 2* pi * (vecindex+1) * h,
        puresine = cos( phi )
#if defined(PETSC_USE_COMPLEX)
        + PETSC_I * sin(phi)
#endif
;
    ierr = VecSetValue(signal,vecindex,puresine,INSERT_VALUES); CHKERRQ(ierr);
}
ierr = VecAssemblyBegin(signal); CHKERRQ(ierr);
ierr = VecAssemblyEnd(signal); CHKERRQ(ierr);

```

```
ierr = VecView(signal,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
ierr = MatMult(transform,signal,frequencies); CHKERRQ(ierr);
ierr = VecView(frequencies,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);

ierr = MatDestroy(&transform); CHKERRQ(ierr);
ierr = VecDestroy(&signal); CHKERRQ(ierr);
ierr = VecDestroy(&frequencies); CHKERRQ(ierr);

ierr = PetscFinalize(); CHKERRQ(ierr);
return 0;
}
```

#### 28.9.4 Listing of code examples/petsc/f/vecset.F90

```
Program VecSetF90

#include <petsc/f/include/petsc.h>
use petsc
implicit none

Vec :: vector
PetscScalar,dimension(:),pointer :: elements
PetscErrorCode :: ierr
PetscInt :: globalsize
integer :: myrank,vecidx,comm
PetscScalar :: vecelt
character*80 :: msg

call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
CHKERRA(ierr)
comm = MPI_COMM_WORLD
call MPI_Comm_rank(comm,myrank,ierr)

call VecCreate(comm,vector,ierr)
call VecSetType(vector,VECMPI,ierr)
call VecSetSizes(vector,2,PETSC_DECIDE,ierr)
call VecGetSize(vector,globalsize,ierr)

if (myrank==0) then
  do vecidx=0,globalsize-1
    vecelt = vecidx
    call VecSetValue(vector,vecidx,vecelt,INSERT_VALUES,ierr)
  end do
end if
call VecAssemblyBegin(vector,ierr)
call VecAssemblyEnd(vector,ierr)
call VecView(vector,PETSC_VIEWER_STDOUT_WORLD,ierr)

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)

call VecDestroy(vector,ierr)
call PetscFinalize(ierr); CHKERRQ(ierr);

End Program VecSetF90
```

### 28.9.5 Listing of code examples/petsc/c/grid1d.c

```
#include <stdlib.h>
#include <stdio.h>
#include "petsc.h"
#include "petscdmda.h"

int main(int argc,char **argv) {

    PetscErrorCode ierr;
    ierr = PetscInitialize(&argc,&argv,0,0); CHKERRQ(ierr);

    MPI_Comm comm = MPI_COMM_WORLD;
    int nprocs,procno;
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_rank(comm,&procno);

    PetscInt i_global = 10*nprocs;

    /*
     * Create a 2d grid and a matrix on that grid.
     */
    DM grid;
    ierr = DMDACreate1d
        ( comm,                      // IN:
          DM_BOUNDARY_NONE,          // collective on this communicator
          i_global,                  // no periodicity and such
          1,                         // global size 100x100; can be changed with options
          1,                         // degree of freedom per node
          1,                         // stencil width
          NULL,                      // arrays of local sizes in each direction
          &grid                       // OUT: resulting object
        ); CHKERRQ(ierr);
    ierr = DMSetUp(grid); CHKERRQ(ierr);

    Mat A;
    ierr = DMCreateMatrix(grid,&A); CHKERRQ(ierr);

    /*
     * Print out how the grid is distributed over processors
     */
    PetscInt i_first,i_local;
    ierr = DMDAGetCorners(grid,&i_first,NULL,NULL,&i_local,NULL,NULL);CHKERRQ(ierr);
    /* ierr = PetscSynchronizedPrintf */
```

```
/*      (comm, */
/*      "[%d] Local part = %d-%d x %d-%d\n", */
/*      procno,info.xs,info.xs+info.xm,info.ys,info.ys+info.ym); CHKERRQ(ierr); */
/* ierr = PetscSynchronizedFlush(comm,stdout); CHKERRQ(ierr); */

/*
 * Fill in the elements of the matrix
*/
for (PetscInt i_index=i_first; i_index<i_first+i_local; i_index++) {
    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++; }
    ierr = MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES);CHKERRQ(ierr);
}

ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);

/*
 * Create vectors on the grid
*/
Vec x,y;
ierr = DMCreateGlobalVector(grid,&x); CHKERRQ(ierr);
ierr = VecDuplicate(x,&y); CHKERRQ(ierr);

/*
 * Set vector values: first locally, then global
*/
PetscReal one = 1.;
{
    Vec xlocal;
    ierr = DMCreateLocalVector(grid,&xlocal); CHKERRQ(ierr);
    ierr = VecSet(xlocal,one); CHKERRQ(ierr);
    ierr = DMLocalToGlobalBegin(grid,xlocal,INSERT_VALUES,x); CHKERRQ(ierr);
    ierr = DMLocalToGlobalEnd(grid,xlocal,INSERT_VALUES,x); CHKERRQ(ierr);
    ierr = VecDestroy(&xlocal); CHKERRQ(ierr);
}

/*
 * Solve a linear system on the grid
*/
KSP solver;
ierr = KSPCreate(comm,&solver); CHKERRQ(ierr);
ierr = KSPSetType(solver,KSPBCGS); CHKERRQ(ierr);
ierr = KSPSetOperators(solver,A,A); CHKERRQ(ierr);
ierr = KSPSetFromOptions(solver); CHKERRQ(ierr);
ierr = KSPSolve(solver,x,y); CHKERRQ(ierr);
```

```
/*
 * Report on success of the solver, or lack thereof
 */
{
    PetscInt its; KSPConvergedReason reason;
    ierr = KSPGetConvergedReason(solver,&reason);
    ierr = KSPGetIterationNumber(solver,&its); CHKERRQ(ierr);
    if (reason<0) {
        PetscPrintf(comm,"Failure to converge after %d iterations; reason %s\n",
                    its,KSPConvergedReasons[reason]);
    } else {
        PetscPrintf(comm,"Number of iterations to convergence: %d\n",its);
    }
}

/*
 * Clean up
 */
ierr = KSPDestroy(&solver); CHKERRQ(ierr);
ierr = VecDestroy(&x); CHKERRQ(ierr);
ierr = VecDestroy(&y); CHKERRQ(ierr);
ierr = MatDestroy(&A); CHKERRQ(ierr);
ierr = DMDestroy(&grid); CHKERRQ(ierr);

PetscFinalize();
return 0;
}
```

### 28.9.6 Listing of code examples/petsc/c/oddeven.c

```
#include "petscksp.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
    PetscErrorCode ierr;
    MPI_Comm comm;

    PetscFunctionBegin;
    PetscInitialize(&argc,&args,0,0);

    comm = MPI_COMM_WORLD;

    int nprocs,procid;
    MPI_Comm_rank(comm,&procid);
    MPI_Comm_size(comm,&nprocs);
    if (nprocs!=2) {
        PetscPrintf(comm,"This example only works on 2 processes, not %d\n",nprocs);
        PetscFunctionReturn(-1); }

    PetscInt Nglobal = 2*nprocs;
```

```
{  
    PetscInt x=1;  
    ierr = PetscOptionsGetInt(PETSC_NULL,PETSC_NULL,"-x",&x,NULL); CHKERRQ(ierr);  
    Nglobal *= x;  
}  
  
Vec in,out;  
ierr = VecCreate(comm,&in); CHKERRQ(ierr);  
ierr = VecSetType(in,VECMPI); CHKERRQ(ierr);  
ierr = VecSetSizes(in,PETSC_DECIDE,Nglobal); CHKERRQ(ierr);  
ierr = VecDuplicate(in,&out); CHKERRQ(ierr);  
  
{  
    PetscInt myfirst,mylast;  
    ierr = VecGetOwnershipRange(in,&myfirst,&mylast); CHKERRQ(ierr);  
    for (PetscInt index=myfirst; index<mylast; index++) {  
        PetscScalar v = index;  
        ierr = VecSetValue(in,index,v,INSERT_VALUES); CHKERRQ(ierr);  
    }  
    ierr = VecAssemblyBegin(in); CHKERRQ(ierr);  
    ierr = VecAssemblyEnd(in); CHKERRQ(ierr);  
}  
  
IS oddeven;  
if (procid==0) {  
    ierr = ISCreateStride(comm,Nglobal/2,0,2,&oddeven); CHKERRQ(ierr);  
} else {  
    ierr = ISCreateStride(comm,Nglobal/2,1,2,&oddeven); CHKERRQ(ierr);  
}  
ISView(oddeven,0);  
  
VecScatter separate;  
ierr = VecScatterCreate  
    (in,oddeven,out,NULL,&separate); CHKERRQ(ierr);  
ierr = VecScatterBegin  
    (separate,in,out,INSERT_VALUES,SCATTER_FORWARD); CHKERRQ(ierr);  
ierr = VecScatterEnd  
    (separate,in,out,INSERT_VALUES,SCATTER_FORWARD); CHKERRQ(ierr);  
  
ierr = ISDestroy(&oddeven); CHKERRQ(ierr);  
ierr = VecScatterDestroy(&separate); CHKERRQ(ierr);  
  
ierr = VecView(in,0); CHKERRQ(ierr);  
ierr = VecView(out,0); CHKERRQ(ierr);  
  
ierr = VecDestroy(&in); CHKERRQ(ierr);  
ierr = VecDestroy(&out); CHKERRQ(ierr);  
  
PetscFunctionReturn(0);  
}
```

## Chapter 29

### Finite Elements support

```
PetscDSSetJacobian
Set the pointwise Jacobian function for given test and basis fields

Synopsis

#include "petscds.h"
PetscErrorCode PetscDSSetJacobian(PetscDS prob, PetscInt f, PetscInt g,
    void (*g0)(PetscInt dim, PetscInt Nf, PetscInt NfAux,
        const PetscInt uOff[], const PetscInt uOff_x[], const PetscScalar u
    [], const PetscScalar u_t[], const PetscScalar u_x[],
        const PetscInt aOff[], const PetscInt aOff_x[], const PetscScalar a
    [], const PetscScalar a_t[], const PetscScalar a_x[],
        PetscReal t, PetscReal u_tShift, const PetscReal x[], PetscInt
    numConstants, const PetscScalar constants[], PetscScalar g0[]),
    void (*g1)(PetscInt dim, PetscInt Nf, PetscInt NfAux,
        const PetscInt uOff[], const PetscInt uOff_x[], const PetscScalar u
    [], const PetscScalar u_t[], const PetscScalar u_x[],
        const PetscInt aOff[], const PetscInt aOff_x[], const PetscScalar a
    [], const PetscScalar a_t[], const PetscScalar a_x[],
        PetscReal t, PetscReal u_tShift, const PetscReal x[], PetscInt
    numConstants, const PetscScalar constants[], PetscScalar g1[]),
    void (*g2)(PetscInt dim, PetscInt Nf, PetscInt NfAux,
        const PetscInt uOff[], const PetscInt uOff_x[], const PetscScalar u
    [], const PetscScalar u_t[], const PetscScalar u_x[],
        const PetscInt aOff[], const PetscInt aOff_x[], const PetscScalar a
    [], const PetscScalar a_t[], const PetscScalar a_x[],
        PetscReal t, PetscReal u_tShift, const PetscReal x[], PetscInt
    numConstants, const PetscScalar constants[], PetscScalar g2[]),
    void (*g3)(PetscInt dim, PetscInt Nf, PetscInt NfAux,
        const PetscInt uOff[], const PetscInt uOff_x[], const PetscScalar u
    [], const PetscScalar u_t[], const PetscScalar u_x[],
        const PetscInt aOff[], const PetscInt aOff_x[], const PetscScalar a
    [], const PetscScalar a_t[], const PetscScalar a_x[],
        PetscReal t, PetscReal u_tShift, const PetscReal x[], PetscInt
    numConstants, const PetscScalar constants[], PetscScalar g3[])
)
```

$$\int_{\Omega} \phi g_0(u, u_t, \nabla u, x, t) \psi + \phi \vec{g}_1(u, u_t, \nabla u, x, t) \nabla \psi + \nabla \phi \cdot \vec{g}_2(u, u_t, \nabla u, x, t) \psi + \nabla \phi \cdot \overleftarrow{g}_3(u, u_t, \nabla u, x, t) \cdot \nabla \psi$$

**29.1 Sources used in this chapter**

**29.1.1 Listing of code header**

# Chapter 30

## 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.

### 30.1 KSP: linear system solvers

#### 30.1.1 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*:

### 30.1 **KSPCreate**

```
C:
PetscErrorCode KSPCreate(MPI_Comm comm, KSP *v);

Python:
ksp = PETSc.KSP()
ksp.create()
# or:
ksp = PETSc.KSP().create()
```

while not converged

$$\begin{aligned} r_i &= Ax_i - b \\ x_{i+1} &\leftarrow f(r_i) \end{aligned}$$

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.

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

$$\begin{aligned} r_i &= Ax_i - b \\ z_i &= M^{-1}r_i \\ x_{i+1} &\leftarrow f(z_i) \end{aligned}$$

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 30.2.)

#### 30.1.2 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 30.1).

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);
```

### 30.2 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
```

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. The call **KSPSetFromOptions** can cover almost all of the settings discussed next.

#### 30.1.3 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 30.2), 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.

#### 30.1.4 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 30.3) to get a **KSPConvergedReason** parameter that indicates what state the solver stopped in:

- The iteration can have successfully converged; this corresponds to `reason > 0`;

### 30.3 **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
```

- 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, **KSPConvergenceReasonView** (before version 3.14: **KSPReasonView**) can print out the reason in readable form; for instance

```
// KSPConvergenceReasonView(solver,PETSC_VIEWER_STDOUT_WORLD);
// before 3.14:
KSPReasonView(solver,PETSC_VIEWER_STDOUT_WORLD);
```

(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); CHKERRQ(ierr);
ierr = KSPReasonView(Solve,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
if (reason<0) {
  PetscPrintf(comm,"Failure to converge: reason=%d\n",reason);
} else {
  ierr = KSPGetIterationNumber(Solve,&its); CHKERRQ(ierr);
  PetscPrintf(comm,"Number of iterations: %d\n",its);
}
```

For the source of this example, see section 30.4.2

#### 30.1.5 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 30.4), or use the option `-ksp_type`.

### 30.4 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
```

### 30.5 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
```

Here are some values (the full list is in `petscksp.h`):

- **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.

#### 30.1.6 Multiple right-hand sides

For the case of multiple right-hand sides, use **KSPMatSolve** (figure 30.5) .

#### 30.1.7 Preconditioners

Another part of an iterative solver is the *preconditioner*. The mathematical background of this is given in section 30.1.1. 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.

#### 30.1.7.1 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.

#### 30.1.7.2 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.

## 30.1.7.3 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
```

**30.1.7.3.1 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 30.1.7.3.3.
- As part of the `PCHYPRE` package, the parallel variant *parasails* is available.

```
-pc_type hypre -pc_hypre_type parasails
```

**30.1.7.3.2 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 natively 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 30.1.7.3.3.
- 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.

**30.1.7.3.3 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 30.1 illustrates these preconditioners both in matrix and subdomain terms.

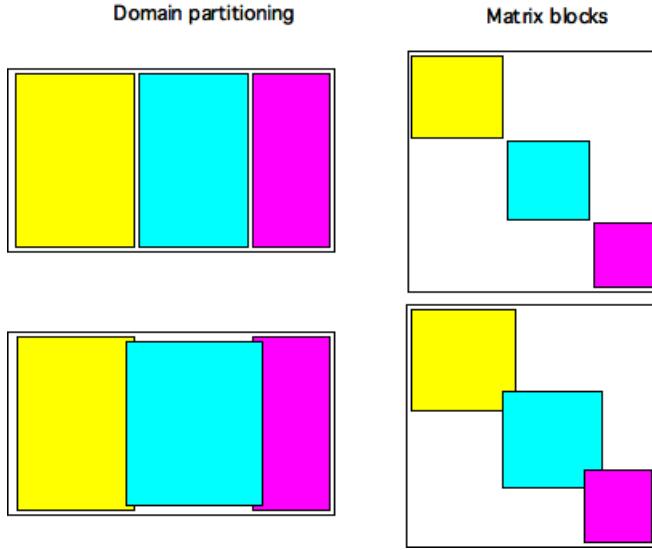


Figure 30.1: Illustration of block Jacobi and Additive Schwarz preconditioners

#### 30.1.7.3.4 Multigrid preconditioners

- There is a native Algebraic MultiGrid (AMG) type: `PCGAMG`;
- the external packages *Hypre* and *ML* have AMG methods.
- There is a general **MG!** (**MG!**) type: `PCMGMG`.

#### 30.1.7.3.5 Field split preconditioners

For background refer to section 28.4.7.

### 30.1.8 Customization: monitoring and convergence tests

PETSc solvers can do various callbacks to user functions.

**30.1.8.0.1 Shell preconditioners** You already saw that, in an iterative methods, the coefficient matrix can be given operationally as a *shell matrix*; section 28.4.6. 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));
```

**30.1.8.0.2 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);
```

### 30.1.8.1 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.

### 30.1.8.2 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.

## 30.6 KSPSetFromOptions

Synopsis

```
#include "petscksp.h"
PetscErrorCode  KSPSetFromOptions(KSP ksp)
```

Collective on ksp

Input Parameters  
ksp - the Krylov space context

### 30.1.8.3 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);
```

## 30.2 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, <solvertype> )
```

where `solvertype` can be mumps, superlu, umfpack, or a number of others. Note that availability of these packages depends on how PETSc was installed on your system.

## 30.3 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 30.6) 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
```

**30.4 Sources used in this chapter**

**30.4.1 Listing of code header**

**30.4.2 Listing of code code/petsc/c**

# Chapter 31

## PETSc tools

### 31.1 Error checking and debugging

#### 31.1.1 Debug mode

During installation (see section 27.3), 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 31.1.3);
- Has the macro `PETSC_USE_DEBUG` set to 1.

#### 31.1.2 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.

1. Every PETSc routine is a function returning a parameter of type `PetscErrorCode`.
2. For a good traceback, surround the executable part of any subprogram with `PetscFunctionBegin` and `PetscFunctionReturn`, where the latter has the return value as parameter.
3. 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);
```

4. You can effect your own error return by using `SETERRQ` (figure 31.1) `SETERRQ1` (figure 31.1), `SETERRQ2` (figure 31.1).

*Fortran note.* 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.

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");
```

### 31.1 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)

||| PetscFunctionReturn(0);
||}
```

For the source of this example, see section 31.6.2

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 fo
[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. 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
```

For the source of this example, see section ??

```
[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 19** 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 31.1.** Look up the definition of `SETERRQ1`. 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
```

### 31.1.3 Memory corruption

PETSc has its own memory management (section 31.5) 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.

## 31.2 Program output

PETSc has a variety of mechanisms to export or visualize program data. We will consider a few possibilities here.

### 31.2.1 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.

#### 31.2.1.1 *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 31.2).

### 31.2 `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
```

### 31.3 `PetscSynchronizedPrintf`

```
C:  
PetscErrorCode PetscSynchronizedPrintf(  
    MPI_Comm comm,const char format[],...)  
  
Fortran:  
PetscSynchronizedPrintf(MPI_Comm, character(*), PetscErrorCode ierr)  
  
python:  
PETSc.Sys.syncPrint(type cls, *args, **kwargs)  
kwargs:  
comm : communicator object  
flush : if True, do synchronizedFlush  
other keyword args as for python3 print function
```

If all processes need to print, you can use `PetscSynchronizedPrintf` (figure 31.3) that forces the output to appear in process order.

To make sure that output is properly flushed from all system buffers use `PetscSynchronizedFlush` (figure 31.4) where for ordinary screen output you would use `stdout` for the file.

*Fortran note.* 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.* 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.

#### 31.2.1.2 *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 31.5), which operates in terms of `PetscDataType`.

#### 31.2.2 Exporting internal data structures

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

### 31.4 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)
```

### 31.5 PetscViewerRead

Synopsis

```
#include "petscviewer.h"  
PetscErrorCode PetscViewerRead(PetscViewer viewer, void *data, PetscInt num, PetscInt *cou  
  
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
```

to a running Matlab process. Calls such as `MatView` or `KSPView` accept a `PetscViewer` argument.

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`.)

Some viewers, such as `KSPView` are mostly for diagnostic purposes. Thus, it is most convenient to activate them through commandline options such `-ksp_view`.

### 31.2.2.1 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 ... ]
```

### 31.2.2.2 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`.

### 31.2.2.3 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`

## 31.3 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.

### 31.3.1 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;
int domain_size = 100;
ierr = PetscOptionsGetInt
    (NULL, PETSC_NULL, "-n", &domain_size, &flag); CHKERRQ(ierr);
PetscPrintf(comm, "Using domain size %d\n", domain_size);
```

For the source of this example, see section [31.6.3](#)

declares the existence of an option `-n` to be followed by an integer.

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.

*Python note.* 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
ierr = PetscOptionsBegin(comm, NULL, "Parameters", NULL); CHKERRQ(ierr);
ierr = PetscOptionsInt("-i", "i value", FILE, i_value, &i_value, &i_flag);
CHKERRQ(ierr);
ierr = PetscOptionsInt("-j", "j value", FILE, j_value, &j_value, &j_flag);
CHKERRQ(ierr);
ierr = PetscOptionsEnd(); CHKERRQ(ierr);
if (i_flag)
    PetscPrintf(comm, "Option '-i' was used\n");
if (j_flag)
    PetscPrintf(comm, "Option '-j' was used\n");
```

For the source of this example, see section [31.6.4](#)

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.

### 31.3.2 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 define 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.

### 31.3.3 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.

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.

## 31.4 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 31.6). 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.

### 31.6 **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)
```

### 31.7 **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

## 31.5 Memory management

Allocate the memory for a given pointer: **PetscNew**, allocate arbitrary memory with **PetscMalloc**, allocate a number of objects with **PetscMalloc1** (figure 31.7) (this does not zero the memory allocated, use **PetscCalloc1** to obtain memory that has been zeroed); use **PetscFree** (figure 31.8) to free.

```
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**.

### 31.8 **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 sucess)

## 31.6 Sources used in this chapter

### 31.6.1 Listing of code header

### 31.6.2 Listing of code examples/petsc/c/backtrace.c

```
#include <stdlib.h>
#include <stdio.h>

#include <petsc.h>

PetscErrorCode this_function_bombs() {
    PetscFunctionBegin;
    SETERRQ(PETSC_COMM_SELF,1,"We cannot go on like this");
    PetscFunctionReturn(0);
}

int main(int argc,char **argv)
{
    PetscErrorCode ierr;

    char help[] = "\nInit example.\n\n";
    ierr = PetscInitialize(&argc,&argv,(char*)0,help); CHKERRQ(ierr);
    ierr = this_function_bombs(); CHKERRQ(ierr);
    ierr = PetscFinalize(); CHKERRQ(ierr);
    return 0;
}
```

### 31.6.3 Listing of code examples/petsc/c/kspcg.c

```
#include "petscksp.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
    PetscErrorCode ierr;
    MPI_Comm comm;
```

```
KSP Solver;
Mat A;
Vec Rhs,Sol;
PetscScalar one = 1.0;

PetscFunctionBegin;
PetscInitialize(&argc,&args,0,0);

comm = PETSC_COMM_SELF;

/*
 * Read the domain size, and square it to get the matrix size
 */
PetscBool flag;
int matrix_size = 100;
ierr = PetscOptionsGetInt
    (NULL,PETSC_NULL,"-n",&matrix_size,&flag); CHKERRQ(ierr);
PetscPrintf(comm,"Using matrix size %d\n",matrix_size);

/*
 * Create the five-point laplacian matrix
 */
ierr = MatCreate(comm,&A); CHKERRQ(ierr);
ierr = MatSetType(A,MATSEQAIJ); CHKERRQ(ierr);
ierr = MatSetSizes(A,matrix_size,matrix_size,matrix_size,matrix_size); CHKERRQ(ierr);
ierr = MatSeqAIJSetPreallocation(A, 3, PETSC_NULL); CHKERRQ(ierr);
ierr = MatCreateVecs(A,&Rhs,PETSC_NULL); CHKERRQ(ierr);
for (int i=0; i<matrix_size; i++) {
    PetscScalar
        h = 1./(matrix_size+1), pi = 3.1415926,
        sin1 = i * pi * h, sin2 = 2 * i * pi * h, sin3 = 3 * i * pi * h,
        coefs[3] = {-1,2,-1};
    PetscInt cols[3] = {i-1,i,i+1};
    ierr = VecSetValue(Rhs,i,sin1 + .5 * sin2 + .3 * sin3, INSERT_VALUES); CHKERRQ(ierr);
    if (i==0) {
        ierr = MatSetValues(A,1,&i,2,cols+1,coefs+1,INSERT_VALUES); CHKERRQ(ierr);
    } else if (i==matrix_size-1) {
        ierr = MatSetValues(A,1,&i,2,cols,coefs,INSERT_VALUES); CHKERRQ(ierr);
    } else {
        ierr = MatSetValues(A,1,&i,2,cols,coefs,INSERT_VALUES); CHKERRQ(ierr);
    }
}
ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
//MatView(A,PETSC_VIEWER_STDOUT_WORLD);

/*
 * Create right hand side and solution vectors
 */
ierr = VecDuplicate(Rhs,&Sol); CHKERRQ(ierr);
ierr = VecSet(Rhs,one); CHKERRQ(ierr);

/*
```

```

    * Create iterative method and preconditioner
    */
 ierr = KSPCreate(comm,&Solver);
 ierr = KSPSetOperators(Solver,A,A); CHKERRQ(ierr);
 ierr = KSPSetType(Solver,KSPCG); CHKERRQ(ierr);
 {
    PC Prec;
    ierr = KSPGetPC(Solver,&Prec); CHKERRQ(ierr);
    ierr = PCSetType(Prec,PCNONE); CHKERRQ(ierr);
 }

/*
 * Incorporate any commandline options for the KSP
 */
 ierr = KSPSetFromOptions(Solver); CHKERRQ(ierr);

/*
 * Solve the system and analyze the outcome
 */
 ierr = KSPSolve(Solver,Rhs,Sol); CHKERRQ(ierr);
 {
    PetscInt its; KSPConvergedReason reason;
    ierr = KSPGetConvergedReason(Solver,&reason);
    ierr = KSPGetIterationNumber(Solver,&its); CHKERRQ(ierr);
    if (reason<0) {
        PetscPrintf(comm,"Failure to converge after %d iterations; reason %s\n",
                    its,KSPConvergedReasons[reason]);
    } else {
        PetscPrintf(comm,"Number of iterations to convergence: %d\n",its);
    }
}

ierr = MatDestroy(&A); CHKERRQ(ierr);
ierr = KSPDestroy(&Solver); CHKERRQ(ierr);
ierr = VecDestroy(&Rhs); CHKERRQ(ierr);
ierr = VecDestroy(&Sol); CHKERRQ(ierr);

ierr = PetscFinalize();
PetscFunctionReturn(0);
}

```

#### 31.6.4 Listing of code examples/petsc/c/optionsbegin.c

```

#include "petsc.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
    PetscErrorCode ierr;
    MPI_Comm comm;

```

```
PetscFunctionBegin;
PetscInitialize(&argc,&args,0,0);
comm = MPI_COMM_WORLD;

PetscInt i_value = 1, j_value = 1;
PetscBool i_flag = 0, j_flag = 0;

ierr = PetscOptionsBegin(comm,NULL,"Parameters",NULL); CHKERRQ(ierr);
ierr = PetscOptionsInt("-i","i value",__FILE__,i_value,&i_value,&i_flag); CHKERRQ(ierr);
ierr = PetscOptionsInt("-j","j value",__FILE__,j_value,&j_value,&j_flag); CHKERRQ(ierr);
ierr = PetscOptionsEnd(); CHKERRQ(ierr);
if (i_flag)
    PetscPrintf(comm,"Option '-i' was used\n");
if (j_flag)
    PetscPrintf(comm,"Option '-j' was used\n");

PetscPrintf(comm,"i=%d, j=%d\n",i_value,j_value);

PetscFinalize();
}
```

## Chapter 32

### PETSc topics

#### 32.1 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`.

#### 32.2 Scalars

The definition of `PetscInt`, `PetscReal`, `PetscComplex` depends on how PETSc was installed. This makes interoperability with other libraries such as `MPI` a little tricky.

The equivalent `MPI_Datatype` values are `PetscMPIInt MPIU_INT MPIU_REAL MPIU_SCALAR MPIU_COMPLEX`

Similarly, for the `BLAS` library there are: `PetscBLASInt`

No real types are needed since these are passed by reference.

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)
```

**32.3 Sources used in this chapter**

**32.3.1 Listing of code header**

## **PART IV**

### **OTHER PROGRAMMING MODELS**

# Chapter 33

## Co-array Fortran

This chapter explains the basic concepts of Co-array Fortran (CAF), and helps you get started on running your first program.

### 33.1 History and design

[https://en.wikipedia.org/wiki/Coarray\\_Fortran](https://en.wikipedia.org/wiki/Coarray_Fortran)

### 33.2 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.

### 33.3 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 Message Passing Interface (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.

### 33.3.1 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"
```

*For the source of this example, see section 33.4.2*

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.

### 33.3.2 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]
```

### 33.3.3 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 non-blocking 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
```

For the source of this example, see section [33.4.3](#)

### 33.3.4 Collectives

Collectives are not part of CAF as of the 2008 Fortran standard.

### 33.4 Sources used in this chapter

#### 33.4.1 Listing of code header

#### 33.4.2 Listing of code examples/caf/f08/hello.F90

```
program hello_image

    write(*,*) "Hello from image ", this_image(), &
                "out of ", num_images()," total images"

end program hello_image
```

#### 33.4.3 Listing of code examples/caf/f08/rightcopy.F90

```
program RightCopy

    integer,dimension(2),codimension[*] :: numbers
    integer :: procid,nprocs

    procid = this_image()
    nprocs = num_images()
    numbers(:)[procid] = procid
    if (procid<nprocs) then
        numbers(1)[procid+1] = procid
    end if
    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

    write(*,*) "After shift,",procid," has",numbers(:)[procid]

end program RightCopy
```

## Chapter 34

### Sycl, OneAPI, DPC++

This chapter explains the basic concepts of Data Parallel C++ (DPCPP), and helps you get started on running your first program.

#### 34.1 Logistics

Headers:

```
|| #include <CL/sycl.hpp>
|| using namespace cl::sycl;
```

**Remark 20** *Warning! The `cl::sycl` name space has its own versions of `cout` and `endl`. Make sure to use explicitly `std::cout` and `std::endl`. Using the wrong I/O will cause tons of inscrutable error messages.*

#### 34.2 Platforms and devices

Since DPCPP is cross-platform, we first need to discover the devices.

```
|| vector<sycl::platform> platforms = cl::sycl::platform::get_platforms();
|| for (const auto &plat : platforms) {
||     cout << plat.get_info<sycl::info::platform::name>() << endl;
||     cout << plat.get_info<sycl::info::platform::vendor>() << " ";
||     cout << plat.get_info<sycl::info::platform::version>() << endl;
||     vector<cl::sycl::device> devices = plat.get_devices();
||     for (const auto &dev : devices) {
||         for (const auto &dev : devices) {
||             cout << dev.get_info<sycl::info::device::name>() << " " << endl;
||             cout << " " << (dev.is_gpu() ? "is a gpu" : "is not a gpu") << endl;
||             cout << " " << (dev.is_cpu() ? "is a cpu" : "is not a cpu") << endl;
||         }
||     }
|| }
```

### 34.3 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.

#### 34.3.1 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`. The `sycl::host_selector` bypasses any devices and make the code run on the host.

```
{ // open scope!
    sycl::cpu_selector selector;
    sycl::queue myqueue(selector);
    cout << myqueue.get_device().get_info<sycl::info::device::name>() << endl;
    myqueue.submit(
        ( [&] (sycl::handler &handle) {
            // some action
        })
    );
} // queue will be executed at the close of the scope
```

If you try to select a device that is not available, a `sycl::runtime_error` exception will be thrown.

#### 34.3.2 Queue execution

It seems that queue kernels will also be executed when only they go out of scope, but not the queue:

```
|| cpu_selector selector;
|| queue q(selector);
{
    q.submit( /* some kernel */ );
} // here the kernel executes
```

### 34.4 Kernels

One kernel per submit.

```

    myqueue.submit( [&] ( handler &commandgroup ) {
        commandgroup.parallel_for<uniquename>
        ( range<1>{N},
        [=] ( id<1> idx ) { ... idx }
        )
    }

    cgh.single_task(
        [=] () {
            // kernel function is executed EXACTLY once on a SINGLE work-item
        });
}

```

### 34.4.1 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
});

```

#### 34.4.1.1 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;
  }
);
```

### 34.4.2 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>(...);
}
);
```

### 34.5 Unified memory

Arrays need to be declared in a way such that they can be access from any device.

```
unsigned long global_range = 5000000;

// create traditional data
vector<float> myarray(global_range);

// make sycl buffer around the C++ data
sycl::range<1> vector_range{global_range};
sycl::buffer<float, 1> myarray_buffer(myarray.data(), vector_range);

myqueue.submit
( [&] (sycl::handler &handle) {
    auto myarray =
        myarray_buffer.get_access<sycl::access::mode::read_write>(handle);
    // and now you can operate on that buffer
```

## 34.6 Parallel output

There is a *sycl::cout* and *sycl::endl*.

```
namespace sycl = cl::sycl;

myQueue.submit
( [&] (sycl::handler &cgh) {
    // Create a output stream
    sycl::stream cout(1024, 256, cgh);

    // Submit a unique task, using a lambda
    cgh.single_task<class hello_world>
    ( [=] () {
        cout << "Hello, World!" << sycl::endl; }
```

**34.7 Sources used in this chapter**

**34.7.1 Listing of code header**

## **PART V**

### **THE REST**

## **Chapter 35**

### **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?

#### **35.1 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.

##### **35.1.1 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.

##### **35.1.2 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.

**35.2 Sources used in this chapter**

**35.2.1 Listing of code header**

## Chapter 36

### Process and thread affinity

In the preceding chapters we mostly considered all MPI nodes or OpenMP threads 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.

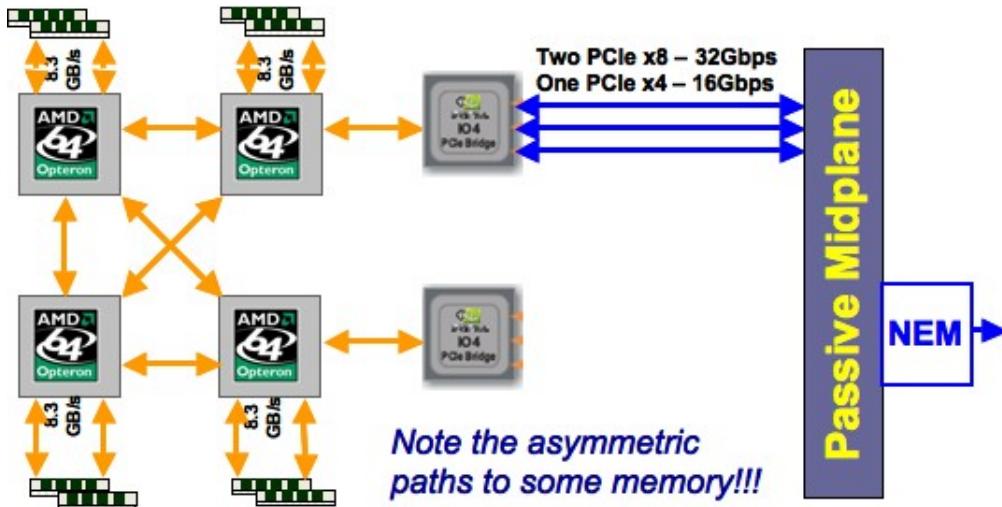


Figure 36.1: The NUMA structure of a Ranger node

Here are some situations where 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 10.2.
- Even on a single node there can be asymmetries. Figure 36.1 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 22.2.
- 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.

## 36.1 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 [6] (<https://www.open-mpi.org/projects/hwloc/>).

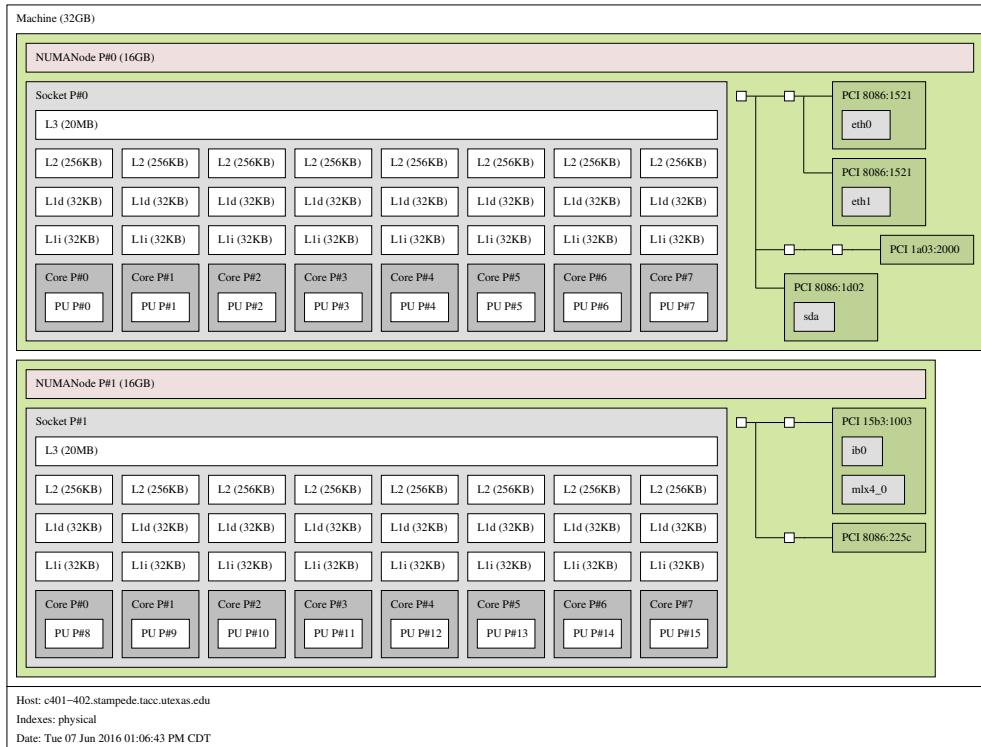


Figure 36.2: Structure of a Stampede compute node

Figure 36.2 depicts a *Stampede compute node*, which is a two-socket *Intel SandyBridge* design; figure 36.3 shows a *Stampede largemem node*, which is a four-socket design. Finally, figure 36.4 shows a *Lonestar5* compute node, a two-socket design with 12-core *Intel Haswell* processors with two hardware threads each.

## 36. Process and thread affinity

---

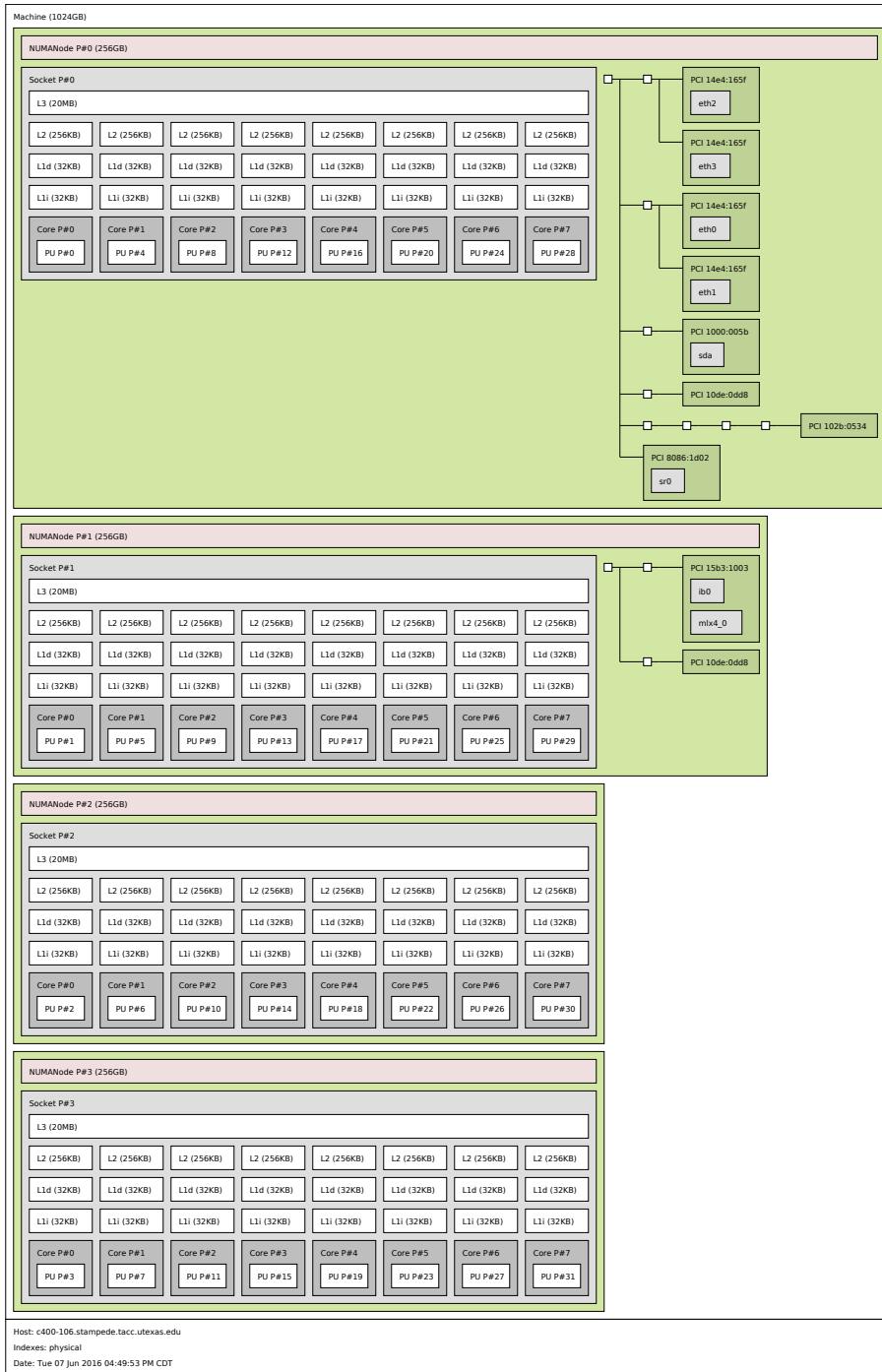


Figure 36.3: Structure of a Stampede largemem four-socket compute node

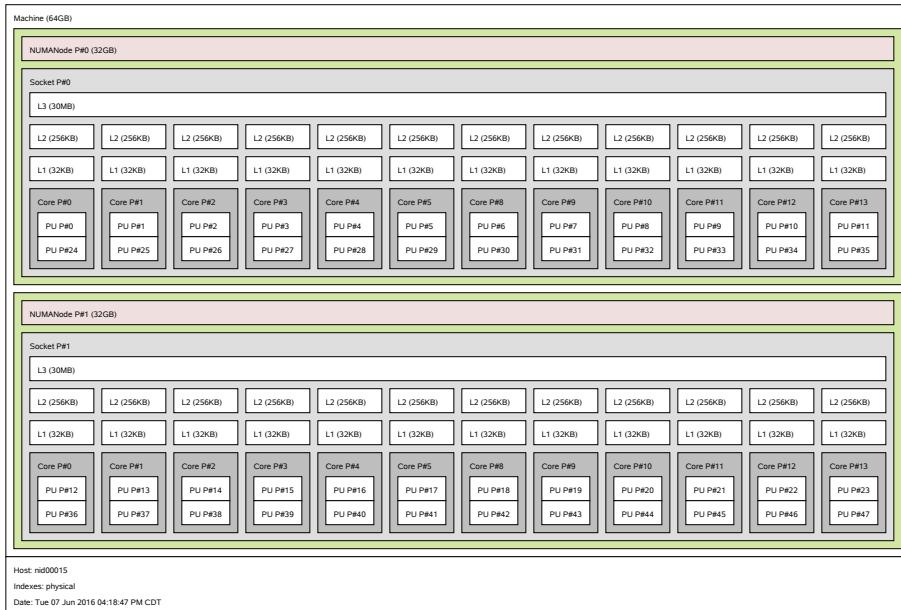


Figure 36.4: Structure of a Lonestar5 compute node

## 36.2 Affinity control

See chapter 22 for OpenMP affinity control.

**36.3 Sources used in this chapter**

**36.3.1 Listing of code header**

## Chapter 37

### 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.

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.

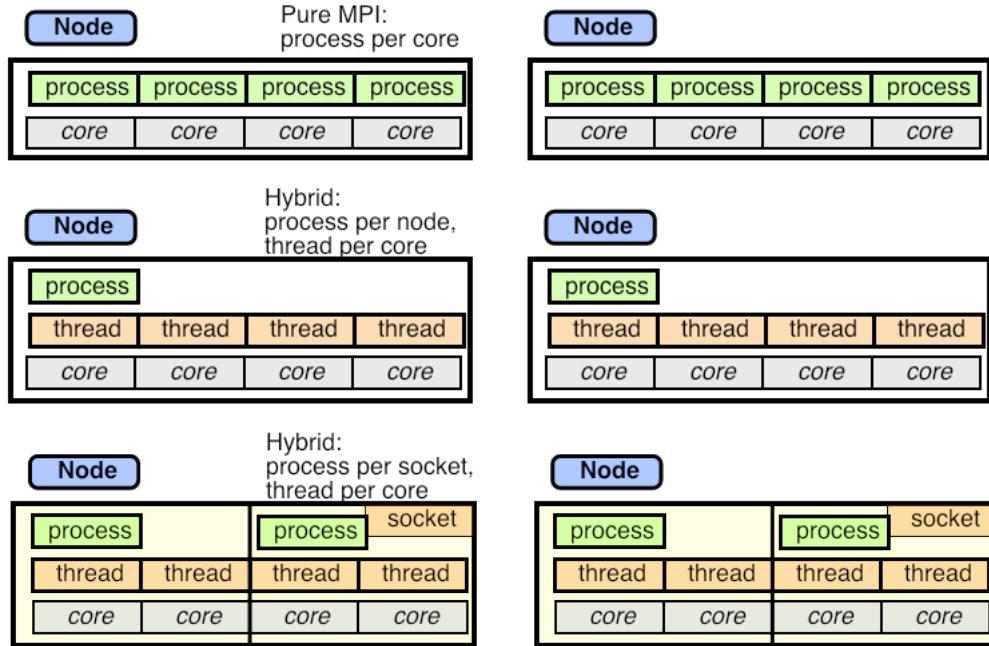


Figure 37.1: Three modes of MPI/OpenMP usage on a multi-core cluster

Figure 37.1 illustrates these 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.

### 37.1 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 behaviour 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.

### 37.1 MPI\_Init\_thread

```
C:  
int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)  
  
Fortran:  
MPI_Init_thread(required, provided, ierror)  
INTEGER, INTENT(IN) :: required  
INTEGER, INTENT(OUT) :: provided  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**Exercise 37.1.** Review the scalability argument for 1D versus 2D matrix decomposition in HPSC-?. 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 37.2) 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.

## 37.2 Hybrid MPI-plus-threads execution

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 37.1). 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 37.2).

In case more than one thread performs communication, `MPI_Is_thread_main` (figure 37.3) can determine whether a thread is the main thread.

### 37.2 MPI\_Query\_thread

```
C:  
int MPI_Query_thread(int *provided)  
  
Fortran:  
MPI_Query_thread(provided, ierror)  
INTEGER, INTENT(OUT) :: provided  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### 37.3 MPI\_Is\_thread\_main

```
C:  
int MPI_Is_thread_main(int *flag)  
  
Fortran:  
MPI_Is_thread_main(flag, ierror)  
LOGICAL, INTENT(OUT) :: flag  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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 *mpirun* program usually propagates *environment variables*, so the value of OMP\_NUM\_THREADS when you call mpirun will be seen by each MPI process.

- 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 37.2.** 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();
```

For the source of this example, see section [37.3.2](#)

### 37.3 Sources used in this chapter

#### 37.3.1 Listing of code header

#### 37.3.2 Listing of code examples/mpi/c/thread.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

    MPI_Comm comm;
    int procno=-1,nprocs,threading,err;

    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();
    return 0;
}
```

## Chapter 38

### Random number generation

Here is how you initialize the random number generator uniquely on each process:

C:

```
// Initialize the random number generator
srand((int)(mytid*(double)RAND_MAX/ntids));
// compute a random float between [0,1]
randomfraction = (rand() / (double)RAND_MAX);
// compute random integer between [0,N-1]
randomfraction = rand() % N;
```

Fortran:

```
integer :: randsize
integer,allocatable,dimension(:) :: randseed
real :: random_value

call random_seed(size=randsize)
allocate(randseed(randsize))
randseed(:) = 1023*mytid
call random_seed(put=randseed)
```

**38.1 Sources used in this chapter**

**38.1.1 Listing of code header**

## Chapter 39

### Parallel I/O

Parallel I/O is a tricky subject. You can try to let all processors jointly write one file, or to write a file per process and combine them later. With the standard mechanisms of your programming language there are the following considerations:

- On clusters where the processes have individual file systems, the only way to write a single file is to let it be generated by a single processor.
- Writing one file per process is easy to do, but
  - You need a post-processing script;
  - if the files are not on a shared file system (such as *Lustre*), it takes additional effort to bring them together;
  - if the files are on a shared file system, writing many files may be a burden on the metadata server.
- On a shared file system it is possible for all files to open the same file and set the file pointer individually. This can be difficult if the amount of data per process is not uniform.

Illustrating the last point:

```
// pseek.c
FILE *pfile;
pfile = fopen("pseek.dat", "w");
fseek(pfile, procid*sizeof(int), SEEK_CUR);
fseek(pfile, procid*sizeof(char), SEEK_CUR);
fprintf(pfile, "%d\n", procid);
fclose(pfile);
```

*For the source of this example, see section 39.1.2*

MPI also has its own portable I/O: *MPI I/O*, for which see chapter 9.

Alternatively, one could use a library such as *hdf5*.

**39.1 Sources used in this chapter****39.1.1 Listing of code header****39.1.2 Listing of code code/mpi/c/pseek.c**

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

int main(int argc,char **argv) {
    int ntids,mytid;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&procid);

    FILE *pfile;
    pfile = fopen("pseek.dat","w");
    fseek(pfile,procid*sizeof(int),SEEK_CUR);
    fseek(pfile,procid*sizeof(char),SEEK_CUR);
    fprintf(pfile,"%d\n",procid);
    fclose(pfile);

    MPI_Finalize();

    return 0;
}
```

## **Chapter 40**

### **Support libraries**

There are many libraries related to parallel programming to make life easier, or at least more interesting, for you.

#### **40.1 SimGrid**

SimGrid [10] 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 **HPL!** (**HPL!**) [2].

#### **40.2 Other**

ParaMesh

Global Arrays

Hdf5 and Silo

**40.3 Sources used in this chapter**

**40.3.1 Listing of code header**

## **PART VI**

### **TUTORIALS**

---

here are some tutorials specific to parallel programming.

1. Debugging, first sequential, then parallel. Chapter [40.4](#).
2. Tracing and profiling. Chapter [40.5](#).
3. SimGrid: a simulation tool for cluster execution. Chapter [40.6](#).
4. Batch systems, in particular Simple Linux Utility for Resource Management (SLURM). Chapter [40.7](#).

## 40.4 Debugging

When a program misbehaves, *debugging* is the process of finding out *why*. There are various strategies of finding errors in a program. The crudest one is debugging by print statements. If you have a notion of where in your code the error arises, you can edit your code to insert print statements, recompile, rerun, and see if the output gives you any suggestions. There are several problems with this:

- The edit/compile/run cycle is time consuming, especially since
- often the error will be caused by an earlier section of code, requiring you to edit, compile, and rerun repeatedly. Furthermore,
- the amount of data produced by your program can be too large to display and inspect effectively, and
- if your program is parallel, you probably need to print out data from all processors, making the inspection process very tedious.

For these reasons, the best way to debug is by the use of an interactive *debugger*, a program that allows you to monitor and control the behaviour of a running program. In this section you will familiarize yourself with *gdb*, which is the open source debugger of the *GNU* project. Other debuggers are proprietary, and typically come with a compiler suite. Another distinction is that *gdb* is a commandline debugger; there are graphical debuggers such as *ddd* (a frontend to *gdb*) or *DDT* and *TotalView* (debuggers for parallel codes). We limit ourselves to *gdb*, since it incorporates the basic concepts common to all debuggers.

In this tutorial you will debug a number of simple programs with *gdb* and *valgrind*. The files can be found in the repository in the directory `tutorials/debug_tutorial_files`.

### 40.4.1 Step 0: compiling for debug

You often need to recompile your code before you can debug it. A first reason for this is that the binary code typically knows nothing about what variable names corresponded to what memory locations, or what lines in the source to what instructions. In order to make the binary executable know this, you have to include the *symbol table* in it, which is done by adding the `-g` option to the compiler line.

Usually, you also need to lower the *compiler optimization level*: a production code will often be compiled with flags such as `-O2` or `-Xhost` that try to make the code as fast as possible, but for debugging you need to replace this by `-O0` ('oh-zero'). The reason is that higher levels will reorganize your code, making it hard to relate the execution to the source<sup>1</sup>.

### 40.4.2 Invoking gdb

There are three ways of using *gdb*: using it to start a program, attaching it to an already running program, or using it to inspect a *core dump*. We will only consider the first possibility.

Here is an example of how to start *gdb* with a program that has no arguments (Fortran users, use `hello.F`):

```
tutorials/gdb/c/hello.c
```

---

1. Typically, actual code motion is done by `-O3`, but at level `-O2` the compiler will inline functions and make other simplifications.

---

```

%% cc -g -o hello hello.c
# regular invocation:
%% ./hello
hello world
# invocation from gdb:
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
Copyright 2004 Free Software Foundation, Inc. .... copyright info ....
(gdb) run
Starting program: /home/eijkhout/tutorials/gdb/hello
Reading symbols for shared libraries +. done
hello world

Program exited normally.
(gdb) quit
%%

```

Important note: the program was compiled with the *debug flag* `-g`. This causes the *symbol table* (that is, the translation from machine address to program variables) and other debug information to be included in the binary. This will make your binary larger than strictly necessary, but it will also make it slower, for instance because the compiler will not perform certain optimizations<sup>2</sup>.

To illustrate the presence of the symbol table do

```

%% cc -g -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list

```

and compare it with leaving out the `-g` flag:

```

%% cc -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list

```

For a program with commandline input we give the arguments to the `run` command (Fortran users use `say.F`):

`tutorials/gdb/c/say.c`

```

%% cc -o say -g say.c
%% ./say 2

```

---

2. Compiler optimizations are not supposed to change the semantics of a program, but sometimes do. This can lead to the nightmare scenario where a program crashes or gives incorrect results, but magically works correctly with compiled with debug and run in a debugger.

```

hello world
hello world
%% gdb say
.... the usual messages ...
(gdb) run 2
Starting program: /home/eijkhout/tutorials/gdb/c/say 2
Reading symbols for shared libraries +. done
hello world
hello world

Program exited normally.

```

### 40.4.3 Finding errors

Let us now consider some programs with errors.

#### 40.4.3.1 C programs

tutorials/gdb/c/square.c

```

%% cc -g -o square square.c
%% ./square
5000
Segmentation fault

```

The *segmentation fault* (other messages are possible too) indicates that we are accessing memory that we are not allowed to, making the program abort. A debugger will quickly tell us where this happens:

```

%% gdb square
(gdb) run
50000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x000000000000eb4a
0x00007fff824295ca in __svfscanf_l ()

```

Apparently the error occurred in a function `__svfscanf_l`, which is not one of ours, but a system function. Using the backtrace (or `bt`, also `where` or `w`) command we quickly find out how this came to be called:

```

(gdb) backtrace
#0 0x00007fff824295ca in __svfscanf_l ()
#1 0x00007fff8244011b in fscanf ()
#2 0x0000000100000e89 in main (argc=1, argv=0x7fff5fbfc7c0) at square.c:7

```

---

We take a close look at line 7, and see that we need to change nmax to &nmax.

There is still an error in our program:

```
(gdb) run  
50000  
  
Program received signal EXC_BAD_ACCESS, Could not access memory.  
Reason: KERN_PROTECTION_FAILURE at address: 0x000000010000f000  
0x000000010000ebe in main (argc=2, argv=0x7fff5fbfc7a8) at square1.c:9  
9           squares[i] = 1. / (i * i); sum += squares[i];
```

We investigate further:

```
(gdb) print i  
$1 = 11237  
(gdb) print squares[i]  
Cannot access memory at address 0x10000f000
```

and we quickly see that we forgot to allocate squares.

By the way, we were lucky here: this sort of memory errors is not always detected. Starting our programm with a smaller input does not lead to an error:

```
(gdb) run  
50  
Sum: 1.625133e+00  
  
Program exited normally.
```

#### 40.4.3.2 Fortran programs

Compile and run the following program:

tutorials/gdb/f/square.F

It should abort with a message such as ‘Illegal instruction’. Running the program in gdb quickly tells you where the problem lies:

```
(gdb) run  
Starting program: tutorials/gdb//fsquare  
Reading symbols for shared libraries +++. done  
  
Program received signal EXC_BAD_INSTRUCTION, Illegal instruction/operand.  
0x000000010000da3 in square () at square.F:7  
7           sum = sum + squares(i)
```

We take a close look at the code and see that we did not allocate squares properly.

#### 40.4.4 Memory debugging with Valgrind

Insert the following allocation of `squares` in your program:

```
squares = (float *) malloc( nmax*sizeof(float) );
```

Compile and run your program. The output will likely be correct, although the program is not. Can you see the problem?

To find such subtle memory errors you need a different tool: a memory debugging tool. A popular (because open source) one is *valgrind*; a common commercial tool is *purify*.

**tutorials/gdb/c/square1.c** Compile this program with `cc -o square1 square1.c` and run it with `valgrind square1` (you need to type the input value). You will lots of output, starting with:

```
%% valgrind square1
==53695== Memcheck, a memory error detector
==53695== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==53695== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==53695== Command: a.out
==53695==
10
==53695== Invalid write of size 4
==53695==   at 0x100000EB0: main (square1.c:10)
==53695==   Address 0x10027e148 is 0 bytes after a block of size 40 alloc'd
==53695==   at 0x1000101EF: malloc (vg_replace_malloc.c:236)
==53695==   by 0x100000E77: main (square1.c:8)
==53695==
==53695== Invalid read of size 4
==53695==   at 0x100000EC1: main (square1.c:11)
==53695==   Address 0x10027e148 is 0 bytes after a block of size 40 alloc'd
==53695==   at 0x1000101EF: malloc (vg_replace_malloc.c:236)
==53695==   by 0x100000E77: main (square1.c:8)
```

Valgrind is informative but cryptic, since it works on the bare memory, not on variables. Thus, these error messages take some exegesis. They state that a line 10 writes a 4-byte object immediately after a block of 40 bytes that was allocated. In other words: the code is writing outside the bounds of an allocated array. Do you see what the problem in the code is?

Note that valgrind also reports at the end of the program run how much memory is still in use, meaning not properly freed.

If you fix the array bounds and recompile and rerun the program, valgrind still complains:

```
==53785== Conditional jump or move depends on uninitialised value(s)
==53785==   at 0x10006FC68: __ dtoa (in /usr/lib/libSystem.B.dylib)
==53785==   by 0x10003199F: __ vfprintf (in /usr/lib/libSystem.B.dylib)
==53785==   by 0x1000738AA: vfprintf_l (in /usr/lib/libSystem.B.dylib)
==53785==   by 0x1000A1006: printf (in /usr/lib/libSystem.B.dylib)
==53785==   by 0x100000EF3: main (in ./square2)
```

---

Although no line number is given, the mention of `printf` gives an indication where the problem lies. The reference to an ‘uninitialized value’ is again cryptic: the only value being output is `sum`, and that is not uninitialized: it has been added to several times. Do you see why valgrind calls `is uninitialized` all the same?

#### 40.4.5 Stepping through a program

Often the error in a program is sufficiently obscure that you need to investigate the program run in detail. Compile the following program

tutorials/gdb/c/roots.c and run it:

```
%% ./roots
sum: nan
```

Start it in `gdb` as follows:

```
%% gdb roots
GNU gdb 6.3.50-20050815 (Apple version gdb-1469) (Wed May 5 04:36:56 UTC 2005)
Copyright 2004 Free Software Foundation, Inc.
...
(gdb) break main
Breakpoint 1 at 0x100000ea6: file root.c, line 14.
(gdb) run
Starting program: tutorials/gdb/c/roots
Reading symbols for shared libraries +. done

Breakpoint 1, main () at roots.c:14
14          float x=0;
```

Here you have done the following:

- Before calling `run` you set a *breakpoint* at the main program, meaning that the execution will stop when it reaches the main program.
- You then call `run` and the program execution starts;
- The execution stops at the first instruction in `main`.

If execution is stopped at a breakpoint, you can do various things, such as issuing the `step` command:

```
Breakpoint 1, main () at roots.c:14
14          float x=0;
(gdb) step
15          for (i=100; i>-100; i--)
(gdb)
16          x += root(i);
(gdb)
```

(if you just hit return, the previously issued command is repeated). Do a number of steps in a row by hitting return. What do you notice about the function and the loop?

Switch from doing step to doing next. Now what do you notice about the loop and the function?

Set another breakpoint: break 17 and do cont. What happens?

Rerun the program after you set a breakpoint on the line with the sqrt call. When the execution stops there do where and list.

- If you set many breakpoints, you can find out what they are with info breakpoints.
- You can remove breakpoints with delete n where n is the number of the breakpoint.
- If you restart your program with run without leaving gdb, the breakpoints stay in effect.
- If you leave gdb, the breakpoints are cleared but you can save them: save breakpoints <file>. Use source <file> to read them in on the next gdb run.

#### 40.4.6 Inspecting values

Run the previous program again in gdb: set a breakpoint at the line that does the sqrt call before you actually call run. When the program gets to line 8 you can do print n. Do cont. Where does the program stop?

If you want to repair a variable, you can do set var=value. Change the variable n and confirm that the square root of the new value is computed. Which commands do you do?

If a problem occurs in a loop, it can be tedious keep typing cont and inspecting the variable with print. Instead you can add a condition to an existing breakpoint: the following:

```
condition 1 if (n<0)
```

or set the condition when you define the breakpoint:

```
break 8 if (n<0)
```

Another possibility is to use ignore 1 50, which will not stop at breakpoint 1 the next 50 times.

Remove the existing breakpoint, redefine it with the condition n<0 and rerun your program. When the program breaks, find for what value of the loop variable it happened. What is the sequence of commands you use?

#### 40.4.7 Parallel debugging

Debugging parallel programs is harder than sequential programs, because every sequential bug may show up, plus a number of new types, caused by the interaction of the various processes.

Here are a few possible parallel bugs:

- Processes can deadlock because they are waiting for a message that never comes. This typically happens with blocking send/receive calls due to an error in program logic.
- If an incoming message is unexpectedly larger than anticipated, a memory error can occur.

- 
- A collective call will hang if somehow one of the processes does not call the routine.

There are few low-budget solutions to parallel debugging. The main one is to create an xterm for each process. We will describe this next. There are also commercial packages such as *DDT* and *TotalView*, that offer a GUI. They are very convenient but also expensive. The *Eclipse* project has a parallel package, *Eclipse PTP*, that includes a graphic debugger.

#### 40.4.7.1 MPI debugging with gdb

You can not run parallel programs in gdb, but you can start multiple gdb processes that behave just like MPI processes! The command

```
mpirun -np <NP> xterm -e gdb ./program
```

create a number of xterm windows, each of which execute the commandline `gdb ./program`. And because these xterms have been started with `mpirun`, they actually form a communicator.

#### 40.4.8 Further reading

A good tutorial: <http://www.dirac.org/linux/gdb/>.

Reference manual: [http://www.ofb.net-gnu/gdb/gdb\\_toc.html](http://www.ofb.net-gnu/gdb/gdb_toc.html).

## 40.5 Tracing and profiling with TAU

TAU <http://www.cs.uoregon.edu/Research/tau/home.php> is a utility for profiling and tracing your parallel programs. Profiling is the gathering and displaying of bulk statistics, for instance showing you which routines take the most time, or whether communication takes a large portion of your runtime. When you get concerned about performance, a good profiling tool is indispensable.

Tracing is the construction and displaying of time-dependent information on your program run, for instance showing you if one process lags behind others. For understanding a program's behaviour, and the reasons behind profiling statistics, a tracing tool can be very insightful.

### 40.5.1 Workflow

#### 40.5.1.1 Instrumentation

Unlike such tools as VTune which profile your binary as-is, TAU works by adding *instrumentation* to your code: in effect it is a source-to-source translator that takes your code and turns it into one that generates run-time statistics.

This instrumentation is largely done for you; you mostly need to recompile your code with a script that does the source-to-source translation, and subsequently compiles that instrumented code. You could for instance have the following in your makefile:

```
ifdef TACC_TAU_DIR
    CC = tau_cc.sh
else
    CC = mpicc
endif

% : %.c
<TAB>${CC} -o $@ $^
```

If TAU is to be used (which we detect here by checking for the environment variable TACC\_TAU\_DIR), we define the CC variable as one of the TAU compilation scripts; otherwise we set it to a regular MPI compiler.

#### 40.5.1.2 Running

You can now run your instrumented code; trace/profile output will be written to file if environment variables TAU\_PROFILE and/or TAU\_TRACE are set:

```
export TAU_PROFILE=1
export TAU_TRACE=1
```

A TAU run can generate many files: typically at least one per process. It is therefore advisable to create a directory for your tracing and profiling information. You declare them to TAU by setting the environment variables PROFILEDIR and TRACEDIR.

---

```
mkdir tau_trace
mkdir tau_profile
export PROFILEDIR=tau_profile
export TRACEDIR=tau_trace
```

The actual program invocation is then unchanged:

```
mpirun -np 26 myprogram
```

*TACC note.* At TACC, use `i brun` without a processor count; the count is derived from the queue submission parameters.

While this example uses two separate directories, there is no harm in using the same for both.

#### 40.5.1.3 Output

The tracing/profiling information is spread over many files, and hard to read as such. Therefore, you need some further programs to consolidate and display the information.

You view profiling information with `paraprof`

```
paraprof tau_profile
```

Viewing the traces takes a few steps:

```
cd tau_trace
rm -f tau.trc tau.edf align.trc align.edf
tau_treemerge.pl
tau_timecorrect tau.trc tau.edf align.trc align.edf
tau2slog2 align.trc align.edf -o yourprogram.slog2
```

If you skip the `tau_timecorrect` step, you can generate the `slog2` file by:

```
tau2slog2 tau.trc tau.edf -o yourprogram.slog2
```

The `slog2` file can be viewed with `jumpshot`:

```
jumpshot yourprogram.slog2
```

## 40.5.2 Examples

### 40.5.2.1 Bucket brigade

Let's consider a *bucket brigade* implementation of a broadcast: each process sends its data to the next higher rank.

```

int sendto =
    ( procno<nprocs-1 ? procno+1 : MPI_PROC_NULL )
;
int recvfrom =
    ( procno>0 ? procno-1 : MPI_PROC_NULL )
;

MPI_Recv( leftdata,1,MPI_DOUBLE,recvfrom,0,comm,MPI_STATUS_IGNORE);
myvalue = leftdata
MPI_Send( myvalue,1,MPI_DOUBLE,sendto,0,comm);

```

We implement the bucket brigade with blocking sends and receives: each process waits to receive from its predecessor, before sending to its successor.

```

// bucketblock.c
for (int i=0; i<N; i++)
    myvalue[i] = (procno+1)*(procno+1) + leftdata[i];
MPI_Send( myvalue,N,MPI_DOUBLE,sendto,0,comm);

```

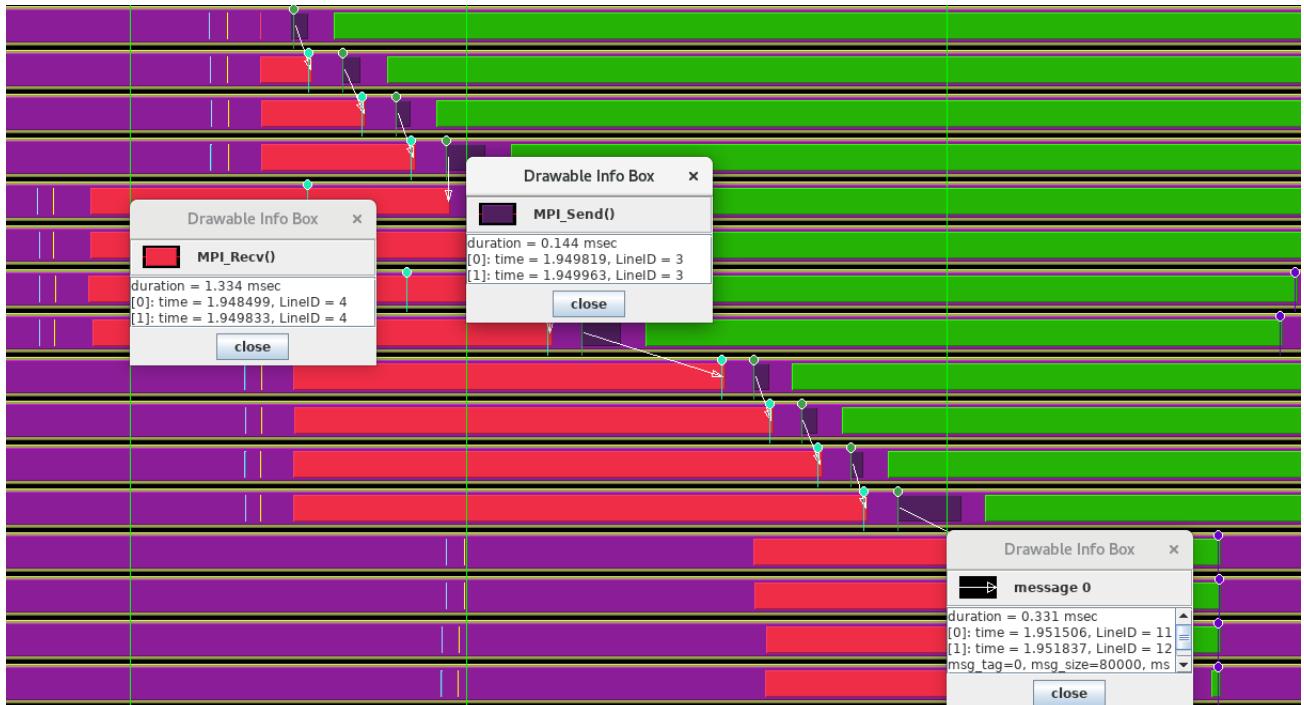


Figure 40.1: Trace of a bucket brigade broadcast

The TAU trace of this is in figure 40.1, using 4 nodes of 4 ranks each. We see that the processes within each node are fairly well synchronized, but there is less synchronization between the nodes. However, the bucket brigade then imposes its own synchronization on the processes because each has to wait for its predecessor, no matter if it posted the receive operation early.

Next, we introduce pipelining into this operation: each send is broken up into parts, and these parts are sent

and received with non-blocking calls.

```
// bucketpipenonblock.c
MPI_Request rrequests[PARTS];
for (int ipart=0; ipart<PARTS; ipart++) {
    MPI_Irecv
    (
        leftdata+partition_starts[ipart], partition_sizes[ipart],
        MPI_DOUBLE, recvfrom, ipart, comm, rrequests+ipart);
}
```

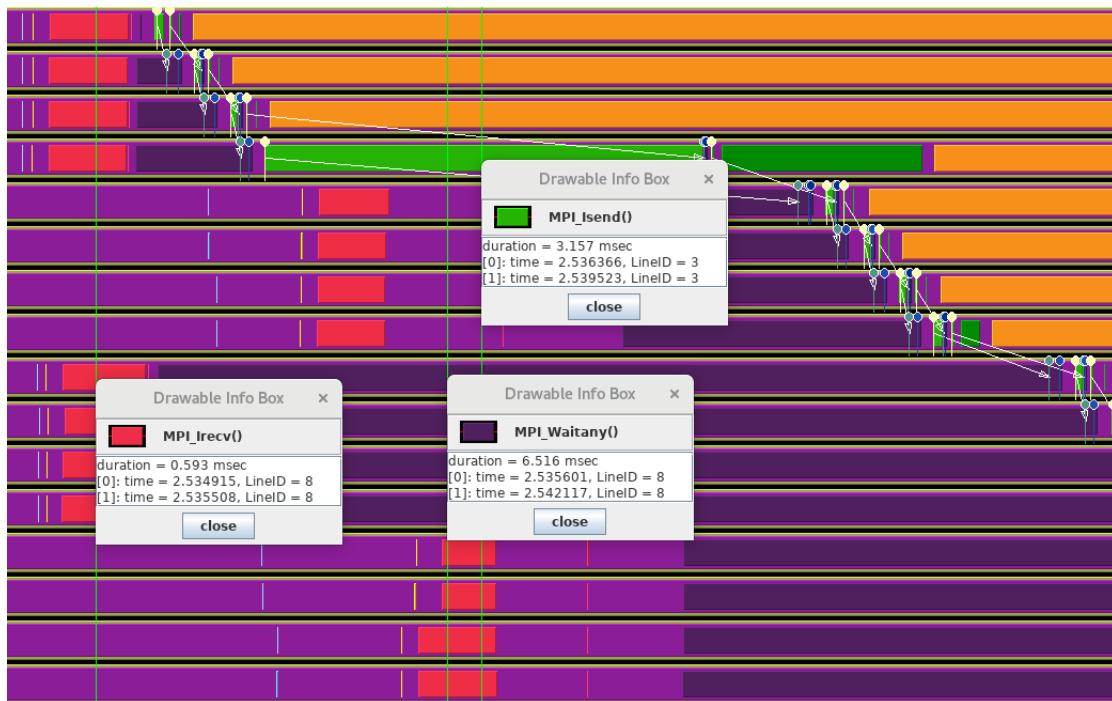


Figure 40.2: Trace of a pipelined bucket brigade broadcast

The TAU trace is in figure 40.2.

#### 40.5.2.2 Butterfly exchange

The NAS Parallel Benchmark suite [14] contains a Conjugate Gradients (CG) implementation that spells out its all-reduce operations as a *butterfly exchange*.

```
// cgb.f
do i = 1, l2npcols
    call mpi_irecv( d,
    >           1,
    >           dp_type,
    >           reduce_exch_proc(i),
    >           i,
```

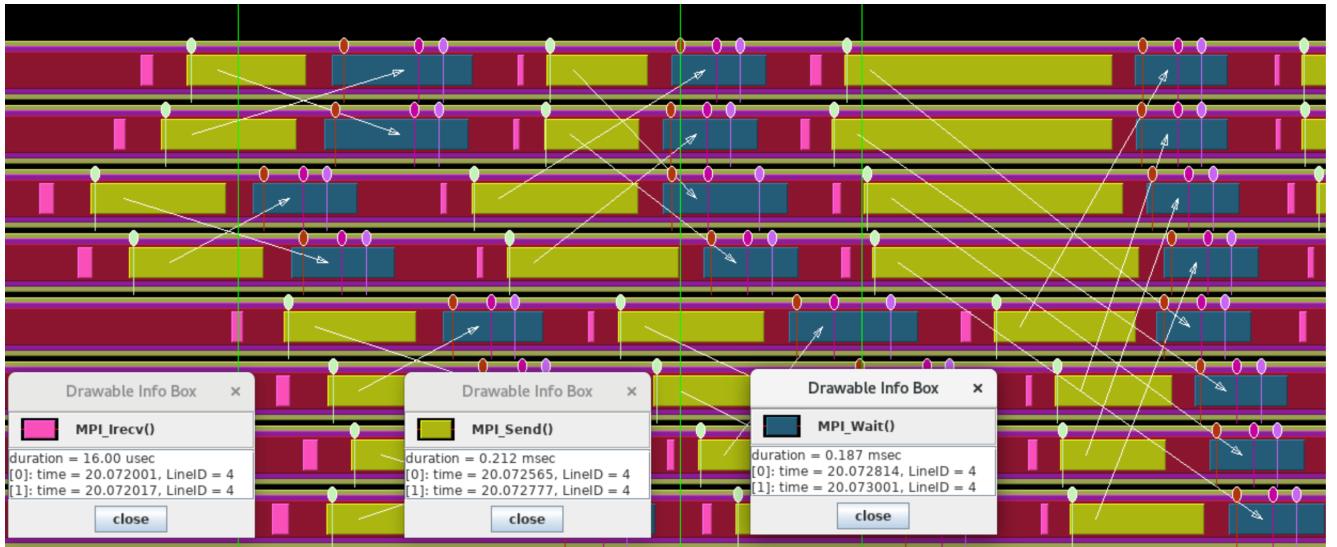


Figure 40.3: Trace of a butterfly exchange

```

>                               mpi_comm_world,
>                               request,
>                               ierr )
call mpi_send(   sum,
>
>                               1,
>                               dp_type,
>                               reduce_exch_proc(i),
>                               i,
>                               mpi_comm_world,
>                               ierr )

call mpi_wait( request, status, ierr )

sum = sum + d
enddo

```

We recognize this structure in the TAU trace: figure 40.3. Upon closer examination, we see how this particular algorithm induces a lot of wait time. Figures 40.5 and 40.6 show a whole cascade of processes waiting for each other.

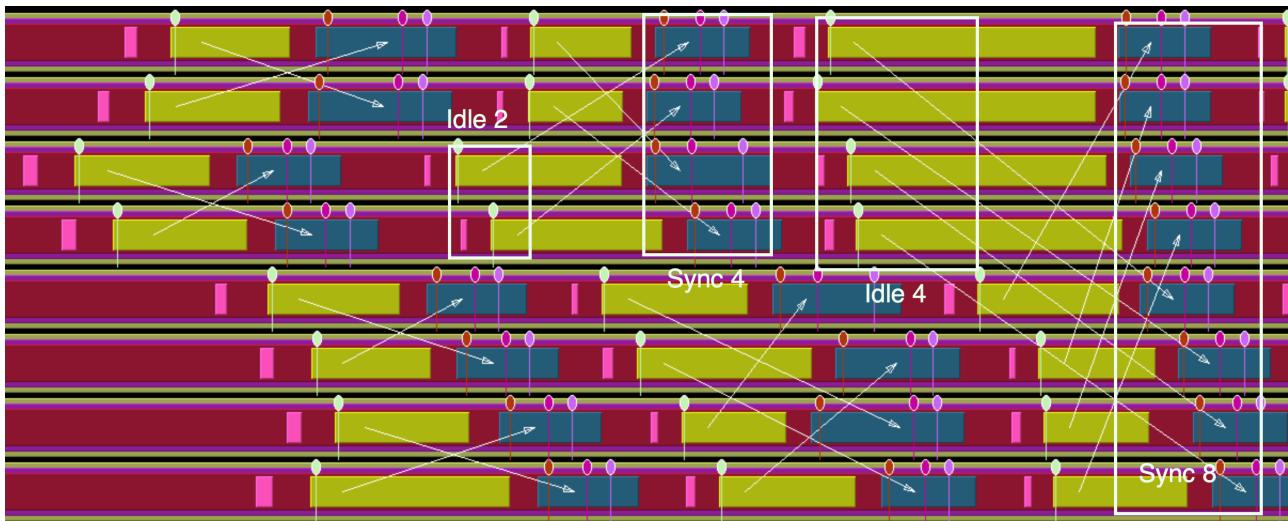


Figure 40.4: Trace of a butterfly exchange

## 40.6 SimGrid

Many readers of this book will have access to some sort of parallel machine so that they can run simulations, maybe even some realistic scaling studies. However, not many people will have access to more than one cluster type so that they can evaluate the influence of the *interconnect*. Even then, for didactic purposes one would often wish for interconnect types (fully connected, linear processor array) that are unlikely to be available.

In order to explore architectural issues pertaining to the network, we then resort to a simulation tool, *SimGrid*.

### **Installation**

**Compilation** You write plain MPI files, but compile them with the *SimGrid compiler* `smpicc`.

**Running** SimGrid has its own version of `mpirun`: `smpirun`. You need to supply this with options:

- `-np 123456` for the number of (virtual) processors;
- `-hostfile simgridhostfile` which lists the names of these processors. You can basically make these up, but are defined in:
- `-platform arch.xml` which defines the connectivity between the processors.

For instance, with a hostfile of 8 hosts, a linearly connected network would be defined as:

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid/simgrid.c...
```

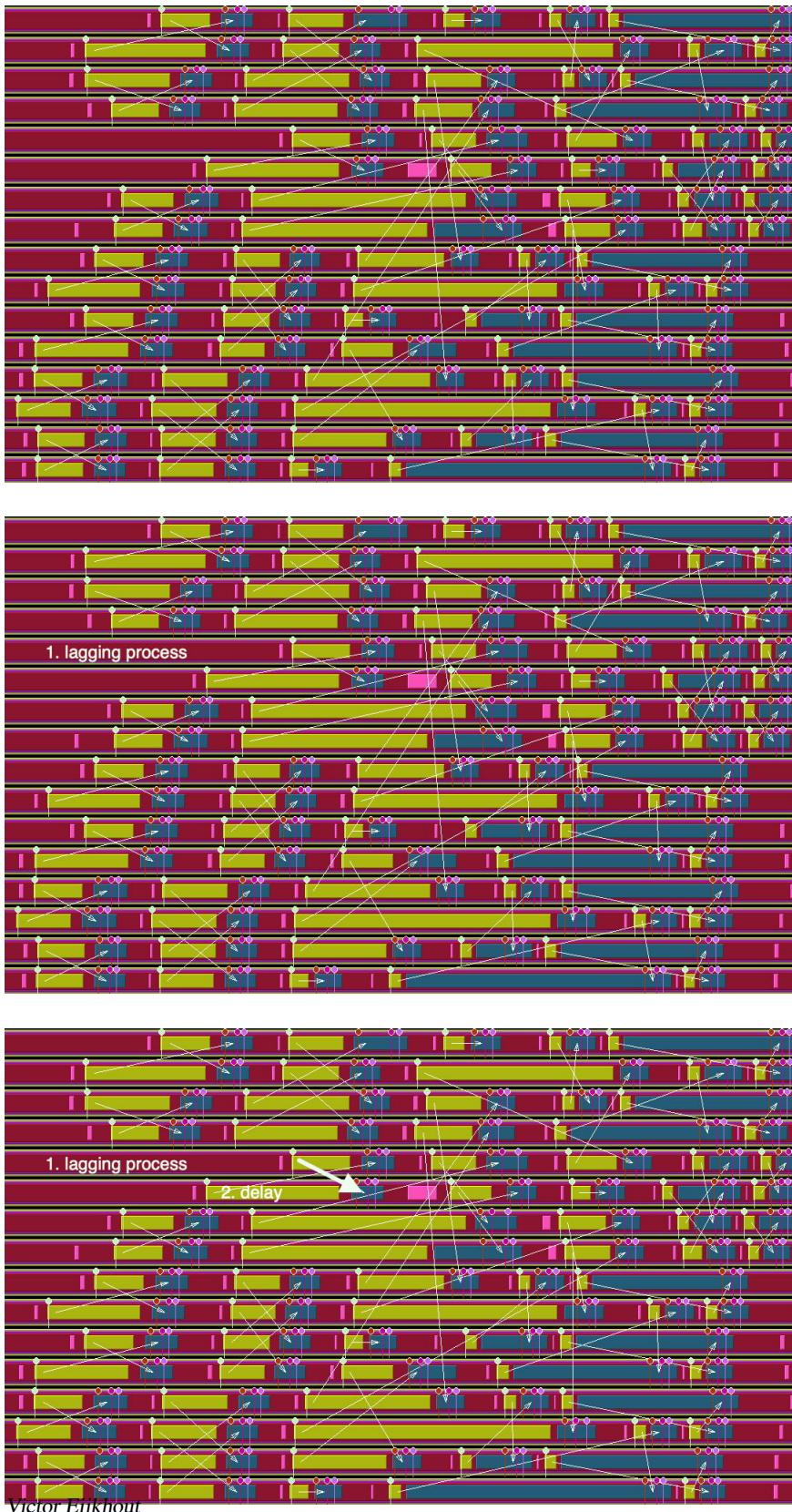


Figure 40.5: Four stages of processes waiting caused by a single lagging process

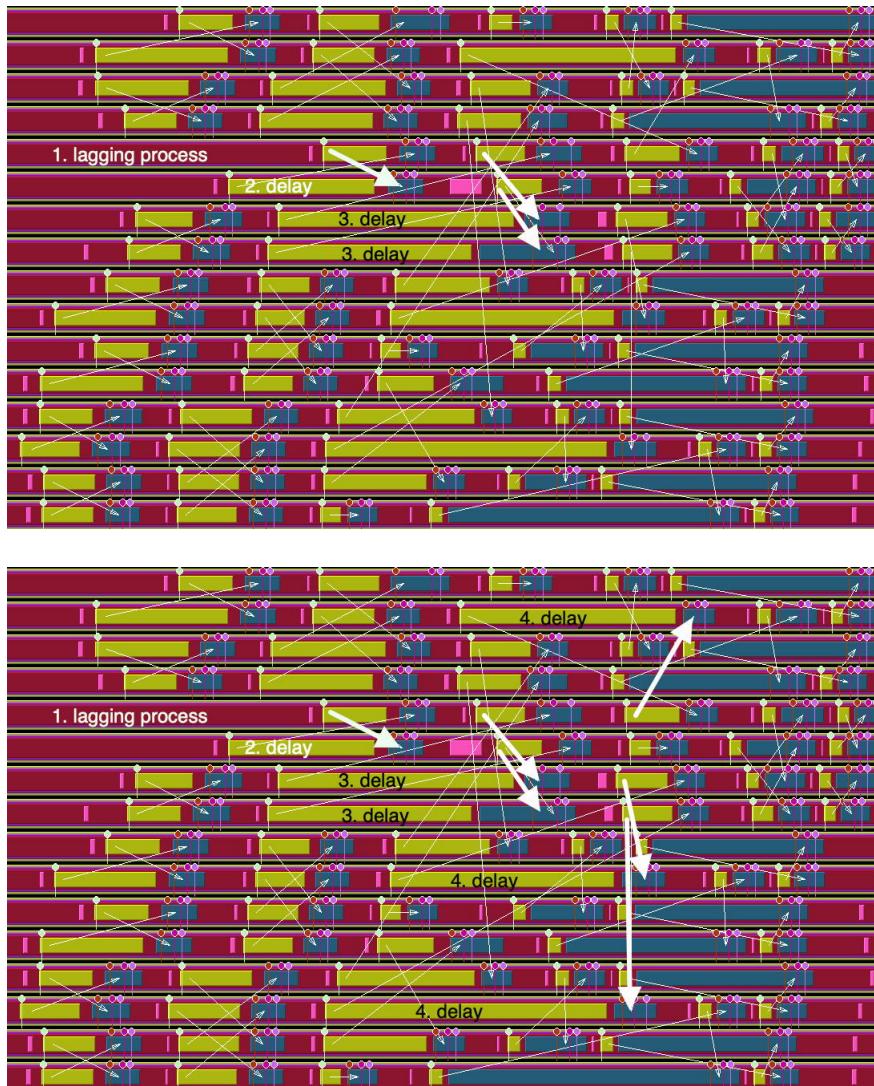


Figure 40.6: Four stages of processes waiting caused by a single lagging process

```
<platform version="4">

<zone id="first zone" routing="Floyd">
    <!-- the resources -->
    <host id="host1" speed="1Mf"/>
    <host id="host2" speed="1Mf"/>
    <host id="host3" speed="1Mf"/>
    <host id="host4" speed="1Mf"/>
    <host id="host5" speed="1Mf"/>
    <host id="host6" speed="1Mf"/>
    <host id="host7" speed="1Mf"/>
    <host id="host8" speed="1Mf"/>
    <link id="link1" bandwidth="125MBps" latency="100us"/>
    <!-- the routing: specify how the hosts are interconnected -->
    <route src="host1" dst="host2"><link_ctn id="link1"/></route>
    <route src="host2" dst="host3"><link_ctn id="link1"/></route>
    <route src="host3" dst="host4"><link_ctn id="link1"/></route>
    <route src="host4" dst="host5"><link_ctn id="link1"/></route>
    <route src="host5" dst="host6"><link_ctn id="link1"/></route>
    <route src="host6" dst="host7"><link_ctn id="link1"/></route>
    <route src="host7" dst="host8"><link_ctn id="link1"/></route>
</zone>

</platform>
```

(such files are easily generated with a shell script).

The `Floyd` designation of the routing means that any route using the transitive closure of the paths given can be used. It is also possible to use `routing="Full"` which requires full specification of all pairs that can communicate.

---

## 40.7 Batch systems

Supercomputer *clusters* can have a large number of *nodes*, but not enough to let all their users run simultaneously, and at the scale that they want. Therefore, users are asked to submit *jobs*, which may start executing immediately, or may have to wait until resources are available.

The decision when to run a job, and what resources to give it, is not done by a human operator, but by software called a *batch system*. (The *Stampede* cluster at *TACC* ran close to 10 million jobs over its lifetime, which corresponds to starting a job every 20 seconds.)

This tutorial will cover the basics of such systems, and in particular Simple Linux Utility for Resource Management (SLURM).

### 40.7.1 Cluster structure

A supercomputer cluster usually has two types of nodes:

- *login nodes*, and
- *compute nodes*.

When you make an *ssh connection* to a cluster, you are connecting to a login node. The number of login nodes is small, typically less than half a dozen.

**Exercise 40.1.** Connect to your favourite cluster. How many people are on that login node? If you disconnect and reconnect, do you find yourself on the same login node?

Compute nodes are where your jobs are run. Different clusters have different structures here:

- Compute nodes can be shared between users, or they can be assigned exclusively.
  - Sharing makes sense if user jobs have less parallelism than the core count of a node.
  - ... on the other hand, it means that users sharing a node can interfere with each other's jobs, with one job using up memory or bandwidth that the other job needs.
  - With exclusive nodes, a job has access to all the memory and all the bandwidth of that node.
- Clusters can be homogeneous, having the same processor type on each compute node, or they can have more than one processor type. For instance, the TACC *Stampede2* cluster has *Intel KNL* and *Intel Skylake* nodes.
- Often, clusters have a number of 'large memory' nodes, on the order of a Terabyte of memory or more. Because of the cost of such hardware, there is usually only a small number of these nodes.

### 40.7.2 Queues

Jobs often cannot start immediately, because not enough resources are available, or because other jobs may have higher priority (see section 40.7.7). It is thus typical for a job to be put on a *queue*, scheduled, and started, by a batch system such as SLURM.

Batch systems do not put all jobs in one big pool: jobs are submitted to any of a number of queues, that are all scheduled separately.

Queues can differ in the following ways:

- If a cluster has different processor types, those are typically in different queues. Also, there may be separate queues for the nodes that have a Graphics Processing Unit (GPU) attached. Having multiple queues means you have to decide what processor type you want your job to run on, even if your executable is binary compatible with all of them.
- There can be ‘development’ queues, which have restrictive limits on runtime and node count, but where jobs typically start faster.
- Some clusters have ‘premium’ queues, which have a higher charge rate, but offer higher priority.
- ‘Large memory nodes’ are typically also in a queue of their own.
- There can be further queues for jobs with large resource demands, such as large core counts, or longer-than-normal runtimes.

For slurm, the `sinfo` command can tell you much about the queues.

```
# what queues are there?  
sinfo -o "%P"  
# what queues are there, and what is their status?  
sinfo -o "%20P %.5a"
```

**Exercise 40.2.** Enter these commands. How many queues are there? Are they all operational at the moment?

#### 40.7.2.1 Queue limits

Queues have limits on

- the runtime of a job;
- the node count of a job; or
- how many jobs a user can have in that queue.

### 40.7.3 Job running

There are two main ways of starting a job on a cluster that is managed by slurm. You can start a program run synchronously with `srun`, but this may hang until resources are available. In this section, therefore, we focus on asynchronously executing your program by submitting a job with `sbatch`.

#### 40.7.3.1 The job submission cycle

In order to run a *batch job*, you need to write a *job script*, or *batch script*. This script describes what program you will run, where its inputs and outputs are located, how many processes it can use, and how long it will run.

In its simplest form, you submit your script without further parameters:

```
sbatch yourscript
```

All options regarding the job run are contained in the script file, as we will now discuss.

As a result of your job submission you get a job id. After submission you can query your job with `squeue`:

---

```
squeue -j 123456
```

or query all your jobs:

```
squeue -u yourname
```

The `squeue` command reports various aspects of your job, such as its status (typically pending or running); and if it is running, the queue (or ‘partition’) where it runs, its elapsed time, and the actual nodes where it runs.

```
squeue -j 5807991
      JOBID      PARTITION      NAME      USER ST      TIME  NODES NODELIST(REASON)
      5807991 development packingt eijkhout    R      0:04          2 c456-[012,034]
```

If you discover errors in your script after submitting it, including when it has started running, you can cancel your job with `scancel`:

```
scancel 1234567
```

#### 40.7.4 The script file

A job script looks like an executable shell script:

- It has an ‘interpreter’ line such as

```
#!/bin/bash
```

at the top, and

- it contains ordinary unix commands, including
- the (parallel) startup of your program:

```
# sequential program:
./yourprogram youroptions
# parallel program, general:
mpexec -n 123 parallelprogram options
# parallel program, TACC:
ibrun parallelprogram options
```

- ... and then it has many options specifying the parallel run.

##### 40.7.4.1 *sbatch* options

In addition to the regular unix commands and the interpreter line, your script has a number of SLURM directives, each starting with `#SBATCH`. (This makes them comments to the shell interpreter, so a batch script is actually a legal shell script.)

Directives have the form

```
#SBATCH -option value
```

Common options are:

- `-J`: the jobname. This will be displayed when you call `squeue`.
- `-o`: name of the output file. This will contain all the stdout output of the script.
- `-e`: name of the error file. This will contain all the stderr output of the script, as well as slurm error messages.  
It can be a good idea to make the output and error file unique per job. To this purpose, the macro `%j` is available, which at execution time expands to the job number. You will then get an output file with a name such as `my.job.o2384737`.
- `-p`: the *partition* or queue. See above.
- `-t hh:mm:ss`: the maximum running time. If your job exceeds this, it will get *cancelled*. Two considerations:
  1. You can not specify a duration here that is longer than the queue limit.
  2. The shorter your job, the more likely it is to get scheduled sooner rather than later.
- `-A`: the name of the account to which your job should be billed.
- `--mail-user=you@where` Slurm can notify you when a job starts or ends. You may for instance want to connect to a job when it starts (to run `top`), or inspect the results when it's done, but not sit and stare at your terminal all day. The action of which you want to be notified is specified with (among others) `--mail-type=begin/end/fail/all`
- `--dependency=after:123467` indicates that this job is to start after jobs 1234567 finished. Use `afterok` to start only if that job successfully finished. (See [https://cvw.cac.cornell.edu/slurm/submission\\_depend](https://cvw.cac.cornell.edu/slurm/submission_depend) for more options.)
- `--nodelist` allows you to specify specific nodes. This can be good for getting reproducible timings, but it will probably increase your wait time in the queue.
- `--array=0-30` is a specification for ‘array jobs’: a task that needs to be executed for a range of parameter values.  
*TACC note.* Array jobs are not supported at TACC; use a launcher instead; section 40.7.5.3.
- `--mem=10000` specifies the desired amount of memory per node. Default units are megabytes, but can be explicitly indicated with K/M/G/T.  
*TACC note.* This option can not be used to request arbitrary memory: jobs always have access to all available physical memory, and use of shared memory is not allowed.

See <https://slurm.schedmd.com/sbatch.html> for a full list.

**Exercise 40.3.** Write a script that executes the `date` command twice, with a `sleep` in between. Submit the script and investigate the output.

#### 40.7.4.2 Environment

Your job script acts like any other shell script when it is executed. In particular, it inherits the calling *environment* with all its environment variables. Additionally, slurm defines a number of environment variables, such as the job ID, the hostlist, and the node and process count.

---

## 40.7.5 Parallelism handling

We discuss parallelism options separately.

### 40.7.5.1 MPI jobs

On most clusters there is a structure with compute nodes, that contain one or more multi-core processors. Thus, you want to specify the node and core count. For this, there are options `-N` and `-n` respectively.

```
#SBATCH -N 4                      # Total number of nodes  
#SBATCH -n 4                      # Total number of mpi tasks
```

It would be possible to specify only the node count or the core count, but that takes away flexibility:

- If a node has 40 cores, but your program stops scaling at 10 MPI ranks, you would use:  

```
#SBATCH -N 1  
#SBATCH -n 10
```
- If your processes use a large amount of memory, you may want to leave some cores unused. On a 40-core node you would either use  

```
#SBATCH -N 2  
#SBATCH -n 40
```

or

```
#SBATCH -N 1  
#SBATCH -n 20
```

Rather than specifying a total core count, you can also specify the core count per node with `--ntasks-per-node`.

**Exercise 40.4.** Go through the above examples and replace the `-n` option by an equivalent `--ntasks-per-node` values.

*Python note.* MPI programs, except that instead of an executable name you specify the python executable and the script name:

```
ibrun python3 mympi4py.py
```

### 40.7.5.2 Threaded jobs

The above discussion was mostly of relevance to MPI programs. Some other cases:

- For pure-OpenMP programs you need only one node, so the `-N` value is 1. Maybe surprisingly, the `-n` value is also 1, since only one process needs to be created: OpenMP uses thread-level parallelism, which is specified through the `OMP_NUM_THREADS` environment variable.
- A similar story holds for the *Matlab parallel computing toolbox* (note: note the distributed computing toolbox), and the *Python multiprocessing* module.

**Exercise 40.5.** What happens if you specify an `-n` value greater than 1 for a pure-OpenMP program?

For *hybrid computing* MPI-OpenMP programs, you use a combination of slurm options and environment variables, such that, for instance, the product of the `--tasks-per-node` and `OMP_NUM_THREADS` is less than the core count of the node.

#### 40.7.5.3 Parameter sweeps / ensembles / massively parallel

So far we have focused on jobs where a single parallel executable is scheduled. However, there are use cases where you want to run a sequential (or very modestly parallel) executable for a large number of inputs. This is called variously a *parameter sweep* or an *ensemble*.

Slurm can support this natively with *array jobs*.

*TACC note.* TACC clusters do not support array jobs. Instead, use the `launcher` or `pylauncher` modules.

#### 40.7.6 Job running

When your job is running, its status is reported as `R` by `squeue`. That command also reports which nodes are allocated to it.

```
squeue -j 5807991
      JOBID      PARTITION      NAME      USER ST      TIME   NODES NODELIST(RE
      5807991 development packingt eijkhout R      0:04      2 c456-[012,0
```

You can then `ssh` into the compute nodes of your job; normally, compute nodes are off-limits. This is useful if you want to run `top` to see how your processes are doing.

#### 40.7.7 Scheduling strategies

Such a system looks at resource availability and the user's priority to determine when a job can be run.

Of course, if a user is requesting a large number of nodes, it may never happen that that many become available simultaneously, so the batch system will force the availability. It does so by determining a time when that job is set to run, and then let nodes go *idle* so that they are available at that time.

An interesting side effect of this is that, right before the really large job starts, a 'fairly' large job can be run, if it only has a short running time. This is known as *backfill*, and it may cause jobs to be run earlier than their priority would warrant.

#### 40.7.8 File systems

File systems come in different types:

- They can be backed-up or not;
- they can be shared or not; and
- they can be permanent or purged.

---

On many clusters each node has a local disc, either spinning or a *RAM disc*. This is usually limited in size, and should only be used for temporary files during the job run.

Most of the file system lives on discs that are part of *RAID arrays*. These discs have a large amount of redundancy to make them fault-tolerant, and in aggregate they form a *shared file system*: one unified file system that is accessible from any node and where files can take on any size, or at least much larger than any individual disc in the system.

*TACC note.* The HOME file system is limited in size, but is both permanent and backed up. Here you put scripts and sources.

The WORK file system is permanent but not backed up. Here you can store output of your simulations. However, currently the work file system can not immediately sustain the output of a large parallel job.

The SCRATCH file system is purged, but it has the most bandwidth for accepting program output. This is where you would write your data. After post-processing, you can then store on the work file system, or write to tape.

**Exercise 40.6.** If you install software with *cmake*, you typically have

1. a script with all your *cmake* options;
2. the sources,
3. the installed header and binary files
4. temporary object files and such.

How would you organize these entities over your available file systems?

## 40.7.9 Examples

Very sketchy section.

### 40.7.9.1 Job dependencies

```
JOB=`sbatch my_batchfile.sh | egrep -o -e "\b[0-9]+\$" `

#!/bin/sh

# Launch first job
JOB=`sbatch job.sh | egrep -o -e "\b[0-9]+\$" `

# Launch a job that should run if the first is successful
sbatch --dependency=afterok:${JOB} after_success.sh

# Launch a job that should run if the first job is unsuccessful
sbatch --dependency=afternotok:${JOB} after_fail.sh
```

#### 40.7.9.2 Multiple runs in one script

```
ibrun stuff &
sleep 10
for h in hostlist ; do
    ssh $h "top"
done
wait
```

#### 40.7.10 Review questions

For all true/false questions, if you answer False, what is the right answer and why?

**Exercise 40.7.** T/F? When you submit a job, it starts running immediately once sufficient resources are available.

**Exercise 40.8.** T/F? If you submit the following script:

```
#!/bin/bash
#SBATCH -N 10
#SBATCH -n 10
echo "hello world"
```

you get 10 lines of ‘hello world’ in your output.

**Exercise 40.9.** T/F? If you submit the following script:

```
#!/bin/bash
#SBATCH -N 10
#SBATCH -n 10
hostname
```

you get the hostname of the login node from which your job was submitted.

**Exercise 40.10.** Which of these are shared with other users when your job is running:

- Memory;
- CPU;
- Disc space?

**Exercise 40.11.** What is the command for querying the status of your job?

- sinfo
- squeue
- sacct

**Exercise 40.12.** On 4 nodes with 40 cores each, what’s the largest program run, measured in

- MPI ranks;
- OpenMP threads?



**PART VII**

**PROJECTS, INDEX**

# Chapter 41

## Class projects

### 41.1 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.

**Structure of your writeup** Most of the projects in this book use a scientific question to allow you to prove your coding skills. That does not mean that turning in the code is sufficient, nor code plus sample output. Turn in a writeup in pdf form that was generated from a text processing program such (preferably)  $\text{\LaTeX}$  (for a tutorial, see HPSC-??).

Your writeup should have:

- Foremost, a short description of the purpose of your project and your results;
- An explanation of your algorithms or solution strategy;
- Relevant fragments of your code;
- A scientific discussion of what you observed,
- Any code-related observations.
- If applicable: graphs, both of application quantities and performance issues. (For parallel runs possibly TAU plots; see 40.5).

*Observe, measure, hypothesize, deduce* Your project may be 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 observe 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.

**Including code** If you include code samples in your writeup, make sure they look good. For starters, use a mono-spaced font. In L<sup>A</sup>T<sub>E</sub>X, you can use the `verbatim` environment or the `verbatiminput` command. In that section option the source is included automatically, rather than cut and pasted. This is to be preferred, since your writeup will stay current after you edit the source file.

Including whole source files makes for a long and boring writeup. The code samples in this book were generated as follows. In the source files, the relevant snippet was marked as

```
... boring stuff
//snippet samplex
    .. interesting! ..
//snippet end
... more boring stuff
```

The files were then processed with the following command line (actually, included in a makefile, which requires doubling the dollar signs):

```
for f in *.{c,cxx,h} ; do
    cat $x | awk 'BEGIN {f=0}
                    /snippet end/ {f=0}
                    f==1 {print $0 > file}
                    /snippet/ && !/end/ {f=1; file=$2 }
        '
done
```

which gives (in this example) a file `samplex`. Other solutions are of course possible.

**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}

|| for (int i=0; i<N; i++)
||     s += 1;
```

**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 sizes, be aware that an average processor can do a billion operations per second: you need

to make your problem large enough for the timings to rise above the level of random variations and startup phenomena.

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.

*Reporting scaling* If you do a scaling analysis, a graph reporting runtimes should not have a linear time axis: a logarithmic graph is much easier to read. A speedup graph can also be informative.

Some algorithms are mathematically equivalent in their sequential and parallel versions. Others, such as iterative processes, can take more operations in parallel than sequentially, for instance because the number of iterations goes up. In this case, report both the speedup of a single iteration, and the total improvement of running the full algorithm in parallel.

*Repository organization* If you submit your work through a repository, make sure you organize your submissions in subdirectories, and that you give a clear name to all files. Object files and binaries should not be in a repository since they are dependent on hardware and things like compilers.

## 41.2 Warmup Exercises

We start with some simple exercises.

### 41.2.1 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!

### 41.2.2 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:

```
// time_max.cxx
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 41.1.** Change this program to compute the average jitter by changing the reduction operator.

**Exercise 41.2.** 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 41.3.** 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?

### 41.2.3 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);
}
```

#### 41.2.3.1 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 4.2.2.

```
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 12.8 for the explanation why.)

**Exercise 41.4.** 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 41.5.** First write this code more elegantly by using MPI\_PROC\_NULL.

#### 41.2.3.2 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 41.6.** 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.

#### 41.2.3.3 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 41.7.** Implement a fully parallel version by using `MPI_Isend` and `MPI_Irecv`.

#### 41.2.3.4 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 41.8.** 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 41.9.** 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?

### 41.3 Mandelbrot set

If you've never heard the name *Mandelbrot set*, you probably recognize the picture; figure 41.1 Its formal

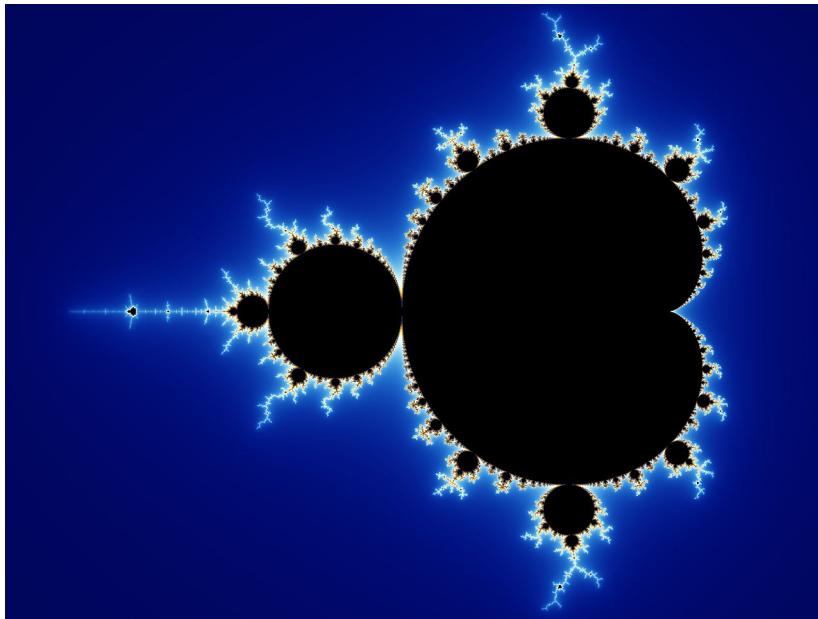


Figure 41.1: The Mandelbrot set

definition 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.

#### 41.3.1 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.

### 41.3.2 Tools

The driver part of the Mandelbrot program is simple. There is a circle object that can generate coordinates

```
// mandel.h
class circle {
public :
    circle(int pxls,int bound,int bs);
    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 41.10.** Explain why this solution is very inefficient. Make a trace of its execution that bears this out.

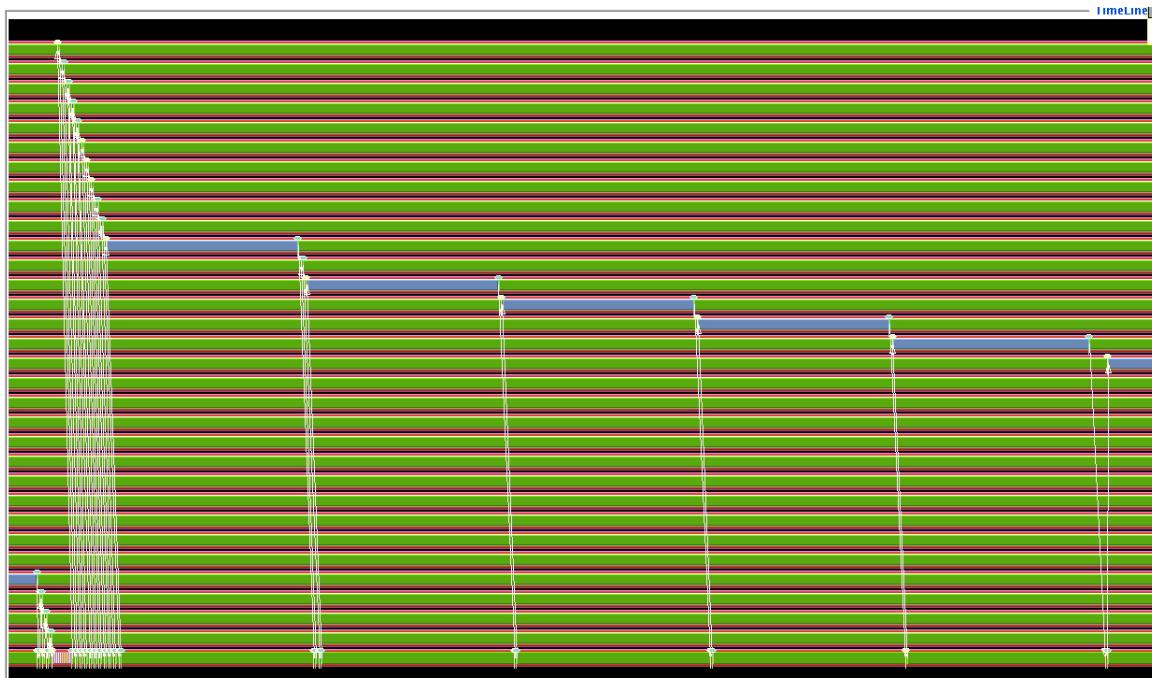


Figure 41.2: Trace of a serial Mandelbrot calculation

### 41.3.3 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 41.11.** 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.

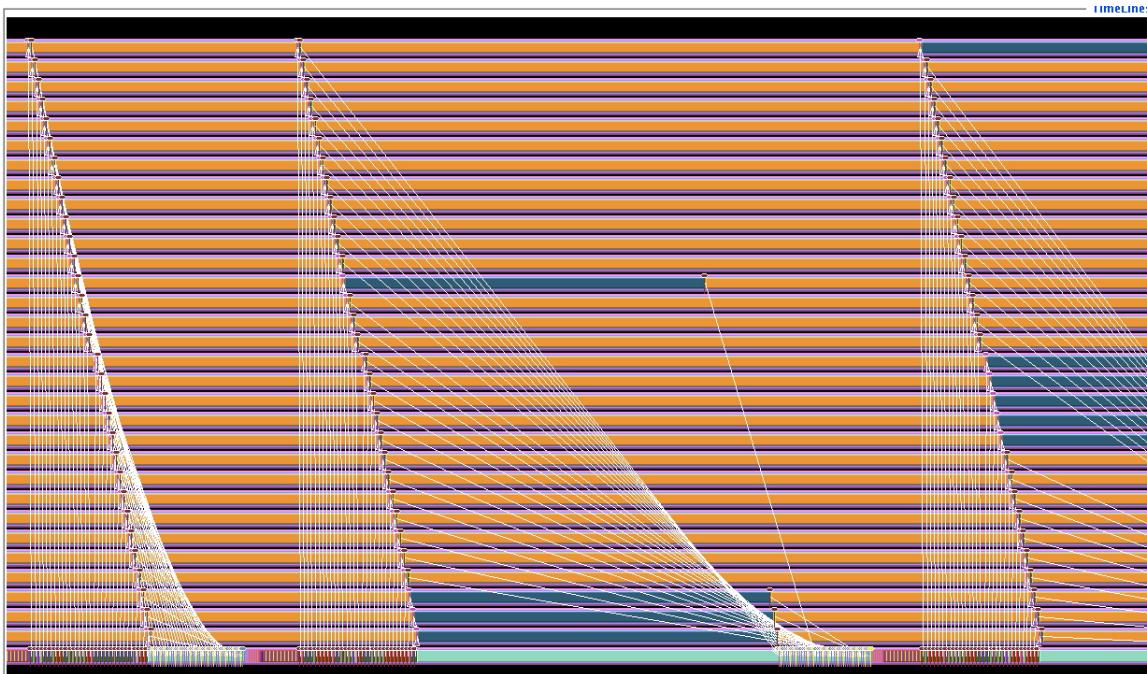


Figure 41.3: Trace of a bulk Mandelbrot calculation

#### 41.3.4 Collective task scheduling

Another implementation of the bulk scheduling of the previous section would be through using collectives.

**Exercise 41.12.** 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?

#### 41.3.5 Asynchronous task scheduling

At the start of section 41.3.3 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 41.13.** Code a fully dynamic solution that uses MPI\_Probe or MPI\_Waitany.

Make an execution trace and report on the total running time.

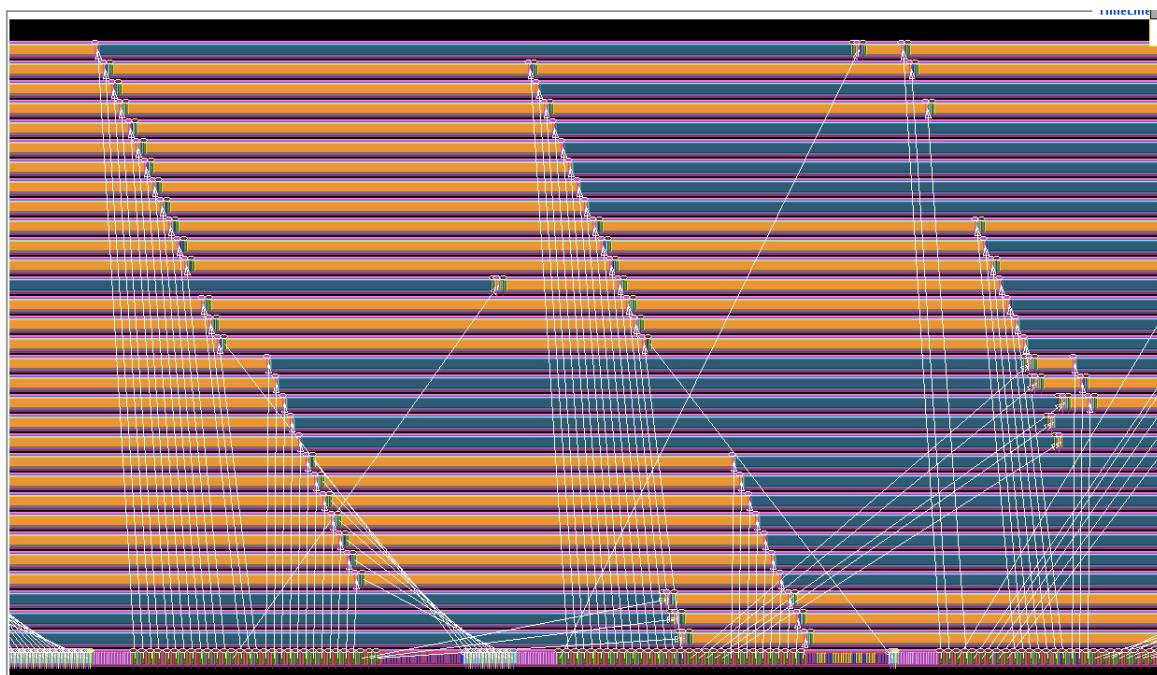


Figure 41.4: Trace of an asynchronous Mandelbrot calculation

#### 41.3.6 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.

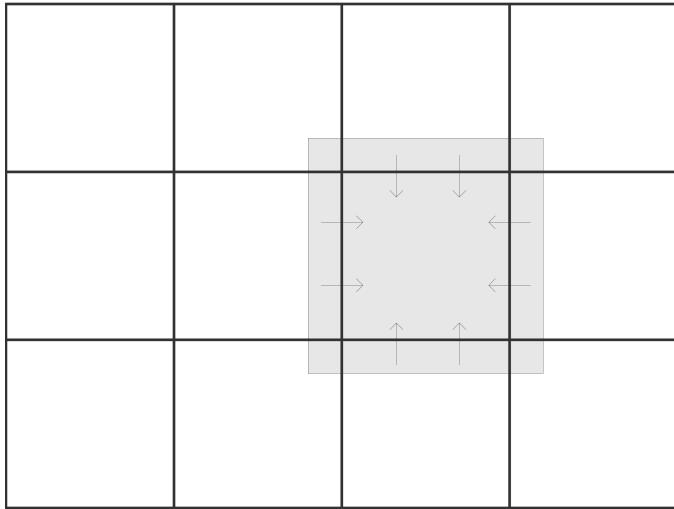


Figure 41.5: A grid divided over processors, with the ‘ghost’ region indicated

## 41.4 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`.

### 41.4.1 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

$$g'_{ij} = 1/4 \cdot (g_{i+1,j} + g_{i-1,j} + g_{i,j+1} + g_{i,j-1}). \quad (41.1)$$

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 HPSC-??.) 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 (41.1) 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 41.5. 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.

#### 41.4.2 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.

```
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();
```

This routine does the sequenced printing that you implemented in exercise ??.

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 HPSC-?? and HPSC-??.) Note how the `INDEX` macro is used to index in a `ilength × 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.;
    }
}
```

## 41.5 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 HPSC-??.

### 41.5.1 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.

### 41.5.2 Shared memory approaches

### 41.5.3 Distributed memory approaches

## Chapter 42

### Bibliography, index, and list of acronyms

#### 42.1 Bibliography

- [1] 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.
- [2] 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.
- [3] Lisandro Dalcin. MPI for Python, homepage. <https://mpi4py.bitbucket.io/>.
- [4] Victor Eijkhout. Performance of MPI sends of non-contiguous data. *arXiv e-prints*, page arXiv:1809.10778, Sep 2018.
- [5] Eijkhout, Victor with Robert van de Geijn and Edmond Chow. *Introduction to High Performance Scientific Computing*. lulu.com, 2011. <http://www.tacc.utexas.edu/~eijkhout/istc/istc.html>.
- [6] 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.
- [7] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1994.
- [8] Torsten Hoefer, 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.
- [9] Torsten Hoefer, Christian Siebert, and Andrew Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. *SIGPLAN Not.*, 45(5):159–168, January 2010.
- [10] INRIA. SimGrid homepage. <http://simgrid.gforge.inria.fr/>.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] NASA Advanced Supercomputing Division. NAS parallel benchmarks. <https://www.nas.nasa.gov/publications/npb.html>.
  - [15] Jeff Squyres. Mpi-request-free is evil. Cisco Blogs, January 2013. [https://blogs.cisco.com/performance/mpi\\_request\\_free\\_is\\_evil](https://blogs.cisco.com/performance/mpi_request_free_is_evil).
  - [16] 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.

---

## 42.2 List of acronyms

<b>API</b>	Application Programmer Interface	<b>NUMA</b>	Non-Uniform Memory Access
<b>AMG</b>	Algebraic MultiGrid	<b>OO</b>	Object-Oriented
<b>AVX</b>	Advanced Vector Extensions	<b>OS</b>	Operating System
<b>BLAS</b>	Basic Linear Algebra Subprograms	<b>PGAS</b>	Partitioned Global Address Space
<b>BSP</b>	Bulk Synchronous Parallel	<b>PDE</b>	Partial Differential Equation
<b>CAF</b>	Co-array Fortran	<b>PRAM</b>	Parallel Random Access Machine
<b>CRS</b>	Compressed Row Storage	<b>RDMA</b>	Remote Direct Memory Access
<b>CG</b>	Conjugate Gradients	<b>RMA</b>	Remote Memory Access
<b>CUDA</b>	Compute-Unified Device Architecture	<b>SAN</b>	Storage Area Network
<b>DAG</b>	Directed Acyclic Graph	<b>SaaS</b>	Software as-a Service
<b>DPCPP</b>	Data Parallel C++	<b>SFC</b>	Space-Filling Curve
<b>DSP</b>	Digital Signal Processing	<b>SIMD</b>	Single Instruction Multiple Data
<b>FEM</b>	Finite Element Method	<b>SIMT</b>	Single Instruction Multiple Thread
<b>FPU</b>	Floating Point Unit	<b>SLURM</b>	Simple Linux Utility for Resource Management
<b>FFT</b>	Fast Fourier Transform	<b>SM</b>	Streaming Multiprocessor
<b>FSA</b>	Finite State Automaton	<b>SMP</b>	Symmetric Multi Processing
<b>GPU</b>	Graphics Processing Unit	<b>SOR</b>	Successive Over-Relaxation
<b>HPC</b>	High-Performance Computing	<b>SP</b>	Streaming Processor
<b>HPF</b>	High Performance Fortran	<b>SPMD</b>	Single Program Multiple Data
<b>ICV</b>	Internal Control Variable	<b>SPD</b>	symmetric positive definite
<b>LAPACK</b>	Linear Algebra Package	<b>SSE</b>	SIMD Streaming Extensions
<b>MIC</b>	Many Integrated Cores	<b>TACC</b>	Texas Advanced Computing Center
<b>MPMD</b>	Multiple Program Multiple Data	<b>TLB</b>	Translation Look-aside Buffer
<b>MIMD</b>	Multiple Instruction Multiple Data	<b>UMA</b>	Uniform Memory Access
<b>MPI</b>	Message Passing Interface	<b>UPC</b>	Unified Parallel C
<b>MPL</b>	Message Passing Layer	<b>WAN</b>	Wide Area Network
<b>MTA</b>	Multi-Threaded Architecture		
<b>NIC</b>	Network Interface Card		

## 42.3 General Index

- malloc\_debug, 546
- malloc\_test, 546
- active target synchronization, 278, 283
- address
  - physical, 349
  - virtual, 349
- adjacency
  - graph, 520
- affinity, 574
  - process and thread, 574–577
  - thread
    - on multi-socket nodes, 452
- alignment, 204
- all-to-all, 37
- allocate
  - and private/shared data, 421
- allreduce, 36
- argc, 26, 28
- argv, 26, 28
- array
  - static, 42
- atomic operation, 434, 460
  - file, 334
  - MPI, 293–298
  - OpenMP, 436
- backfill, 615
- bandwidth, 78
  - bisection, 83
- barrier, 368
  - implicit, 460
  - non-blocking, 75
- Basic Linear Algebra Subprograms (BLAS), 496
- batch
  - job, 15, 611
  - scheduler, 15
  - script, 611
  - system, 610
- Beowulf cluster, 14
- block row, 506
- Boolean satisfiability, 33
- boost, 17
- breakpoint, 598
- broadcast, 35
- btl\_openib\_eager\_limit, 121
- btl\_openib\_rndv\_eager\_limit, 121
- bucket brigade, 82, 122, 602
- buffer
  - MPI, in C, 41
  - MPI, in Fortran, 41
  - MPI, in MPL, 41
  - MPI, in Python, 41
  - receive, 118
- butterfly exchange, 604
- C
  - MPI bindigs, see MPI, C bindings
- C++
  - MPI bindings, see MPI, C++ bindings
  - standard library, 199
  - vector, 199
- C++ iterators
  - in OMP reduction, 431
- C99, 180
- c\_sizeof, 203
- cacheline, 428
- callback, 540
- cast, 41
- CC, 601
- Charmpp, 15
- chunk, 409
- chunk, 409
- client, 271
- cluster, 610
- cmake, 616
- Codimension, 560
- collectives, 35
  - neighbourhood, 340, 371
  - non-blocking, 72
    - cancelling, 377
- column-major storage, 191
- communication
  - asynchronous, 149–150

blocking, **118–123**  
    vs non-blocking, **370**  
buffered, **154**, **371**  
non-blocking, **129–142**  
one-sided, **278–308**  
one-sided, implementation of, **307–308**  
persistent, **150–154**, **371**  
synchronous, **149–150**  
two-sided, **159**

communicator, **29**, **246–259**  
    info object, **360**  
    inter, **254**, **255**, **268**  
    intra, **255**, **257**  
    object, **29**  
    peer, **255**  
    variable, **29**

compare-and-swap, **126**

compiler, **202**  
    optimization level, **593**

completion, **300**  
    local, **300**  
    remote, **300**

compute  
    node, **610**

configure.log, **491**

construct, **392**

contention, **83**

contiguous  
    data type, **184**

core, **23**, **387**

core dump, **593**

cosubscript, **561**

cpp, **390**

cpuinfo, **572**

Cray  
    MPI, **16**  
    T3E, **374**

critical section  
    flush at, **461**

critical section, **427**, **435**, **469**

curly braces, **391**

Dalcin  
    Lisandro, **18**

data dependency, **446**

data race, see race condition

datatype, **178–212**  
    big, **209–210**  
    derived, **184–324**  
    different on sender and receiver, **188**  
    elementary, **179–183**  
        in C, **179**  
        in Fortran, **180**  
        in Python, **182**  
    signature, **200**

datatypes  
    derived, **178**  
    elementary, **178**

date, **613**

ddd, **593**

DDT, **593**, **600**

deadlock, **75**, **114**, **119**, **129**, **370**, **372**, **599**

debug flag, **594**

debug\_mt, **366**

debugger, **593**

debugging, **593–600**  
    parallel, **599**

dense linear algebra, **251**

destructor, **249**

directive  
    end-of, **391**

directives, **391**, **391**  
    cpp, **399**

displacement unit, **304**

distributed array, **111**

distributed shared memory, **278**

doubling  
    recursive, see recursive doubling

dynamic mode, **388**

eager limit, **119–121**, **370**

Eclipse, **600**  
    PTP, **600**

edge  
    cuts, **520**  
    weight, **520**

ensemble, **615**

environment

of batch job, 613  
environment variables, 582  
epoch, 283  
    access, 284, 292  
    communication, 283  
    completion, 303  
    exposure, 284, 291  
    passive target, 298  
error return, 17  
ethernet, 16  
  
false sharing, 406, 428  
Fast Fourier Transform (FFT), 516, 518  
fat-tree, 574  
fence, 283  
fftw, 492, 516  
Fibonacci sequence, 438–440  
file  
    pointer  
        advance by write, 332  
        individual, 331  
    system  
        shared, 616  
file system  
    shared, 324  
first-touch, 456, 575  
five-point stencil, 75  
fork/join model, 387, 394, 445  
Fortran  
    1-based indexing, 136  
    90  
        bindings, 18  
    2008, 29  
        array syntax, 416  
        assumed-shape arrays in MPI, 365  
        fixed-form source, 391  
        forall loops, 416  
        Fortran90, 26  
        line length, 544  
        MPI bindings, see MPI, Fortran bindings  
        MPI equivalences of scalar types, 180  
        MPI issues, 364–365  
    Fortran2008, 286  
        MPI bindings, see MPI, Fortran2008 bindings  
Fortran77  
    PETSc interface, 487  
Fortran90  
    PETSc interface, 487  
Fortran90 types  
    in MPI, 364  
  
gather, 35  
Gauss-Jordan algorithm, 49  
Gaussian elimination, 532  
gcc  
    thread affinity, 457  
gdb, 593–600  
gemv, 512  
getrusage, 551  
ghost region, 634  
GNU, 593  
    gdb, see gdb  
Gram-Schmidt, 40  
graph  
    partitioning  
        packages, 520  
        topology, 340, 371  
        unweighted, 342  
graph topology, 574  
grid  
    Cartesian, 337, 516  
    periodic, 337  
    processor, 574  
group, 291  
group of  
    processors, 293  
  
halo, 634  
    update, 288  
halo region, 518  
handshake, 372  
hdf5, 324, 587  
heap, 394  
heat equation, 454  
histogram, 437  
hostname, 362  
hwloc, 572  
hwloc, 575

hybrid computing, 615  
hyperthreading, 575  
Hypre, 540  
hypre, 492, 538  
  pilut, 539

I/O  
  in OpenMP, 415  
`I_MPI_ASYNC_PROGRESS`, 366  
`I_MPI_ASYNC_PROGRESS_...`, 366  
`I_MPI_ASYNC_PROGRESS_THREADS`, 366  
`I_MPI_EAGER_THRESHOLD`, 121  
ibrun, 270, 490, 602  
idle, 615  
image, 560  
`image_index`, 561  
implicit barrier, 435  
  after single directive, 415  
indexed  
  data type, 184  
inner product, 40  
input redirection  
  shell, 373  
instrumentation, 601  
Intel, 371  
  compiler  
    thread affinity, 457  
  compiler suite, 572  
  Haswell, 575  
  Knight's Landing, 387, 464  
    thread placement, 455  
  KNL, 610  
  MPI, 16, 83, 366  
  mpi, 121, 270  
  Paragon, 366  
  SandyBridge, 575  
  Sandybridge, 387  
  Skylake, 610  
interconnect, 606  
Internal Control Variable (ICV), 468–470

job, 610  
  array, 615

cancel, 613  
job script, 611  
jumpshot, 602

KIND, 203  
`KMP_AFFINITY`, 457

Laplace equation, 509  
latency, 78  
  hiding, 139, 370, 502, 504  
latency hiding, 511  
launcher, 615  
lcobound, 561  
lexical scope, 418  
Linear Algebra Package (LAPACK), 496  
linked list, 442  
linker  
  weak symbol, 369  
listing, 621  
load balancing, 406  
load imbalance, 406  
local refinement, 76  
lock, 436, 436–438  
  flush at, 461  
  nested, 438  
login  
  node, 610  
Lonestar5, 575  
LU factorization, 408, 532  
Lustre, 587

`make.log`, 491  
malloc  
  and private/shared data, 421  
`malloc`, 394, 456  
manager-worker, 627  
manager-worker model, 140, 145  
Mandelbrot set, 33, 81, 424, 627  
matching, 372  
matching queue, 144  
Matlab  
  parallel computing toolbox, 614  
matrix  
  sparse, 72, 511

transposition, 252  
matrix-vector product, 532  
  dense, 53  
  sparse, 56  
Mellanox, 371  
memory  
  coherent, 303  
  shared, MPI, 280  
memory leak, 142  
message  
  collision, 371  
  count, 147  
  source, 118  
  status, 118, 128, 144  
    error, 147  
    source, 145  
  tag, 147  
  tag, 116, 582  
Message Passing Layer (MPL), 17  
messsage  
  target, 114  
MKL, 496  
mkl, 492  
ML, 540  
Monte Carlo codes, 32  
motherboard, 386  
move\_pages, 457  
MPI  
  1, 337, 343  
  3, 72, 195, 199, 293, 365  
    C++ bindings removed, 17  
    Fortran2008 interface, 17  
  C bindings, 17  
  C++ bindings, 17  
  constants, 374–375, 382–383  
    compile-time, 374, 382  
    link-time, 374, 382  
  datatype  
    extent, 204  
    size, 204  
    subarray, 205  
    vector, 204  
  Fortran bindings, 17–18

Fortran issues, see Fortran, MPI issues  
Fortran2008 bindings, 17–18  
I/O, 364, 587  
initialization, 26  
MPI-2, 360  
MPI-3, 209, 346  
MPI-4, 346  
Python bindings, 18  
Python issues, 376  
semantics, 371–372  
tools interface, 369–370  
version, 362  
MPI-4, 203  
mpi.h, 25  
mpi.h, 18  
MPI/O, 324–334  
mpi4py, 18, 31  
MPI\_COMM\_WORLD, 29  
mpi\_f08, 17, 18, 286  
mpich, 16  
mpicxx, 17  
mpieexec, 15, 25, 28, 245, 268  
  options, 16  
mpieexec, 25  
mpif.h, 25  
mpirun, 15, 16, 28  
  and environment variables, 582  
MPL, 17, 31  
  compiling and linking, 17  
mulpd, 464  
mulsd, 464  
multicore, 388  
Multiple Program Multiple Data (MPMD), 374  
mumps, 492  
MV2\_IBA\_EAGER\_THRESHOLD, 121  
mvapich, 582  
mvapich2, 121, 580  
nested parallelism, 399–401  
netcdf, 324  
network  
  card, 371  
  contention, 371  
  port

oversubscription, 371  
new, 394  
node, 23, 610  
    cluster, 386  
norm  
    one, 71  
np.frombuffer, 70  
num\_images, 561  
numactl, 457, 575  
numerical integration, 405  
Numpy, 182  
numpy, 18, 41, 183, 282, 376  
od, 324  
offloading  
    vs onloading, 371  
omp  
    barrier  
        implicit, 435  
    for  
        barrier behaviour, 435  
        reduction, 427–431  
        user-defined, 430–431  
OMP\_NUM\_THREADS, 582, 614  
onloading, see offloading, vs onloading  
opaque handle, 29, 41  
OpenMP, 346  
    accelerator support in, 472  
    co-processor support in, 472  
    compiling, 389–390  
    environment variables, 390, 394, 468–470  
    library routines, 394  
    library routines, 468–470  
    places, 452  
    running, 390  
    tasks, 442–449  
        data, 443–444  
        dependencies, 446–447  
        synchronization, 444–446  
    version 4, 448  
OpenMPI, 121  
operating system, 472  
operator, 67–71  
    predefined, 67  
    user-defined, 68  
option  
    prefix, 551  
origin, 278, 292  
overlapping computation and communication, see  
    latency, hiding  
owner computes, 113  
package, 572  
packing, 210  
page  
    small, 349  
    table, 349  
parallel  
    data, 457  
    embarrassingly, 457  
parallel region, 388, 397–401, 414  
    barrier at the end of, 435  
    dynamic scope, 400, 419  
    flush at, 461  
parallel regions  
    nested, 469  
parallel\_for, 567  
parameter sweep, 615  
paraprof, 602  
parasails, 538  
ParMetis, 371, 520  
partition, 613  
passive target synchronization, 279, 293, 298  
pbng, 457  
persistent communication, see communication, per-  
    sistent  
PETSc, 371  
    log files, 491  
Petsc  
    interoperability with BLAS, 557  
    interoperability with MPI, 557  
PETSC\_ARCH, 488  
PETSC\_DIR, 488  
PETSC\_OPTIONS, 551  
pin a thread, 575  
ping-pong, 113, 368, 562  
PMI\_RANK, 374  
point-to-point, 113

pointer  
    null, 52  
polling, 137, 366  
posting  
    of send/receive, 132  
pragma, see for list see under ‘omp’, 391  
preconditioner, 533, 536  
    block jacobi, 551  
    field-split, 516  
prefix operation, 64  
private variables, 394  
proc\_bind, 398  
process, 23  
processes status of, 373  
producer-consumer, 459  
PROFILEDIR, 601  
progress  
    asynchronous, 139, 140, 366  
purify, 597  
PVM, 15, 268  
pylauncher, 615  
Python  
    MPI bindigs, see MPI, Python bindings  
    multiprocessing, 614  
    PETSc interface, 488

queue, 610  
    SYCL, 566

race condition, 293, 427, 435, 460, 470, 561  
    in OpenMP, 460–461

radix sort, 56

RAID  
    array, 616

RAM  
    disc, 616

random number generation, 423

random number generator, 471

range, 567

Ranger, 574

rar, 305

raw, 305

ray tracing, 351

recursive doubling, 83

redirection, see shell, input redirection

reduction, 35

region of code, 392

register  
    SSE2, 464  
release\_mt, 366

residual, 533

Riemann sums, 405

RMA  
    active, 278  
    passive, 279

root, 42

root process, 35

sbatch, 611

scalable  
    in space, 81  
    in time, 81

scalapack, 542

scan, 37  
    exclusive, 66  
    inclusive, 65

scancel, 612

scanf, 547

scatter, 35

sched\_setaffinity, 457

schedule  
    clause, 409

scope  
    lexical, 388, 393  
    of variables, 388, 393

SEEK\_SET, 334

segfault, 546

segmentation fault, 595

segmented scan, 67

send  
    buffer, 114  
    buffered, 154

sentinel, 391, 398

sequential  
    semantics, 487

sequential consistency, 461

serialization, 122

server, 271

SetThreadAffinityMask, 457  
shared data, 388  
shared memory, see memory, shared  
shared variables, 394  
shell  
    matrix, 540  
shmem, 374  
silo, 324  
SimGrid, 81, 606–609  
    compiler, 606  
sinfo, 611  
Single Program Multiple Data (SPMD), 391, 397  
sizeof, 280, 365  
sleep, 613  
SLURM, 335  
smpicc, 606  
smpirun, 606  
socket, 23, 347, 386, 579  
sort  
    odd-even transposition, 126  
    radix, 56  
    swap, 126  
sorting  
    radix, 56  
sparse approximate inverse, 537, 538  
sparse matrix vector product, 67  
spin-lock, 469  
squeue, 611–613, 615  
srun, 611  
ssh, 15  
    connection, 610  
ssh, 615  
stack, 394, 469  
    overflow, 419  
    per thread, 419  
Stampede, 610  
    compute node, 575  
    largemem node, 575  
    node, 387  
Stampede2, 610  
standard deviation, 36  
start/affinity, 457  
status  
    of received message, 128  
stderr, 373  
stdout, 373  
stdout/err of, 373  
stencil, 516  
    nine-point, 338  
    star, 338  
storage association, 419, 422  
storage\_size, 203  
stride, 191  
stringstream, 398  
struct  
    data type, 184  
structured block, 392  
Sun  
    compiler, 457  
SUNW\_MP\_PROCBind, 457  
sycl::cpu\_selector, 566  
sycl::host\_selector, 566  
sycl::runtime\_error, 566  
symbol table, 593, 594  
sync, 561  
synchronization  
    in OpenMP, 434–440  
TACC, 610  
    portal, 488  
tacc\_affinity, 457, 575, 579  
TACC\_TAU\_DIR, 601  
tag, see message, tag  
    bound on value, 362  
target, 278, 292  
    active synchronization, see active target synchronization  
    passive synchronization, see passive target synchronization  
task  
    scheduler, 442  
    scheduling point, 447  
taskset, 457  
TAU, 601–605  
TAU\_PROFILE, 601  
tau\_timecorrect, 602  
TAU\_TRACE, 601

this\_image, 561  
thread  
    affinity, 452–456  
    initial, 398  
    master, 398  
    migrating a, 455  
    primary, 398  
    private data, 422  
thread-safe, 470–471  
threads, 387  
    hardware, 388, 575  
    master, 388  
    team of, 387, 398  
time slicing, 23, 388  
time-slicing, 268  
timing  
    MPI, 367–368  
top, 613, 615  
topology  
    virtual, 337  
TotalView, 593, 600  
TRACEDIR, 601  
transpose, 74–75  
    and all-to-all, 55–56  
    data, 518  
    recursive, 252  
    through derived types, 208–209  
tree  
    traversal  
        post-order, 449  
        pre-order, 449  
tunnel  
    ssh, 372  
ucobound, 561  
ulimit, 419  
Unix  
    process, 419  
valarray, 457  
valgrind, 597–598  
vector  
    data type, 184  
    instructions, 463  
verbatim, 621  
virtual shared memory, 278  
VTune, 601  
wall clock, 367  
war, 305  
waw, 305  
weak symbol, see linker, weak symbol  
while loop, 442  
while loops, 411  
window, 278–284  
    consistency, 300  
    displacement, 285  
    displacement unit, 305  
    info object, 360  
    memory, see also memory model  
        model, 303  
        separate, 302, 303  
        unified, 303  
    memory allocation, 280–283  
    private, 303  
    public, 303  
work sharing, 388  
work sharing construct, 394, 414  
workshare  
    flush after, 461  
worksharing constructs  
    implied barriers at, 435  
wormhole routing, 83  
wraparound connections, 337  
XSEDE  
    portal, 488  
Zoltan, 371, 520

## 42.4 Index of MPI commands

0\_MPI\_OFFSET\_KIND, **333**  
access\_style, **360**  
accumulate\_ops, **305**  
accumulate\_ordering, **305**  
alloc\_shared\_noncontig, **349**  
  
cb\_block\_size, **360**  
cb\_buffer\_size, **360**  
cb\_nodes, **360**  
chunked, **360**  
chunked\_item, **360**  
chunked\_size, **360**  
collective\_buffering, **360**  
  
file\_perm, **360**  
  
io\_node\_list, **360**  
irequest\_pool, **136**  
  
KSPSolve, **534**  
  
MPI\_2DOUBLE\_PRECISION, **68**  
MPI\_2INT, **68**  
MPI\_2INTEGER, **68**  
MPI\_2REAL, **68**  
MPI\_Abort, **27**, **71**, **364**, **373**  
MPI\_Accumulate, **284**, **289**, **297**, **305**  
MPI\_ADDRESS\_KIND, **180**, **183**, **286**, **375**  
MPI\_AINT, **179**, **183**  
MPI\_Aint, **180**, **183**, **183**, **285**  
    in Fortran, **183**  
MPI\_Aint\_add, **183**  
MPI\_Aint\_diff, **183**  
MPI\_Allgather, **53**, **85**  
MPI\_Allgatherv, **61**  
MPI\_Alloc\_mem, **280**, **282**  
MPI\_Allreduce, **38**, **38**, **44**, **56**, **85**  
MPI\_Alltoall, **55**, **56**, **519**  
MPI\_Alltoallv, **56**, **61**, **64**  
MPI\_ANY\_SOURCE, **58**, **81**, **118**, **128**, **140**, **145**,  
    **149**, **278**, **372**, **375**, **376**  
MPI\_ANY\_TAG, **118**, **123**, **128**, **147**, **375**  
  
MPI\_APPNUM, **362**  
MPI\_ARGV\_NULL, **375**  
MPI\_ARGVS\_NULL, **375**  
MPI\_ASYNC\_PROTECTS\_NONBLOCKING, **375**  
MPI\_Attr\_get, **361**  
MPI\_Barrier, **60**, **334**, **368**  
MPI\_Bcast, **46**, **85**  
MPI\_BOTTOM, **183**, **306**, **374**, **375**, **382**  
MPI\_Bsend, **154**, **154**, **155**  
MPI\_Bsend\_init, **154**, **155**, **155**  
MPI\_BSEND\_OVERHEAD, **155**, **212**, **375**  
MPI\_Buffer\_attach, **155**  
MPI\_Buffer\_detach, **155**  
MPI\_BYTE, **179**, **184**  
MPI\_Cancel, **376**, **377**  
MPI\_CART, **337**  
MPI\_Cart\_coords, **338**  
MPI\_Cart\_create, **338**  
MPI\_Cart\_rank, **338**  
MPI\_CHAR, **179**  
MPI\_CHARACTER, **180**  
MPI\_Close\_port, **272**  
MPI\_Comm, **17**, **19**, **20**, **29**, **245**, **374**, **382**  
MPI\_Comm\_accept, **271**, **272**  
MPI\_Comm\_compare, **257**  
MPI\_Comm\_connect, **271**, **272**, **272**, **363**  
MPI\_Comm\_create, **247**, **253**  
MPI\_Comm\_create\_errhandler, **364**  
MPI\_Comm\_create\_group, **253**  
MPI\_Comm\_disconnect, **272**  
MPI\_Comm\_dup, **247**, **248**, **248**, **360**, **491**  
MPI\_Comm\_dup\_with\_info, **360**  
MPI\_Comm\_free, **247**, **249**  
MPI\_Comm\_get\_attr, **268**, **361**  
MPI\_Comm\_get\_errhandler, **364**  
MPI\_Comm\_get\_info, **360**  
MPI\_Comm\_get\_parent, **257**, **271**  
MPI\_Comm\_group, **253**, **257**  
MPI\_Comm\_join, **273**  
MPI\_COMM\_NULL, **245**, **247**  
MPI\_Comm\_rank, **30**, **30**, **31**, **250**, **257**, **342**

MPI\_Comm\_remote\_group, 257  
MPI\_Comm\_remote\_size, 257, 271  
MPI\_COMM\_SELF, 245, 497  
MPI\_Comm\_set\_errhandler, 363, 364  
MPI\_Comm\_set\_info, 360  
MPI\_Comm\_set\_name, 253  
MPI\_Comm\_size, 30, 30, 31, 114, 257, 342  
MPI\_Comm\_spawn, 247, 268, 365  
MPI\_Comm\_spawn\_multiple, 271, 362  
MPI\_Comm\_split, 60, 247, 250, 251, 259  
MPI\_Comm\_split\_type, 251, 346, 351  
MPI\_Comm\_test\_inter, 257  
MPI\_COMM\_TYPE\_HW\_GUIDED, 346  
MPI\_COMM\_TYPE\_SHARED, 346  
MPI\_COMM\_WORLD, 245–247, 254, 255, 259, 268,  
    271, 351, 362, 364, 365, 374, 377, 382,  
    487, 491, 497  
MPI\_Compare\_and\_swap, 296  
MPI\_COMPLEX, 180  
MPI\_Count, 180, 209, 210  
MPI\_COUNT\_KIND, 180, 375  
MPI\_Datatype, 52, 178, 185, 185  
MPI\_DATATYPE\_NULL, 180, 186  
MPI\_DISPLACEMENT\_CURRENT, 333  
MPI\_DIST\_GRAPH, 337  
MPI\_Dist\_graph\_create, 340, 341, 342, 343  
MPI\_Dist\_graph\_create\_adjacent, 340,  
    341  
MPI\_Dist\_graph\_neighbors, 342, 343, 343  
MPI\_Dist\_graph\_neighbors\_count, 342,  
    343  
MPI\_DOUBLE, 179  
MPI\_DOUBLE\_COMPLEX, 180  
MPI\_DOUBLE\_INT, 68  
MPI\_DOUBLE\_PRECISION, 179, 180  
MPI\_ERR\_ARG, 363  
MPI\_ERR\_BUFFER, 155, 363  
MPI\_ERR\_COMM, 363, 365  
MPI\_ERR\_INFO, 363  
MPI\_ERR\_INTERN, 155, 363  
MPI\_ERR\_OTHER, 363  
MPI\_ERR\_PORT, 272, 363  
MPI\_ERR\_SERVICE, 273, 363  
MPI\_ERRCODES\_IGNORE, 268, 375  
MPI\_Errhandler, 363  
MPI\_ERROR, 147, 364  
MPI\_Error\_string, 363, 364  
MPI\_ERRORS\_ARE\_FATAL, 363  
MPI\_ERRORS\_RETURN, 363, 364  
MPI\_Exscan, 66, 66  
MPI\_Fetch\_and\_op, 293, 293, 297, 302  
MPI\_File, 325  
MPI\_File\_close, 325  
MPI\_File\_delete, 326  
MPI\_File\_get\_size, 333  
MPI\_File\_get\_view, 333  
MPI\_File\_iread, 330  
MPI\_File\_iread\_shared, 333  
MPI\_File\_iwrite, 330  
MPI\_File\_iwrite\_shared, 333  
MPI\_File\_open, 37, 325, 360  
MPI\_File\_preeallocate, 333  
MPI\_File\_read, 326, 335  
MPI\_File\_read\_all, 326  
MPI\_File\_read\_all\_begin, 330  
MPI\_File\_read\_all\_end, 330  
MPI\_File\_read\_at, 326  
MPI\_File\_read\_at\_all, 326  
MPI\_File\_read\_ordered, 334  
MPI\_File\_read\_shared, 333, 335  
MPI\_File\_seek, 326, 332, 335  
MPI\_File\_seek\_shared, 333, 335  
MPI\_File\_set\_atomicity, 334  
MPI\_File\_set\_info, 360  
MPI\_File\_set\_size, 333  
MPI\_File\_set\_view, 332, 333, 335, 360  
MPI\_File\_sync, 326  
MPI\_File\_write, 326, 332, 333  
MPI\_File\_write\_all, 326, 330  
MPI\_File\_write\_all\_begin, 330  
MPI\_File\_write\_all\_end, 330  
MPI\_File\_write\_at, 330, 331, 332  
MPI\_File\_write\_at\_all, 330  
MPI\_File\_write\_ordered, 334  
MPI\_File\_write\_shared, 333  
MPI\_Finalize, 26, 27, 491

MPI\_Finalized, 27  
MPI\_FLOAT, 179  
MPI\_FLOAT\_INT, 68  
MPI\_Gather, 52, 61, 63, 85, 193, 324  
MPI\_Gatherv, 61, 61, 63  
MPI\_Get, 284, 286, 296, 299, 302, 304  
MPI\_Get\_accumulate, 291, 293, 293, 305  
MPI\_Get\_address, 183, 199, 202, 304  
MPI\_Get\_count, 143, 147, 209  
MPI\_Get\_count\_x, 210  
MPI\_Get\_elements, 148, 209  
MPI\_Get\_elements\_x, 148, 210  
MPI\_Get\_hw\_resource\_types, 346  
MPI\_Get\_processor\_name, 27, 28, 28, 362  
MPI\_Get\_version, 362  
MPI\_GRAPH, 337  
MPI\_Graph\_create, 343  
MPI\_Graph\_get, 343  
MPI\_Graph\_neighbors, 343  
MPI\_Graph\_neighbors\_count, 343  
MPI\_Graphdims\_get, 343  
MPI\_Group, 253, 253  
MPI\_Group\_difference, 253  
MPI\_Group\_excl, 253  
MPI\_Group\_incl, 253  
MPI\_HOST, 362  
MPI\_Iallgather, 74  
MPI\_Iallgatherv, 74  
MPI\_Iallreduce, 73, 74  
MPI\_Ialltoall, 74  
MPI\_Ialltoallv, 74  
MPI\_Ialltoallw, 74  
MPI\_Ibarrier, 72, 74, 75  
MPI\_Ibcast, 74  
MPI\_Ibsend, 155  
MPI\_Iexscan, 74  
MPI\_Igather, 74, 75  
MPI\_Igatherv, 74  
MPI\_IN\_PLACE, 44, 52, 375  
MPI\_Ineighbor\_allgather, 343  
MPI\_Ineighbor\_allgatherv, 343  
MPI\_Ineighbor\_alltoall, 343  
MPI\_Ineighbor\_alltoallv, 343  
MPI\_Ineighbor\_alltoallw, 343  
MPI\_Info, 305, 333, 349, 358, 360  
MPI\_Info\_create, 358  
MPI\_Info\_delete, 358  
MPI\_Info\_dup, 358  
MPI\_INFO\_ENV, 28, 358  
MPI\_Info\_free, 358  
MPI\_Info\_get, 358  
MPI\_Info\_get\_nkeys, 358  
MPI\_Info\_get\_nthkey, 358  
MPI\_INFO\_NULL, 333  
MPI\_Info\_set, 358  
MPI\_Init, 26, 27, 28, 374, 377, 382, 490, 557  
    in Fortran, 365  
MPI\_Init\_thread, 27, 377, 581, 581  
MPI\_Initialized, 27  
MPI\_INT, 178, 179, 377  
MPI\_INTEGER, 180  
MPI\_INTEGER1, 180  
MPI\_INTEGER16, 180  
MPI\_INTEGER2, 180  
MPI\_INTEGER4, 180  
MPI\_INTEGER8, 180  
MPI\_INTEGER\_KIND, 375  
MPI\_Intercomm\_create, 247, 254  
MPI\_Intercomm\_merge, 257  
MPI\_IO, 362  
MPI\_Iprobe, 76, 143, 366  
MPI\_Irecv, 73, 83, 130, 132, 137, 139, 142, 144,  
    145, 150  
MPI\_Ireduce, 74  
MPI\_Ireduce\_scatter, 74  
MPI\_Ireduce\_scatter\_block, 74  
MPI\_Is\_thread\_main, 581  
MPI\_Iscan, 74  
MPI\_Iscatter, 74, 75  
MPI\_Iscatterv, 74  
MPI\_Isend, 83, 130, 132, 150, 151, 291  
MPI\_Issend, 150  
MPI\_KEYVAL\_INVALID , 375  
MPI\_LOCK\_EXCLUSIVE, 299, 375  
MPI\_LOCK\_SHARED, 299, 375  
MPI\_LOGICAL, 180

---

MPI\_LONG, 179  
 MPI\_LONG\_DOUBLE, 179  
 MPI\_LONG\_DOUBLE\_INT, 68  
 MPI\_LONG\_INT, 68  
 MPI\_LONG\_LONG\_INT, 179  
 MPI\_MAX, 43  
 MPI\_MAX\_DATAREP\_STRING, 375  
 MPI\_MAX\_ERROR\_STRING, 364, 375  
 MPI\_MAX\_INFO\_KEY, 358, 375  
 MPI\_MAX\_INFO\_VAL, 375  
 MPI\_MAX\_LIBRARY\_VERSION\_STRING, 375  
 MPI\_MAX\_OBJECT\_NAME, 375  
 MPI\_MAX\_PORT\_NAME, 271, 375  
 MPI\_MAX\_PROCESSOR\_NAME, 29, 29, 362, 374, 382  
 MPI\_MAXLOC, 68  
 MPI\_Message, 144  
 MPI\_MINLOC, 68  
 MPI\_MODE\_APPEND, 326  
 MPI\_MODE\_CREATE, 326  
 MPI\_MODE\_DELETE\_ON\_CLOSE, 326  
 MPI\_MODE\_EXCL, 326  
 MPI\_MODE\_NOCHECK, 300, 306, 307  
 MPI\_MODE\_NOPRECEDE, 307  
 MPI\_MODE\_NOPUT, 307  
 MPI\_MODE\_NOSTORE, 307  
 MPI\_MODE\_NOSUCCEED, 307  
 MPI\_MODE\_RDONLY, 326  
 MPI\_MODE\_RDWR, 326  
 MPI\_MODE\_SEQUENTIAL, 326  
 MPI\_MODE\_UNIQUE\_OPEN, 326  
 MPI\_MODE\_WRONLY, 326  
 MPI\_Mprobe, 144  
 MPI\_Mrecv, 144  
 MPI\_Neighbor\_allgather, 342, 343  
 MPI\_Neighbor\_allgatherv, 343  
 MPI\_Neighbor\_allreduce, 343  
 MPI\_Neighbor\_alltoall, 343  
 MPI\_Neighbor\_alltoallv, 343  
 MPI\_Neighbor\_alltoallw, 343  
 MPI\_NO\_OP, 291, 293, 305  
 MPI\_Offset, 180, 331  
 MPI\_OFFSET\_KIND, 180, 333, 375  
 MPI\_Op, 40, 65, 66, 67, 68, 71, 284, 293, 364  
 MPI\_Op\_commutative, 71  
 MPI\_Op\_create, 67, 68, 71  
 MPI\_Op\_free, 71  
 MPI\_OP\_NULL, 71  
 MPI\_Open\_port, 271, 271  
 MPI\_ORDER\_C, 195  
 MPI\_ORDER\_FORTRAN, 195  
 MPI\_Pack, 211  
 MPI\_Pack\_size, 155, 212  
 MPI\_PACKED, 179, 180, 211  
 MPI\_Probe, 143, 143, 366  
 MPI\_PROC\_NULL, 123, 123, 125, 128, 138, 256, 284, 339, 362, 375, 625  
 MPI\_PROD, 43, 66  
 MPI\_Publish\_name, 273  
 MPI\_Put, 284, 285, 286, 288, 296, 297, 299, 302  
 MPI\_Query\_thread, 581  
 MPI\_Raccumulate, 291  
 MPI\_REAL, 179, 180  
 MPI\_REAL2, 180  
 MPI\_REAL4, 180  
 MPI\_REAL8, 180  
 MPI\_Recv, 116, 118, 119, 122, 128, 129, 139, 145, 147, 366, 376  
 MPI\_Recv\_init, 150, 151  
 MPI\_Reduce, 42, 44, 61, 85, 289  
 MPI\_Reduce\_local, 71  
 MPI\_Reduce\_scatter, 56, 58, 58, 60, 85, 308  
 MPI\_Reduce\_scatter\_block, 56, 57  
 MPI\_REPLACE, 289, 291, 293  
 MPI\_Request, 20, 72, 132, 142, 150, 330  
 MPI\_Request\_free, 142, 142, 151, 377  
 MPI\_Request\_get\_status, 142  
 MPI\_REQUEST\_NULL, 142  
 MPI\_Rget, 291  
 MPI\_Rget\_accumulate, 291  
 MPI\_ROOT, 256, 375  
 MPI\_Rput, 291  
 MPI\_Rsend, 372  
 MPI\_Rsend\_init, 154  
 MPI\_Scan, 64, 65, 66  
 MPI\_Scatter, 49, 50, 85

MPI\_Scatterv, 61  
MPI\_SEEK\_CUR, 326, 332  
MPI\_SEEK\_END, 326  
MPI\_SEEK\_SET, 326, 334  
MPI\_Send, 73, 82, 114, 118–120, 122, 128, 129, 139, 366, 372  
MPI\_Send\_init, 150, 151  
MPI\_Sendrecv, 123, 123, 125, 128, 129, 139, 625  
MPI\_Sendrecv\_replace, 126  
MPI\_SHORT, 179  
MPI\_SHORT\_INT, 68  
MPI\_SIGNED\_CHAR, 179  
MPI\_Sizeof, 183, 183, 203, 365  
MPI\_SOURCE, 137, 140, 145, 145, 149  
MPI\_Ssend, 120, 150, 372  
MPI\_Ssend\_init, 154  
MPI\_Start, 150, 151  
MPI\_Startall, 150, 151  
MPI\_Status, 118, 123, 128, 132, 133, 138, 142, 144, 144, 149, 326  
MPI\_STATUS\_IGNORE, 118, 128, 133, 138, 374, 375, 382  
MPI\_STATUS\_SIZE, 375  
MPI\_STATUSES\_IGNORE, 134, 138, 375  
MPI\_SUBARRAYS\_SUPPORTED, 375  
MPI\_SUBVERSION, 362, 375  
MPI\_SUCCESS, 19, 155, 363  
MPI\_SUM, 43, 61, 66  
MPI\_TAG, 147  
MPI\_TAG\_UB, 116, 362, 362  
MPI\_Test, 76, 140, 142, 330, 366, 377  
MPI\_Testall, 140  
MPI\_Testany, 140  
MPI\_THREAD\_FUNNELED, 581  
MPI\_THREAD\_MULTIPLE, 581  
MPI\_THREAD\_SERIALIZED, 581  
MPI\_THREAD\_SINGLE, 581  
MPI\_Topo\_test, 337  
MPI\_Type\_commit, 186  
MPI\_Type\_contiguous, 186, 186, 375  
MPI\_Type\_create\_f90\_complex, 180  
MPI\_Type\_create\_f90\_integer, 180  
MPI\_Type\_create\_f90\_real, 180  
MPI\_Type\_create\_hindexed, 198  
MPI\_Type\_create\_hindexed\_block, 198  
MPI\_Type\_create\_resized, 207  
MPI\_Type\_create\_struct, 186, 199  
MPI\_Type\_create\_subarray, 186, 193, 193  
MPI\_Type\_extent, 205  
MPI\_Type\_free, 186  
MPI\_Type\_get\_extent, 204, 204, 205  
MPI\_Type\_get\_extent\_x, 210  
MPI\_Type\_get\_true\_extent, 205  
MPI\_Type\_get\_true\_extent\_x, 206, 210  
MPI\_Type\_hindexed, 186  
MPI\_Type\_indexed, 186, 195, 198  
MPI\_Type\_lb, 205  
MPI\_Type\_match\_size, 203  
MPI\_Type\_size, 204  
MPI\_Type\_size\_x, 180  
MPI\_Type\_struct, 199  
MPI\_Type\_ub, 205  
MPI\_Type\_vector, 186, 188  
MPI\_TYPECLASS\_COMPLEX, 203  
MPI\_TYPECLASS\_INTEGER, 203  
MPI\_TYPECLASS\_REAL, 203  
MPI\_UB, 202, 207  
MPI\_UNDEFINED, 337, 375  
MPI\_UNIVERSE\_SIZE, 268, 362  
MPI\_Unpack, 211  
MPI\_Unpublish\_name, 273, 363  
MPI\_UNSIGNED, 179  
MPI\_UNSIGNED\_CHAR, 179  
MPI\_UNSIGNED\_LONG, 179  
MPI\_UNSIGNED\_SHORT, 179  
MPI\_UNWEIGHTED, 342, 375  
MPI\_VERSION, 362, 374, 375, 382  
MPI\_Wait, 72, 132, 134, 138, 144, 151, 283, 330, 377  
MPI\_Wait..., 128, 134, 145, 151  
MPI\_Waitall, 134, 134, 138, 279  
MPI\_Waitany, 135, 137, 138  
MPI\_Waitsome, 138  
MPI\_WEIGHTS\_EMPTY, 342, 375  
MPI\_Win, 183, 279, 346  
MPI\_Win\_allocate, 280, 282, 303, 306

MPI\_Win\_allocate\_shared, 280, 303, 306, striping\_unit, 361  
347, 347, 349  
MPI\_Win\_attach, 303 wtime, 367  
MPI\_Win\_complete, 292  
MPI\_Win\_create, 183, 280, 280, 282, 303, 305,  
376  
MPI\_Win\_create\_dynamic, 280, 303, 306  
MPI\_Win\_detach, 304  
MPI\_Win\_fence, 283, 299, 300, 306, 307, 626  
MPI\_WIN\_FLAVOR\_ALLOCATE, 306  
MPI\_WIN\_FLAVOR\_CREATE, 305  
MPI\_WIN\_FLAVOR\_DYNAMIC, 306  
MPI\_WIN\_FLAVOR\_SHARED, 306  
MPI\_Win\_flush, 302  
MPI\_Win\_flush..., 291  
MPI\_Win\_flush\_all, 302  
MPI\_Win\_flush\_local, 300, 302  
MPI\_Win\_flush\_local\_all, 302  
MPI\_Win\_get\_info, 360  
MPI\_Win\_lock, 293, 298, 299, 307  
MPI\_Win\_lock\_all, 299, 299, 302, 307  
MPI\_Win\_post, 291, 306, 307  
MPI\_WIN\_SEPARATE, 306  
MPI\_Win\_set\_info, 360  
MPI\_Win\_shared\_query, 347, 349  
MPI\_Win\_start, 292, 306, 307  
MPI\_Win\_sync, 302  
MPI\_WIN\_UNIFIED, 306  
MPI\_Win\_unlock, 299, 300  
MPI\_Win\_unlock\_all, 299, 300, 300  
MPI\_Win\_wait, 291  
MPI\_Wtick, 367, 368  
MPI\_Wtime, 118, 367, 551  
MPI\_WTIME\_IS\_GLOBAL, 362, 367

nb\_proc, 360  
no\_locks, 305  
num\_io\_nodes, 361

PMPI\_..., 368

same\_op, 305  
same\_op\_no\_op, 305  
striping\_factor, 361

## 42.5 MPL commands and topics

absolute, 203  
blocking-send-and-receive, 116  
`bsend_buffer`, 154  
`bsend_size`, 154  
`buffer-attach`, 154  
`buffer-handling`, 41  
`buffered-send`, 154  
  
`comm_world`, 31  
`communicator`, 31, 251  
`communicator-errhandler`, 364  
`communicator-splitting`, 251  
`communicator-types`, 246  
`communicator::split`, 251  
`contiguous-type`, 188  
`contiguous_layout`, 42, 188  
  
`data-types`, 179  
`derived-type-handling`, 185  
  
`gather`, 52  
`gather-scatter`, 52  
  
`header-file`, 26  
`heterogeneous_layout`, 202  
  
`indexed-type`, 198  
`indexed_block_layout`, 198  
`indexed_layout`, 198  
`init,-finalize`, 27  
`irequest`, 74, 133, 133  
`irequest_pool`, 133, 136  
`iterator_layout`, 191  
  
`make_absolute`, 203  
`message-tag`, 147  
`mpl-operators`, 68  
  
`non-blocking-collectives`, 74  
`notes-format`, 17  
  
`processor-name`, 29  
`processor_name`, 29

rank, 31  
rank-and-size, 31  
receive-count, 148  
reduce-in-place, 45  
reduction-operator, 40  
requests-from-non-blocking-calls, 133  
  
`scatter`, 52  
`send-recv-call`, 123  
`size`, 31  
`status`, 133, 146  
`status-querying`, 146  
`strided_vector_layout`, 191  
`struct-type`, 202  
`subarray-layout`, 193  
`subarray_layout`, 193  
`submit`, 568  
  
`tag`, 147  
`testall`, 136  
`testany`, 136  
`testsome`, 136  
`timing`, 367  
  
`vector-type`, 191  
  
`wait-any`, 136  
`waitall`, 136  
`waitany`, 136  
`waitsome`, 136  
`world-communicator`, 30  
`wtick`, 367  
`wtime_is_global`, 367

## 42.6 MPI4Python notes

Big data, 210

Buffers from numpy, 41

Comm split key is optional, 251

Communicator methods, 31

Communicator object, 491

Communicator types, 246

Data types, 179

Derived type handling, 185

Displacement byte computations, 282

File open is class method, 325

Graph communicators, 342

Import mpi module, 26

In-place collectives, 45

Init, and with commandline options, 491

No initialize/finalize calls, 27

P, 614

Petsc options, 550

Petsc print and python print, 547

Python notes, 18

Request arrays, 135

Running mpi4py programs, 16

Sending objects, 47

User-defined operators, 70

Vector creation, 497

Vector type, 190

Window buffers, 282

## 42.7 Index of OpenMP commands

\_OPENMP, 390

omp

- atomic, 436, 460
- barrier, 434
- cancel, 448
- critical, 436, 471
- declare simd, 463
- flush, 437, 461
- lastprivate, 412
- master, 415, 416, 581
- ordered, 410
- parallel, 391, 397, 455
- parallel for, 404
- private, 419
- section, 414
- sections, 414, 422
- simd, 463, 463
- single, 415
- task, 442, 445, 446
- taskgroup, 445, 446
- taskwait, 445--447
- taskyield, 447
- threadprivate, 422, 457, 471
- workshare, 416

omp clause

- aligned, 463
- collapse, 410
- copyin, 423
- copyprivate, 416, 423
- default, 420
  - firstprivate, 421
  - none, 421
  - private, 420
  - shared, 420
- depend, 446
- firstprivate, 422, 443
- lastprivate, 422
- linear, 463
- nowait, 411, 435, 471
- ordered, 410
- private, 419

proc\_bind, 453, 455

reduction, 427, 430

safelen( $n$ ), 463

schedule, 471

- auto, 408
- chunk, 407
- guided, 407
- runtime, 408
- untied, 447

OMP\_CANCELLATION, 468

OMP\_DEFAULT\_DEVICE, 469

omp\_destroy\_nest\_lock, 438

OMP\_DISPLAY\_ENV, 452, 468

OMP\_DYNAMIC, 424, 469, 469

omp\_get\_active\_level, 468

omp\_get\_ancestor\_thread\_num, 468

omp\_get\_dynamic, 468, 469

omp\_get\_level, 468

omp\_get\_max\_active\_levels, 468

omp\_get\_max\_threads, 468, 469

omp\_get\_nested, 468, 469

omp\_get\_num\_procs, 395, 468, 469

omp\_get\_num\_threads, 395, 397, 468, 469

omp\_get\_schedule, 409, 468, 469

omp\_get\_team\_size, 468

omp\_get\_thread\_limit, 468

omp\_get\_thread\_num, 397, 468, 469

omp\_get\_wtick, 468, 470

omp\_get\_wtime, 468, 470

omp\_in, 430

omp\_in\_parallel, 400, 468, 469

omp\_init\_nest\_lock, 438

OMP\_MAX\_ACTIVE\_LEVELS, 469

OMP\_MAX\_TASK\_PRIORITY, 469

OMP\_NESTED, 395, 469, 469

OMP\_NUM\_THREADS, 390, 394, 469, 469

omp\_out, 430

OMP\_PLACES, 452, 454, 469

omp\_priv, 430

OMP\_PROC\_BIND, 452, 455, 469, 470

omp\_sched\_affinity, 409

omp\_sched\_auto, 409  
omp\_sched\_dynamic, 409  
omp\_sched\_guided, 409  
omp\_sched\_runtime, 409  
omp\_sched\_static, 409  
OMP\_SCHEDULE, 408, 409, 469, 469  
omp\_set\_dynamic, 468, 469  
omp\_set\_max\_active\_levels, 468  
omp\_set\_nest\_lock, 438  
omp\_set\_nested, 468, 469  
omp\_set\_num\_threads, 394, 468, 469  
omp\_set\_schedule, 409, 468, 469  
OMP\_STACKSIZE, 419, 469, 469  
omp\_test\_nest\_lock, 438  
OMP\_THREAD\_LIMIT, 469  
omp\_unset\_nest\_lock, 438  
OMP\_WAIT\_POLICY, 399, 469, 469  
  
schedule, 406  
  
wait-policy-var, 469

## 42.8 Index of PETSc commands

--ksp\_monitor, 551  
--ksp\_view, 551  
--sub\_ksp\_monitor, 551  
-download-blas-lapack, 496  
-download\_mpich, 492  
-ksp\_atol, 534  
-ksp\_converged\_reason, 535  
-ksp\_divtol, 534  
-ksp\_gmres\_restart, 536  
-ksp\_max\_it, 534  
-ksp\_monitor, 541  
-ksp\_monitor\_true\_residual, 541  
-ksp\_rtol, 534  
-ksp\_type, 535  
-ksp\_view, 549  
-log\_summary, 549  
-malloc\_dump, 552  
-pc\_factor\_levels, 539  
-with-precision, 492  
-with-scalar-type, 492  
  
ADD\_VALUES, 504, 511  
AO, 520  
  
CHKERRA, 544  
CHKERRQ, 544  
CHKMMA, 546  
CHKMEMQ, 544, 546  
  
DM, 516, 517  
DMCreateGlobalVector, 517  
DMCreateLocalVector, 517  
DMDA, 517  
DMDACreate1d, 516  
DMDACreate2d, 516  
DMDAGetCorners, 516, 518  
DMDAGetLocalInfo, 516  
DMDALocalInfo, 516  
DMGlobalToLocal, 517  
DMGlobalToLocalBegin, 517  
DMGlobalToLocalEnd, 517  
DMLocalToGlobal, 517  
  
DMLocalToGlobalBegin, 517  
DMLocalToGlobalEnd, 517  
  
INSERT\_VALUES, 504, 511  
IS, 521  
ISCreate, 519  
ISCreateBlock, 519  
ISCreateGeneral, 519  
ISCreateStride, 519  
ISGetIndices, 519  
ISRestoreIndices, 519  
  
KSP, 532  
KSPBuildResidual, 542  
KSPBuildSolution, 542  
KSPConvergedDefault, 541  
KSPConvergedReason, 534  
KSPConvergenceReasonView, 535  
KSPCreate, 533  
KSPGetConvergedReason, 534  
KSPGetIterationNumber, 535  
KSPGetRhs, 542  
KSPGetSolution, 542  
KSPGMRESSetRestart, 536  
KSPMatSolve, 536  
KSPMonitorDefault, 541  
KSPMonitorSet, 541  
KSPMonitorTrueResidualNorm, 541  
KSPReasonView, 535  
KSPSetConvergenceTest, 541  
KSPSetFromOptions, 534, 536, 542  
KSPSetOperators, 534  
KSPSetOptionsPrefix, 551  
KSPSetTolerances, 534  
KSPSetType, 535  
KSPView, 549  
  
MAT\_FLUSH\_ASSEMBLY, 511  
MatAssemblyBegin, 511, 511  
MatAssemblyEnd, 511, 511  
MatCreate, 506  
MatCreateFFT, 516

MatCreateShell, **513**  
MatCreateSubMatrices, **513**  
MatCreateSubMatrix, **513, 521**  
MatCreateVecs, **500, 508**  
MatCreateVecsFFTW, **516**  
MatGetRow, **511**  
MatMatMult, **513**  
MATMPIAIJ, **506**  
MatMPIAIJSetPreallocation, **509**  
MATMPIBIJ, **516**  
MATMPIDENSE, **506**  
MatMult, **512, 513**  
MatMultAdd, **512**  
MatMultHermitianTranspose, **512**  
MatMultTranspose, **512**  
MatPartitioning, **520**  
MatPartitioningApply, **521**  
MatPartitioningCreate, **521**  
MatPartitioningDestroy, **521**  
MatPartitioningSetType, **521**  
MatRestoreRow, **511**  
MATSEQAIJ, **506**  
MatSeqAIJSetPreallocation, **509**  
MATSEQDENSE, **506**  
MatSetSizes, **508**  
MatsetType, **506**  
MatSetValue, **510**  
MatSetValues, **510**  
MatShellGetContext, **515**  
MatShellSetContext, **515**  
MatShellSetOperation, **513**  
MatSizes, **508**  
MatView, **549**  
MPI\_Datatype, **557**  
MPIU\_COMPLEX, **496, 557**  
MPIU\_INT, **557**  
MPIU\_REAL, **496, 557**  
MPIU\_SCALAR, **496, 557**  
  
PCCOMPOSITE, **541**  
PCFactorSetLevels, **539**  
PCGAMG, **540**  
PCHYPRESetType, **538**  
PCM, **540**  
  
PCSHELL, **540**  
PCShellGetContext, **540**  
PCShellSetApply, **540**  
PCShellSetContext, **540**  
PCShellSetSetUp, **541**  
PETSC\_ARCH, **489**  
PETSC\_CC\_INCLUDES, **489**  
PETSC\_COMM\_SELF, **497, 517, 545**  
PETSC\_COMM\_WORLD, **491, 497, 545**  
PETSC\_DECIDE, **495, 498**  
PETSC\_DEFAULT, **534**  
PETSC\_DIR, **489**  
PETSC\_ERR\_ARG\_OUTOFRANGE, **496**  
PETSC\_FC\_INCLUDES, **489**  
PETSC\_i, **496**  
PETSC\_MEMALIGN, **552**  
PETSC\_NULL\_CHARACTER, **490**  
PETSC\_NULL\_INTEGER, **488**  
PETSC\_NULL\_IS, **520**  
PETSC\_NULL\_OBJECT, **488**  
PETSC\_NULL\_VIEWER, **549**  
PETSC\_STDOUT, **547**  
PETSC\_USE\_DEBUG, **544**  
PETSC\_VIEWER\_STDOUT\_WORLD, **549**  
PetscBLASInt, **496, 496, 557**  
PetscBLASIntCast, **496**  
PetscCalloc1, **552**  
PetscComm, **557**  
PetscComplex, **496, 496, 557**  
PetscDataType, **547**  
PetscErrorCode, **496, 544**  
PetscFinalize, **491, 552**  
PetscFree, **552**  
PetscFunctionBegin, **544**  
PetscFunctionReturn, **544**  
PetscGetCPUTime, **551**  
PetscImaginaryPart, **496**  
PetscInitialize, **490, 490, 491, 549, 551, 557**  
PetscInitializeFortran, **491**  
PetscInt, **496, 496, 519, 557**  
petscksp.h, **536**  
PetscLogDouble, **551**  
PetscMalloc, **505, 552**

PetscMalloc1, **552**  
PetscMallocDump, **552**  
PetscMPIInt, **496, 557**  
PetscNew, **552**  
PetscOptionsBegin, **550**  
PetscOptionsEnd, **550**  
PetscOptionsGetInt, **550**  
PetscOptionsSetValue, **551, 551**  
PetscPrintf, **546, 547**  
PetscReal, **496, 496, 551, 557**  
PetscRealPart, **496**  
PetscScalar, **495, 496**  
PetscSplitOwnership, **495**  
PetscSynchronizedFlush, **547**  
PetscSynchronizedPrintf, **547**  
PetscTime, **551**  
PetscViewer, **547**  
PETSCVIEWERASCII, **549**  
PETSCVIEWERBINARY, **549**  
PetscViewerCreate, **549**  
PETSCVIEWERDRAW, **549**  
PETSCVIEWERHDF5, **549**  
PetscViewerPopFormat, **549**  
PetscViewerPushFormat, **549**  
PetscViewerRead, **547**  
PetscViewerSetType, **549**  
PETSCVIEWERSOCKET, **549**  
PETSCVIEWERSTRING, **549**  
PETSCVIEWERVTK, **549**

SETERRA, **544**  
SETERRQ, **544**  
SETERRQ1, **544, 546**  
SETERRQ2, **544**

VecAssemblyBegin, **503, 504**  
VecAssemblyEnd, **503, 504**  
VecCreate, **497**  
VecCreateMPIWithArray, **500**  
VecCreateSeqWithArray, **500**  
VecDestroy, **497**  
VecDestroyVecs, **498**  
VecDot, **502**  
VecDotBegin, **502**

VecDotEnd, **502**  
VecDuplicate, **497**  
VecDuplicateVecs, **497**  
VecGetArray, **504**  
VecGetArrayF90, **506**  
VecGetArrayRead, **504**  
VecGetLocalSize, **498**  
VecGetOwnershipRange, **498**  
VecGetSize, **498**  
VECMPI, **497**  
VecNorm, **502**  
VecNormBegin, **502**  
VecNormEnd, **502**  
VecPlaceArray, **505, 506**  
VecReplaceArray, **505**  
VecResetArray, **505**  
VecRestoreArray, **504**  
VecRestoreArrayF90, **506**  
VecRestoreArrayRead, **504**  
VecScale, **502**  
VecScatter, **519**  
VecScatterCreate, **519**  
VECSEQ, **497**  
VecSet, **502**  
VecSetSizes, **498, 516**  
VecSetType, **497**  
VecSetValue, **502, 502, 504**  
VecSetValues, **502, 502, 504**  
VecView, **500**

