MPI Course

Victor Eijkhout

2024 COE 379L / CSE 392

Materials

Textbooks and repositories:
`https://theartofhpc.com`

Justification

The MPI library is the main tool for parallel programming on a large scale. This course introduces the main concepts through lecturing and exercises.

Table of Contents

1. The SPMD Model *link*
2. Collectives *link*
3. Point-to-point *link*
4. Derived datatypes *link*
5. Communicators *link*
6. MPI I/O *link*
7. One-sided communication *link*
8. Big data *link*
9. Advanced collectives *link*
10. Shared memory *link*
11. Process management *link*
12. Process topologies *link*
13. Trace and performance *link*

Basics

Basics6 The SPMD model7 Collectives54 Concepts57

All-reduce

Buffers69 Collective basics74 Scan86 Gather/Scatter, Barrier, and others92 User-defined operators112 Performance of collectives118 Point-to-point communication127 Point-to-point communication129 Distributed data137 Local information exchange143 Blocking communication149 Pairwise exchange157 Irregular exchanges: non-blocking communication172 Communicating other than in pairs174

Intermediate topics

Intermediate topics202 Derived Datatypes204 Discussion206 Datatypes209 Subarray type223 Extent and resizing229 Packed data241 Communicator manipulations244 Cartesian topologies263 MPI File I/O270 One-sided communication282 Basic mechanisms284 Ordering and synchronization299 Passive target synchronization304 Big data communication322

Advanced (MPI-3/4) topics

Part

The SPMD model

Overview

In this section you will learn how to think about parallelism in MPI.

Commands learned:

- `MPI_Init`, `MPI_Finalize`,
- `MPI_Comm_size`, `MPI_Comm_rank`
- `MPI_Get_processor_name`,

The MPI worldview: SPMD

Computers when MPI was designed

One processor and one process per node;
all communication goes through the network.

⇒ process model, no data sharing.

Lab setup

- Clone the repository
- Directory: `exercises-mpi-c` or `cxx` or `f` or `f08` or `p` or `mpl`
- Open a terminal window on a TACC cluster.
- Type `idev -N 2 -n 10 -t 2:0:0` which gives you an interactive session of 2 nodes, 10 cores, for the next 2 hours.
- Type `make exercisename` to compile it
- Run with `ibrun` or `mpiexec` (see above)

Exercise 1 (`hello`)

Write a 'hello world' program, without any MPI in it, and run it in parallel with `mpiexec` or your local equivalent. Explain the output.

1. In the directories `exercises-mpi-xxx` do `make hello` to compile;
2. do `ibrun hello` to execute.

In a picture

int main() {
.....
printf("Hello world!\n");
.....
}
a.out
proc1
proc2
proc3
proc4
int main() {
.....
printf("Hello world!\n");
.....
}
int main() {
.....
printf("Hello world!\n");
.....
}
int main() {
.....
printf("Hello world!\n");
.....
}
int main() {
.....
printf("Hello world!\n");
.....
}

SPMD

The basic model of MPI is
'Single Program Multiple Data':
each process is an instance of the same program.

Symmetry: There is no 'master process', all processes are equal, start and end at the same time.

Communication calls do not see the cluster structure:
data sending/receiving is the same for all neighbors.

Practicalities

Pure MPI

A node has multiple sockets, each with multiple cores.
Pure MPI puts a process on each core: pretend shared memory doesn't exist.

Quad socket node

Hybrid programming

Hybrid programming puts a process per node or per socket; further parallelism comes from threading.

Terminology

'Processor' is ambiguous: is that a chip or one independent instruction processing unit?

- Socket: the processor chip
- Processor: we don't use that word
- Core: one instruction-stream processing unit
- Process: preferred terminology in talking about MPI.

Compiling

MPI compilers are usually called `mpicc`, `mpif90`, `mpicxx`.

These are not separate compilers, but scripts around the regular C/Fortran compiler. You can use all the usual flags.

```
$ mpicc -show
icc -I/intel/include/stuff -L/intel/lib/stuff -Wwarnings # et ce
```

(For CMake see slide 400.)

Running

Running your program at TACC:

```
#SBATCH -N 4
#SBATCH -n 200
ibrun yourprog
```

the number of processes is determined by SLURM. General case of running code

```
mpiexec -n 4 hostfile ... yourprogram arguments
mpirun -np 4 hostfile ... yourprogram arguments
```

Do I need a supercomputer?

- With `mpiexec` and such, you start a bunch of processes that execute your MPI program.
- Does that mean that you need a cluster or a big multicore?
- No! You can start a large number of MPI processes, even on your laptop. The OS will use 'time slicing'.
- Of course it will not be very efficient...

Installing your own MPI

It is convenient to do MPI development on your laptop/desktop.

- Use a package manager
  Apple: brew or macports
  Linux: yum, aptget, ...
  Windows: I'll have to get back to you on that
- ... or download and compile from source `mpich.org`

Cluster setup

Typical cluster:

- Login nodes, where you ssh into; usually shared with 100 (or so) other people. You don't run your parallel program there!
- Compute nodes: where your job is run. They are often exclusive to you: no other users getting in the way of your program.

Hostfile: the description of where your job runs. Usually generated by a *job scheduler*.

Interactive run

- Do not run your programs on a login node.
- Acquire compute nodes with `idev`
  (other systems: `qsub -I`)
- Caveat: only small short jobs; nodes may not be available.

Batch run

- Submit batch job with `sbatch` or `qsub`
- Your job will be executed . . . Real Soon Now.
- See userguide for details about queues, sizes, runtimes, . . .

We start learning MPI!

MPI Init / Finalize

These calls need to be around the MPI part of your code:

```
MPI_Init(&argc,&argv); // zeros allowed
// your code
MPI_Finalize();
```

This is not a 'parallel region'.
Only internal library initialization:
allocate buffers, discover network, . . .

Init Finalize, Fortran

No possibility for commandline arguments:

```
call MPI_Init()     ! F08 style
! your code
call MPI_Finalize()

call MPI_Init(ierr) ! F90 style
! your code
call MPI_Finalize(ierr)
```

Exercise 2 (`hello`)

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.

About library calls and bindings

Bindings

The standard defines interfaces to MPI from C and Fortran.
These look very similar; sometimes we will only show the C variant.

MPI can also be used from C++ and Python

MPI headers: C

You need an include file:

```
#include "mpi.h"
```

This defines all routines and constants.

MPI headers: Fortran

Fortran bindings are declared in the standard.

- Come in two flavors, Fortran90 vs Fortran2008
  you will find many examples online in old style
- We teach you modern style.

You need an include file:

```
! Modern Fortran2008
use mpi_f08
! Legacy Fortran90
use mpi
! Deprecated
#include "mpif.h"
```

Ranks

Process identification

- Processes are organized in 'communicators'.
- For now only the 'world' communicator
- Each process has a unique 'rank' wrt the communicator.

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

Lowest number is always zero.

This is a logical view of parallelism: mapping to physical processors/cores is invisible here.

Exercise 3 (`commrank`)

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

```
Hello from process 2 out of 5!
```

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

About routine signatures: C/C++

Signature:

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

Use:

```
MPI_Comm comm = MPI_COMM_WORLD;
int nprocs;
int errorcode;
errorcode = MPI_Comm_size( comm,&nprocs );
```

But forget about that error code most of the time:

```
MPI_Comm_size( comm,&nprocs );
```

About routine signatures: Fortran2008

Signature

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

Use:

```
Type(MPI_Comm) :: comm = MPI_COMM_WORLD
integer :: size
CALL MPI_Comm_size( comm, size ) ! F2008 style
```

- final parameter optional.
- `MPI_...` types are `Type`.

About routine signatures: Fortran90

Signature

```
MPI_Comm_size(comm, size, ierror)
Integer, Intent(in) :: comm
Integer, Intent(out) :: ierror
```

Use:

```
Integer :: comm = MPI_COMM_WORLD
Integer :: size,ierr
CALL MPI_Comm_size( comm, size, ierr ) ! F90 style
```

- Final parameter always error parameter. Do not forget!
- `MPI_...` types are `INTEGER`.

Exercise 4 (`commrank`)

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

Illustration

About errors

MPI routines invoke an error handler (slide 405)
default action: abort

Every routine is defined as returning integer error code

- In C: function result.

```
ierr = MPI_Init(0,0);
if (ierr!=MPI_SUCCESS) /* do something */
```

  But really: can often be ignored; is ignored in this course.

```
MPI_Init(0,0);
```

- In Fortran: as optional (F08 only) parameter.
- MPL: throws exceptions
- In Python: throwing exception.

There's not a lot you can do with an error code:
very hard to recover from errors in parallel.
By default code bombs with (hopefully informative) message.

Communicators

For now, the communicator will be `MPI_COMM_WORLD`.

C:

```
MPI_Comm comm = MPI_COMM_WORLD;
```

F:

```
Type(MPI_Comm) :: comm = MPI_COMM_WORLD
```

P:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
```

MPL:

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

Have you been paying attention?

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 5

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

TACC nodes have a hostname `cRRR-CNN`, where RRR is the rack number, C is the chassis number in the rack, and NN is the node number within the chassis. Communication is faster inside a rack than between racks!

Go to `examples/mpi/xxx` and do `make procname`, then `ibrun procname`

Processor name

Processes (can) run on physically distinct locations.

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

In a picture

Four processes on two nodes (`idev -N 2 -n 4`)

Processor name: Fortran

Processes (can) run on physically distinct locations.

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

Your first useful parallel program

Functional Parallelism

Parallelism by letting each process do a different thing.

Example: divide up a search space.

Each process knows its rank, so it can find its part of the search space.

Exercise 6 (`prime`)

Is the number $N = 2,000,000,111$ prime? Let each process test a disjoint set of integers, and print out any factor they find. You don't have to test all integers $< N$: any factor is at most $\sqrt{N} \approx 45,200$.

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

Can you find more than one solution?

Exercise 7

Allocate on each process an array:

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

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

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

Collectives

Overview

In this section you will learn 'collective' operations, that combine information from all processes.

Commands learned:

- `MPI_Bcast`, `MPI_Reduce`, `MPI_Gather`, `MPI_Scatter`
- *`MPI_All_...`* variants, *`MPI_....v`* variants
- `MPI_Barrier`, `MPI_Alltoall`, `MPI_Scan`

Technically

Routines can be ‘collective on a communicator’:

- They involve a communicator;
- if one process calls that routine, every process in that communicator needs to call it
- Mostly about combining data, but also opening shared files, declaring ‘windows’ for one-sided communication.

Concepts

Collectives

Gathering and spreading information:

- Every process has data, you want to bring it together;
- One process has data, you want to spread it around.

Root process: the one doing the collecting or disseminating.

Basic cases:

- Collect data: gather.
- Collect data and compute some overall value (sum, max): reduction.
- Send the same data to everyone: broadcast.
- Send individual data to each process: scatter.

Broadcast
1
2
3
4
10
Reduce
Scatter
Gather

Exercise 8

How would you realize the following scenarios with MPI collectives?

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

Think about time and space complexity of your suggestions.

Allreduce: reduce-to-all

Exercise 2 above contains a common case:
do a reduction, but everyone needs the result.

- `MPI_Allreduce` does the same as:
  `MPI_Reduce` (reduction) followed by `MPI_Bcast` (broadcast)
- Same running time as either, half of reduce-followed-by-broadcast (no proof given here)
- Common use case, symmetrical expression.

Motivation for allreduce

Example: normalizing a vector

$$y \leftarrow x/\|x\|$$

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

- Vectors x, y are distributed: every process has certain elements
- The norm calculation is an all-reduce: every process gets same value
- Every process scales its part of the vector.
- Question: what kind of reduction do you use for an inf-norm? One-norm? Two-norm?

Another Allreduce

Standard deviation:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i}^{N}(x_i - \mu)} \qquad \text{where} \qquad \mu = \frac{\sum_i^N x_i}{N}$$

and assume that every process stores just one x_i value.

How do we compute this?

1. The calculation of the average μ is a reduction.
2. Every process needs to compute $x_i - \mu$ for its value x_i, so use allreduce operation, which does the reduction and leaves the result on all processes.
3. $\sum_i (x_i - \mu)$ is another sum of distributed data, so we need another reduction operation. Might as well use allreduce.

Conceptual picture

Recall SPMD: every process has the input and output variable

(What actually happens is a different story!)

Allreduce syntax

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

- All processes have send and recv buffer
- (No root argument)
- *count* is number of items in the buffer: 1 for scalar.
  > 1: pointwise application of the reduction operator
- `MPI_Datatype` is `MPI_INT`, `MPI_FLOAT`, `MPI_REAL8` et cetera.
- `MPI_Op` is `MPI_SUM`, `MPI_MAX` et cetera.

Exercise 9 (`randommax`)

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

$$\xi = \sum_i x_i$$

Each process then scales its value by this sum.

$$x_i' \leftarrow x_i / \xi$$

Compute the sum of the scaled numbers

$$\xi' = \sum_i x_i'$$

and check that it is 1.

Exercise 10

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

MPI_Allreduce

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

Buffers

Buffers in Fortran

General principle: buffer is address in memory of the data.

- Fortran always passes by reference:
- write `x` for scalar
- write `x` for array

```
integer :: i
integer,dimension(:) :: iar
call MPI_Bcast( i, .... );
call MPI_Bcast( iar, .... );
```

Buffers in C++

- Scalars same as in C.
- Use of *std::vector* or *std::array*:

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

- Can not send from iterator / let recv determine size/capacity.

Large buffers

As of MPI-4 a buffer can be longer than 2^{31} elements.

- Use `MPI_Count` for count
- In C: use `MPI_Reduce_c`
- in Fortran: polymorphism means no change to the call.
- MPL: *long int* and *size_t* supported for layouts.

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

Collective basics

Elementary datatypes

C	Fortran	Python	meaning
MPI_CHAR	MPI_CHARACTER		only for text
MPI_SHORT	MPI_BYTE		8 bits
MPI_INT	MPI_INTEGER		like the C/F types
MPI_FLOAT	MPI_REAL		
MPI_DOUBLE	MPI_DOUBLE_PRECISION	*MPI.DOUBLE*	
	MPI_COMPLEX		
	MPI_LOGICAL		
unsigned	extensions		
			MPI_Aint MPI_Offset

A bunch more.

MPI operators

MPI_Op	description
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bitwise and
MPI_LOR	logical or
MPI_BOR	bitwise or
MPI_LXOR	logical xor
MPI_BXOR	bitwise xor
MPI_MAXLOC	location of max
MPI_MINLOC	location of min

A couple more.

Reduction to single process

Regular reduce: great for printing out summary information at the end of your job.

Reduction to root

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

- Buffers: *sendbuf*, *recvbuf* are ordinary variables/arrays.
- Every process has data in its *sendbuf*,
  Root combines it in *recvbuf* (ignored on non-root processes).
- *count* is number of items in the buffer: 1 for scalar.
- MPI_Op is MPI_SUM, MPI_MAX et cetera.

In-place operations

```
// 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);
```

More in-place operations

```
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);
```

or

```
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);
```

In-place (Fortran)

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

or

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

Broadcast

```
int MPI_Bcast(
    void *buffer, int count, MPI_Datatype datatype,
    int root, MPI_Comm comm )
```

- All processes call with the same argument list
- *root* is the rank of the process doing the broadcast
- Each process allocates buffer space;
  root explicitly fills in values,
  all others receive values through broadcast call.
- Datatype is `MPI_FLOAT`, `MPI_INT` et cetera, different between C/Fortran.
- *comm* is usually `MPI_COMM_WORLD`

Gauss-Jordan elimination

https://youtu.be/aQYuwatlWME

Exercise 11 (`jordan`)

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

for pivot $k = 1, \ldots, n$
    let the vector of scalings $\ell_i^{(k)} = A_{ik}/A_{kk}$
    for row $r \neq k$
        for column $c = 1, \ldots, n$
            $A_{rc} \leftarrow A_{rc} - \ell_r^{(k)} A_{kc}$

where we ignore the update of the righthand side, or the formation of the inverse.

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

Exercise (optional) 12

Bonus exercise: can you extend your program to have multiple columns per process?

Scan

Scan

Scan or 'parallel prefix': reduction with partial results

- Useful for indexing operations:
- Each process has an array of n_p elements;
- My first element has global number $\sum_{q<p} n_q$.
- Two variants: `MPI_Scan` inclusive, and `MPI_Exscan` exclusive.

In vs Exclusive

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

MPI_Scan

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

MPI_Exscan

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

Exercise 13 (scangather)

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

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

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

Gather/Scatter, Barrier, and others

MPI_Gather

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

MPI_Scatter

Name	Param name	Explanation	C type	F type
MPI_Scatter (				
	sendbuf	address of send buffer	const void*	TYPE(*), DIMENSION(..)
	sendcount	number of elements sent to each process	int	INTEGER
	sendtype	datatype of send buffer elements	MPI_Datatype	TYPE(MPI_Datatype)
	recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)
	recvcount	number of elements in receive buffer	int	INTEGER
	recvtype	datatype of receive buffer elements	MPI_Datatype	TYPE(MPI_Datatype)
	root	rank of sending process	int	INTEGER
	comm	communicator	MPI_Comm	TYPE(MPI_Comm)
	)			

Gather/Scatter

- Compare buffers to reduce
- Scatter: the `sendcount` / Gather: the `recvcount`:
  this is not, as you might expect, the total length of the buffer;
  instead, it is the amount of data to/from each process.

Gather pictured

Popular application of gather

Matrix is constructed distributed, but needs to be brought to one process:

This is not efficient in time or space. Do this only when strictly necessary.
Remember SPMD: try to keep everything symmetrically parallel.

MPI_Allgather

Name	Param name	Explanation	C type	F type
MPI_Allgather (				
	sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)
	sendcount	number of elements in send buffer	int	INTEGER
	sendtype	datatype of send buffer elements	MPI_Datatype	TYPE(MPI_Datatype)
	recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)
	recvcount	number of elements received from any process	int	INTEGER
	recvtype	datatype of receive buffer elements	MPI_Datatype	TYPE(MPI_Datatype)
	comm	communicator	MPI_Comm	TYPE(MPI_Comm)
	)			

Allgather pictured

V-type collectives

- Gather/scatter but with individual sizes
- Requires displacement in the gather/scatter buffer

MPI_Gatherv

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

Exercise 14 (scangather)

Take the code from exercise 15 and extend it to gather all local buffers onto rank zero. Since the local arrays are of differing lengths, this requires `MPI_Gatherv`.

How do you construct the lengths and displacements arrays?

Exercise 15 (scangather)

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

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

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

Review 1

An `MPI_Scatter` call puts the same data on each process

```
/poll "A scatter call puts the same data on each process" "T" "F"
```

All-to-all

- Every process does a scatter;
- (equivalently: every process gather)
- each individual data, but amounts are identical
- Example: data transposition in FFT

Data transposition

Example: each process knows who to send to,
all-to-all gives information who to receive from

All-to-allv

- Every process does a scatter or gather;
- each individual data and individual amounts.
- Example: radix sort by least-significant digit.

Radix sort

Sort 4 numbers on two processes:

	proc0		proc1		
array	2	5	7	1	
binary	010	101	111	001	
stage 1					
last digit	0	1	1	1	
	(this serves as bin number)				
sorted	010		101	111	001
stage 2					
next digit	1		0	1	0
	(this serves as bin number)				
sorted	101	001	010	111	
stage 3					
next digit	1	0	0	1	
	(this serves as bin number)				
sorted	001	010	101	111	
decimal	1	2	5	7	

Reduce-scatter

- Pointwise reduction (one element per process) followed by scatter
- Somewhat related to all-to-all: data transpose but reduced information, rather than gathered.
- Applications in both sparse and dense matrix-vector product.

Example: sparse matrix setup

Example: each process knows who to send to,
all-to-all gives information how many messages to expect
reduce-scatter leaves only relevant information

Barrier

```
int MPI_Barrier( MPI_Comm comm )
```

- Synchronize processes:
- each process waits at the barrier until all processes have reached the barrier
- **This routine is almost never needed**:
  collectives are already a barrier of sorts, two-sided communication is a local synchronization
- One conceivable use: timing

User-defined operators

MPI Operators

Define your own reduction operator

- Define operator between partial result and new operand

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

- Don't forget to free:

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

- Make your own reduction scheme `MPI_Reduce_local`

User defined operators, Fortran

```
FUNCTION user_function( invec(*), inoutvec(*), length, mpitype)
<fortrantype> :: invec(length), inoutvec(length)
INTEGER :: length, mpitype
```

MPI_Op_create

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

Example

Smallest nonzero:

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

Review 2

The $\| \cdot \|_2$ norm (sum of squares) needs a custom operator.

```
/poll "The sum of squares norm needs a custom operators" "T" "F"
```

Performance of collectives

Naive realization of collectives

Broadcast:

Single message:

$$\alpha = \text{message startup} \approx 10^{-6}s, \qquad \beta = \text{time per word} \approx 10^{-9}s$$

- Time for message of n words:

$$\alpha + \beta n$$

- Time for collective? Can you improve on that?

Better implementation of collective

- What is the running time now?
- Can you come up with lower bounds on the α, β terms? Are these achieved here?
- How about the case of really long buffers?

Implementation of Reduce

	$t = 1$	$t = 2$	$t = 3$
p_0	$x_0^{(0)}, x_1^{(0)}, x_2^{(0)}, x_3^{(0)}$	$x_0^{(0:1)}, x_1^{(0:1)}, x_2^{(0:1)}, x_3^{(0:1)}$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$
p_1	$x_0^{(1)} \uparrow, x_1^{(1)} \uparrow, x_2^{(1)} \uparrow, x_3^{(1)} \uparrow$		
p_2	$x_0^{(2)}, x_1^{(2)}, x_2^{(2)}, x_3^{(2)}$	$x_0^{(2:3)} \uparrow, x_1^{(2:3)} \uparrow, x_2^{(2:3)} \uparrow, x_3^{(2:3)} \uparrow$	
p_3	$x_0^{(3)} \uparrow, x_1^{(3)} \uparrow, x_2^{(3)} \uparrow, x_3^{(3)} \uparrow$		

Implementation of Allreduce

	$t = 1$	$t = 2$	$t = 3$
p_0	$x_0^{(0)} \downarrow, x_1^{(0)} \downarrow, x_2^{(0)} \downarrow, x_3^{(0)} \downarrow$	$x_0^{(0:1)} \downarrow\downarrow, x_1^{(0:1)} \downarrow\downarrow, x_2^{(0:1)} \downarrow\downarrow, x_3^{(0:1)} \downarrow\downarrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(}$
p_1	$x_0^{(1)} \uparrow, x_1^{(1)} \uparrow, x_2^{(1)} \uparrow, x_3^{(1)} \uparrow$	$x_0^{(0:1)} \downarrow\downarrow, x_1^{(0:1)} \downarrow\downarrow, x_2^{(0:1)} \downarrow\downarrow, x_3^{(0:1)} \downarrow\downarrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(}$
p_2	$x_0^{(2)} \downarrow, x_1^{(2)} \downarrow, x_2^{(2)} \downarrow, x_3^{(2)} \downarrow$	$x_0^{(2:3)} \uparrow\uparrow, x_1^{(2:3)} \uparrow\uparrow, x_2^{(2:3)} \uparrow\uparrow, x_3^{(2:3)} \uparrow\uparrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(}$
p_3	$x_0^{(3)} \uparrow, x_1^{(3)} \uparrow, x_2^{(3)} \uparrow, x_3^{(3)} \uparrow$	$x_0^{(2:3)} \uparrow\uparrow, x_1^{(2:3)} \uparrow\uparrow, x_2^{(2:3)} \uparrow\uparrow, x_3^{(2:3)} \uparrow\uparrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(}$

Review 3

True of false: there are collectives that do not communicate data

```
/poll "there are collectives that do not communicate data" "T" "F"
```

Reduction operators

User-defined operators

Given a reduction function:

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

create a new operator:

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

Exercise 16 (`onenorm`)

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

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

Point-to-point communication

Overview

This section concerns direct communication between two processes.
Discussion of distributed work, deadlock and other parallel phenomena.

Commands learned:

- `MPI_Send`, `MPI_Recv`, `MPI_Sendrecv`, `MPI_Isend`, `MPI_Irecv`
- `MPI_Wait...`
- Mention of `MPI_Test`, `MPI_Bsend`/*Ssend*/*Rsend*.

Point-to-point communication

MPI point-to-point mechanism

- Two-sided communication
- Matched send and receive calls
- One process sends to a specific other process
- Other process does a specific receive.

Ping-pong

A sends to B, B sends back to A

What is the code for A? For B?

Ping-pong in MPI

Remember SPMD:

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

MPI_Send

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

MPI_Recv

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

Status object

Use `MPI_STATUS_IGNORE` unless . . .

- Receive call can have various wildcards:
  `MPI_ANY_SOURCE`, `MPI_ANY_TAG`
- Receive buffer size is actually upper bound, not exact
- Use status object to retrieve actual description of the message

```
int s = status.MPI_SOURCE;
int t = status.MPI_TAG;
MPI_Get_count(status,MPI_FLOAT,&c);
```

Exercise 17 (pingpong)

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

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

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

Distributed data

Distributed data

Distributed array: each process stores disjoint local part

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

Local numbering $0, \ldots, n_{\text{local}}$;
global numbering is 'in your mind'.

Local and global indexing

Every local array starts at 0 (Fortran: 1);
you have to translate that yourself to global numbering:

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

Exercise (optional) 18

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

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

which we approximate by

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

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

Load balancing

If the distributed array is not perfectly divisible:

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

This gives a load balancing problem. Better solution?

(for future reference)

Let

$$f(i) = \lfloor iN/p \rfloor$$

and give process i the points $f(i)$ up to $f(i+1)$.
Result:

$$\lfloor N/p \rfloor \leq f(i+1) - f(i) \leq \lceil N/p \rceil$$

Local information exchange

Motivation

Partial differential equations:

$-\Delta u = -u_{xx}(\bar{x}) - u_{yy}(\bar{x}) = f(\bar{x})$ for $\bar{x} \in \Omega = [0,1]^2$ with $u(\bar{x}) = u_0$ on $\delta\Omega$.

Simple case:

$$-u_{xx} = f(x).$$

Finite difference approximation:

$$\frac{2u(x) - u(x+h) - u(x-h)}{h^2} = f(x, u(x), u'(x)) + O(h^2),$$

Finite dimensional: $u_i \equiv u(ih)$.

Motivation (continued)

Equations

$$\begin{cases} -u_{i-1} + 2u_i - u_{i+1} &= h^2 f(x_i) \qquad 1 < i < n \\ 2u_1 - u_2 &= h^2 f(x_1) + u_0 \\ 2u_n - u_{n-1} &= h^2 f(x_n) + u_{n+1}. \end{cases}$$

$$\begin{pmatrix} 2 & -1 & & \emptyset \\ -1 & 2 & -1 & \\ \emptyset & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} h^2 f_1 + u_0 \\ h^2 f_2 \\ \vdots \end{pmatrix} \tag{1}$$

So we are interested in sparse/banded matrices.

Matrix vector product

Most common operation: matrix vector product

$$y \leftarrow Ax, \qquad A = \begin{pmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \end{pmatrix}$$

- Component operation: $y_i = 2x_i - x_{i-1} - x_{i+1}$
- Parallel execution: each process has range of i-coordinates
- $\Rightarrow$ segment of vector, block row of matrix

Partitioned matrix-vector product

We need a point-to-point mechanism:

each process with ones before/after it.

Operating on distributed data

Array of numbers $x_i : i = 0, \ldots, N$
compute

$$y_i = -x_{i-1} + 2x_i - x_{i+1} : i = 1, \ldots, N-1$$

'owner computes'
This leads to communication:

so we need a point-to-point mechanism.

Blocking communication

Blocking send/recv

`MPI_Send` and `MPI_Recv` are *blocking* operations:

- The process waits ('blocks') until the operation is concluded.
- A send can not complete until the receive executes.

Ideal vs actual send/recv behaviour.

Deadlock

Exchange between two processes:

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

A subtlety.
This code may actually work:

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

Small messages get sent even if there is no corresponding receive.
(Often a system parameter)

Protocol

Communication is a ‘rendez-vous’ or ‘hand-shake’ protocol:

- Sender: ‘I have data for you’
- Receiver: ‘I have a buffer ready, send it over’
- Sender: ‘Ok, here it comes’
- Receiver: ‘Got it.’

Small messages bypass this: ‘eager’ send.
Definition of ‘small message’ controlled by environment variables:
`I_MPI_EAGER_THRESHOLD` `MV2_IBA_EAGER_THRESHOLD`

Exercise 19

(Classroom exercise) Each student holds a piece of paper in the right hand – keep your left hand behind your back – and we want to execute:

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

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

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

TAU trace: serialization

The problem here...

Here you have a case of a program that computes the right output, just way too slow.

Beware! Blocking sends/receives can be trouble.
(How would you solve this particular case?)

Food for thought: what happens if you flip the send and receive call?

Exercise (optional) 20

Implement the above algorithm using `MPI_Send` and `MPI_Recv` calls. Run the code, and use TAU to reproduce the trace output of figure 151. If you don't have TAU, can you show this serialization behavior using timings, for instance running it on an increasing number of processes?

Pairwise exchange

Operating on distributed data

Take another look:

$$y_i = x_{i-1} + x_i + x_{i+1} : i = 1, \ldots, N-1$$

- One-dimensional data and linear process numbering;
- Operation between neighboring indices: communication between neighboring processes.

Two steps

First do all the data movement to the right, later to the left.

- Each process does a send and receive
- So everyone does the send, then the receive? We just saw the problem with that.
- Better solution coming up!

Sendrecv

Instead of separate send and receive: use

`MPI_Sendrecv`

Combined calling sequence of send and receive;
execute such that no deadlock or sequentialization.

(Also: `MPI_Sendrecv_replace` with single buffer.)

MPI_Sendrecv

Name	Param name	Explanation	C type	F type
MPI_Sendrecv (				
	sendbuf	initial address of send buffer	const void*	TYPE(*), DIMENSION(..)
	sendcount	number of elements in send buffer	int	INTEGER
	sendtype	type of elements in send buffer	MPI_Datatype	TYPE(MPI_Datatype)
	dest	rank of destination	int	INTEGER
	sendtag	send tag	int	INTEGER
	recvbuf	initial address of receive buffer	void*	TYPE(*), DIMENSION(..)
	recvcount	number of elements in receive buffer	int	INTEGER
	recvtype	type of elements receive buffer element	MPI_Datatype	TYPE(MPI_Datatype)
	source	rank of source or MPI_ANY_SOURCE	int	INTEGER
	recvtag	receive tag or MPI_ANY_TAG	int	INTEGER
	comm	communicator	MPI_Comm	TYPE(MPI_Comm)
	status	status object	MPI_Status*	TYPE(MPI_Status)
	)			

SPMD picture

What does process p do?

Sendrecv with incomplete pairs

```
MPI_Comm_rank( .... &procno );
if ( /* I am not the first process */ )
  predecessor = procno-1;
else
  predecessor = MPI_PROC_NULL;

if ( /* I am not the last process */ )
  successor = procno+1;
else
  successor = MPI_PROC_NULL;

sendrecv(from=predecessor,to=successor);
```

(Receive from `MPI_PROC_NULL` succeeds without altering the receive buffer.)

Proc null

Use *mpl::proc_null*.

A point of programming style

The previous slide had:

- a conditional for computing the sender and receiver rank;
- a single Sendrecv call.

Also possible:

```
if ( /* i am first */ )
  Sendrecv( to=right, from=NULL );
else if ( /* i am last */
  Sendrecv( to=NULL,  from=left );
else
  Sendrecv( to=right, from=left );
```

```
if ( /* i am first */ )
  Send( to=right );
else if ( /* i am last */
  Recv( from=left );
else
  Sendrecv( to=right, from=left );
```

But:
Code duplication is error-prone, also chance of deadlock by missing a case

Exercise (optional) 21 (`rightsend`)

Revisit exercise 19 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.

Exercise 22 (sendrecv)

Implement the above three-point combination scheme using `MPI_Sendrecv`; every processor only has a single number to send to its neighbor.

Odd-even transposition sort

Odd-even transposition sort on 4 elements.

Exercise (optional) 23

A very simple sorting algorithm is *swap sort* or *odd-even transposition sort*: pairs of processors compare data, and if necessary exchange. The elementary step is called a *compare-and-swap*: in a pair of processors each sends their data to the other; one keeps the minimum values, and the other the maximum. For simplicity, in this exercise we give each processor just a single number.

The transposition sort algorithm is split in even and odd stages, where in the even stage processors $2i$ and $2i+1$ compare and swap data, and in the odd stage processors $2i+1$ and $2i+2$ compare and swap. You need to repeat this $P/2$ times, where P is the number of processors; see figure 165.

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.

Bucket brigade

Sometimes you really want to pass information from one process to the next: 'bucket brigade'

Exercise 24 (bucketblock)

Take the code of exercise 20 and modify it so that the data from process zero gets propagated to every process. Specifically, compute all partial sums $\sum_{i=0}^{p} i^2$:

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

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

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

Irregular exchanges: non-blocking communication

Sending with irregular connections

Graph operations:

Communicating other than in pairs

PDE, 2D case

A difference stencil applied to a two-dimensional square domain, distributed over processes. A cross-process connection is indicated ⇒ complicated to express pairwise

PDE matrix

$$
A = \left(\begin{array}{cccccc|cccccc|cc}
4 & -1 & & & & \emptyset & -1 & & & & & \emptyset & & \\
-1 & 4 & -1 & & & & & -1 & & & & & & \\
 & \ddots & \ddots & \ddots & & & & & \ddots & & & & & \\
 & & \ddots & \ddots & -1 & & & & & \ddots & & & & \\
\emptyset & & & -1 & 4 & & \emptyset & & & & & -1 & & \\
\hline
-1 & & & & & \emptyset & 4 & -1 & & & & & -1 & \\
 & -1 & & & & & -1 & 4 & -1 & & & & & -1 \\
 & \uparrow & \ddots & & & & \uparrow & \uparrow & \uparrow & & & & & \uparrow \\
 & k-n & & & & & k-1 & k & k+1 & & -1 & & & k+n \\
 & & & & & -1 & & & & -1 & 4 & & & \\
\hline
 & & & & & & \ddots & & & & & & \ddots &
\end{array}\right)
$$

Halo region

The halo region of a process, induced by a stencil

How do you approach this?

- It is very hard to figure out a send/receive sequence that does not deadlock or serialize
- Even if you manage that, you may have process idle time.

Instead:

- Declare 'this data needs to be sent' or 'these messages are expected', and
- then wait for them collectively.

Non-blocking send/recv

- `MPI_Isend` / `MPI_Irecv` does not send/receive:
- They declare a buffer.
- The buffer contents are there after a wait call.
- In between the `MPI_Isend` and `MPI_Wait` the data may not have been sent.
- In between the `MPI_Irecv` and `MPI_Wait` the data may not have arrived.

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

Syntax

Very much like blocking MPI_Send/MPI_Recv:

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

Basic wait:

```
MPI_Wait( MPI_Request*, MPI_Status* );
```

Most common way of waiting for completion:

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

- ignore status: MPI_STATUSES_IGNORE
- also MPI_Wait, MPI_Waitany, MPI_Waitsome

Exercise 25 (`isendirecv`)

Now use nonblocking send/receive routines to implement the three-point averaging operation

$$y_i = \big(x_{i-1} + x_i + x_{i+1}\big)/3 \colon i = 1, \ldots, N-1$$

on a distributed array. There are two approaches to the first and last process:

1. you can use `MPI_PROC_NULL` for the 'missing' communications;
2. you can skip these communications altogether, but now you have to count the requests carefully.

(Can you think of a different way of handling the end points?)

Comparison

- Obvious: blocking vs non-blocking behaviour.
- Buffer reuse: when a blocking call returns, the buffer is safe for reuse or free;
- A buffer in a non-blocking call can only be reused/freed after the wait call.

Buffer use in blocking/non-blocking case

Blocking:

```
double *buffer;
// allocate the buffer
for ( ... p ... ) {
   buffer = // fill in the data
   MPI_Send( buffer, ... /* to: */ p );
```

Non-blocking:

```
double **buffers;
// allocate the buffers
for ( ... p ... ) {
   buffers[p] = // fill in the data
   MPI_Isend( buffers[p], ... /* to: */ p );
MPI_Waitsomething(.....)
```

Pitfalls

- Strictly one request/wait per `isend`/`irecv`:
  can not use one request for multiple simultaneous `isends`
- Some people argue:

  > *Wait for the send is not necessary: if you wait for the receive, the message has arrived safely*

  This leads to memory leaks! The `wait` call deallocates the request object.

Matrices in parallel

$$y \leftarrow Ax$$

and A, x, y all distributed:

Hiding the halo

Interior of a process domain can overlap with halo transfer:

Latency hiding

Other motivation for non-blocking calls:
overlap of computation and communication, provided hardware support.

Also known as 'latency hiding'.

Example: three-point combination operation (see above):

1. Start communication for edge points,
2. Do local operations while communication goes on,
3. Wait for edge points from neighbor processes
4. Incorporate incoming data.

Exercise 26 (`isendirecvarray`)

Take your code of exercise 25 and modify it to use latency hiding. Operations that can be performed without needing data from neighbors should be performed in between the `MPI_Isend` / `MPI_Irecv` calls and the corresponding `MPI_Wait` calls.

Write your code so that it can achieve latency hiding.

Mix and match

You can match blocking and non-blocking:

```
if ( /* I am Process A */ ) {
  MPI_Irecv( /* from B */, &req1 );
  MPI_Isend( /* to B */, &req2 );
  MPI_Waitall( /* requests 1 and 2 */ );
} else if ( /* I am Process B */ ) {
  MPI_Recv( /* from A */ );
  MPI_Send( /* to A */ );
}
```

Test: non-blocking wait

- Post non-blocking receives
- test on the request(s) for incoming messages
- if nothing comes in, do local work

```
while (1) {
  MPI_Test( some_request, &flag );
  if (flag)
    // do something with incoming message
  else
    // do local work
}
```

Local operation.
Also `MPI_Testall` et cetera.

Probe for message

Is there a message?

```
// 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);
}
```

(Also non-blocking `MPI_Iprobe`.)
These commands force global progress.

The Pipeline Pattern

- Remember the bucket brigade: data propagating through processes
- If you have many buckets being passed: pipeline
- This is very parallel: only filling and draining the pipeline is not completely parallel
- Application to long-vector broadcast: pipelining gives overlap

Exercise (optional) 27 (`bucketpipenonblock`)

Implement a pipelined broadcast for long vectors:
use non-blocking communication to send the vector in parts.

Exercise 28 (`setdiff`)

Create two distributed arrays of positive integers. Take the set difference of the two: the first array needs to be transformed to remove from it those numbers that are in the second array.

How could you solve this with an `MPI_Allgather` call? Why is it not a good idea to do so? Solve this exercise instead with a circular bucket brigade algorithm.

Consider: `MPI_Send` and `MPI_Recv` vs `MPI_Sendrecv` vs `MPI_Sendrecv_replace` vs `MPI_Isend` and `MPI_Irecv`

The wheel of reinvention

The circular bucket brigade is the idea behind the 'Horovod' library, which is the key to efficient parallel Deep Learning.

More sends and receive

- `MPI_Bsend`, `MPI_Ibsend`: buffered send
- `MPI_Ssend`, `MPI_Issend`: synchronous send
- `MPI_Rsend`, `MPI_Irsend`: ready send
- Persistent communication: repeated instance of same proc/data description.

  MPI-4:

  - *Partitioned sends.*

too obscure to go into.

Review 4

Does this code deadlock?

```
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);
```

/poll "This code deadlocks" "Yes" "No" "Maybe"

Review 5

Does this code deadlock?

```
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);
```

/poll "This code deadlocks" "Yes" "No" "Maybe"

Review 6

Does this code deadlock?

```
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);
```

/poll "This code deadlocks" "Yes" "No" "Maybe"

Review 7

Does this code deadlock?

```
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);
```

/poll "This code deadlocks" "Yes" "No" "Maybe"

Where to go from here...

- Derived data types: send strided/irregular/inhomogeneous data
- Sub-communicators: work with subsets of `MPI_COMM_WORLD`
- I/O: efficient file operations
- One-sided communication: 'just' put/get the data somewhere
- Process management
- Non-blocking collectives
- Graph topology and neighborhood collectives
- Shared memory

Part

Justification

MPI basic concepts suffice for many applications. The Intermediate Topics section deals with more complicated data, process groups, file I/O, and the basics of one-sided communication.

Derived Datatypes

Overview

In this section you will learn about derived data types.

Commands learned:

- `MPI_Type_contiguous`/`vector`/`indexed`/`struct`
  `MPI_Type_create_subarray`
- `MPI_Pack` / `MPI_Unpack`
- F90 types

Discussion

Motivation: datatypes in MPI

All examples so far:

- contiguous buffer
- elements of single type

We need data structures with gaps, or heterogeneous types.

- Send real or imaginary parts out of complex array.
- Gather/scatter cyclicly.
- Send *struct* or *Type* data.

MPI allows for recursive construction of data types.

Datatype topics

- Elementary types: built-in.
- Derived types: user-defined.
- Packed data: not really a datatype.

Datatypes

Elementary datatypes

C/C++	Fortran
MPI_CHAR	MPI_CHARACTER
MPI_UNSIGNED_CHAR	
MPI_SIGNED_CHAR	
	MPI_LOGICAL
MPI_SHORT	
MPI_UNSIGNED_SHORT	
MPI_INT	MPI_INTEGER
MPI_UNSIGNED	
MPI_LONG	
MPI_UNSIGNED_LONG	
MPI_FLOAT	MPI_REAL
MPI_DOUBLE	MPI_DOUBLE_PRECISION
MPI_LONG_DOUBLE	
	MPI_COMPLEX
	MPI_DOUBLE_COMPLEX

How to use derived types

Create, commit, use, free:

```
MPI_Datatype newtype;
MPI_Type_xxx( ... oldtype ... &newtype);
MPI_Type_commit ( &newtype );

  // code using the new type

MPI_Type_free ( &newtype );
```

```
Type(MPI_Datatype) :: newtype ! F2008
Integer            :: newtype ! F90
```

The *oldtype* can be elementary or derived.
Recursively constructed types.

Contiguous type

```
int MPI_Type_contiguous(
  int count, MPI_Datatype old_type, MPI_Datatype *new_type_p)
```

This one is indistinguishable from just sending *count* instances of the *old_type*.

Example: non-contiguous data

Matrix in column storage:

- Columns are contiguous
- Rows are not contiguous

Logical:

Physical:

Vector type

```
int MPI_Type_vector(
  int count, int blocklength, int stride,
  MPI_Datatype old_type, MPI_Datatype *newtype_p
);
```

Used to pick a regular subset of elements from an array.

Different send and receive types

Send and receive type can differ. Example:
Sender type: vector
receiver type: contiguous or elementary

Receiver has no knowledge of the stride of the sender.

Send vs recv type

```
// 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);
}
```

Illustration of the next exercise

Sending strided data from process zero to all others

Exercise 29 (`stridesend`)

Let processor 0 have an array x of length $10P$, where P is the number of processors. Elements $0, P, 2P, \ldots, 9P$ should go to processor zero, $1, P+1, 2P+1, \ldots$ 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 30

Allocate a matrix on processor zero, using Fortran column-major storage. Using P sendrecv calls, distribute the rows of this matrix among the processors.

Indexed type

```
int MPI_Type_indexed(
  int count, int blocklens[], int displacements[],
  MPI_Datatype old_type, MPI_Datatype *newtype);
```

Hindexed type

Similar to indexed but using byte offsets:
explicit memory address.

Example usage scenario: send linked list.
Use `MPI_Get_address`

Heterogeneous: Structure type

```
int MPI_Type_create_struct(
  int count, int blocklengths[], MPI_Aint displacements[],
  MPI_Datatype types[], MPI_Datatype *newtype);
```

This gets very tedious. . .

Subarray type

Submatrix storage

- Location of first element
- Stride, blocksize

BLAS/Lapack storage

Three parameter description:

Subarray type

- Vector type is convenient for 2D subarrays,
- it gets tedious in higher dimensions.
- Better solution: `MPI_Type_create_subarray`

```
MPI_Type_create_subarray(
    ndims, array_of_sizes, array_of_subsizes,
    array_of_starts, order, oldtype, newtype)
```

Subtle: data does not start at the buffer start

Exercise 31 (cubegather)

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.

If you haven't started `idev` with the right number of processes, use

```
ibrun -np 27 cubegather
```

Normally you use `ibrun` without process count argument.

Fortran 'kind' types

Check out MPI_Type_create_f90_integer, MPI_Type_create_f90_real, MPI_Type_create_f90_complex

Example:

```
REAL ( KIND = SELECTED_REAL_KIND(15 ,300) ) , &
 DIMENSION(100) :: array
Type(MPI_Datatype) :: realtype
CALL MPI_Type_create_f90_real( 15 , 300 , realtype , error )
```

Extent and resizing

Extent

Extent: ‘size’ of a type,
especially useful for derived types.

Extent resizing: enlarging

Multiple derived types may not be what you intended
extent resizing makes it artificually larger:

Extent computation

Use MPI_Type_get_extent to query extent
note: parameters are measured in bytes.

```
MPI_Aint lb,asize;
MPI_Type_vector(count,bs,stride,MPI_DOUBLE,&newtype);
MPI_Type_commit(&newtype);
MPI_Type_get_extent(newtype,&lb,&asize);
ASSERT( lb==0 );
ASSERT( asize==((count-1)*stride+bs)*sizeof(double) );
MPI_Type_free(&newtype);
```

Naive code

Send multiple derived types from

0 1 2 3 4 5 6 7 8 9 10

```
// vectorpadsend.c
for (int i=0; i<max_elements; i++) sendbuffer[i] = i;
MPI_Type_vector(count,blocklength,stride,MPI_INT,&stridetype);
MPI_Type_commit(&stridetype);
MPI_Send( sendbuffer,ntypes,stridetype, receiver,0, comm );
```

Receive as single block:

```
MPI_Recv( recvbuffer,max_elements,MPI_INT, sender,0, comm,&status );
int count; MPI_Get_count(&status,MPI_INT,&count);
printf("Receive %d elements:",count);
for (int i=0; i<count; i++) printf(" %d",recvbuffer[i]);
printf("\n");
```

giving an output of:

```
Receive 6 elements: 0 2 4 5 7 9
```

Resizing code

Extend the vector type with padding:

```
MPI_Type_get_extent(stridetype,&l,&e);
printf("Stride type l=%ld e=%ld\n",l,e);
e += ( stride-blocklength) * sizeof(int);
MPI_Type_create_resized(stridetype,l,e,&paddedtype);
MPI_Type_get_extent(paddedtype,&l,&e);
printf("Padded type l=%ld e=%ld\n",l,e);
MPI_Type_commit(&paddedtype);
MPI_Send( sendbuffer,ntypes,paddedtype, receiver,0, comm );
```

giving:

```
Strided type l=0 e=20
Padded  type l=0 e=24
Receive 6 elements: 0 2 4 6 8 10
```

Extent resizing: shrinking

Elements are placed at distance equal to extent:

Exercise 32 (`stridescatter`)

Change the `stridesend` code to use a scatter call, rather than a sequence of sends.

Extent of subarray type

The ‘subarray’ type:
data does not start at the start of the type.

`MPI_Type_get_true_extent` returns non-zero lower bound.

Transposition

- Basic block of FFT
- Before: Each process stores a block column,
- After: Each process stores a block row

in a picture

Exercise 33 (`transposeblock`)

Fill in the missing bits of the skeleton code.

Packed data

Packing into buffer

```c
int MPI_Pack(
  void *inbuf, int incount, MPI_Datatype datatype,
  void *outbuf, int outcount, int *position,
  MPI_Comm comm);

int MPI_Unpack(
  void *inbuf, int insize, int *position,
  void *outbuf, int outcount, MPI_Datatype datatype,
  MPI_Comm comm);
```

Example

```
if (procno==sender) {
  position = 0;
  MPI_Pack(&nsends,1,MPI_INT,
           buffer,buflen,&position,comm);
  for (int i=0; i<nsends; i++) {
    double value = rand()/(double)RAND_MAX;
    printf("[%d] pack %e\n",procno,value);
    MPI_Pack(&value,1,MPI_DOUBLE,
             buffer,buflen,&position,comm);
  }
  MPI_Pack(&nsends,1,MPI_INT,
           buffer,buflen,&position,comm);
  MPI_Send(buffer,position,MPI_PACKED,other,0,comm);
} else if (procno==receiver) {
  int irecv_value;
  double xrecv_value;
  MPI_Recv(buffer,buflen,MPI_PACKED,other,0,
           comm,MPI_STATUS_IGNORE);
  position = 0;
  MPI_Unpack(buffer,buflen,&position,
             &nsends,1,MPI_INT,comm);
  for (int i=0; i<nsends; i++) {
    MPI_Unpack(buffer,buflen,
```

Communicator manipulations

Overview

In this section you will learn about various subcommunicators.

Commands learned:

- `MPI_Comm_dup`, discussion of library design
- `MPI_Comm_split`
- discussion of groups
- discussion of inter/intra communicators.

Sub-computations

Simultaneous groups of processes, doing different tasks, but loosely interacting:

- Simulation pipeline: produce input data, run simulation, post-process.
- Climate model: separate groups for air, ocean, land, ice.
- Quicksort: split data in two, run quicksort independently on the halves.
- Process grid: do broadcast in each column.

New communicators are formed recursively from `MPI_COMM_WORLD`.

Communicator duplication

Simplest new communicator: identical to a previous one.

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

This is useful for library writers:

```
MPI_Isend(...); MPI_Irecv(...);
// library call
MPI_Waitall(...);
```

- Naively, the library can 'catch' the user messages.
- With a duplicate communicator there is no confusion:
  user and library both have their own 'context' for their messages.

Interleaved library and user code

```
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();
```

Library internally has messages

```
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;
}
```

Wrong way of setting up the library

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

Right way of setting up the library

```
// commdupright.cxx
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();
};
```

Disjoint splitting

Split a communicator in multiple disjoint others.

Give each process a ‘color’, group processes by color:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
```

(`key` determines ordering: use rank unless you want special effects)

Row/column example

Simulate a process grid
create subcommunicator per column (or row)
assume processes numbered by rows

```
MPI_Comm_rank( MPI_COMM_WORLD, &procno );
proc_i = procno / proc_column_length;
proc_j = procno % proc_column_length;

MPI_Comm column_comm;
MPI_Comm_split( MPI_COMM_WORLD, proc_j, procno, &column_comm );

MPI_Bcast( data, ... column_comm );
```

Food for thought: there are many columns, but only one `column_comm` variable. Why?

Row and column communicators

Row and column broadcasts in subcommunicators

Exercise 34 (procgrid)

Organize your processes in a grid, and make subcommunicators for the rows and columns. For this compute the row and column number of each process.

In the row and column communicator, compute the rank. For instance, on a 2×3 processor grid you should find:

```
Global ranks:  Ranks in row:  Ranks in colum:
  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. This is one occasion where you could use `ibrun -np 9`; normally you would *never* put a processor count on `ibrun`.

Exercise 35

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.

Splitting by shared memory

- `MPI_Comm_split_type` splits into communicators of same type.
- MPI-3: only `MPI_COMM_TYPE_SHARED` splitting by shared memory.
- MPI-4: `MPI_COMM_TYPE_HW_GUIDED` split using an `info` value from `MPI_Get_hw_resource_types`.

```
// 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);
```

Inter-communicators

- Communicators so far are of *intra-communicator* type.
- Bridge between two communicators: *inter-communicator*.
- Example: communicator with newly spawned processes

In a picture

Illustration of ranks in an inter-communicator setup

```
// intercomm.c
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 );
```

Concepts

- Two local communicators
- The ‘peer’ communicator that contains them
- Leaders in each of them
- An inter-communicator over the leaders.

Routines

- `MPI_Intercomm_create`: create
- `MPI_Comm_get_parent`: the other leader (see process management)
- `MPI_Comm_remote_size`, `MPI_Comm_remote_group`: query the other communicator
- `MPI_Comm_test_inter`: is this an inter or intra?

More

- Non-disjoint subcommunicators through process groups.
- Process topologies: cartesian and graph.
  There will also be a section about this, later.

Cartesian topologies

Cartesian decomposition

Code:

```
// cartdims.c
int *dimensions = (int*)malloc(dim*sizeof(int));
for (int idim=0; idim<dim; idim++)
  dimensions[idim] = 0;
MPI_Dims_create(nprocs,dim,dimensions);
```

Output:

```
mpicc -o cartdims cartdims.o
Cartesian grid size: 3 dim: 1
 3
Cartesian grid size: 3 dim: 2
 3 x  1
Cartesian grid size: 4 dim: 1
 4
Cartesian grid size: 4 dim: 2
 2 x  2
Cartesian grid size: 4 dim: 3
 2 x  2 x  1
Cartesian grid size: 12 dim: 1
12
Cartesian grid size:
```

Create/test Cartesian topology

```
MPI_Comm cart_comm;
int *periods = (int*) malloc(dim*sizeof(int));
for ( int id=0; id<dim; id++ ) periods[id] = 0;
MPI_Cart_create
  ( comm,dim,dimensions,periods,
    0,&cart_comm );
```

```
int dim;
MPI_Cartdim_get( cart_comm,&dim );
int *dimensions = (int*) malloc(dim*sizeof(int));
int *periods    = (int*) malloc(dim*sizeof(int));
int *coords     = (int*) malloc(dim*sizeof(int));
MPI_Cart_get( cart_comm,dim,dimensions,periods,coords );
```

Rank translation

```
// cart.c
MPI_Comm comm2d;
int periodic[ndim]; periodic[0] = periodic[1] = 0;
MPI_Cart_create(comm,ndim,dimensions,periodic,1,&comm2d);
if (comm2d==MPI_COMM_NULL) {
  printf("Process %d not included\n",procno);
} else {
  MPI_Cart_coords(comm2d,procno,ndim,coord_2d);
  MPI_Cart_rank(comm2d,coord_2d,&rank_2d);
  printf("I am %d: (%d,%d); originally %d\n",
           rank_2d,coord_2d[0],coord_2d[1],procno);
```

Cartesian communication

```
// cartcoord.c
for ( int id=0; id<dim; id++)
  periods[id] = id==0 ? 1 : 0;
MPI_Cart_create
  ( comm,dim,dimensions,periods,
    0,&period_comm );
```

Code:

```
int pred,succ;
MPI_Cart_shift
  (period_comm,/* dim: */ 0,/* up: */ 1,
   &pred,&succ);
printf
  ("periodic dimension 0:\n  src=%d,
    ↪tgt=%d\n",
   pred,succ);
MPI_Cart_shift
  (period_comm,/* dim: */ 1,/* up: */ 1,
   &pred,&succ);
printf
  ("non-periodic dimension 1:\n  src=%d,
    ↪tgt=%d\n",
   pred,succ);
```

Output:

```
Grid of size 6 in 3
    ↪dimensions:
   3 x  2 x  1
Shifting process 0.
periodic dimension 0:
  src=4, tgt=2
non-periodic dimension
    ↪1:
  src=-1, tgt=1
```

Subgrids

Code:

```
MPI_Cart_sub(
    ↪period_comm,remain,&hyperplane );
if (procno==0) {
  MPI_Topo_test( hyperplane,&topo_type );
  MPI_Cartdim_get( hyperplane,&hyperdim );
  printf("hyperplane has dimension %d,
     ↪type %d\n",
         hyperdim,topo_type);
  MPI_Cart_get(
     ↪hyperplane,dim,dims,period,coords );
  printf("  periodic: ");
  for (int id=0; id<2; id++)
    printf("%d,",period[id]);
  printf("\n");
```

Output:

```
Grid of size 6 in 3
    ↪dimensions:
   3 x  2 x  1
hyperplane has
    ↪dimension 2,
    ↪type 2
  periodic: 1,0,
```

Exercise 36 (`isendirecvcart`)

Use Cartesian topology routines to extend exercise 26 to two dimensions.

MPI File I/O

Overview

This section discusses parallel I/O. What is the problem with regular I/O in parallel?

Commands learned:

- `MPI_File_open`/*`write`*/*`close`* and variants
- parallel file pointer routines: `MPI_File_set_view`/*`write_at`*

The trouble with parallel I/O

- Multiple process reads from one file: no problem.
- Multiple writes to one file: big problem.
- Everyone writes to separate file: stress on the file system, and requires post-processing.

MPI I/O

- Part of MPI since MPI-2
- Joint creation of one file from bunch of processes.
- You could also use `hdf5`, `netcdf`, `silo` ...

The usual bits

```
MPI_File mpifile;
MPI_File_open(comm,"blockwrite.dat",
        MPI_MODE_CREATE | MPI_MODE_WRONLY,MPI_INFO_NULL,
        &mpifile);
if (procno==0) {
  MPI_File_write
    (mpifile,output_data,nwords,MPI_INT,MPI_STATUS_IGNORE);
}
MPI_File_close(&mpifile);
```

```
type(MPI_File) :: mpifile ! F08
integer        :: mpifile ! F90
```

How do you make it unique for a process?

```
MPI_File_write_at
   (mpifile,offset,output_data,nwords,
    MPI_INT,MPI_STATUS_IGNORE);
```

or

```
MPI_File_set_view
  (mpifile,
   offset,datatype,
   MPI_INT,"native",MPI_INFO_NULL);
MPI_File_write // no offset, we have a view
  (mpifile,output_data,nwords,MPI_INT,MPI_STATUS_IGNORE);
```

Write at an offset

Write to a view

Write to a view

Made

Exercise 37 (`blockwrite`)

The given code works for one writing process. Compute a unique offset for each process (in bytes!) so that all the local arrays are placed in the output file in sequence.

Exercise 38 (viewwrite)

Solve the previous exercise by using `MPI_File_write` (that is, without offset), but by using `MPI_File_set_view` to specify the location.

Exercise 39 (`scatterwrite`)

Now write the local arrays cyclically to the file: with 5 processes and 3 elements per process the file should contain

```
1 4 7 10 13 | 2 5 8 11 14 | 3 6 9 12 15
```

Do this by defining a vector derived type and setting that as the file view.

One-sided communication

Overview

This section concernes one-sided operations, which allows 'shared memory' type programming. (Actual shared memory later.)

Commands learned:

- `MPI_Put`, `MPI_Get`, `MPI_Accumulate`
- Window commands: `MPI_Win_create`, `MPI_Win_allocate`
- Active target synchronization `MPI_Win_fence`
- `MPI_Win_post`/`wait`/*start*/*complete*
- Passive target synchronization `MPI_Win_lock` / `MPI_Win_lock`
- Atomic operations: `MPI_Fetch_and_op`

Basic mechanisms

Motivation

With two-sided messaging, you can not just put data on a different process: the other has to expect it and receive it.

- Sparse matrix: it is easy to know what you are receiving, not what you need to send. Usually solved with complicated preprocessing step.
- Neuron simulation: spiking neuron propagates information to neighbors. Uncertain when this happens.
- Other irregular data structures: distributed hash tables.

Dynamic data

```
x = f();
p = hash(x);
MPI_Send( x, /* to: */ p );
```

Problem: how does p know to post a receive, and how does everyone else know not to?

One-sided concepts

- A process has a *window* that other processes can access.
- *origin*: process doing a one-sided call
  target: process being accessed.
- One-sided calls: `MPI_Put`, `MPI_Get`, `MPI_Accumulate`.
- Various synchronization mechanisms.

Window creation

```
MPI_Win_create (void *base, MPI_Aint size,
  int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

- `size`: in bytes
- *disp_unit*: `sizeof(type)`

Also call `MPI_Win_free` when done. This is important!

Window allocation

Instead of passing buffer, let MPI allocate with `MPI_Win_allocate` and return the buffer pointer:

```
int MPI_Win_allocate
    (MPI_Aint size, int disp_unit, MPI_Info info,
     MPI_Comm comm, void *baseptr, MPI_Win *win)
```

can use dedicated fast memory.

Active target synchronization

All processes call `MPI_Win_fence`. Epoch is between fences:

```
MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
if (procno==producer)
  MPI_Put( /* operands */, win);
MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
```

Second fence indicates that one-sided communication is concluded: target knows that data has been put.

MPI_Put

Name	Param name	Explanation	C type	F type
MPI_Put (				
MPI_Put_c (				
	origin_addr	initial address of origin buffer	const void*	TYPE(*), DIMENSION(..)
	origin_count	number of entries in origin buffer	[int MPI_Count	INTEGER
	origin_datatype	datatype of each entry in origin buffer	MPI_Datatype	TYPE(MPI_Datatype)
	target_rank	rank of target	int	INTEGER
	target_disp	displacement from start of window to target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND
	target_count	number of entries in target buffer	[int MPI_Count	INTEGER
	target_datatype	datatype of each entry in target buffer	MPI_Datatype	TYPE(MPI_Datatype)
	win	window object used for communication	MPI_Win	TYPE(MPI_Win)
	)			

Location in the window

Location to write:

$$\texttt{window_base} + \texttt{target_disp} \times \texttt{disp_unit}.$$

Exercise 40 (`rightput`)

Revisit exercise 19 and solve it using `MPI_Put`.

Exercise 41 (`randomput`)

Write code where:

- process 0 computes a random number r
- if $r < .5$, zero writes in the window on 1;
- if $r \geq .5$, zero writes in the window on 2.

Exercise (optional) 42 (`randomput`)

Replace `MPI_Win_create` by `MPI_Win_allocate`.

Remaining simple routines: Get, Accumulate

- `MPI_Get` is converse of `MPI_Put`. Like Recv, but no status argument.
- `MPI_Accumulate` is a Put plus a reduction on the result: multiple accumulate calls in one epoch well-defined.
  Can use any predefined `MPI_Op` (not user-defined) or `MPI_REPLACE`.

MPI_Get

Name	Param name	Explanation	C type	F type
MPI_Get (				
MPI_Get_c (				
	origin_addr	initial address of origin buffer	void*	TYPE(*), DIMENSION(..)
	origin_count	number of entries in origin buffer	[int MPI_Count	INTEGER
	origin_datatype	datatype of each entry in origin buffer	MPI_Datatype	TYPE(MPI_Datatype)
	target_rank	rank of target	int	INTEGER
	target_disp	displacement from window start to the beginning of the target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND
	target_count	number of entries in target buffer	[int MPI_Count	INTEGER
	target_datatype	datatype of each entry in target buffer	MPI_Datatype	TYPE(MPI_Datatype)
	win	window object used for communication	MPI_Win	TYPE(MPI_Win)
	)			

MPI_Accumulate

Name	Param name	Explanation	C type	F type
MPI_Accumulate (				
MPI_Accumulate_c (				
	origin_addr	initial address of buffer	const void*	TYPE(*), DIMENSION(..)
	origin_count	number of entries in buffer	int / MPI_Count	INTEGER
	origin_datatype	datatype of each entry	MPI_Datatype	TYPE(MPI_Datatype)
	target_rank	rank of target	int	INTEGER
	target_disp	displacement from start of window to beginning of target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND
	target_count	number of entries in target buffer	int / MPI_Count	INTEGER
	target_datatype	datatype of each entry in target buffer	MPI_Datatype	TYPE(MPI_Datatype)
	op	reduce operation	MPI_Op	TYPE(MPI_Op)
	win	window object	MPI_Win	TYPE(MPI_Win)
	)			

Ordering and synchronization

Fence synchronization

Already mentioned active target synchronization:
the target indicates the start/end of an epoch.

Simplest mechanism: `MPI_Win_fence`, collective.

After the closing fence, buffers have been sent / windows have been updated.

Ordering of operations

Ordering is often undefined:

- No ordering of Get and Put/Accumulate operations
- No ordering of multiple Puts. Use 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 ordered by default.

Exercise (optional) 43 (`countdown`)

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?

A second active synchronization

Use `MPI_Win_post`, `MPI_Win_wait`, `MPI_Win_start`, `MPI_Win_complete` calls

More fine-grained than fences

Passive target synchronization

Passive target synchronization

Lock a window on the target:

```
MPI_Win_lock
   (int locktype, int rank, int assert, MPI_Win win)
MPI_Win_unlock
   (int rank, MPI_Win win)
```

with types: `MPI_LOCK_SHARED` `MPI_LOCK_EXCLUSIVE`

Justification

MPI-1/2 lacked tools for race condition-free one-sided communication.
These have been added in MPI-3.

Emulating shared memory with one-sided communicatio

- One process stores a table of work descriptors, and a 'stack pointer' stating how many there are.
- Each process reads the pointer, reads the corresponding descriptor, and decrements the pointer; and
- A process that has read a descriptor then executes the corresponding task.
- Non-collective behavior: processes only take a descriptor when they are available.

In a picture

Simplified model

- One process has a counter, which models the shared memory;
- Each process, if available, reads the counter; and
- . . . decrements the counter.
- No actual work: random decision if process is available.

Shared memory problems: what is a race condition?

Race condition: outward behavior depends on timing/synchronization of low-level events.
In shared memory associated with shared data.

Example:

Init: `I=0`
process 1: `I=I+2`
process 2: `I=I+3`

scenario 1.		scenario 2.		scenario 3.	
I = 0					
read I = 0	read I = 0	read I = 0	read I = 0	read I = 0	
local I = 2	local I = 3	local I = 2	local I = 3	local I = 2	
write I = 2			write I = 3	write I = 2	
	write I = 3	write I = 2			read I = 2
					local I = 5
					write I = 5
I = 3		I = 2		I = 5	

(In MPI, the read/write would be `MPI_Get` / `MPI_Put` calls)

Case study in shared memory: 1, wrong

```
// countdownput.c
MPI_Win_fence(0,the_window);
int counter_value;
MPI_Get( &counter_value,1,MPI_INT,
         counter_process,0,1,MPI_INT,
         the_window);
MPI_Win_fence(0,the_window);
if (i_am_available) {
  int decrement = -1;
  counter_value += decrement;
  MPI_Put
    ( &counter_value,   1,MPI_INT,
      counter_process,0,1,MPI_INT,
      the_window);
}
MPI_Win_fence(0,the_window);
```

Discussion

- The multiple `MPI_Put` calls conflict.
- Code is correct if in each iteration there is only one writer.
- Question: In that case, can we take out the middle fence?
- Question: what is wrong with

```
MPI_Win_fence(0,the_window);
if (i_am_available) {
  MPI_Get( &counter_value, ... )
  MPI_Win_fence(0,the_window);
  MPI_Put( ... )
}
MPI_Win_fence(0,the_window);
```

?

Case study in shared memory: 2, hm

```
// countdownacc.c
MPI_Win_fence(0,the_window);
int counter_value;
MPI_Get( &counter_value,1,MPI_INT,
         counter_process,0,1,MPI_INT,
         the_window);
MPI_Win_fence(0,the_window);
if (i_am_available) {
  int decrement = -1;
  MPI_Accumulate
    ( &decrement,       1,MPI_INT,
      counter_process,0,1,MPI_INT,
      MPI_SUM,
      the_window);
}
MPI_Win_fence(0,the_window);
```

Discussion: need for atomics

- `MPI_Accumulate` is atomic, so no conflicting writes.
- What is the problem?
- Answer: Processes are not reading unique *`counter_value`* values.
- Conclusion: Read and update need to come together:
  read unique value and immediately update.

Atomic 'get-and-set-with-no-one-coming-in-between':
`MPI_Fetch_and_op` / `MPI_Get_accumulate`.
Former is simple version: scalar only.

MPI_Fetch_and_op

Name	Param name	Explanation	C type	F type
MPI_Fetch_and_op (				
	origin_addr	initial address of buffer	const void*	TYPE(*), DIMENSION(..)
	result_addr	initial address of result buffer	void*	TYPE(*), DIMENSION(..)
	datatype	datatype of the entry in origin, result, and target buffers	MPI_Datatype	TYPE(MPI_Datatype)
	target_rank	rank of target	int	INTEGER
	target_disp	displacement from start of window to beginning of target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)
	op	reduce operation	MPI_Op	TYPE(MPI_Op)
	win	window object	MPI_Win	TYPE(MPI_Win)
	)			

Case study in shared memory: 3, good

```
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;
}
```

Allowable operators. (Hint!)

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

No user-defined operators.

Problem

We are using fences, which are collective.
What if a process is still operating on its local work?

Better (but more tricky) solution:
use passive target synchronization and locks.

Passive target epoch

```
if (rank == 0) {
  MPI_Win_lock (MPI_LOCK_EXCLUSIVE, 1, 0, win);
  MPI_Put (outbuf, n, MPI_INT, 1, 0, n, MPI_INT, win);
  MPI_Win_unlock (1, win);
}
```

No action on the target required!

Exercise 44 (lockfetch)

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 processs 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 process spins until the others have performed their update.

Use an atomic operation for the latter process to read out the shared value.
Can you replace the exclusive lock with a shared one?

Exercise 45 (`lockfetchshared`)

As exercise 44, 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`.

Big data communication

Overview

This section discusses big messages.

Commands learned:

- `MPI_Send_c`, `MPI_Allreduce_c`, `MPI_Get_count_c` etc. (MPI-4)

The problem with large messages

- There is no problem allocating large buffers:

```
size_t bigsize = 1<<33;
double *buffer =
    (double*) malloc(bigsize*sizeof(double));
```

- But you can not tell MPI how big the buffer is:

```
MPI_Send(buffer,bigsize,MPI_DOUBLE,...) // WRONG
```

because the size argument has to be `int`.

MPI 3 count type

Count type since MPI 3
C:

```
MPI_Count count;
```

Fortran:

```
Integer(kind=MPI_COUNT_KIND) :: count
```

Big enough for

- `int`;
- `MPI_Aint`, used in one-sided (and therefore big enough for *intptr_t* and *ptrdiff_t*);
- `MPI_Offset`, used in file I/O.

However, this type could not be used in MPI-3 to describe send buffers.

MPI 4 large count routines

C: routines with _c suffix

```
MPI_Count count;
MPI_Send_c( buff,count,MPI_INT, ... );
```

also MPI_Reduce_c, MPI_Get_c, ... (some 190 routines in all)

Fortran: polymorphism rules

```
Integer(kind=MPI_COUNT_KIND) :: count
call MPI_Send( buff,count, MPI_INTEGER, ... )
```

Big count example

```
// pingpongbig.c
assert( sizeof(MPI_Count)>4 );
for ( int power=3; power<=10; power++) {
  MPI_Count length=pow(10,power);
    buffer = (double*)malloc( length*sizeof(double) );
      MPI_Ssend_c
        (buffer,length,MPI_DOUBLE,
         processB,0,comm);
      MPI_Recv_c
        (buffer,length,MPI_DOUBLE,
         processB,0,comm,MPI_STATUS_IGNORE);
```

Same in F08

```fortran
!! pingpongbig.F90
  integer :: power,countbytes
  Integer(KIND=MPI_COUNT_KIND) :: length
  call MPI_Sizeof(length,countbytes,ierr)
  if (procno==0) &
       print *,"Bytes in count:",countbytes
        length = 10**power
        allocate( senddata(length),recvdata(length) )
           call MPI_Send(senddata,length,MPI_DOUBLE_PRECISION, &
                processB,0, comm)
           call MPI_Recv(recvdata,length,MPI_DOUBLE_PRECISION, &
                processB,0, comm,MPI_STATUS_IGNORE)
```

MPI_Send

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

MPI 4 large count querying

C:

```
MPI_Count count;
MPI_Get_count_c( &status,MPI_INT, &count );
MPI_Get_elements_c( &status,MPI_INT, &count );
```

Fortran:

```
Integer(kind=MPI_COUNT_KIND) :: count
call MPI_Get_count( status,MPI_INTEGER,count )
call MPI_Get_elements( status,MPI_INTEGER,count )
```

Compound messages

MPI-3 mechanism, deprecated in MPI-4.1:
send a number of contiguous types:

```
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);
```

By composing types you can make a 'big type'. Use
`MPI_Type_get_extent_x`, `MPI_Type_get_true_extent_x`, `MPI_Get_elements_x`
to query.

```
MPI_Count recv_count;
MPI_Get_elements_x(&recv_status,MPI_FLOAT,&recv_count);
```

Part

Justification

Recent additions to the MPI standard allow your code to deal with unusual scenarios or very large scale runs.

Advanced collectives

Non-blocking collectives

- Collectives are blocking.
- Compare blocking/non-blocking sends:

  `MPI_Send` → `MPI_Isend`

  immediate return of control, produce request object.
- Non-blocking collectives:

  `MPI_Bcast` → `MPI_Ibcast`

  Same:

```
MPI_Isomething( <usual arguments>, MPI_Request *req);
```

- Considerations:
  - Calls return immediately;
  - the usual story about buffer reuse
  - Requires `MPI_Wait...` for completion.
  - Multiple collectives can complete in any order
- Why?
  - Use for overlap communication/computation
  - Imbalance resilience
  - Allows pipelining

MPI_Ibcast

Name	Param name	Explanation	C type	F type
MPI_Ibcast (				
MPI_Ibcast_c (				
	buffer	starting address of buffer	void*	TYPE(*), DIMENSION(..)
	count	number of entries in buffer	int MPI_Count	INTEGER
	datatype	datatype of buffer	MPI_Datatype	TYPE(MPI_Datatype)
	root	rank of broadcast root	int	INTEGER
	comm	communicator	MPI_Comm	TYPE(MPI_Comm)
	request	communication request	MPI_Request*	TYPE(MPI_Request)
	)			

Overlapping collectives and computation

Independent collective and local operations:

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

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

Simultaneous reductions

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

$$\alpha \leftarrow x^t y$$
$$\beta \leftarrow \|z\|_\infty$$

which translates to:

```
MPI_Request reqs[2];
MPI_Iallreduce
   ( &local_xy,  &global_xy, 1,MPI_DOUBLE,MPI_SUM,comm,
     &(reqs[0]) );
MPI_Iallreduce
   ( &local_xinf,&global_xin,1,MPI_DOUBLE,MPI_MAX,comm,
     &(reqs[1]) );
MPI_Waitall(2,reqs,MPI_STATUSES_IGNORE);
```

Matching collectives

Blocking and non-blocking collectives 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 /* ... */ root,comm,&req);
```

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`.

Transpose as gather/scatter

Every process needs to do a scatter or gather.

Simultaneous collectives

Transpose matrix by scattering all rows simultaneously.
Each scatter involves all processes, but with a different spanning tree.

```
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);
```

Persistent collectives

Persistent collectives (MPI-4)

Similar to persistent send/recv:

```
double *buffer;
MPI_Allreduce_init( buffer ...., &request );
for ( ... ) {
  // fill buffer
  MPI_Start( request );
  // possibly other activities
  MPI_Wait( &request );
}
MPI_Request_free( &request );
```

Available for all collectives and neighborhood collectives.

Example

```
// powerpersist1.c
double localnorm,globalnorm=1.;
MPI_Request reduce_request;
MPI_Allreduce_init
  ( &localnorm,&globalnorm,1,MPI_DOUBLE,MPI_SUM,
    comm,MPI_INFO_NULL,&reduce_request);
for (int it=0; ; it++) {
  /*
   * Matrix vector product
   */
  matmult(indata,outdata,buffersize);

// start computing norm of output vector
  localnorm = local_l2_norm(outdata,buffersize);
  double old_globalnorm = globalnorm;
  MPI_Start( &reduce_request );

// end computing norm of output vector
  MPI_Wait( &reduce_request,MPI_STATUS_IGNORE );
  globalnorm = sqrt(globalnorm);
// now ‘globalnorm’ is the L2 norm of ‘outdata’
  scale(outdata,indata,buffersize,1./globalnorm);
}
```

Persistent vs non-blocking

Both request-based.

- Non-blocking is 'ad hoc': buffer info not known before the collective call.
- Persistent allows 'planning ahead': management of internal buffers and such.

Request handling:

- Non-blocking: wait deallocates the request
- Persistent: wait deactivates the request, still requires `MPI_Request_free`.

Non-blocking barrier

Just what is a barrier?

- Barrier is not *time* synchronization but *state* synchronization.
- Test on non-blocking barrier: 'has everyone reached some state'

Use case: adaptive refinement

- Some processes decide locally to alter their structure
- ... need to communicate that to neighbors
- Problem: neighbors don't know whether to expect update calls, if at all.
- Solution:
  - send update msgs, if any;
  - then post barrier.
  - Everyone probe for updates, test for barrier.

Use case: distributed termination detection

- Distributed termination detection (Matocha and Kamp, 1998): draw a global conclusion with local operations
- Everyone posts the barrier when done;
- keeps doing local computation while testing for the barrier to complete

MPI_Ibarrier

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

Step 1

Do sends, post 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-blocking barrier
 * and gets a request to test/wait for
 */
MPI_Request barrier_request;
MPI_Ibarrier(comm,&barrier_request);
```

Poll for barrier and messages

```
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!
```

Shared memory

Shared memory myths

Myth:

MPI processes use network calls, whereas OpenMP threads access memory directly, therefore OpenMP is more efficient for shared memory.

Truth:

MPI implementations use copy operations when possible, whereas OpenMP has thread overhead, and affinity/coherence problems.

Main problem with MPI on shared memory: data duplication.

MPI shared memory

- Shared memory access: two processes can access each other's memory through `double*` (and such) pointers, if they are on the same shared memory.
- Limitation: only window memory.
- Non-use case: remote update. This has all the problems of traditional shared memory (race conditions, consistency).
- Good use case: every process needs access to large read-only dataset Example: ray tracing.

Shared memory threatments in MPI

- MPI uses optimizations for shared memory: copy instead of socket call
- One-sided offers 'fake shared memory': you can access another process' data, but only through function calls.
- MPI-3 shared memory gives you a pointer to another process' memory,
  if that process is on the same shared memory.

Shared memory per cluster node

- Cluster node has shared memory
- Memory is attached to specific socket
- beware Non-Uniform Memory Access (NUMA) effects

Shared memory interface

Here is the high level overview; details next.

- Use `MPI_Comm_split_type` to find processes on the same shared memory
- Use `MPI_Win_allocate_shared` to create a window between processes on the same shared memory
  (MPI-4.1: other window creation calls allowed, but the burden is on you!)
- Use `MPI_Win_shared_query` to get pointer to another process' window data.
- You can now use `memcpy` instead of `MPI_Put`.

Discover shared memory

- `MPI_Comm_split_type` splits into communicators of same type.
- Use type: `MPI_COMM_TYPE_SHARED` splitting by shared memory.

Code:

```
// commsplittype.c
MPI_Info info;
MPI_Comm_split_type
  (MPI_COMM_WORLD,
   MPI_COMM_TYPE_SHARED,
   procno,info,&sharedcomm);
MPI_Comm_size
  (sharedcomm,&new_nprocs);
MPI_Comm_rank
  (sharedcomm,&new_procno);
```

Output:

```
make[3]: 'commsplittype' is up to date.
TACC:  Starting up job 4356245
TACC:  Starting parallel tasks...
There are 10 ranks total
[0] is processor 0 in a shared group of 5,
    ↪running on
    ↪c209-010.frontera.tacc.utexas.edu
[5] is processor 0 in a shared group of 5,
    ↪running on
    ↪c209-011.frontera.tacc.utexas.edu
TACC:  Shutdown complete. Exiting.
```

Exercise 46 (shareddata)

Write a program that uses `MPI_Comm_split_type` to analyze for a run

1. How many nodes there are;
2. How many processes there are on each node.

If you run this program on an unequal distribution, say 10 processes on 3 nodes, what distribution do you find?

```
Nodes: 3; processes: 10
TACC:  Starting up job 4210429
TACC:  Starting parallel tasks...
There are 3 nodes
Node sizes: 4 3 3
TACC:  Shutdown complete. Exiting.
```

Allocate shared window

Use `MPI_Win_allocate_shared` to create a window that can be shared;

- Has to be on a communicator on shared memory
- Example: window is one double.

```
// sharedbulk.c
MPI_Win node_window;
MPI_Aint window_size; double *window_data;
if (onnode_procid==0)
  window_size = sizeof(double);
else window_size = 0;
MPI_Win_allocate_shared
  ( window_size,sizeof(double),MPI_INFO_NULL,
    nodecomm,
    &window_data,&node_window);
```

Get pointer to other windows

Use MPI_Win_shared_query:

```
MPI_Aint window_size0; int window_unit; double *win0_addr;
MPI_Win_shared_query
  ( node_window,0,
    &window_size0,&window_unit, &win0_addr );
```

MPI_Win_shared_query

Name	Param name	Explanation	C type	F type
MPI_Win_shared_query (				
MPI_Win_shared_query_c (				
	win	shared memory window object	MPI_Win	TYPE(MPI_Win)
	rank	rank in the group of window win or MPI_PROC_NULL	int	INTEGER
	size	size of the window segment	MPI_Aint*	INTEGER (KIND=MPI_ADDRESS_KIND)
	disp_unit	local unit size for displacements, in bytes	int* / MPI_Aint*	INTEGER
	baseptr	address for load/store access to window segment	void*	TYPE(C_PTR)
	)			

Allocated memory

Memory will be allocated contiguously
convenient for address arithmetic,
not for NUMA: set `alloc_shared_noncontig` true in `MPI_Info` object.

Example: each window stores one double. Measure distance in bytes:

Strategy: default behavior of shared window allocation

```
Distance 1 to zero: 8
Distance 2 to zero: 16
```

Strategy: allow non-contiguous shared window allocation

```
Distance 1 to zero: 4096
Distance 2 to zero: 8192
```

Question: what is going on here?

Exciting example: bulk data

- Application: ray tracing:
  large read-only data strcture describing the scene
- traditional MPI would duplicate:
  excessive memory demands
- Better: allocate shared data on process 0 of the shared communicator
- Everyone else points to this object.

Split by other hardware types

MPI-4: split by other hardware features through
`MPI_COMM_TYPE_HW_GUIDED` and `MPI_Get_hw_resource_types`)

Process management

Overview

This section discusses processes management; intra communicators.

Commands learned:

- `MPI_Comm_spawn`, `MPI_UNIVERSE_SIZE`
- `MPI_Comm_get_parent`, `MPI_Comm_remote_size`

Process management

- PVM was a precursor of MPI: could dynamically create new processes.
- It took MPI a while to catch up.
- Use `MPI_Attr_get` to retrieve `MPI_UNIVERSE_SIZE` attribute indicating space for creating more processes outside `MPI_COMM_WORLD`.
- New processes have their own `MPI_COMM_WORLD`.
- Communication between the two communicators: 'inter communicator'
  (the old type is 'intra communicator')

Space for processes

Probably a machine dependent component.

Suggested standard:

```
mpiexec -n 4 -usize 8 spawn_manager
```

Intel MPI at TACC:

```
MY_MPIRUN_OPTIONS="-usize 8" ibrun -np 4 spawn_manager
```

Discover size of the universe:

```
MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
  (void*)&universe_sizep, &flag);
```

Manager program

```
int universe_size, *universe_size_attr,uflag;
MPI_Comm_get_attr
  (comm_world,MPI_UNIVERSE_SIZE,
   &universe_size_attr,&uflag);
if (uflag) {
  universe_size = *universe_size_attr;
} else {
  printf("This MPI does not support UNIVERSE_SIZE.\nUsing world
     ↪size");
  universe_size = world_n;
}
int work_n = universe_size - world_n;
if (world_p==0) {
  printf("A universe of size %d leaves room for %d workers\n",
         universe_size,work_n);
  printf(".. spawning from %s\n",procname);
}
```

Manager program (cont'd)

```
const char *workerprogram = "./spawnapp";
MPI_Comm_spawn(workerprogram,MPI_ARGV_NULL,
               work_n,MPI_INFO_NULL,
               0,comm_world,&comm_inter,NULL);
```

Worker program

```
// spawnworker.c
MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
MPI_Comm_get_parent(&parent);
```

Were you spawned?

```
// spawnapp.c
MPI_Comm comm_parent;
MPI_Comm_get_parent(&comm_parent);
int is_child = (comm_parent!=MPI_COMM_NULL);
if (is_child) {
  int nworkers,workerno;
  MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
  MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
  printf("I detect I am worker %d/%d running on %s\n",
         workerno,nworkers,procname);
```

Process topologies

Overview

This section discusses topologies:

- Cartesian topology
- MPI-1 Graph topology
- MPI-3 Graph topology

Commands learned:

- `MPI_Dist_graph_create`, `MPI_DIST_GRAPH`, `MPI_Dist_graph_neighbors_count`
- `MPI_Neighbor_allgather` and such

Process topologies

- Processes don't communicate at random
- Example: Cartesian grid, each process 4 (or so) neighbors
- Express operations in terms of topology
- Elegance of expression
- MPI can optimize

Process reordering

- Consecutive process numbering often the best:
  divide array by chunks
- Not optimal for grids or general graphs:
- MPI is allowed to renumber ranks
- Graph topology gives information from which MPI can deduce renumbering

MPI-1 topology

- Cartesian topology
- Graph topology, globally specified.
  Not scalable, do not use!

MPI-3 topology

- Graph topologies locally specified: scalable!
  Limit cases: each process specifies its own connectivity one process specifies whole graph.
- Neighborhood collectives:
  expression close to the algorithm.

Graph topologies

Example: 5-point stencil

Neighbor exchange, spelled out:

- Each process communicates down/right/up/left
- Send and receive at the same time.
- Can optimally be done in four steps

Step 1

Step 2

The middle node is blocked because all its targets are already receiving or a channel is occupied:
one missed turn

Neighborhood collective

This is really a 'local gather':
each node does a gather from its neighbors in whatever order.

`MPI_Neighbor_allgather`

Why neighborhood collectives?

- Using `MPI_Isend` / `MPI_Irecv` is like spelling out a collective, imposes order;
- Collectives can use pipelining as opposed to sending a whole buffer;
- Collectives can use spanning trees as opposed to direct connections.

Create graph topology

```
int MPI_Dist_graph_create
   (MPI_Comm comm_old, int nsources, const int sources[],
    const int degrees[], const int destinations[],
    const int weights[], MPI_Info info, int reorder,
    MPI_Comm *comm_dist_graph)
```

- *nsources* how many source nodes described? (Usually 1)
- *sources* the processes being described (Usually `MPI_Comm_rank` value)
- *degrees* how many processes to send to
- *destinations* their ranks
- *weights*: usually set to `MPI_UNWEIGHTED`.
- *info*: `MPI_INFO_NULL` will do
- *reorder*: 1 if dynamically reorder processes

Neighborhood collectives

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

Like an ordinary `MPI_Allgather`, but
the receive buffer has a length enough for *degree* messages
(instead of comm size).

Neighbor querying

After `MPI_Neighbor_allgather` data in the buffer is *not* in normal rank order.

- `MPI_Dist_graph_neighbors_count` gives actual number of neighbors. (Why do you need this?)
- `MPI_Dist_graph_neighbors` lists neighbor numbers.

MPI_Dist_graph_neighbors_count

Name	Param name	Explanation	C type	F type
MPI_Dist_graph_neighbors_count (				
	comm	communicator with distributed graph topology	MPI_Comm	TYPE(MPI_Comm)
	indegree	number of edges into this process	int*	INTEGER
	outdegree	number of edges out of this process	int*	INTEGER
	weighted	false if MPI_UNWEIGHTED was supplied during creation, true otherwise	int*	LOGICAL
	)			

MPI_Dist_graph_neighbors

Name	Param name	Explanation	C type	F type
MPI_Dist_graph_neighbors (				
	comm	communicator with distributed graph topology	MPI_Comm	TYPE(MPI_Comm)
	maxindegree	size of sources and sourceweights arrays	int	INTEGER
	sources	processes for which the calling process is a destination	int[]	INTEGER(maxindegree)
	sourceweights	weights of the edges into the calling process	int[]	INTEGER(*)
	maxoutdegree	size of destinations and destweights arrays	int	INTEGER
	destinations	processes for which the calling process is a source	int[]	INTEGER(maxoutdegree)
	destweights	weights of the edges out of the calling process	int[]	INTEGER(*)
	)			

Example: Systolic graph

Code:

```
// graph.c
for ( int i=0; i<=1; i++ ) {
  int neighb_i = proci+i;
  if (neighb_i<0 || neighb_i>=idim)
    continue;
  int j = 1-i;
  int neighb_j = procj+j;
  if (neighb_j<0 || neighb_j>=jdim)
        continue;
  destinations[ degree++ ] =
        PROC(neighb_i,neighb_j,idim,jdim);
}
MPI_Dist_graph_create
  (comm,
   /* I specify just one proc: me */ 1,
   &procno,&degree,destinations,weights,
   MPI_INFO_NULL,0,
   &comm2d
   );
```

Output

Code:

```
int indegree,outdegree,
  weighted;
MPI_Dist_graph_neighbors_count
  (comm2d,
   &indegree,&outdegree,
   &weighted);
int
  my_ij[2] = {proci,procj},
  other_ij[4][2];
MPI_Neighbor_allgather
  ( my_ij,2,MPI_INT,
    other_ij,2,MPI_INT,
    comm2d );
```

Output:

```
[ 0 = (0,0)] has  2 outbound: 1, 2,
   0 inbound:
[ 1 = (0,1)] has  1 outbound: 3,
   1 inbound: (0,0)=0
[ 2 = (1,0)] has  2 outbound: 3, 4,
   1 inbound: (0,0)=0
[ 3 = (1,1)] has  1 outbound: 5,
   2 inbound: (0,1)=1 (1,0)=2
[ 4 = (2,0)] has  1 outbound: 5,
   1 inbound: (1,0)=2
[ 5 = (2,1)] has  0 outbound:
   2 inbound: (1,1)=3 (2,0)=4
```

Note that the neighbors are listed in correct order. This need not be the case.

Query

Explicit query of neighbor process ranks.

Code:

```
int indegree,outdegree,
  weighted;
MPI_Dist_graph_neighbors_count
  (comm2d,
   &indegree,&outdegree,
   &weighted);
int
  my_ij[2] = {proci,procj},
  other_ij[4][2];
MPI_Neighbor_allgather
  ( my_ij,2,MPI_INT,
    other_ij,2,MPI_INT,
    comm2d );
```

Output:

```
    0 inbound:
    1 inbound: 0
    1 inbound: 0
    2 inbound: 1 2
    1 inbound: 2
    2 inbound: 4 3
```

Exercise 47 (rightgraph)

Earlier rightsend exercise

Revisit exercise 19 and solve it using `MPI_Dist_graph_create`. Use figure 393 for inspiration.

Use a degree value of 1.

Inspiring picture for the previous exercise

Solving the right-send exercise with neighborhood collectives

Hints for the previous exercise

Two approaches:

1. Declare just one source: the previous process. Do this! Or:
2. Declare two sources: the previous and yourself. In that case bear in mind slide 386.

More graph collectives

- Heterogeneous: `MPI_Neighbor_alltoallw`.
- Non-blocking: `MPI_Ineighbor_allgather` and such
- Persistent: `MPI_Neighbor_allgather_init`, `MPI_Neighbor_allgatherv_init`.

Part

Tracing, performance, and such

Overview

We briefly touch on peripheral issues issues to MPI.

CMake

cmake

. MPI is discoverable by cmake.

CMake for C++

```
cmake_minimum_required( VERSION 3.12 )
project( ${PROJECT_NAME} VERSION 1.0 )

# https://cmake.org/cmake/help/latest/module/FindMPI.html
find_package( MPI )

add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.cxx )
target_compile_features( ${PROJECT_NAME} PRIVATE cxx_std_20 )
target_include_directories(
        ${PROJECT_NAME} PUBLIC
        ${MPI_CXX_INCLUDE_DIRS} ${CMAKE_CURRENT_SOURCE_DIR} )
target_link_libraries(
        ${PROJECT_NAME} PUBLIC
        ${MPI_CXX_LIBRARIES} )

install( TARGETS ${PROJECT_NAME} DESTINATION . )
```

CMake for F90

```
cmake_minimum_required( VERSION 3.12 )
project( ${PROJECT_NAME} VERSION 1.0 )

enable_language(Fortran)

# https://cmake.org/cmake/help/latest/module/FindMPI.html
find_package( MPI )

add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.F90 )
target_include_directories(
        ${PROJECT_NAME} PUBLIC
        ${MPI_INCLUDE_DIRS} ${CMAKE_CURRENT_SOURCE_DIR} )
target_link_directories(
        ${PROJECT_NAME} PUBLIC
        ${MPI_LIBRARY_DIRS} )
target_link_libraries(
        ${PROJECT_NAME} PUBLIC
        ${MPI_Fortran_LIBRARIES} )

install( TARGETS ${PROJECT_NAME} DESTINATION . )
```

CMake for F2008

```
cmake_minimum_required( VERSION 3.12 )
project( ${PROJECT_NAME} VERSION 1.0 )

enable_language(Fortran)

# https://cmake.org/cmake/help/latest/module/FindMPI.html
find_package( MPI )

if( MPI_Fortran_HAVE_F08_MODULE )
else()
  message( FATAL_ERROR "No f08 module for this MPI" )
endif()

add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.F90 )
target_include_directories(
        ${PROJECT_NAME} PUBLIC
        ${MPI_Fortran_INCLUDE_DIRS} ${CMAKE_CURRENT_SOURCE_DIR} )
target_link_directories(
        ${PROJECT_NAME} PUBLIC
        ${MPI_LIBRARY_DIRS} )
target_link_libraries(
        ${PROJECT_NAME} PUBLIC
        ${MPI_Fortran_LIBRARIES} )
```

Errors

Built-in handlers

Default: global termination.

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_ARE_FATAL);
```

MPI-4: *Only terminate on communicator:* `MPI_ERRORS_ABORT`.

Local handling: `MPI_ERRORS_RETURN`:

Handlers on specific classes

Associate error handler with communicator:

`MPI_Comm_set_errhandler` `MPI_Comm_get_errhandler`

Other:

- `MPI_File_set_errhandler`, `MPI_File_call_errhandler`,
  MPI-4: *`MPI_Session_set_errhandler`*, *`MPI_Session_call_errhandler`*,
  `MPI_Win_set_errhandler`, `MPI_Win_call_errhandler`.

Handling errors

```
char errtxt[MPI_MAX_ERROR_STRING];
int err = status.MPI_ERROR;
int len=MPI_MAX_ERROR_STRING;
MPI_Error_string(err,errtxt,&len);
printf("Waitall error: %d %s\n",err,errtxt);
```

Define new errors

```
int nonzero_code;
MPI_Add_error_code(nonzero_class,&nonzero_code);
MPI_Add_error_string(nonzero_code,"Attempting to send zero buffer");
```

Performance measurement

Timers

MPI has a *wall clock* timer: `MPI_Wtime` which gives the number of seconds from a certain point in the past.

The timer has a resolution of `MPI_Wtick`

Timers can be 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);
```

but probably aren't.

Example

```
// pingpong.c
if (procno==processA) {
  t = MPI_Wtime();
  for (int n=0; n<NEXPERIMENTS; n++) {
    MPI_Send(send,1,MPI_DOUBLE,
    MPI_Recv(recv,1,MPI_DOUBLE,
  }
  t = MPI_Wtime()-t; t /= NEXPERIMENTS;
```

Global timing

Processes don't start/end simultaneously. What does a timing result mean overall? Take average or maximum?

Alternative:

```
MPI_Barrier(comm)
t = MPI_Wtime();
// something happens here
MPI_Barrier(comm)
t = MPI_Wtime()-t;
```

Profiling

See other lecture: MPIP, TAU, et cetera.

Your own profiling interface

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.

```
main() {
  MPI_Send ( buffer,ct,tp, ... );
}
```

call

```
int MPI_Send ( buffer,ct,tp, ... ) {
  PMPI_Send( buffer,ct,tp, ... ) ;
```

Programming for performance

Eager limit

- Optimization for small messages: bypass rendez-vous protocol (slide 149)
- Cross-over point: 'Eager limit'.
- Force efficient messages by increasing the eager limit.
- Beware: decreasing payoff for large messages, and
- Beware: buffers for eager send eat into your available memory.

Eager limit setting

- For Intel MPI: `I_MPI_EAGER_THRESHOLD`
- mvapich2: `MV2_IBA_EAGER_THRESHOLD`
- OpenMPI: `OpenMPI` the `--mca` options `btl_openib_eager_limit` and `btl_openib_rndv_eager_limit`.

Blocking versus non-blocking

- Non-blocking sends `MPI_Isend` / `MPI_Irecv` can be more efficient than blocking
- Also: allow overlap computation/communication (latency hiding)
- However: can usually not be considered a replacement.

Progress

MPI is not magically active in the background, so latency hiding is not automatic. Same for passive target synchronization and non-blocking barrier completion.

- Dedicated communications processor or thread.
  This is implementation dependent; for instance, Intel MPI: `I_MPI_ASYNC_PROGRESS_...` variables.
- Force progress by occasional calls to a polling routine such as `MPI_Iprobe`.

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*.

- `MPI_Send_init`
- `MPI_Recv_init`
- `MPI_Start`

Buffering

- MPI has internal buffers: copying costs performance
- Use your own buffer:
  - `MPI_Buffer_attach`
  - `MPI_Bsend`
- Copying is also a problem for derived datatypes.

Graph topology and neighborhood collectives

- Mapping problem to architecture sometimes not trivial
- Load balancers: *ParMetis*, *Zoltan*
- Graph topologies: *MPI_Dist_graph_adjacent*:
  allowed to reorder ranks for proximity
- Neighborhood collectives allow MPI to schedule optimally.
  - `MPI_Neighbor_allgather` (and *MPI_Neighbor_allgather_v*)
  - `MPI_Neighbor_alltoall`

Network issues

Network contention means that

- Your messages can collide with other jobs
- messages within your job can collide

Output routing

Contention

Offloading and onloading

- Network cards can offer assistance
- Mellanox: off-loading
  limited repertoire of scenarios where it helps
- Intel disagrees: on-loading
- Either way, investigate the capabilities of your network.