PARALLEL COMPUTING
FOR
SCIENCE AND ENGINEERING

VICTOR EIJKHOUT

0TH EDITION 2013

This book covers OpenMP (ok, it will at some point) and MPI. For both systems it covers some of the latest features. There is a healthy emphasis on practical examples.

Contents

Chapter 1

MPI

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

1.1 Distributed memory and message passing

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

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

Another reason that MPI was a big hit with programmers is that it does not ask you to learn a new language: it is a library that can be interface to C/C++ or Fortran; there are even bindings to Python. A related point is that it is easy to install: there are free implementations that you can download and install on any computer that has a Unix-like operating system, even if that is not a parallel machine.

1.1.1 History

Mid 1990s, many parties involved, big concensus. Many competing packages before, few after.

1.1.2 Basic model

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

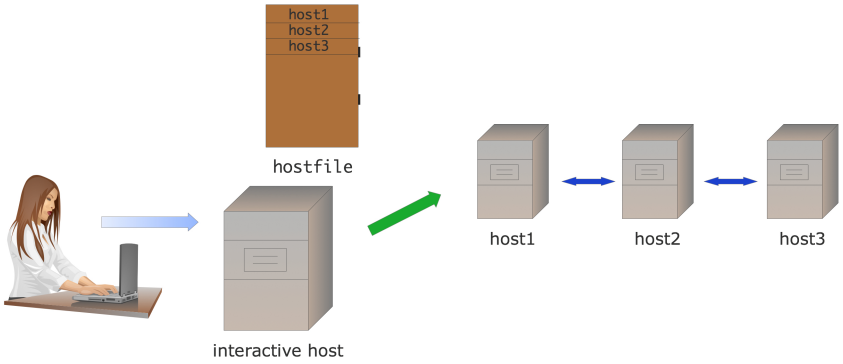

Figure 1.1: Interactive MPI setup

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

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

In the second scenario (figure 1.2) the user prepares a *batch job* script with commands, and these will be run when the *batch scheduler* gives a number of hosts to the job. Now the batch script contains the `mpirun`

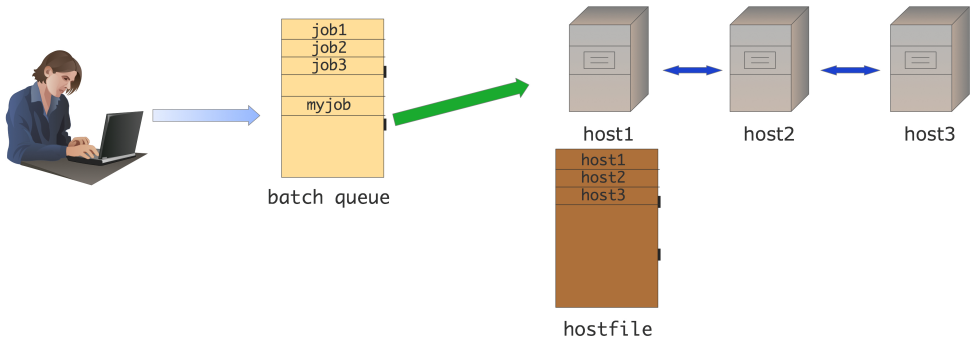

Figure 1.2: Batch MPI setup

command, or some variant such as `ibrun`, and the hostfile is dynamically generated when the job starts. Since the job now runs at a time when the user may not be logged in, any screen output goes into an output file.

You see that in both scenarios the parallel program is started by the `mpirun` command, which only supports an Single Program Multiple Data (SPMD) mode of execution: all hosts execute the same program.

To first order, the network is symmetric. However, the truth is more complicated ('topology-aware communication').

1.2 Basic concepts

1.2.1 Initialization / finalization

Every program that uses MPI needs to initialize and finalize exactly once. In C, the calls are

```
ierr = MPI_Init(&argc,&argv);
// your code
ierr = MPI_Finalize();
```

where `argc` and `argv` are the arguments of the main program. The corresponding Fortran calls are

```
call MPI_Init(ierr)
// your code
call MPI_Finalize()
```

(There is a call `MPI_Abort` if you want to abort execution completely.)

We make a few observations.

- MPI routines return an error code. In C, this is a function result; in Fortran it is the final parameter in the calling sequence.
- For most routines, this parameter is the only difference between the C and Fortran calling sequence, but some routines differ in some respect related to the languages. In this case, C has a mechanism for dealing with commandline arguments that Fortran lacks.
- This error parameter is zero if the routine completes succesfully, and nonzero otherwise. You can write code to deal with the case of failure, but by default your program will simply abort on any MPI error. See section 1.8.2 for more details.

The commandline arguments `argc` and `argv` are only guaranteed to be passed to process zero, so the best way to pass commandline information is by a broadcast (section 1.4.1).

1.2.2 Making and running an MPI program

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

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

However, MPI libraries may have different names between different architectures, making it hard to have a portable makefile. Therefore, MPI typically has shell scripts around your compiler call:

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

MPI programs can be run on many different architectures. Obviously it is your ambition (or at least your dream) to run your code on a cluster with a hundred thousand processors and a fast network. But maybe you only have a small cluster with plain *ethernet*. Or maybe you're sitting in a plane, with just your laptop. An MPI program can be run in all these circumstances – within the limits of your available memory of course.

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

1.2.3 Distinguishing between processes

The reference for the commands introduced here can be found in section 1.10.1.1.

In the SPMD model you run the same executable on each of a set of processors; see section HPSC-2.2.2. So how can you do anything useful if all processors run the same code? Here is where your first two MPI routines come in.

With `MPI_Comm_size` a processor can query how many processes there are in total, and with `MPI_Comm_rank` it can find out what its number is. This rank is a number from zero to the comm size minus one. (Zero-based indexing is used even if you program in Fortran.)

Using these calls, the simplest MPI programs does this:

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&ntids);
MPI_Comm_rank(MPI_COMM_WORLD,&mytid);
printf("Hello, this is processor %d out of %d\n",mytid,ntids);
MPI_Finalize();
```

1.3 Point-to-point communication

MPI has two types of message passing routines: point-to-point and collectives. In this section we will discuss point-to-point communication, which involves the interaction of a unique sender and a unique receiver. Collectives, which involve all processes in some joint fashion, will be discussed in the next section.

There is a lot to be said about simple sending and receiving of data. We will go into three broad categories of operations: blocking and non-blocking two-sided communication, and the somewhat more tricky one-sided communication.

Two-sided communication is a little like email: one party send data, which needs to be specified, to another party. The other party can then be expecting a message from a specified sender or it can be open to receiving from any source, but in either case the receiver indicates that something is to be expected. One-sided communication is very different in nature. Compare it to leaving your front-door open and people can bring things to your house, or take them, without you noticing.

1.3.1 Blocking communication

The reference for the commands introduced here can be found in section 1.10.2.

In two-sided communication, one process issues a send call and the other a receive call. Life would be easy if the send call put the data somewhere in the network for the receiving process to find whenever it gets around to its receive call. This ideal scenario is pictured figure 1.3. Of course, if the receiving process gets

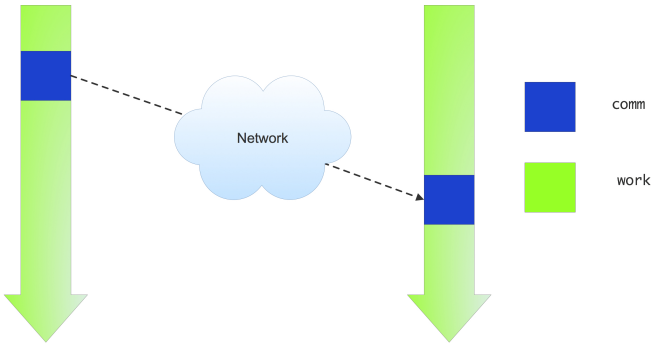

Figure 1.3: Illustration of an ideal send-receive interaction

to the receive call before the send call has been issued, it will be idle until that happens.

Even if processes are optimally synchronized communication introduces some overhead: there is an inital latency connected with every message, and the network also has a limited bandwidth which leads to a transfer time per byte; see HPSC-2.6.7.

The above ideal scenario is not realistic: it assumes that somewhere in the network there is buffer capacity for all messages that are in transit. Since this message volume can be large, we have to worry explicitly about management of send and receive *buffers*.

The easiest scenario is that the sending process keeps the message data in its address space until the receiving process has indicated that it is ready to receive it. This is pictured in figure 1.4. This is known as *blocking* communication: a process that issues a send or receive call will then block until the corresponding receive or send call is successfully concluded.

It is clear what the first problem with this scenario is: if your processes are not perfectly synchronized your performance may degrade because processes spend time waiting for each other; see HPSC-2.10.1.

But there is a more insidious, more serious problem. Suppose two process need to exchange data, and consider the following pseudo-code, which purports to exchange data between processes 0 and 1:

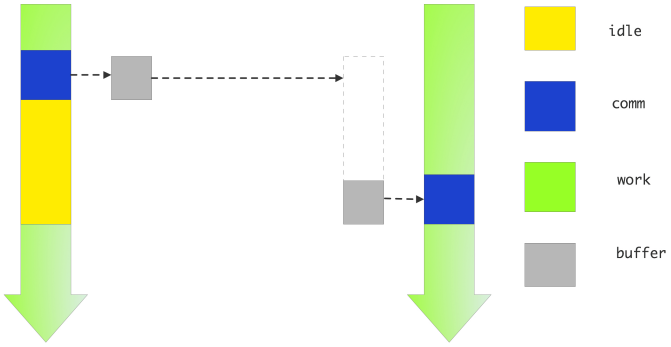

Figure 1.4: Illustration of a blocking communication: the sending processor is idle until the receiving processor issues and finishes the receive call

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

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

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

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

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

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

This code does not deadlock. All processors but the last one block on the send call, but the last processor

executes the receive call. Thus, the processor before the last one can do its send, and subsequently continue to its receive, which enables another send, et cetera.

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

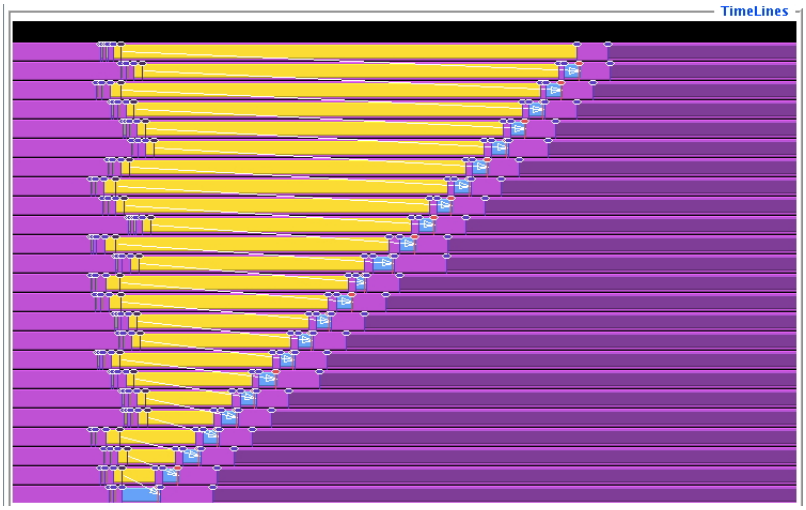

Figure 1.5: Trace of a simple send-recv code

a sequential one. This is illustrated in figure 1.5.

Exercise 1.1. Modify your earlier code, run it and reproduce the trace output of figure 1.5.

It is possible to orchestrate your processes to get an efficient and deadlock-free execution, but doing so is a bit cumbersome. There are better solutions which we will explore next.

Exercise 1.2. Give pseudo-code for a solution that uses blocking operations, and is parallel, although maybe not optimally so.

Exercise 1.3. There are rare circumstances where you actually want this serial behaviour. Recall from exercise **??** that output from processes is not automatically serialized. Take your code from that exercise, and use the serialization behaviour your observed to force the processes to output their print statements in sequence.

Above, you saw a code fragment with a conditional send:

```
MPI_Comm_rank( .... &mytid );
successor = mytid+1
if ( /* I am not the last processor */ )
  send(target=successor);
```

MPI allows for the following variant which makes the code slightly more homogeneous:

```
MPI_Comm_rank( .... &mytid );
if ( /* I am not the last processor */ )
```

```
  successor = mytid+1
else
  successor = MPI_PROC_NULL;
send(target=successor);
```

where the send call is executed by all processors; the target value of `MPI_PROC_NULL` means that no actual send is done. The null processor value is also of use with the `MPI_Sendrecv` call.

1.3.1.1 Deadlock-free blocking communication

Above you saw that with blocking sends the precise ordering of the send and receive calls is crucial. Use the wrong ordering and you get either deadlock, or something that is not efficient at all in parallel. MPI has a way out of this problem that is sufficient for many purposes: the combined send/recv call

```
MPI_Send_recv( /* send data */ ....
               /* recv data */ .... );
```

This call makes it easy to exchange data between two processors: both specify the other as both target and source. However, there need not be any such relation between target and source: it is possible to receive from a predecessor in some ordering, and send to a successor in that ordering.

Above you saw some examples that had most processors doing both a send and a receive, but some only a send or only a receive. You can still use `MPI_Sendrecv` in this call if you use `MPI_PROC_NULL` for the unused source or target argument.

Exercise 1.4. Take your code from exercise 1.1 and rewrite it to use the `MPI_Sendrecv` call. Run it and produce a trace output. Do you see the serialization behaviour of your earlier code?

1.3.1.2 Subtleties with processor synchronization

Blocking communication involves a complicated dialog between the two processors involved. Processor one says 'I have this much data to send; do you have space for that?', to which processor two replies 'yes, I do; go ahead and send', upon which processor one does the actual send. This back-and-forth (technically known as a *handshake*) takes a certain amount of communication overhead. For this reason, network hardware will sometimes forgo the handshake for small messages, and just send them regardless, knowing that the other process has a small buffer for such occasions.

One strange side-effect of this strategy is that a code that should *deadlock* according to the MPI specification does not do so. In effect, you may be shielded from you own programming mistake! Of course, if you then run a larger problem, and the small message becomes larger than the threshold, the deadlock will suddenly occur. So you find yourself in the situation that a bug only manifests itself on large problems, which are usually harder to debug. In this case, replacing every `MPI_Send` with a `MPI_Ssend` will force the handshake, even for small messages.

Conversely, you may sometimes wish to avoid the handshake on large messages. MPI as a solution for this: the `MPI_Rsend` ('ready send') routine sends its data immediately, but it needs the receiver to be ready for

this. How can you guarantee that the receiving process is ready? You could for instance do the following (this uses non-blocking routines, which are explained below in section 1.3.2):

```
if ( receiving ) {
  MPI_Irecv()   // post non-blocking receive
  MPI_Barrier() // synchronize
else if ( sending ) {
  MPI_Barrier() // synchronize
  MPI_Rsend()   // send data fast
```

When the barrier is reached, the receive has been posted, so it is safe to do a ready send. However, global barriers are not a good idea. Instead you would just synchronize the two processes involved.

Exercise 1.5. Give pseudo-code for a scheme where you synchronize the two processes through the exchange of a blocking zero-size message.

1.3.2 Non-blocking communication

The reference for the commands introduced here can be found in section 1.10.3.

In the previous section you saw that blocking communication makes programming tricky if you want to avoid deadlock and performance problems. The main advantage of these routines is that you have full control about where the data is. By contrast, *non-blocking* communication routines are easier to use, but more wasteful of memory.

The non-blocking calls `MPI_Isend` and `MPI_Irecv` do not wait for their counterpart: in effect they tell the runtime system 'here is some data and please send it as follows' or 'here is some buffer space, and expect such-and-such data to come'. This is illustrated in figure 1.6.

While the use of non-blocking routines prevents deadlock, it introduces two new problems:

1. When the send call returns, the send buffer may not be safe to overwrite; when the recv call returns, you do not know for sure that the expected data is in it. Thus, you need a mechanism to make sure that data was actually sent or received.
2. With a blocking send call, you could repeatedly fill the send buffer and send it off. To send multiple messages with non-blocking calls you have to allocate multiple buffers.

For the first problem, MPI has two types of routines. The `MPI_Wait...` calls are blocking: when you issue such a call, your execution will wait until the specified requests have been completed. A typical way of using them is:

```
// start non-blocking communication
MPI_Isend( ... ); MPI_Irecv( ... );
// do work that does not depend on incoming data
....
// wait for the Isend/Irecv calls to finish
MPI_Wait( ... );
// now do the work that absolutely needs the incoming data
....
```

Figure 1.7 shows the trace of a non-blocking execution using `MPI_Waitall`.

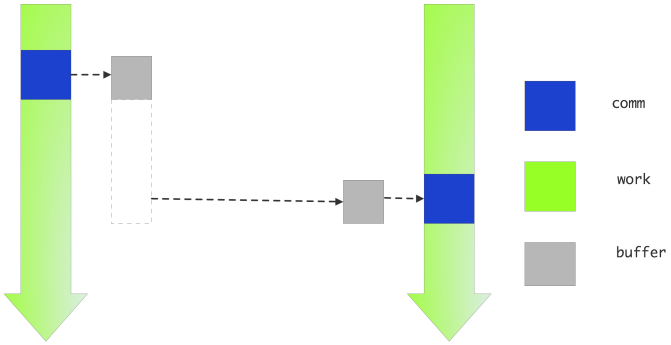

Figure 1.6: Illustration of a non-blocking communication: the sending processor immediately continues execution after issueing the send call

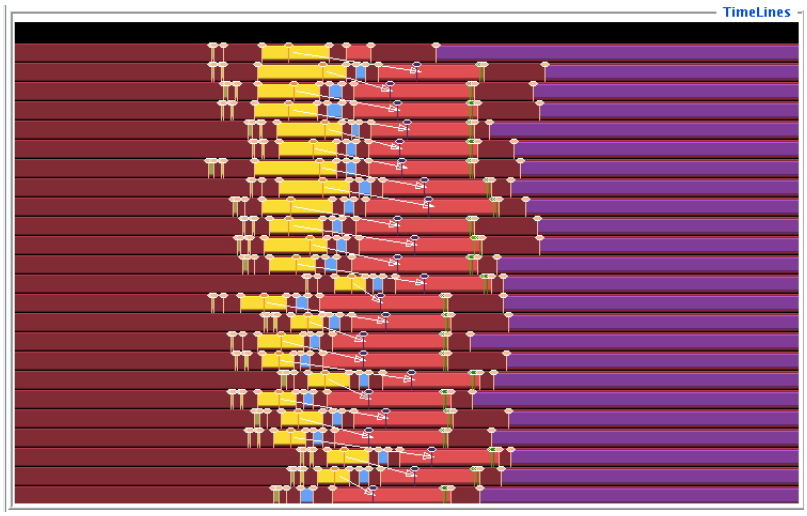

Figure 1.7: A trace of a nonblocking send between neighbouring processors

The `MPI_Test...` calls are themselves non-blocking: they test for whether one or more requests have been fullfilled. Here a typical idiom would be:

```
// start non-blocking communication
for ( ... p ... ) {
  MPI_Isend( .. p .. ); MPI_Irecv( .. p .. );
}
// wait for incoming data and process
while ( .. there are still requests .. ) {
  MPI_Test( ... );
  p = message.source;
```

```
  // process data from p
}
```

Exercise 1.6. Read section HPSC-6.4 and give pseudo-code for the distributed sparse matrix-vector product using the above idiom for using `MPI_Test...` calls. Discuss the advantages and disadvantages of this approach. The answer is not going to be black and white: discuss when you expect which approach to be preferable.

Another scenario where `MPI_Wait` calls are convenient is in the *master-worker model*. The master process creates tasks, and sends them to whichever worker process has finished its work:

```
while ( not done ) {
  // create new inputs for a while
  ....
  // see if anyone has finished
  MPI_Test( .... &index, &flag );
  if ( flag ) {
    // receive processed data and send new
}
```

1.3.2.1 Overlap of computation and communication

Non-blocking routines have long held the promise of letting a program *overlap its computation and communication*. The idea was that after posting the non-blocking calls the program could proceed to do non-communication work, while another part of the system would take care of the communication. Unfortunately, a lot of this communication involved activity in user space, so the solution would have been to let it be handled by a separate thread. Until recently, processors were not efficient at doing such multi-threading, so true overlap stayed a promise for the future.

1.3.2.2 More about non-blocking

Above we used `MPI_Irecv`, but we could have used the `MPI_Recv` routine. There is nothing special about a non-blocking or synchronous message once it arrives; the `MPI_Recv` call can match any of the send routines you have seen so far (but not `MPI_Sendrecv`).

1.3.3 One-sided communication

The reference for the commands introduced here can be found in section 1.10.4.

Above, you saw collectives and point-to-point operations. The latter type had in common that they require the co-operation of a sender and receiver. This co-operation could be loose: you can post a receive with `MPI_ANY_SOURCE` as sender, but there had to be both a send and receive call. On the other hand, in one-sided communication a process can do a 'put' or 'get' operation, writing data to or reading it from another processor, without that other processor's involvement.

In one-sided MPI operations, also known as Remote Direct Memory Access (RDMA) or Remote Memory Access (RMA) operations, there are still two processes involved: the *origin*, which is the process that originates the transfer, whether this is a 'put' or a 'get', and the *target* whose memory is being accessed. On the target, you declare an area of user-space memory that is accessible to other processes. This is known as a *window*. Windows limit how origin processes can access the target's memory: you can only 'get' data from a window or 'put' it into a window; all the other memory is not reachable from other processes.

The alternative to having windows is to use *distributed shared memory* or *virtual shared memory*: memory is distributed but acts as if it shared. The so-called Partitioned Global Address Space (PGAS) languages such as Unified Parallel C (UPC) use this model. The MPI RMA model makes it possible to lock a window which makes programming slightly more cumbersome, but the implementation more efficient.

Within one-sided communication, MPI has two modes: active RMA and passive RMA. In *active RMA*, both parties are involved; the main advantage of this mode is that the origin program can perform many small transfers, which are aggregated behind the scenes. Active RMA acts much like asynchronous transfer with a concluding `Waitall`.

In *passive RMA*, the target process is not involved. (PGAS languages such as UPC are based on this model: data is simply read or written at will.) While intuitively it is attractive to be able to write to and read from arbitrary memory, there are practical problems. For instance, it requires a remote agent on the target, which may interfere with execution of the main thread, or conversely it may not be activated at the optimal time. Passive RMA is also very hard to debug and can lead to strange deadlocks.

1.3.3.1 Windows and epochs

The reference for the commands introduced here can be found in section 1.10.4.1.

A window is a contiguous area of memory, defined with respect to a communicator: each process specifies

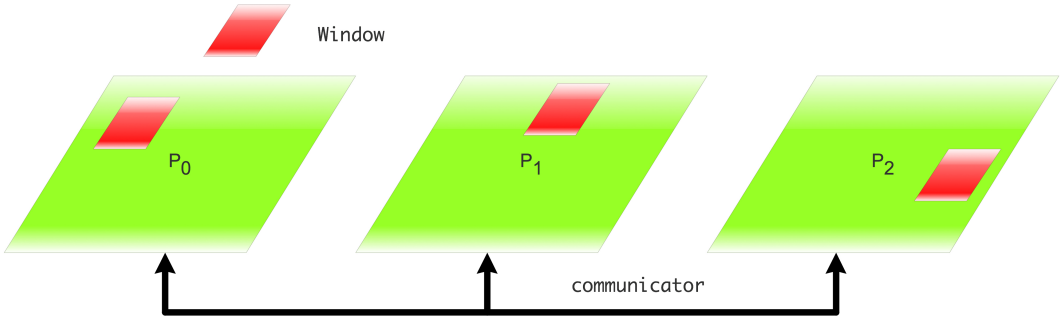

Figure 1.8: Collective definition of a window for one-sided data access

a memory area. Routine for creating and releasing windows are collective, so each process *has* to call them; see figure 1.8.

```
MPI_Info info;
MPI_Win window;
MPI_Win_create( /* size */, info, comm, &window );
MPI_Win_free( window );
```

However, each processor can specify its own window size, including zero, and even the type of the elements in it.

With a two-sided operation it is clear when the message has been completed; with a one-sided operation this is not so clear. Therefore we need a mechanism to ensure successful completion. One possibility is to use a *fence*.

```
MPI_WIN_FENCE (int assert, MPI_Win win, ierr)
```

This is comparable to doing a barrier on the communicator on which the window was based, and it guarantees that all communication on the window is completed.

Two fences delineate a so-called *epoch*. You can give various hints to the system about this epoch versus the ones before and after through the `assert` parameter.

```
MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
MPI_Get( /* operands */, win);
MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
```

In between the two fences the window is exposed, and while it is you should not access it locally. If you absolutely need to access it locally, you can use an RMA operation for that. Also, there can be only one remote process that does a `put`; multiple `accumulate` accesses are allowed.

Assertions are an integer parameter: you can add or logical-or values. The value zero is always correct. There are two types of parameters. Local assertions are:

- `MPI_MODE_NOSTORE` The preceding epoch did not store anything in this window.
- `MPI_MODE_NOPUT` The following epoch will not store anything in this window.

Global assertions:

- `MPI_MODE_NOPRECEDE` This process made no RMA calls in the preceding epoch.
- `MPI_MODE_NOSUCCEED` This process will make no RMA calls in the next epoch.

1.3.3.2 Put, get, accumulate

Window areas are accessible to other processes in the communicator by specifying the process rank and an offset from the base of the window.

```
MPI_PUT (void *origin_addr, int origin_count, MPI_Datatype
origin_datatype, int target_rank, MPI_Aint target_disp, int
target_count, MPI_Datatype target_datatype, MPI_Win
window, ierr)
```

The `MPI_Get` call is very similar; a third one-sided routine is `MPI_Accumulate` which does a reduction operation on the results that are being put:

```
MPI_ACCUMULATE (void *origin_addr, int
origin_count, MPI_Datatype origin_datatype,
int target_rank, MPI_Aint target_disp, int
target_count, MPI_Datatype
target_datatype,MPI_Op op,MPI_Win
window,ierr)
```

Exercise 1.7. Implement an 'all-gather' operation using one-sided communication: each processor stores a single number, and you want each processor to build up an array that contains the values from all processors. Note that you do not need a special case for a processor collecting its own value: doing 'communication' between a processor and itself is perfectly legal.

Accumulate is a reduction with remote result. As with reduction, the order is undefined. The same predefined operators are available, but no user-defined ones. One extra: `MPI_REPLACE`.

1.3.3.3 *Put vs Get*

```
while(!converged(A)){
  update(A);
  MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
  for(i=0; i < toneighbors; i++)
    MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                         todisp[i], 1, totype[i], win);
  MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
  }

  while(!converged(A)){
  update_boundary(A);
  MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
  for(i=0; i < fromneighbors; i++)
    MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
                    fromdisp[i], 1, fromtype[i], win);
  update_core(A);
  MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
  }
```

1.3.3.4 *Active and passive target synchronization*

There are two more fine-grained ways of doing one-sided communication that are suitable if only a small number of processor pairs is involved. In *active target synchronization* both processors are involved through the calls `MPI_Win_start`, `MPI_Win_complete`, `MPI_ Win_post`, `Win_wait`. What routine you use depends on whether the processor is an *origin* or *target*.

If the current process is going to have the data in its window accessed, you define an *exposure epoch* by:

```
MPI_Win_post( /* group of origin processes */ )
MPI_Win_wait()
```

This turns the current processor into a target for access operations issued by a different process.

If the current process is going to be issuing one-sided operations, you define an *access epoch* by:

```
MPI_Win_start()
MPI_Win_complete()
```

This turns the current process into the origin of a number of one-sided access operations.

The 'post' and 'start' routines open the window to a *group of processors*; see section 1.6.3 for how to get such a group from a communicator.

In *passive target synchronization* only one processor is involved in the communication. This involves using a lock: one process can lock the window on another, specified, process.

```
MPI_WIN_LOCK (int locktype, int rank, int assert, MPI_Win win,ierr)
MPI_WIN_UNLOCK (int rank, MPI_Win win, ierr)
```

During an access epoch, a process can initiate and finish a one-sided transfer.

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

These routines make MPI behave like a shared memory system; the instructions between locking and unlocking the window effectively become *atomic operations*.

1.3.3.5 Details

Sometimes an architecture has memory that is shared between processes, or that otherwise is fast for one-sided communication. To put a window in such memory, it can be placed in memory that is especially allocated:

```
MPI_Alloc_mem() and MPI_Free_mem()
```

These calls reduce to `malloc` and `free` is there is no special memory area.

1.4 Collectives

The reference for the commands introduced here can be found in section 1.10.5.

Collectives are operations that involve all processes in a communicator. The simplest example is a broadcast: one processor has some data and all others need to get a copy of it. A collective is a single call, and it blocks on all processors. That does not mean that all processors exit the call at the same time: because of network latency some processors can receive their data later than others.

There are several variants of most collectives. For instance, the simplest type has a *root of the collective* where information originates or is collected. Then there is an 'all' variant of operations such as the reduction, where the result is not left just on the root but everywhere. Finally, there are 'v' variants where the amount of data coming from or going to each processor is variable.

1.4.1 Broadcast, reduce, scan

The simplest collective is the broadcast, where one process has some data that needs to be shared with all others. One scenario is that processor zero can parse the commandline arguments of the executable. The call has the following structure:

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

The root is the process that is sending its data; see figure 1.9. Typically, it will be the root of a broadcast

Figure 1.9: A simple broadcast

tree. You see that there is no message tag, because collectives are blocking, so you can have only one active at a time. (In MPI 3 there are non-blocking collectives; see section 1.4.4.)

It is possible for the data to be an array; in that case MPI acts as if you did a separate scalar broadcast on each array index.

If a processor has only one outgoing connection, the broadcast in figure 1.9 would take a time proportional to the number of processors. One way to ameliorate that is to structure the broadcast in a tree-like fashion. This is depicted in figure 1.10. How does the communication time now depend on the number of processors? The theory of the complexity of collectives is described in more detail in HPSC-6.2.2.1; see also [1].

The reverse of a broadcast is a reduction:

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

Now there is a separate buffer for outgoing data, on all processors, and incoming data, only relevant on the root. Also, you have to indicate how the data is to be combined. Popular choices are `MPI_SUM`, `MPI_PROD` and `MPI_MAX`, but complicated operators such as finding the location of the maximum value exist. You can also define your own operators.

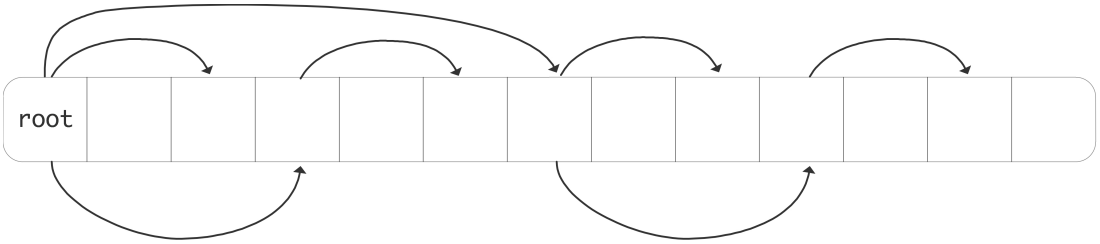

Figure 1.10: A tree-based broadcast

On processes that are not the root, the receive buffer is ignored. On the root, you have two buffers, but by specifying `MPI_IN_PLACE`, the reduction call uses the value in the receive buffer as the root's contribution to the operation. On the `Allreduce` call, `MPI_IN_PLACE` can be used for the send buffer of every process.

```
void *buffer = malloc(size*sizeof(double));
// write data into the buffer
if (mytid==0) {
  sendbuf = MPI_IN_PLACE; recvbuf = buffer;
} else {
  sendbuf = buffer;       recvbuf = NULL;
}
MPI_Reduce(sendbuf,recvbuf,size,MPI_DOUBLE,MPI_MAX,0,comm);
```

The `MPI_Scan` operation also performs a reduction, but it keeps the partial results. That is, if processor i contains a number x_i, and $\oplus$ is an operator, then the scan operation leaves $x_0 \oplus \cdot \oplus x_i$ on processor i.

```
MPI_Scan( send data, recv data, operator, communicator);
```

This is an inclusive scan operation. The exclusive definition, which computes $x_0 \oplus \cdot \oplus x_{i-1}$ on processor i, can easily be derived from the inclusive operation for operations such as `MPI_PLUS` or `MPI_MULT`. This is not the case for `MPI_MIN` or `MPI_MAX`, so there is an exclusive routine

```
MPI_Exscan( send data, recv data, operator, communicator);
```

with the same prototype.

The `MPI_Scan` operation is often useful with indexing data. Suppose that every processor p has a local vector where the number of elements n_p is dynamically determined. In order to translate the local numbering $0 \ldots n_p - 1$ to a global numbering one does a scan with the number of local elements as input. The output is then the global number of the first local variable.

Exercise 1.8. Do you use `MPI_Scan` or `MPI_Exscan` for this operation? How would you describe the result of the other scan operation, given the same input?

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

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

This means that X_i sums the segments between locations where $y_i = 0$ and the first subsequent place where $y_i = 1$. To implement this, you need a user-defined operator

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

1.4.2 Gather and scatter

In gather and scatter, the root collects information from, or conversely spreads it to, all other processes. The difference with reduce and broadcast is that it involves individual information from/to every process. Thus, the gather operation typically has an array of items, one coming from each sending process, and scatter has an array, with an individual item for each receiving process; see figure 1.11.

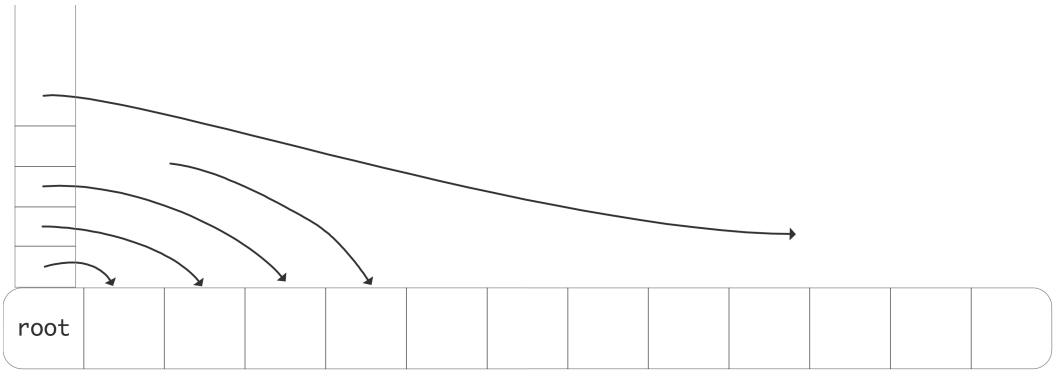

Figure 1.11: A scatter operation

To make this more precise, consider, arbitrarily, the scatter operation. The root process specifies an out buffer:

```
outbuffer, outcount, outtype
```

but the `outcount` is not the length of the buffer: it is the number of elements to send to each process. On the receiving processes other than the root the outbuffer arguments are irrelevant.

1.4.3 Variable gather and scatter

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

Looking again at the scatter, nothing changes in the receive buffer, but now the send buffer on the root is more complicated. It now consists of:

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

1.4.4 Non-blocking collectives

MPI version 3 has non-blocking collectives.

1.4.5 Barrier and all-to-all

There are two collectives we have not mentioned yet. A barrier is a call that blocks all processes until they have all reached the barrier call. This call's simplicity is contrasted with its usefulness, which is very limited. It is almost never necessary to synchronize processes through a barrier: for most purposes it does not matter if processors are out of sync. Conversely, collectives (except the new non-blocking ones) introduce a barrier of sorts themselves.

The all-to-all call is a generalization of a scatter and gather: every process is scattering an array of data, and every process is gathering an array of data. There is also a 'v' variant of this routine.

1.5 Data types

In many cases the data you send is a single element or an array of some elementary type such as byte, int, or real. You pass the data by specifying the start address and the number of data elements.

Since MPI is a library, and not a language, there is a possibility of confusion here: you first declare your variable or array as, say, `double`, and subsequently declare it to be `MPI_DOUBLE`. First of all, you can make errors, and the compiler will most likely not catch these. Secondly, the definition of datatypes is complicated; for instance there are 32-bit and 64-bit integers. How can you keep your code both correct and portable then?

1.5.1 Elementary data types

C/C++:

`MPI_CHAR`	only for text data, do not use for small integers
`MPI_UNSIGNED_CHAR`	
`MPI_SIGNED_CHAR`	
`MPI_SHORT`	
`MPI_UNSIGNED_SHORT`	
`MPI_INT`	
`MPI_UNSIGNED`	
`MPI_LONG`	
`MPI_UNSIGNED_LONG`	
`MPI_FLOAT`	
`MPI_DOUBLE`	
`MPI_LONG_DOUBLE`	

There is some, but not complete, support for *C99* types.

Fortran:

`MPI_CHARACTER`	Character(Len=1)
`MPI_LOGICAL`	
`MPI_INTEGER`	
`MPI_REAL`	
`MPI_DOUBLE_PRECISION`	
`MPI_COMPLEX`	
`MPI_DOUBLE_COMPLEX`	Complex(Kind=Kind(0.d0))

1.6 Communicators

A communicator is an object describing a group of processes. In many applications all processes work together closely coupled, and the only communicator you need is `MPI_COMM_WORLD`. However, there are circumstances where you want one subset of processes to operate independently of another subset. For example:

- If processors are organized in a 2×2 grid, you may want to do broadcasts inside a row or column.
- For an application that includes a producer and a consumer part, it makes sense to split the processors accordingly.

In this section we will see mechanisms for defining new communicators and sending messages between communicators.

An important reason for using communicators is the development of software libraries. If the routines in a library use their own communicator (even if it is a duplicate of the 'outside' communicator), there will never be a confusion between message tags inside and outside the library.

1.6.1 Basics

There are three predefined communicators:

- `MPI_COMM_WORLD` comprises all processes that were started together by *mpirun* (or some related program).
- `MPI_COMM_SELF` is the communicator that contains only the current process.
- `MPI_COMM_NULL` is the invalid communicator. Routines that construct communicators can give this as result if an error occurs.

In some applications you will find yourself regularly creating new communicators, using the mechanisms described below. In that case, you should de-allocate communicators with `MPI_Comm_free` when you're done with them.

1.6.2 Creating new communicators

There are various ways of making new communicators. We discuss three mechanisms, from simple to complicated.

1.6.2.1 Duplicating communicators

With `MPI_Comm_dup` you can make an exact duplicate of a communicator. This may seem pointless, but it is actually very useful for the design of software libraries. Image that you have a code

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

and suppose that the library has receive calls. Now it is possible that the receive in the library inadvertently catches the message that was sent in the outer environment.

To prevent this confusion, the library should duplicate the outer communicator, and send all messages with respect to its duplicate. Now messages from the user code can never reach the library software, since they are on different communicators.

1.6.2.2 Splitting a communicator

Splittingf a communicator into multiple disjoint communicators can be done with `MPI_Comm_split`. This uses a 'colour':

```
MPI_Comm_split( old_comm, colour, new_comm, .... );
```

and all processes in the old communicator with the same colour wind up in a new communicator together. The old communicator still exists, so processes now have two different contexts in which to communicate.

Here is one example of communicator splitting. Suppose your processors are in a two-dimensional grid:

```
MPI_Comm_rank( &mytid );
proc_i = mytid % proc_column_length;
proc_j = mytid / proc_column_length;
```

You can now create a communicator per column:

```
MPI_Comm column_comm;
MPI_Comm_split( MPI_COMM_WORLD, proj_j, &column_comm );
```

and do a broadcast in that column:

```
MPI_Bcast( data, /* tag: */ 0, column_comm );
```

Because of the SPMD nature of the program, you are now doing in parallel a broadcast in every processor column. Such operations often appear in *dense linear algebra*.

1.6.2.3 Process groups

The most general mechanism is based on groups: you can extract the group from a communicator, combine different groups, and form a new communicator from the resulting group.

The group mechanism is more involved. You get the group from a communicator, or conversely make a communicator from a group with `MPI_Comm_group` and `MPI_Comm_create`:

```
MPI_Comm_group( comm, &group);
MPI_Comm_create( old_comm, group, &new_comm );
```

and groups are manipulated with `MPI_Group_incl`, `MPI_Group_excl`, `MPI_Group_difference` and a few more.

You can name your communicators with `MPI_Comm_set_name`, which could improve the quality of error messages when they arise.

1.6.3 Intra-communicators

We start by exploring the mechanisms for creating a communicator that encompasses a subset of `MPI_COMM_WORLD`.

There is a simple function that splits a communicator into disjoint communicators:

```
MPI_COMM_SPLIT(MPI_Comm comm, int color, int key, MPI_Comm newcomm, ierr)
```

Each processor specifies what 'colour' it is, and processes with the same colour are grouped into a joint communicator. The 'key' value is used to determine the rank in the new communicator.

The most general mechanism for creating communicators is through process groups: you can query the group of processes of a communicator, manipulate groups, and make a new communicator out of a group you have formed.

```
MPI_COMM_GROUP (comm, group, ierr)
MPI_COMM_CREATE (MPI_Comm comm,MPI_Group group, MPI_Comm newcomm, ierr)
```

```
MPI_GROUP_UNION(group1, group2, newgroup, ierr)
MPI_GROUP_INTERSECTION(group1, group2, newgroup, ierr)
MPI_GROUP_DIFFERENCE(group1, group2, newgroup, ierr)

MPI_GROUP_INCL(group, n, ranks, newgroup, ierr)
MPI_GROUP_EXCL(group, n, ranks, newgroup, ierr)

MPI_GROUP_SIZE(group, size, ierr)
MPI_GROUP_RANK(group, rank, ierr)
```

1.6.4 Inter-communicators

If two disjoint communicators exist, it may be necessary to communicate between them. This can of course be done by creating a new communicator that overlaps them, but this would be complicated: since the 'inter' communication happens in the overlap communicator, you have to translate its ordering into those of the two worker communicators. It would be easier to express messages directly in terms of those communicators, and this can be done with 'inter-communicators'.

```
MPI_INTERCOMM_CREATE (local_comm, local_leader, bridge_comm, remote_leader,
```

After this, the intercommunicator can be used in collectives such as

```
MPI_BCAST (buff, count, dtype, root, comm, ierr)
```

- In group A, the root process passes `MPI_ROOT` as 'root' value; all others use `MPI_NULL_PROC`.
- In group B, all processes use a 'root' value that is the rank of the root process in the root group.

Gather and scatter behave similarly; the allgather is different: all send buffers of group A are concatenated in rank order, and places on all processes of group B.

Inter-communicators can be used if two groups of process work asynchronously with respect to each other; another application is fault tolerance (section 1.8.3).

1.7 Hybrid programming: MPI and threads

The reference for the commands introduced here can be found in section 1.10.8.

It is not automatic that a program or a library is *thread-safe*. A user can request a certain level of multi-threading with `MPI_Init_thread`, and the system will respond what the highest supported level is.

MPI can be thread-safe on the following levels:

- An MPI implementation can forbid any multi-threading;
- it can allow one thread to make MPI calls;

- it can allow one thread *at a time* to make MPI calls;
- it can allow arbitrary multi-threaded behaviour in MPI calls.

Some points.

- MPI can not distinguish between threads: the communicator rank identifies a process, and is therefore identical for all threads.
- A message sent to a process can be received by any thread that has issued a receive call with the right source/tag specification.
- Multi-threaded calls to an MPI routine have the semantics of an unspecified sequence of calls.
- A blocking MPI call only blocks the thread that makes it.

1.8 Leftover topics

1.8.1 Getting message information

In some circumstances the recipient may not know all details of a message.

- If you are expecting multiple incoming messages, it may be most efficient to deal with them in the order in which they arrive. For that, you have to be able to ask 'who did this message come from, and what is in it'.
- Maybe you know the sender of a message, but the amount of data is unknown. In that case you can overallocate your receive buffer, and after the message is received ask how big it was, or you can 'probe' an incoming message and allocate enough data when you find out how much data is being sent.

1.8.1.1 Status object

The receive calls you saw above has a status argument. If you precisely know what is going to be sent, this argument tells you nothing new. However, if you expect data from multiple senders, or the amount of data is indeterminate, the status will give you that information.

The `MPI_Status` object is a structure with the following freely accessible members: `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`. There is also opaque information: the amount of data received can be retrieved by a function call to `MPI_Get_count`.

```
int MPI_Get_count(
  MPI_Status *status,
  MPI_Datatype datatype,
  int *count
);
```

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

1.8.1.2 Probing messages

MPI receive calls specify a receive buffer, and its size has to be enough for any data sent. In case you really have no idea how much data is being sent, and you don't want to overallocate the receive buffer, you can use a 'probe' call.

The calls `MPI_Probe`, `MPI_Iprobe`, accept a message, but do not copy the data. Instead, when probing tells you that there is a message, you can use `MPI_Get_count` to determine its size, allocate a large enough receive buffer, and do a regular receive to have the data copied.

1.8.2 Error handling

The reference for the commands introduced here can be found in section 1.10.6.

Errors in normal programs can be tricky to deal with; errors in parallel programs can be even harder. This is because in addition to everything that can go wrong with a single executable (floating point errors, memory violation) you now get errors that come from faulty interaction between multiple executables.

A few examples of what can go wrong:

- MPI errors: an MPI routine can abort for various reasons, such as receiving much more data than its buffer can accomodate. Such errors, as well as the more common type mentioned above, typically cause your whole execution to abort. That is, if one incarnation of your executable aborts, the MPI runtime will kill all others.
- Deadlocks and other hanging executions: there are various scenarios where your processes individually do not abort, but are all waiting for each other. This can happen if two processes are both waiting for a message from each other, and this can be helped by using non-blocking calls. In another scenario, through an error in program logic, one process will be waiting for more messages (including non-blocking ones) than are sent to it.

The MPI library has a general mechanism for dealing with errors that it detects. The default behaviour, where the full run is aborted, is equivalent to your code having the following call [1]:

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

Another simple possibility is to specify

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_RETURN);
```

which gives you the opportunity to write code that handles the error return value.

In most cases where an MPI error occurs a complete abort is the sensible thing, since there are few ways to recover. The second possibility can for instance be used to print out debugging information:

```
ierr = MPI_Something();
if (ierr!=0) {
    // print out information about what your programming is doing
    MPI_Abort();
```

1. The routine `MPI_Errhandler_set` is deprecated.

```
}
```

For instance,

```
Fatal error in MPI_Waitall:
See the MPI_ERROR field in MPI_Status for the error code
```

You could code this as

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_RETURN);
ierr = MPI_Waitall(2*ntids-2,requests,status);
if (ierr!=0) {
   char errtxt[200];
   for (int i=0; i<2*ntids-2; i++) {
       int err = status[i].MPI_ERROR; int len=200;
       MPI_Error_string(err,errtxt,&len);
       printf("Waitall error: %d %s\n",err,errtxt);
   }
   MPI_Abort(MPI_COMM_WORLD,0);
}
```

One cases where errors can be handled is that of *MPI file I/OMPI!I/O*: if an output file has the wrong permissions, code can possibly progress without writing data, or writing to a temporary file.

1.8.3 Fault tolerance

Processors are not completely reliable, so it may happen that one 'breaks': for software or hardware reasons it becomes unresponsive. For an MPI program this means that it becomes impossible to send data to it, and any collective operation involving it will hang. Can we deal with this case? Yes, but it involves some programming.

First of all, one of the possible MPI error return codes (section 1.8.2) is `MPI_ERR_COMM`, which can be returned if a processor in the communicator is unavailable. You may want to catch this error, and add a 'replacement processor' to the program. For this, the `MPI_Comm_spawn` can be used:

```
int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info,
                   int root, MPI_Comm comm, MPI_Comm *intercomm,
                   int array_of_errcodes[])
```

But this requires a change of program design: the communicator containing the new process(es) is not part of the old `MPI_COMM_WORLD`, so it is better to set up your code as a collection of inter-communicators to begin with.

1.8.4 Profiling

`MPI_Wtime` gives *wall clock time* from unspecified starting point.

1.8.5 Debugging

There are various ways of debugging an MPI program. Typically there are two cases. In the simple case your program can have a serious error in logic which shows up even with small problems and a small number of processors. In the more difficult case your program can only be run on large scale, or the problem only shows up when you run at large scale. For the second case you, unfortunately, need a dedicated debugging tool, and of course the good ones are expensive. In the first case there are some simpler solutions.

1.8.5.1 Small scale debugging

If your program hangs or crashes even with small numbers of processors, you can try debugging on your local desktop or laptop computer:

```
mpirun -np <n> xterm -e gdb yourprogram
```

This starts up a number of X terminals, each of which runs your program. The magic of `mpirun` makes sure that they all collaborate on a parallel execution of that program. If your program needs commandline arguments, you have to type those in every xterm:

```
run <argument list>
```

See appendix A.2 for more about debugging with `gdb`.

This approach is not guaranteed to work, since it depends on your `ssh` setup; see the discussion in http://www.open-mpi.org/faq/?category=debugging#serial-debuggers.

1.8.5.2 Large scale debugging

Check out *ddt* or *TotalView*.

1.8.5.3 Memory debugging of MPI programs

The commercial parallel debugging tools typically have a memory debugger. For an open source solution you can use *valgrind*, but that requires some setup during installation. See http://valgrind.org/docs/manual/mc-manual.html#mc-manual.mpiwrap for details.

1.8.6 Language issues

MPI is typically written in C, what if you program Fortran?

Assumed shape arrays can be a problem: they need to be copied. That's a problem with Isend.

1.9 Programming projects

1.9.1 Warmpup exercises

We start with some simple exercises.

1.9.1.1 Hello world

The exercises in this section are about the routines introduced in section 1.2.3; for the reference information see section 1.10.1.1.

First of all we need to make sure that you have a working setup for parallel jobs. The example program `helloworld.c` does the following:

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&ntids);
MPI_Comm_rank(MPI_COMM_WORLD,&mytid);
printf("Hello, this is processor %d out of %d\n",mytid,ntids);
MPI_Finalize();
```

Compile this program and run it in parallel. Make sure that the processors do *not* all say that they are `processor 0 out of 1`!

1.9.1.2 Trace output

We want to make trace files of the parallel runs, for which we'll use the TAU utility of the University of Oregon. (For documentation, go to http://www.cs.uoregon.edu/Research/tau/docs.php.) Here are the steps:

- Load two modules:
  ```
  module load tau
  module load jdk64
  ```
- Recompile your program with `make yourprog`. You'll notice a lot more output: that is the TAU preprocessor.
- Now run your program, setting environment variables `TAU_TRACE` and `TAU_PROFILE` to 1, and `TRACEDIR` and `PROFILEDIR` to where you want the output to be. Big shortcut: do
  ```
  make submit EXECUTABLE=yourprog
  ```
  for a batch job or
  ```
  make idevrun EXECUTABLE=yourprog
  ```
  for an interactive parallel run. These last two set all variables for you. See if you can find where the output went...
- Now you need to postprocess the TAU output. Do `make tau EXECUTABLE=yourprog` and you'll get a file `taulog_yourprog.slog2` which you can view with the `jumpshot` program.

1.9.1.3 Collectives

It is a good idea to be able to collect statistics, so before we do anything interesting, we will look at MPI collectives; section 1.4.

Take a look at `time_max.cxx`. This program sleeps for a random number of seconds:

```
wait = (int) ( 6.*rand() / (double)RAND_MAX );
tstart = MPI_Wtime();
sleep(wait);
tstop = MPI_Wtime();
jitter = tstop-tstart-wait;
```

and measures how long the sleep actually was:

```
if (mytid==0)
  sendbuf = MPI_IN_PLACE;
else sendbuf = (void*)&jitter;
MPI_Reduce(sendbuf,(void*)&jitter,1,MPI_DOUBLE,MPI_MAX,0,comm);
```

In the code, this quantity is called 'jitter', which is a term for random deviations in a system.

Exercise 1.9. Change this program to compute the average jitter by changing the reduction operator.

Exercise 1.10. Now compute the standard deviation. For this you need to broadcast the average to all the processors, and do a second reduction. (You can also change the reduction to an `Allreduce`.

Exercise 1.11. Finally, use a gather call to collect all the values on processor zero, and print them out.

Exercise 1.12. Make a TAU trace of these programs. Can you see what algorithms are used for the broadcast and reduction?

Submit your answers by leaving code, output and screenshots in your repository.

1.9.1.4 Linear arrays of processors

In this section you are going to write a number of variations on a very simple operation: all processors pass a data item to the processor with the next higher number.

- In the file `linear-serial.c` you will find an implementation using blocking send and receive calls.
- You will change this code to use non-blocking sends and receives; they require an `MPI_Wait` call to finalize them.
- Next, you will use `MPI_Sendrecv` to arrive at a synchronous, but deadlock-free implementation.
- Finally, you will use two different one-sided scenarios.

In the reference code `linear-serial.c`, each process defines two buffers:

```
int my_number = mytid, other_number=-1.;
```

where `other_number` is the location where the data from the left neighbour is going to be stored.

To check the correctness of the program, there is a gather operation on processor zero:

```
int *gather_buffer=NULL;
if (mytid==0) {
  gather_buffer = (int*) malloc(ntids*sizeof(int));
  if (!gather_buffer) MPI_Abort(comm,1);
}
MPI_Gather(&other_number,1,MPI_INT,
           gather_buffer,1,MPI_INT, 0,comm);
if (mytid==0) {
  int i,error=0;
  for (i=0; i<ntids; i++)
    if (gather_buffer[i]!=i-1) {
      printf("Processor %d was incorrect: %d should be %d\n",
             i,gather_buffer[i],i-1);
      error =1;
    }
  if (!error) printf("Success!\n");
  free(gather_buffer);
}
```

1.9.1.4.1 **Coding with blocking calls** Passing data to a neighbouring processor should be a very parallel operation. However, if we code this naively, with `MPI_Send` and `MPI_Recv`, we get an unexpected serial behaviour, as was explained in section 1.3.1.

```
if (mytid<ntids-1)
  MPI_Ssend( /* data: */ &my_number,1,MPI_INT,
             /* to: */ mytid+1, /* tag: */ 0, comm);
if (mytid>0)
  MPI_Recv( /* data: */ &other_number,1,MPI_INT,
            /* from: */ mytid-1, 0, comm, &status);
```

(Note that this uses an `Ssend`; see section 1.3.1.2 for the explanation why.)

Exercise 1.13. Compile and run this code, and generate a TAU trace file. Confirm that the execution is serial. Does replacing the `Ssend` by `Send` change this?

Let's clean up the code a little.

Exercise 1.14. First write this code more elegantly by using `MPI_PROC_NULL`.

1.9.1.4.2 **A better blocking solution** The easiest way to prevent the serialization problem of the previous exercises is to use the `MPI_Sendrecv` call. This routine acknowledges that often a processor will have a receive call whenever there is a send. For border cases where a send or receive is unmatched you can use `MPI_PROC_NULL`.

Exercise 1.15. Rewrite the code using `MPI_Sendrecv`. Confirm with a TAU trace that execution is no longer serial.

Note that the `Sendrecv` call itself is still blocking, but at least the ordering of its constituent send and recv are no longer ordered in time.

1.9.1.4.3 **Non-blocking calls** The other way around the blocking behaviour is to use `Isend` and `Irecv` calls, which do not block. Of course, now you need a guarantee that these send and receive actions are concluded; in this case, use `MPI_Waitall`.

Exercise 1.16. Implement a fully parallel version by using `MPI_Isend` and `MPI_Irecv`.

1.9.1.4.4 **One-sided communication** Another way to have non-blocking behaviour is to use one-sided communication. During a `Put` or `Get` operation, execution will only block while the data is being transferred out of or into the origin process, but it is not blocked by the target. Again, you need a guarantee that the transfer is concluded; here use `MPI_Win_fence`.

Exercise 1.17. Write two versions of the code: one using `MPI_Put` and one with `MPI_Get`. Make TAU traces.

Investigate blocking behaviour through TAU visualizations.

Exercise 1.18. If you transfer a large amount of data, and the target processor is occupied, can you see any effect on the origin? Are the fences synchronized?

1.9.2 Mandelbrot set

If you've never heard the name *Mandelbrot set*, you probably recognize the picture. Its formal definition is as follows:

A point c in the complex plane is part of the Mandelbrot set if the series x_n defined by

$$\begin{cases} x_0 = 0 \\ x_{n+1} = x_n^2 + c \end{cases}$$

satisfies

$$\forall_n : |x_n| \leq 2.$$

It is easy to see that only points c in the bounding circle $|c| < 2$ qualify, but apart from that it's hard to say much without a lot more thinking. Or computing; and that's what we're going to do.

The way we approach this, is as a *master-worker* model: there is one master processor which gives out work to, and accepts results from, the worker processors.

1.9.2.1 Tools

The driver part of the Mandelbrot program is simple. There is a circle object that can generate coordinates

```
class circle {
 public :
  circle(double stp,int bound);
  void next_coordinate(struct coordinate& xy);
  int is_valid_coordinate(struct coordinate xy);
  void invalid_coordinate(struct coordinate& xy);
```

and a global routine that tests whether a coordinate is in the set, at least up to an iteration bound:

```
int belongs(struct coordinate xy,int itbound) {
  double x=xy.x, y=xy.y; int it;
  for (it=0; it<itbound; it++) {
    double xx,yy;
    xx = x*x - y*y + xy.x;
    yy = 2*x*y + xy.y;
    x = xx; y = yy;
    if (x*x+y*y>4.) {
      return it;
    }
  }
  return 0;
}
```

We use a fairly simple code for the worker processes: they execute a loop in which they wait for input, process it, return the result.

```
void queue::wait_for_work(MPI_Comm comm,circle *workcircle) {
  MPI_Status status; int ntids;
  MPI_Comm_size(comm,&ntids);
  int stop = 0;

  while (!stop) {
    struct coordinate xy;
    int res;

    MPI_Recv(&xy,2,MPI_DOUBLE,ntids-1,0, comm,&status);
    stop = !workcircle->is_valid_coordinate(xy);
    if (stop) res = 0;
    else {
      res = belongs(xy,workcircle->infty);
    }
    MPI_Send(&res,1,MPI_INT,ntids-1,0, comm);
```

```
  }
  return;
}
```

A very simple solution using blocking sends on the master is given:

```
class serialqueue : public queue {
private :
  int free_processor;
public :
  serialqueue(MPI_Comm queue_comm,circle *workcircle)
    : queue(queue_comm,workcircle) {
    free_processor=0;
  };
  /* Send a coordinate to a free processor;
     if the coordinate is invalid, this should stop the process;
     otherwise add the result to the image.
  */
  void addtask(struct coordinate xy) {
    MPI_Status status; int contribution;

    MPI_Send(&xy,2,MPI_DOUBLE,
     free_processor,0,comm);
    MPI_Recv(&contribution,1,MPI_INT,
     free_processor,0,comm, &status);
    if (workcircle->is_valid_coordinate(xy)) {
      coordinate_to_image(xy,contribution);
      total_tasks++;
    }
    free_processor++;
    if (free_processor==ntids-1)
      // wrap around to the first again
      free_processor = 0;
  };
```

Exercise 1.19. Explain why this solution is very inefficient. Make a trace of its execution that bears this out.

1.9.2.2 Bulk task scheduling

The previous section showed a very inefficient solution, but that was mostly intended to set up the code base. If all tasks take about the same amount of time, you can give each process a task, and then wait on them all to finish. A first way to do this is with non-blocking sends.

Exercise 1.20. Code a solution where you give a task to all worker processes using non-blocking sends and receives, and then wait for these tasks with `MPI_Waitall` to finish before you give a new round of data to all workers. Make a trace of the execution of this and report on the total time.
You can do this by writing a new class that inherits from `queue`, and that provides its own `addtask` method:

```
class bulkqueue : public queue {
public :
  bulkqueue(MPI_Comm queue_comm,circle *workcircle)
    : queue(queue_comm,workcircle) {
```

You will also have to override the `complete` method: when the circle object indicates that all coordinates have been generated, not all workers will be busy, so you need to supply the proper `MPI_Waitall` call.

1.9.2.3 Collective task scheduling

Another implementation of the bulk scheduling of the previous section would be through using collectives.

1.9.2.4 Asynchronous task scheduling

At the start of section 1.9.2.2 we said that bulk scheduling mostly makes sense if all tasks take similar time to complete. In the Mandelbrot case this is clearly not the case.

Exercise 1.21. Code a fully dynamic solution that uses `MPI_Probe` or `MPI_Waitany`. Make an execution trace and report on the total running time.

1.9.3 Data parallel grids

1.9.3.1 A realistic programming example

In this section we will gradually build a semi-realistic example program. To get you started some pieces have already been written: as a starting point look at `code/mpi/c/grid.cxx`.

1.9.3.1.1 **Description of the problem** With this example you will investigate several strategies for implementing a simple iterative method. Let's say you have a two-dimensional grid of datapoints $G = \{g_{ij} : 0 \leq i < n_i,\ 0 \leq j < n_j\}$ and you want to compute G' where

$$g'_{ij} = 1/4 \cdot (g_{i+1,j} + g_{i-1,j} + g_{i,j+1} + g_{i,j-1}). \tag{1.1}$$

This is easy enough to implement sequentially, but in parallel this requires some care.

Let's divide the grid G and divide it over a two-dimension grid of $p_i \times p_j$ processors. (Other strategies exist, but this one scales best; see section HPSC-6.4.) Formally, we define two sequences of points

$$0 = i_0 < \cdots < i_{p_i} < i_{p_i+1} = n_i, \quad 0 < j_0 < \cdots < j_{p_j} < i_{p_j+1} = n_j$$

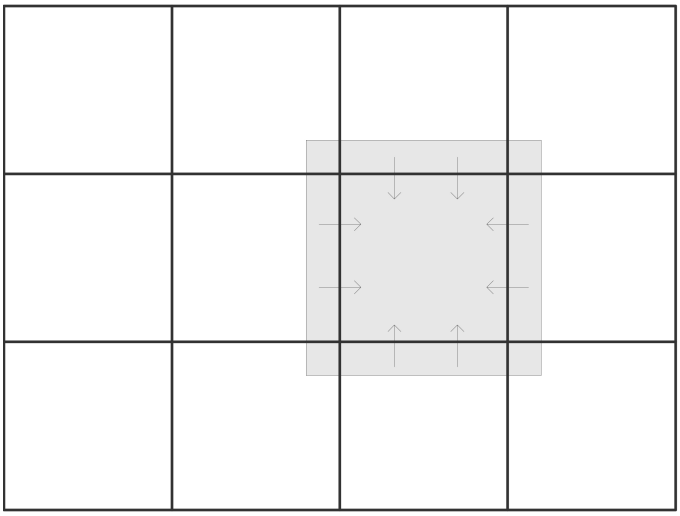

Figure 1.12: A grid divided over processors, with the 'ghost' region indicated

and we say that processor (p, q) computes g_{ij} for

$$i_p \leq i < i_{p+1}, \quad j_q \leq j < j_{q+1}.$$

From formula (1.1) you see that the processor then needs one row of points on each side surrounding its part of the grid. A picture makes this clear; see figure 1.12. These elements surrounding the processor's own part are called the *halo* or *ghost region* of that processor.

The problem is now that the elements in the halo are stored on a different processor, so communication is needed to gather them. In the upcoming exercises you will have to use different strategies for doing so.

1.9.3.1.2 **Code basics** The program needs to read the values of the grid size and the processor grid size from the commandline, as well as the number of iterations. This routine does some error checking: if the number of processors does not add up to the size of `MPI_COMM_WORLD`, a nonzero error code is returned.

```
ierr = parameters_from_commandline
  (argc,argv,comm,&ni,&nj,&pi,&pj,&nit);
if (ierr) return MPI_Abort(comm,1);
```

From the processor parameters we make a processor grid object:

```
processor_grid *pgrid = new processor_grid(comm,pi,pj);
```

and from the numerical parameters we make a number grid:

```
number_grid *grid = new number_grid(pgrid,ni,nj);
```

Number grids have a number of methods defined. To set the value of all the elements belonging to a processor to that processor's number:

```
grid->set_test_values();
```

To set random values:

```
grid->set_random_values();
```

If you want to visualize the whole grid, the following call gathers all values on processor zero and prints them:

```
grid->gather_and_print();
```

Next we need to look at some data structure details.

The definition of the `number_grid` object starts as follows:

```
class number_grid {
public:
  processor_grid *pgrid;
  double *values,*shadow;
```

where `values` contains the elements owned by the processor, and `shadow` is intended to contain the values plus the ghost region. So how does `shadow` receive those values? Well, the call looks like

```
grid->build_shadow();
```

and you will need to supply the implementation of that. Once you've done so, there is a routine that prints out the shadow array of each processor

```
grid->print_shadow();
```

This routine does the sequenced printing that you implemented in exercise **??**.

In the file `code/mpi/c/grid_impl.cxx` you can see several uses of the macro `INDEX`. This translates from a two-dimensional coordinate system to one-dimensional. Its main use is letting you use (i,j) coordinates for indexing the processor grid and the number grid: for processors you need the translation to the linear rank, and for the grid you need the translation to the linear array that holds the values.

A good example of the use of `INDEX` is in the `number_grid::relax` routine: this takes points from the `shadow` array and averages them into a point of the `values` array. (To understand the reason for this particular averaging, see HPSC-4.2.2.2 and HPSC-5.5.3.) Note how the `INDEX` macro is used to index in a `ilength` $\times$ `jlength` target array `values`, while reading from a (`ilength` $+2$) $\times$ (`jlength` $+2$) source array `shadow`.

```
for (i=0; i<ilength; i++) {
  for (j=0; j<jlength; j++) {
    int c=0;
    double new_value=0.;
    for (c=0; c<5; c++) {
```

```
int ioff=i+1+ioffsets[c],joff=j+1+joffsets[c];
new_value += coefficients[c] *
  shadow[ INDEX(ioff,joff,ilength+2,jlength+2) ];
    }
    values[ INDEX(i,j,ilength,jlength) ] = new_value/8.;
  }
}
```

1.10 Reference to the routines

1.10.1 Basics

1.10.1.1 MPI setup

This reference section gives the syntax for routines introduced in section 1.2.3.

1.10.1.2 Language interfaces

The *C++* interface is deprecated as of *MPI 2.2*. It is unclear what is happening.

1.10.1.3 Send and receive buffers

The data is specified as a number of elements in a buffer. The same MPI routine can be used with data of different types, so the standard indicates such buffers as *choice*. The specification of this differs per language:

- In C it is an address, so the clean way is to pass it as `(void*)&myvar`.
- Fortran compilers may complain about type mismatches. This can not be helped.

1.10.1.4 Types

`double` vs `MPI_DOUBLE`, Fortran especially.

Addresses have type `MPI_Aint` or `INTEGER (KIND=MPI_ADDRESS_KIND)` in Fortran. The start of the address range is given in `MPI_BOTTOM`.

1.10.2 Blocking communication

This reference section gives the syntax for routines introduced in section 1.3.1.

The basic send command is

```
int MPI_Send(void *buf,
  int count, MPI_Datatype datatype, int dest, int tag,
  MPI_Comm comm)
```

http://www.mcs.anl.gov/research/projects/mpi/www/www3/MPI_Send.html This routine may not blocking for small messages; to force blocking behaviour use MPI_Ssend with the same argument list. http://www.mcs.anl.gov/research/projects/mpi/www/www3/MPI_Ssend.html

```
int MPI_Recv(void *buf,
  int count, MPI_Datatype datatype, int source, int tag,
  MPI_Comm comm, MPI_Status *status)
```

http://www.mcs.anl.gov/research/projects/mpi/www/www3/MPI_Recv.html The count argument indicates the maximum length of a message; the actual length of the message can be determined with MPI_Get_count.

1.10.3 Non-blocking communication

This reference section gives the syntax for routines introduced in section 1.3.2.

The non-blocking routines have much the same parameter list as the blocking ones, with the addition of an MPI_Request parameter. The MPI_Isend routine does not have a 'status' parameter, which has moved to the 'wait' routine.

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

http://www.mcs.anl.gov/research/projects/mpi/www/www3/MPI_Isend.html

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

http://www.mcs.anl.gov/research/projects/mpi/www/www3/MPI_Irecv.html

There are various 'wait' routines. Since you will often do at least one send and one receive, this routine is useful:

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

http://www.mcs.anl.gov/research/projects/mpi/www/www3/MPI_Waitall.html

It is possible to omit the status array by specifying MPI_STATUSES_IGNORE. Other routines are MPI_Wait for a single request, and MPI_Waitsome, MPI_Waitany.

1.10.4 One-sided communication

This reference section gives the syntax for routines introduced in section 1.3.3.

1.10.4.1 Windows and epochs

This reference section gives the syntax for routines introduced in section 1.3.3.1.

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

The data array must not be `PARAMETER` of `static const`.

1.10.5 Collectives

This reference section gives the syntax for routines introduced in section 1.4.

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

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

On processes that are not the root, the receive buffer is ignored. On the root, you have two buffers, but by specifying `MPI_IN_PLACE`, the reduction call uses the value in the receive buffer as the root's contribution to the operation. On the `Allreduce` call, `MPI_IN_PLACE` can be used for the send buffer of every process.

The scan operations are

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

and

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

The `MPI_Op` operations do not return an error code.

The result of the exclusive scan is undefined on processor 0, and on processor 1 it is a copy of the send value of processor 1. In particular, the `MPI_Op` need not be called on these two processors.

1.10.6 Error handling

This reference section gives the syntax for routines introduced in section 1.8.2.

MPI operators (`MPI_Op`) do not return an error code. In case of an error they call `MPI_Abort`; if `MPI_ERRORS_RETURN` is the error handler, errors may be silently ignore.

1.10.7 More utility stuff

This reference section gives the syntax for routines introduced in section 1.8.4.

MPI has a *wall clock* timer: `MPI_Wtime`

```
double MPI_Wtime(void);
```

which gives the number of seconds from a certain point in the past. Thus, you would write:

```
double tstart,tstop,elapsed;
tstart = MPI_Wtime();
tstop = MPI_Wtime();
elapsed = tstop-tstart;
```

The timer has a resolution of `MPI_Wtick`:

```
double MPI_Wtick(void);
```

Timing in parallel is a tricky issue. For instance, most clusters do not have a central clock, so you can not relate start and stop times on one process to those on another. You can test for a global clock as follows :

```
int *v,flag;
MPI_Attr_get( comm, MPI_WTIME_IS_GLOBAL, &v, &flag );
if (mytid==0) printf(``Time synchronized? %d->%d\n'',flag,*v);
```

1.10.8 Multi-threading

This reference section gives the syntax for routines introduced in section 1.7.

```
int MPI_Init_thread( int *argc, char ***argv, int required, int *provided )
```

- `MPI_THREAD_SINGLE`: each MPI process can only have a single thread.
- `MPI_THREAD_FUNNELED`: an MPI process can be multithreaded, but all MPI calls need to be done from a single thread.
- `MPI_THREAD_SERIALIZED`: a processes can sustain multiple threads that make MPI calls, but these threads can not be simultaneous: they need to be for instance in an OpenMP *critical section*.
- `MPI_THREAD_MULTIPLE`: processes can be fully generally multi-threaded.

1.11 Literature

Online resources:

- MPI 1 Complete reference:
  http://www.netlib.org/utk/papers/mpi-book/mpi-book.html

- Official MPI documents:
  http://www.mpi-forum.org/docs/
- List of all MPI routines:
  http://www.mcs.anl.gov/research/projects/mpi/www/www3/

Tutorial books on MPI:

- Using MPI [2] by some of the original authors.

Chapter 2

Hybrid computing

MPI-2 provides precise interaction with multi-threaded programs `MPI_THREAD_SINGLE` `MPI_THREAD_FUNNELLED` (OpenMP loops) `MPI_THREAD_SERIAL` (Open MP single) `MPI_THREAD_MULTIPLE`

Chapter 3

Support libraries

ParaMesh

Global Arrays

PETSc

Hdf5 and Silo

Appendix A

Practical tutorials

here are some tutorials

A.1 Managing projects with Make

The *Make* utility helps you manage the building of projects: its main task is to facilitate rebuilding only those parts of a multi-file project that need to be recompiled or rebuilt. This can save lots of time, since it can replace a minutes-long full installation by a single file compilation. *Make* can also help maintaining multiple installations of a program on a single machine, for instance compiling a library with more than one compiler, or compiling a program in debug and optimized mode.

Make is a Unix utility with a long history, and traditionally there are variants with slightly different behaviour, for instance on the various flavours of Unix such as HP-UX, AUX, IRIX. These days, it is advisable, no matter the platform, to use the GNU version of Make which has some very powerful extensions; it is available on all Unix platforms (on Linux it is the only available variant), and it is a *de facto* standard. The manual is available at `http://www.gnu.org/software/make/manual/make.html`, or you can read the book [3].

There are other build systems, most notably Scons and Bjam. We will not discuss those here. The examples in this tutorial will be for the C and Fortran languages, but *Make* can work with any language, and in fact with things like TEX that are not really a language at all; see section A.1.6.

A.1.1 A simple example

Purpose. In this section you will see a simple example, just to give the flavour of *Make*.

A.1.1.1 C

Make the following files:

foo.c

bar.c

bar.h and a makefile:

Makefile

The makefile has a number of rules like

```
foo.o : foo.c
<TAB>cc -c foo.c
```

which have the general form

```
target : prerequisite(s)
<TAB>rule(s)
```

where the rule lines are indented by a TAB character.

A rule, such as above, states that a 'target' file `foo.o` is made from a 'prerequisite' `foo.c`, namely by executing the command `cc -c foo.c`. The precise definition of the rule is:

- if the target `foo.o` does not exist or is older than the prerequisite `foo.c`,
- then the command part of the rule is executed: `cc -c foo.c`
- If the prerequisite is itself the target of another rule, than that rule is executed first.

Probably the best way to interpret a rule is:

- if any prerequisite has changed,
- then the target needs to be remade,
- and that is done by executing the commands of the rule.

If you call `make` without any arguments, the first rule in the makefile is evaluated. You can execute other rules by explicitly invoking them, for instance `make foo.o` to compile a single file.

Exercise. Call `make`.

Expected outcome. The above rules are applied: `make` without arguments tries to build the first target, `fooprog`. In order to build this, it needs the prerequisites `foo.o` and `bar.o`, which do not exist. However, there are rules for making them, which `make` recursively invokes. Hence you see two compilations, for `foo.o` and `bar.o`, and a link command for `fooprog`.

Caveats. Typos in the makefile or in file names can cause various errors. In particular, make sure you use tabs and not spaces for the rule lines. Unfortunately, debugging a makefile is not simple. *Make*'s error message will usually give you the line number in the make file where the error was detected.

Exercise. Do `make clean`, followed by `mv foo.c boo.c` and `make` again. Explain the error message. Restore the original file name.

Expected outcome. Make will complain that there is no rule to make `foo.c`. This error was caused when `foo.c` was a prerequisite, and was found not to exist. *Make* then went looking for a rule to make it.

Now add a second argument to the function `bar`. This requires you to edit `bar.c` and `bar.h`: go ahead and make these edits. However, it also requires you to edit `foo.c`, but let us for now 'forget' to do that. We will see how *Make* can help you find the resulting error.

Exercise. Call `make` to recompile your program. Did it recompile `foo.c`?

Expected outcome. Even through conceptually `foo.c` would need to be recompiled since it uses the `bar` function, *Make* did not do so because the makefile had no rule that forced it.

In the makefile, change the line

```
foo.o : foo.c
```

to

```
foo.o : foo.c bar.h
```

which adds `bar.h` as a prerequisite for `foo.o`. This means that, in this case where `foo.o` already exists, *Make* will check that `foo.o` is not older than any of its prerequisites. Since `bar.h` has been edited, it is younger than `foo.o`, so `foo.o` needs to be reconstructed.

Exercise. Confirm that the new makefile indeed causes `foo.o` to be recompiled if `bar.h` is changed. This compilation will now give an error, since you 'forgot' to edit the use of the `bar` function.

A.1.1.2 Fortran

Make the following files:

foomain.F

foomod.F and a makefile:

Makefile If you call `make`, the first rule in the makefile is executed. Do this, and explain what happens.

Exercise. Call `make`.

Expected outcome. The above rules are applied: `make` without arguments tries to build the first target, `foomain`. In order to build this, it needs the prerequisites `foomain.o` and `foomod.o`, which do not exist. However, there are rules for making them, which `make` recursively invokes. Hence you see two compilations, for `foomain.o` and `foomod.o`, and a link command for `fooprog`.

Caveats. Typos in the makefile or in file names can cause various errors. Unfortunately, debugging a makefile is not simple. You will just have to understand the errors, and make the corrections.

Exercise. Do `make clean`, followed by `mv foo.c boo.c` and `make` again. Explain the error message. Restore the original file name.

Expected outcome. Make will complain that there is no rule to make `foo.c`. This error was caused when `foo.c` was a prerequisite, and was found not to exist. *Make* then went looking for a rule to make it.

Now add an extra parameter to `func` in `foomod.F` and recompile.

Exercise. Call `make` to recompile your program. Did it recompile `foomain.F`?

Expected outcome. Even through conceptually `foomain.F` would need to be recompiled, *Make* did not do so because the makefile had no rule that forced it.

Change the line

```
foomain.o : foomain.F
```

to

```
foomain.o : foomain.F foomod.F
```

which adds `foomod.F` as a prerequisite for `foomain.o`. This means that, in this case where `foomain.o` already exists, *Make* will check that `foomain.o` is not older than any of its prerequisites. Since `foomod.F` has ben edited, it is younger than `foomain.o`, so `foomain.o` needs to be reconstructed.

Exercise. Confirm that the corrected makefile indeed causes `foomain.F` to be recompiled.

A.1.2 Variables and template rules

Purpose. In this section you will learn various work-saving mechanism in *Make*, such as the use of variables, and of template rules.

A.1.2.1 Makefile variables

It is convenient to introduce variables in your makefile. For instance, instead of spelling out the compiler explicitly every time, introduce a variable in the makefile:

```
CC = gcc
FC = gfortran
```

and use `${CC}` or `${FC}` on the compile lines:

```
foo.o : foo.c
        ${CC} -c foo.c
foomain.o : foomain.F
        ${FC} -c foomain.F
```

Exercise. Edit your makefile as indicated. First do `make clean`, then `make foo` (C) or `make fooprog` (Fortran).

Expected outcome. You should see the exact same compile and link lines as before.

Caveats. Unlike in the shell, where braces are optional, variable names in a makefile have to be in braces or parentheses. Experiment with what hapens if you forget the braces around a variable name.

One advantage of using variables is that you can now change the compiler from the commandline:

```
make CC="icc -O2"
make FC="gfortran -g"
```

Exercise. Invoke *Make* as suggested (after `make clean`). Do you see the difference in your screen output?

Expected outcome. The compile lines now show the added compiler option `-O2` or `-g`.

Make also has built-in variables:

`$@` The target. Use this in the link line for the main program.
`$^` The list of prerequisites. Use this also in the link line for the program.
`$<` The first prerequisite. Use this in the compile commands for the individual object files.

Using these variables, the rule for `fooprog` becomes

```
fooprog : foo.o bar.o
        ${CC} -o $@ $^
```

and a typical compile line becomes

```
foo.o : foo.c bar.h
        ${CC} -c $<
```

You can also declare a variable

```
THEPROGRAM = fooprog
```

and use this variable instead of the program name in your makefile. This makes it easier to change your mind about the name of the executable later.

Exercise. Construct a commandline so that your makefile will build the executable `fooprog_v2`.

Expected outcome. You need to specify the `THEPROGRAM` variable on the commandline using the syntax `make VAR=value`.

Caveats. Make sure that there are no spaces around the equals sign in your commandline.

A.1.2.2 Template rules

In your makefile, the rules for the object files are practically identical:

- the rule header (`foo.o : foo.c`) states that a source file is a prerequisite for the object file with the same base name;
- and the instructions for compiling (`${CC} -c $<`) are even character-for-character the same, now that you are using *Make*'s built-in variables;
- the only rule with a difference is

  ```
  foo.o : foo.c bar.h
          ${CC} -c $<
  ```

  where the object file depends on the source file and another file.

We can take the commonalities and summarize them in one rule[1]:

```
%.o : %.c
        ${CC} -c $<
%.o : %.F
        ${FC} -c $<
```

This states that any object file depends on the C or Fortran file with the same base name. To regenerate the object file, invoke the C or Fortran compiler with the `-c` flag. These template rules can function as a replacement for the multiple specific targets in the makefiles above, except for the rule for `foo.o`.

The dependence of `foo.o` on `bar.h` can be handled by adding a rule

```
foo.o : bar.h
```

with no further instructions. This rule states, 'if the prerequisite file `bar.h` changed, file `foo.o` needs updating'. *Make* will then search the makefile for a different rule that states how this updating is done.

Exercise. Change your makefile to incorporate these ideas, and test.

A.1.3 Wildcards

Your makefile now uses one general rule for compiling all your source files. Often, these source files will be all the `.c` or `.F` files in your directory, so is there a way to state 'compile everything in this directory'? Indeed there is. Add the following lines to your makefile, and use the variable `COBJECTS` or `FOBJECTS` wherever appropriate.

```
# wildcard: find all files that match a pattern
CSOURCES := ${wildcard *.c}
# pattern substitution: replace one pattern string by another
COBJECTS := ${patsubst %.c,%.o,${SRC}}

FSOURCES := ${wildcard *.F}
FOBJECTS := ${patsubst %.F,%.o,${SRC}}
```

A.1.4 Miscellania

A.1.4.1 What does this makefile do?

Above you learned that issuing the `make` command will automatically execute the first rule in the makefile. This is convenient in one sense[2], and inconvenient in another: the only way to find out what possible actions a makefile allows is to read the makefile itself, or the – usually insufficient – documentation.

---

1. This mechanism is the first instance you'll see that only exists in GNU make, though in this particular case there is a similar mechanism in standard make. That will not be the case for the wildcard mechanism in the next section.
2. There is a convention among software developers that a package can be installed by the sequence `./configure ; make ; make install`, meaning: Configure the build process for this computer, Do the actual build, Copy files to some system directory such as `/usr/bin`.

A better idea is to start the makefile with a target

```
info :
        @echo "The following are possible:"
        @echo "  make"
        @echo "  make clean"
```

Now `make` without explicit targets informs you of the capabilities of the makefile. The at-sign at the start of the commandline means 'do not echo this command to the terminal', which makes for cleaner terminal output; remove the at signs and observe the difference in behaviour.

A.1.4.2 Phony targets

The example makefile contained a target `clean`. This uses the *Make* mechanisms to accomplish some actions that are not related to file creation: calling `make clean` causes *Make* to reason 'there is no file called `clean`, so the following instructions need to be performed'. However, this does not actually cause a file `clean` to spring into being, so calling `make clean` again will make the same instructions being executed.

To indicate that this rule does not actually make the target, declare

```
.PHONY : clean
```

One benefit of declaring a target to be phony, is that the *Make* rule will still work, even if you have a file named `clean`.

A.1.4.3 Predefined variables and rules

Calling `make -p yourtarget` causes make to print out all its actions, as well as the values of all variables and rules, both in your makefile and ones that are predefined. If you do this in a directory where there is no makefile, you'll see that make actually already knows how to compile `.c` or `.F` files. Find this rule and find the definition of the variables in it.

You see that you can customize make by setting such variables as `CFLAGS` or `FFLAGS`. Confirm this with some experimentation. If you want to make a second makefile for the same sources, you can call `make -f othermakefile` to use this instead of the default `Makefile`.

A.1.4.4 Using the target as prerequisite

Suppose you have two different targets that are treated largely the same. You would want to write:

```
PROGS = myfoo other
${PROGS} : $@.o
        ${CC} -o $@ $@.o ${list of libraries goes here}
```

and saying `make myfoo` would cause

```
cc -c myfoo.c
cc -o myfoo myfoo.o ${list of libraries}
```

and likewise for `make other`. What goes wrong here is the use of `$@.o` as prerequisite. In Gnu Make, you can repair this as follows:

```
.SECONDEXPANSION:
${PROGS} : $$@.o
```

A.1.5 Shell scripting in a Makefile

Purpose. In this section you will see an example of a longer shell script appearing in a makefile rule.

In the makefiles you have seen so far, the command part was a single line. You can actually have as many lines there as you want. For example, let us make a rule for making backups of the program you are building.

Add a `backup` rule to your makefile. The first thing it needs to do is make a backup directory:

```
.PHONY : backup
backup :
        if [ ! -d backup ] ; then
          mkdir backup
        fi
```

Did you type this? Unfortunately it does not work: every line in the command part of a makefile rule gets executed as a single program. Therefore, you need to write the whole command on one line:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
```

or if the line gets too long:

```
backup :
        if [ ! -d backup ] ; then \
          mkdir backup ; \
        fi
```

Next we do the actual copy:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
```

But this backup scheme only saves one version. Let us make a version that has the date in the name of the saved program.

The Unix `date` command can customize its output by accepting a format string. Type the following: `date` This can be used in the makefile.

Exercise. Edit the `cp` command line so that the name of the backup file includes the current date.

Expected outcome. Hint: you need the backquote. Consult the Unix tutorial if you do not remember what backquotes do.

If you are defining shell variables in the command section of a makefile rule, you need to be aware of the following. Extend your `backup` rule with a loop to copy the object files:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
        for f in ${OBJS} ; do \
          cp $f backup ; \
        done
```

(This is not the best way to copy, but we use it for the purpose of demonstration.) This leads to an error message, caused by the fact that *Make* interprets `$f` as an environment variable of the outer process. What works is:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
        for f in ${OBJS} ; do \
          cp $$f backup ; \
        done
```

(In this case *Make* replaces the double dollar by a single one when it scans the commandline. During the execution of the commandline, `$f` then expands to the proper filename.)

A.1.6 A Makefile for LaTeX

A.2 Debugging

When a program misbehaves, *debugging* is the process of finding out *why*. There are various strategies of finding errors in a program. The crudest one is debugging by print statements. If you have a notion of where in your code the error arises, you can edit your code to insert print statements, recompile, rerun, and see if the output gives you any suggestions. There are several problems with this:

- The edit/compile/run cycle is time consuming, especially since
- often the error will be caused by an earlier section of code, requiring you to edit, compile, and rerun repeatedly. Furthermore,
- the amount of data produced by your program can be too large to display and inspect effectively, and
- if your program is parallel, you probably need to print out data from all proccessors, making the inspection process very tedious.

For these reasons, the best way to debug is by the use of an interactive *debugger*, a program that allows you to monitor and control the behaviour of a running program. In this section you will familiarize yourself with *gdb*, which is the open source debugger of the *GNU* project. Other debuggers are proprietary, and typically come with a compiler suite. Another distinction is that gdb is a commandline debugger; there are graphical debuggers such as *ddd* (a frontend to gdb) or *DDT* and *TotalView* (debuggers for parallel codes). We limit ourselves to gdb, since it incorporates the basic concepts common to all debuggers.

In this tutorial you will debug a number of simple programs with gdb and valgrind. The files can be downloaded from `http://tinyurl.com/ISTC-debug-tutorial`.

A.2.1 Invoking `gdb`

There are three ways of using gdb: using it to start a program, attaching it to an already running program, or using it to inspect a *core dump*. We will only consider the first possibility.

Here is an exaple of how to start gdb with program that has no arguments (Fortran users, use `hello.F`):

tutorials/gdb/c/hello.c

```
%% cc -g -o hello hello.c
# regular invocation:
%% ./hello
hello world
# invocation from gdb:
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
Copyright 2004 Free Software Foundation, Inc. .... copyright info ....
(gdb) run
Starting program: /home/eijkhout/tutorials/gdb/hello
Reading symbols for shared libraries +. done
hello world

Program exited normally.
```

```
(gdb) quit
%%
```

Important note: the program was compiled with the *debug flag* `-g`. This causes the *symbol table* (that is, the translation from machine address to program variables) and other debug information to be included in the binary. This will make your binary larger than strictly necessary, but it will also make it slower, for instance because the compiler will not perform certain optimizations[3].

To illustrate the presence of the symbol table do

```
%% cc -g -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

and compare it with leaving out the `-g` flag:

```
%% cc -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

For a program with commandline input we give the arguments to the `run` command (Fortran users use `say.F`):

tutorials/gdb/c/say.c

```
%% cc -o say -g say.c
%% ./say 2
hello world
hello world
%% gdb say
.... the usual messages ...
(gdb) run 2
Starting program: /home/eijkhout/tutorials/gdb/c/say 2
Reading symbols for shared libraries +. done
hello world
hello world

Program exited normally.
```

3. Compiler optimizations are not supposed to change the semantics of a program, but sometimes do. This can lead to the nightmare scenario where a program crashes or gives incorrect results, but magically works correctly with compiled with debug and run in a debugger.

A.2.2 Finding errors

Let us now consider some programs with errors.

A.2.2.1 C programs

tutorials/gdb/c/square.c

```
%% cc -g -o square square.c
 %% ./square
5000
Segmentation fault
```

The *segmentation fault* (other messages are possible too) indicates that we are accessing memory that we are not allowed to, making the program abort. A debugger will quickly tell us where this happens:

```
%% gdb square
(gdb) run
50000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x000000000000eb4a
0x00007fff824295ca in __svfscanf_l ()
```

Apparently the error occurred in a function `__svfscanf_l`, which is not one of ours, but a system function. Using the `backtrace` (or `bt`, also `where` or `w`) command we quickly find out how this came to be called:

```
(gdb) backtrace
#0  0x00007fff824295ca in __svfscanf_l ()
#1  0x00007fff8244011b in fscanf ()
#2  0x0000000100000e89 in main (argc=1, argv=0x7fff5fbfc7c0) at square.c:7
```

We take a close look at line 7, and see that we need to change `nmax` to `&nmax`.

There is still an error in our program:

```
(gdb) run
50000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x000000010000f000
0x0000000100000ebe in main (argc=2, argv=0x7fff5fbfc7a8) at square1.c:9
9           squares[i] = 1./(i*i); sum += squares[i];
```

We investigate further:

```
(gdb) print i
$1 = 11237
(gdb) print squares[i]
Cannot access memory at address 0x10000f000
```

and we quickly see that we forgot to allocate `squares`.

By the way, we were lucky here: this sort of memory errors is not always detected. Starting our programm with a smaller input does not lead to an error:

```
(gdb) run
50
Sum: 1.625133e+00

Program exited normally.
```

A.2.2.2 Fortran programs

Compile and run the following program:

tutorials/gdb/f/square.F It should abort with a message such as 'Illegal instruction'. Running the program in gdb quickly tells you where the problem lies:

```
(gdb) run
Starting program: tutorials/gdb//fsquare
Reading symbols for shared libraries ++++. done

Program received signal EXC_BAD_INSTRUCTION, Illegal instruction/operand.
0x0000000100000da3 in square () at square.F:7
7                sum = sum + squares(i)
```

We take a close look at the code and see that we did not allocate `squares` properly.

A.2.3 Memory debugging with Valgrind

Insert the following allocation of `squares` in your program:

```
squares = (float *) malloc( nmax*sizeof(float) );
```

Compile and run your program. The output will likely be correct, although the program is not. Can you see the problem?

To find such subtle memory errors you need a different tool: a memory debugging tool. A popular (because open source) one is *valgrind*; a common commercial tool is *purify*.

tutorials/gdb/c/square1.c Compile this program with `cc -o square1 square1.c` and run it with `valgrind square1` (you need to type the input value). You will lots of output, starting with:

```
%% valgrind square1
==53695== Memcheck, a memory error detector
==53695== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==53695== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==53695== Command: a.out
==53695==
10
==53695== Invalid write of size 4
==53695==    at 0x100000EB0: main (square1.c:10)
==53695==  Address 0x10027e148 is 0 bytes after a block of size 40 alloc'd
==53695==    at 0x1000101EF: malloc (vg_replace_malloc.c:236)
==53695==    by 0x100000E77: main (square1.c:8)
==53695==
==53695== Invalid read of size 4
==53695==    at 0x100000EC1: main (square1.c:11)
==53695==  Address 0x10027e148 is 0 bytes after a block of size 40 alloc'd
==53695==    at 0x1000101EF: malloc (vg_replace_malloc.c:236)
==53695==    by 0x100000E77: main (square1.c:8)
```

Valgrind is informative but cryptic, since it works on the bare memory, not on variables. Thus, these error messages take some exegesis. They state that a line 10 writes a 4-byte object immediately after a block of 40 bytes that was allocated. In other words: the code is writing outside the bounds of an allocated array. Do you see what the problem in the code is?

Note that valgrind also reports at the end of the program run how much memory is still in use, meaning not properly `free`d.

If you fix the array bounds and recompile and rerun the program, valgrind still complains:

```
==53785== Conditional jump or move depends on uninitialised value(s)
==53785==    at 0x10006FC68: __dtoa (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x10003199F: __vfprintf (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x1000738AA: vfprintf_l (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x1000A1006: printf (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x100000EF3: main (in ./square2)
```

Although no line number is given, the mention of `printf` gives an indication where the problem lies. The reference to an 'uninitialized value' is again cryptic: the only value being output is `sum`, and that is not uninitialized: it has been added to several times. Do you see why valgrind calls is uninitialized all the same?

A.2.4 Stepping through a program

Often the error in a program is sufficiently obscure that you need to investigate the program run in detail. Compile the following program

tutorials/gdb/c/roots.c and run it:

```
%% ./roots
```

```
sum: nan
```

Start it in gdb as follows:

```
%% gdb roots
GNU gdb 6.3.50-20050815 (Apple version gdb-1469) (Wed May  5 04:36:56 UTC 20
Copyright 2004 Free Software Foundation, Inc.
....
(gdb) break main
Breakpoint 1 at 0x100000ea6: file root.c, line 14.
(gdb) run
Starting program: tutorials/gdb/c/roots
Reading symbols for shared libraries +. done

Breakpoint 1, main () at roots.c:14
14        float x=0;
```

Here you have done the following:

- Before calling `run` you set a *breakpoint* at the main program, meaning that the execution will stop when it reaches the main program.
- You then call `run` and the program execution starts;
- The execution stops at the first instruction in main.

If execution is stopped at a breakpoint, you can do various things, such as issuing the `step` command:

```
Breakpoint 1, main () at roots.c:14
14        float x=0;
(gdb) step
15        for (i=100; i>-100; i--)
(gdb)
16          x += root(i);
(gdb)
```

(if you just hit return, the previously issued command is repeated). Do a number of `steps` in a row by hitting return. What do you notice about the function and the loop?

Switch from doing `step` to doing `next`. Now what do you notice about the loop and the function?

Set another breakpoint: `break 17` and do `cont`. What happens?

Rerun the program after you set a breakpoint on the line with the `sqrt` call. When the execution stops there do `where` and `list`.

- If you set many breakpoints, you can find out what they are with `info breakpoints`.
- You can remove breakpoints with `delete n` where `n` is the number of the breakpoint.
- If you restart your program with `run` without leaving gdb, the breakpoints stay in effect.
- If you leave gdb, the breakpoints are cleared but you can save them: `save breakpoints <file>`. Use `source <file>` to read them in on the next gdb run.

A.2.5 Inspecting values

Run the previous program again in gdb: set a breakpoint at the line that does the `sqrt` call before you actually call `run`. When the program gets to line 8 you can do `print n`. Do `cont`. Where does the program stop?

If you want to repair a variable, you can do `set var=value`. Change the variable `n` and confirm that the square root of the new value is computed. Which commands do you do?

If a problem occurs in a loop, it can be tedious keep typing `cont` and inspecting the variable with `print`. Instead you can add a condition to an existing breakpoint: the following:

```
condition 1 if (n<0)
```

or set the condition when you define the breakpoint:

```
break 8 if (n<0)
```

Another possibility is to use `ignore 1 50`, which will not stop at breakpoint 1 the next 50 times.

Remove the existing breakpoint, redefine it with the condition `n<0` and rerun your program. When the program breaks, find for what value of the loop variable it happened. What is the sequence of commands you use?

A.2.6 Further reading

A good tutorial: http://www.dirac.org/linux/gdb/.

Reference manual: http://www.ofb.net/gnu/gdb/gdb_toc.html.

Appendix B

Codes

B.1 TAU profiling and tracing

TAU http://www.cs.uoregon.edu/Research/tau/home.php is a utility for profiling and tracing your parallel programs. Profiling is the gathering and displaying of bulk statistics, for instance showing you which routines take the most time, or whether communication takes a large portion of your runtime. When you get concerned about performance, a good profiling tool is indispensible.

Tracing is the construction and displaying of time-dependent information on your program run, for instance showing you if one process lags behind others. For understanding a program's behaviour, and the reasons behind profiling statistics, a tracing tool can be very insightful.

TAU works by adding *instrumentation* to your code: in effect it is a source-to-source translator that takes your code and turns it into one that generates run-time statistics. Doing this instrumentation is fortunately simple: start by having this code fragment in your makefile:

```
ifdef TACC_TAU_DIR
  CC = tau_cc.sh
else
  CC = mpicc
endif

% : %.c
${CC} -o $@ $^
```

To use TAU, do `module load tau`. You have to set the environment variable `TAU_TRACE` to 1, and optionally set `TRACEDIR`.

Bibliography

[1] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19:1749–1783, 2007.

[2] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1994.

[3] Robert Mecklenburg. *Managing Projects with GNU Make*. O'Reilly Media, 3rd edition edition, 2004. Print ISBN:978-0-596-00610-5 ISBN 10:0-596-00610-1 Ebook ISBN:978-0-596-10445-0 ISBN 10:0-596-10445-6.

Appendix C

Index and list of acronyms

AVX Advanced Vector Extensions
BSP Bulk Synchronous Parallel
CAF Co-array Fortran
DAG Directed Acyclic Graph
DSP Digital Signal Processing
FPU Floating Point Unit
FFT Fast Fourier Transform
FSA Finite State Automaton
HPC High-Performance Computing
HPF High Performance Fortran
MIC Many Integrated Cores
MIMD Multiple Instruction Multiple Data
MPI Message Passing Interface
MTA Multi-Threaded Architecture
NUMA Non-Uniform Memory Access
PGAS Partitioned Global Address Space
PDE Partial Diffential Equation
PRAM Parallel Random Access Machine
RDMA Remote Direct Memory Access
RMA Remote Memory Access
SAN Storage Area Network
SaaS Software as-a Service
SFC Space-Filling Curve
SIMD Single Instruction Multiple Data
SIMT Single Instruction Multiple Thread
SM Streaming Multiprocessor
SMP Symmetric Multi Processing
SOR Successive Over-Relaxation
SP Streaming Processor
SPMD Single Program Multiple Data
SPD symmetric positive definite
SSE SIMD Streaming Extensions
TLB Translation Look-aside Buffer
UMA Uniform Memory Access
UPC Unified Parallel C
WAN Wide Area Network

Index