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

Introduction to parallel programming

1.1 Introduction

There is not one way to approach parallel programming. Of course you need to start with the problem that you want to solve, but after that there can be more than one algorithm for that problem, you may have a choice of programming systems to use to implement that algorithm, and finally you have to consider the hardware that will run the software. Sometimes people will argue that certain problems are best solved on certain types of hardware or with certain programming systems. Whether this is so is indeed a question worth discussing, but hard to assess in all its generality.

In this tutorial we will look at one particular problem, Conway's *Game of Life*, and investigate how that is best implemented using different parallel programming systems and different hardware. That is, we will see how different types of parallel programming can all be used to solve the same problem. In the process, you will learn about most of the common parallel programming models and their characteristics.

This tutorial does not teach you to program in any particular system: you will only deal with *pseudo-code* and not run it on actual hardware. However, the discussion will go into detail on the implications of using different types of parallel computers.

(Note. At some points in this discussion there will be references to the book 'Introduction to High-Performance Scientific Computing' by the present author. Such references take the form 'HPSC-1.2.3' for section 1.2.3 of that book.)

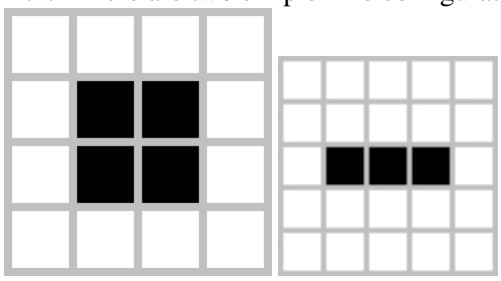
1.1.1 Conway's Game of Life

The Game of Life takes place on a two-dimensional board of *cells*. Each cell can be alive or dead, and it can switch its status from alive to dead or the other way around once per time interval, let's say a second. The rules for cells are as follows. In each time step, each cell counts how many live neighbours it has, where a neighbour is a cell that borders on it horizontallly, vertically, or diagonally. Then:

- If a cell is alive, and it has fewer than two live neighbours, it dies of loneliness.
- A live cell with more than three live neighbours dies from overcrowding.
- A live cell with two or three live neighbours lives on to the next generation.
- A dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The 'game' is that you create an initial configuration of live cells, and then stand back and see what happens.

Exercise 1.1. Here are two simple Life configurations.



Go through the rules and show that the first figure is stationary, and the second figure morphs into something, then morphs back.

The Game of Life is hard to illustrate in a book, since it's so dynamic. Maybe you should search for 'game of life' on YouTube; there are some great animations there.

The rules of Life are very simple, but the results can be surprising. For instance, some simple shapes, called 'gliders', seem to move over the board; others, called 'puffers', move over the board leaving behind other groups of cells. Some configurations of cells quickly disappear, others stay the same or alternate between a few shapes; for a certain type of configuration, called 'garden of Eden', you can prove that it could not have evolved from an earlier configuration. Probably most surprisingly, Life can simulate, very slowly, a computer!

1.1.2 Programming the Game of Life

It is not hard to write a program for Life. Let's say we want to compute a certain number of time steps, and we have a square board of $N \times N$ cells. Also assume that we have a function `life_evaluation` that takes a 3×3 cells and returns the updated status of the center cell:

```
def life_evaluation( cells ):
  count = 0
  for i in [0,1,2]:
    for j in [0,1,2]:
      count += cells[i,j]
  if count<2:
   return 0
  elif count>3:
    return 0
  elif cells[1,1]==1 and (count==2 or count==3):
    return 1
  elif cells[1,1]==0 and count==3:
    return 1
  else: return cells[1,1]
```

The code would then be something like:

```
life_board.create(final_time,N,N)

for t in [0:final_time]:
    for i in [0:N-1]:
        for j in [0:N-1]:
            life_board[t+1,i,j] := life_evaluation( life_board[t,i-1:i+1,j-1:
```

where we don't worry too much about the edge of the board; we can for instance declare that points outside the range $0 \ldots N-1$ are always dead.

The above code created a board for all time steps, which is not strictly necessary. You can save yourself some space by creating only two boards:

```
life_board.create(N,N)
temp_board.create(N,N)

for t in [0:final_time]:
    life_generation( life_board,temp_board )

def life_generation( board,tmp ):
    for i in [0:N-1]:
        for j in [0:N-1]:
            tmp[i,j] = board[i,j]
    for i in [0:N-1]:
        for j in [0:N-1]:
            board[i,j] = life_evaluation( tmp[i-1:i+1,j-1:j+1] )
```

We will call this the basic *sequential implementation*, since it does its computation in a long sequence of steps. We will now explore parallel implementations of this algorithm. You'll see that some look very different from this basic code.

1.1.3 General thoughts on parallelism

In the rest of this tutorial we will use various types of parallelism to explore coding the Game of Life. We start with data parallelim, based on the observation that each point in a Life board undergoes the same computation. Then we go on to task parallelism, which is necessary when we start looking at distributed memory programming on large clusters.

But first we start with some basic thoughts on parallelism.

If you're familiar with programming, you'll have read the above code fragments and agreed that this is a good way to solve the problem. You do one time step after another, and at each time step you compute a new version of the board, one line after another.

Exercise 1.2. The second version used a whole temporary board. Can you come up with an implementation that uses just three temporary lines?

Most programming languages are very explicit about loop constructs: one iteration is done, and then the next, and the next, and so on. This works fine if you have just one processor. However, if you have some form of parallelism, meaning that there is more than one processing unit, you have to figure out which things really have to be done in sequence, and where the sequence is more an artifact of the programming languages.

And by the way, *you* have to think about this yourself. In a distant past it was thought that programmers could write ordinary code, and the compiler would figure out parallelism. This has long proved impossible, so programmers these days accept that parallel code will look differently from sequential code, sometimes very much so.

So let's start looking at Life from a point of analyzing the parallelism. The Life program above used three levels of loops: one for the time steps, and two for the rows and columns of the board. While this is a correct way of programming Life, such explicit sequencing of loop iterations is not strictly necessary for solving the Game of Life problem. For instance, all the cells in the new board are the result of independent computations, and so they can be executed in any order, or indeed simultaneously.

You can view parallel programming as the problem of how to tell multiple processors that they can do certain things simultaneously, and other things only in sequence.

1.2 Parallel variants

We will now discuss various specific parallel realizations of Life.

1.2.1 Data parallelism

In the sequential reference code for Life we updated the whole board in its entirety before we proceeded to the next step. That is, we did the time steps sequentially. We also observed that, in each time step, all cells can be updated independently, and therefore in parallel. If parallelism comes in such small chunks, we call

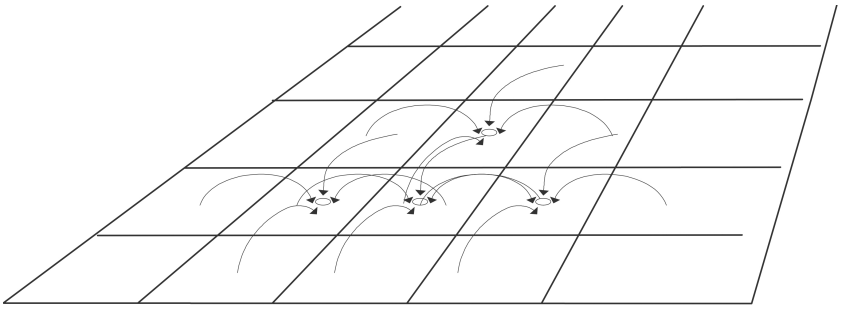


Figure 1.1: Illustration of data parallelism: all points of the board get the same update treatment

it *data parallelism* or *fine-grained parallelism*: the parallelism comes from having lots of data points that are all treated identically. This is illustrated in figure 1.1.

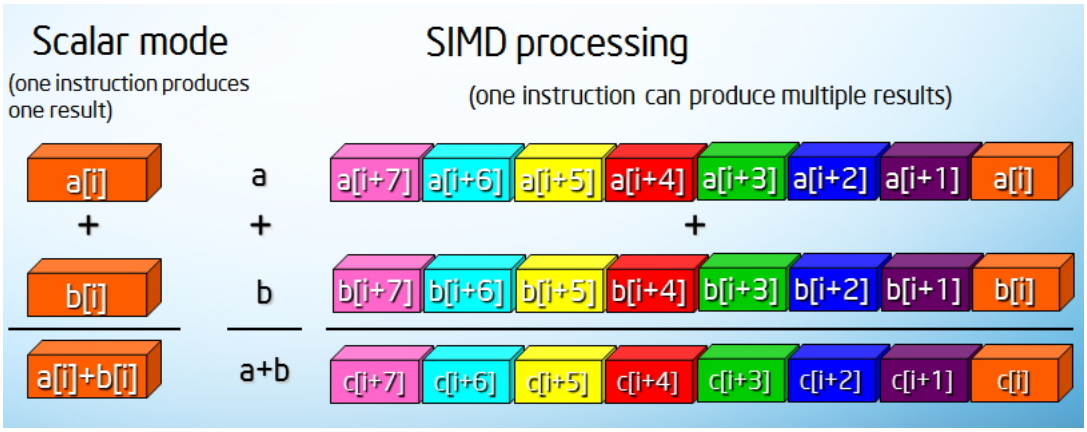


Figure 1.2: SIMD execution of multiple identical instructions

This model of parallel computing is known as Single Instruction Multiple Data (SIMD): the same instruction is performed on multiple data elements; see figure 1.2. An actual computer will of course not have an instruction for computing a Life cell update. Rather, its instructions are things like additions and multiplications. This means that you may need to restructure your code a little for SIMD execution.

A parallel computer that is designed for doing lots of identical operations (on different data elements, of course) has certain advantages. For instance, there needs to be only one central instruction decoding unit that tells the processors what to do, so the design of the individual processors can be much simpler. This means that the processors can be smaller, more power efficient, and easier to manufacture.

In the 1980s and 1990s SIMD computers existed, such as the MasPar and the Connection Machine. They were sometimes called *array processors* since they could operate on an array of data simultaneously. These days, SIMD still exists, but in slightly different guises, and we will now explore what SIMD parallelism looks like in current architectures.

1.2.1.1 Vector instructions

Modern processors have embraced the SIMD concept in an attempt to gain performance without complicating the processor design too much. Instead of operating on a single pair of inputs, you would load two or more pairs of operands, and execute multiple identical operations simultaneously.

Vector instructions constitute SIMD parallelism on a much smaller scale than the old array processors. For instance, Intel processors have had SIMD Streaming Extensions (SSE) instructions for quite some time, which are described as 'two-wide' since they work on two sets of (double precision floating point) operands. The current generation of Intel vector instructions is called Advanced Vector Extensions (AVX), and they can be up to 'eight-wide'; see figure 1.3 for an illustration of four-wide instructions. Since with these instructions you can do four or eight operations per clock cycle, it becomes important to write your code such that the processor can actually use all that available parallelism.

Now suppose that you are coding the Game of Life, which is SIMD in nature, and you want to make sure that is is executed with these vector instructions. Regular programming languages have no way of saying

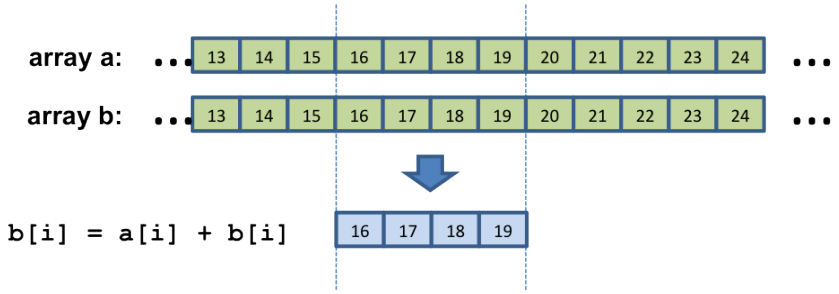


Figure 1.3: Four-wide vector instructions work on four operand pairs at the same time

'do the following operation with vector instructions'. That leaves you with two options:

1. You can start coding in assembly language, or use your compiler's facility for using 'in-line assembly'; or
2. You can hope that the compiler understands your code enough to generate the vector instructions for you.

The first option is no fun, and beyond the capabilities of most programmers, so you'll probably rely on the compiler.

Compilers are pretty smart, but they can not read your mind. If you code is too sophisticated, they may not figure out that vector instructions can be used. On the other hand, you can sometimes help the compiler. For instance, the operation

```
for i in [0:N]:
    count[i,j] += board[i,j+1]
```

can be written as

```
for ii in [0:N/2]:
    i = 2*ii
    count[i,j] += board[i,j+1]
    count[i+1,j] += board[i+1,j+1]
```

In this second version the compiler will have no trouble concluding that there are two operations that can be done simultaneously. This is called *loop unrolling*, specifically, unrolling by 2.

Exercise 1.3. The second code is not actually equivalent to the first. (Hint: consider the case that N is odd.) How can you repair that code? One way of repairing this code is to add a few lines of 'clean-up code' after the unrolled loop. Give the pseudo-code for this.
Now consider the case of unrolling by 4. What does the unrolled code look like now? Think carefully about the clean-up code.

1.2.1.2 Vector pipelining

In the previous section you saw that modern CPUs can deal with applying the same operation to a sequence of data elements. In the case of vector instructions (above), or in the case of GPUs (next section), these

identical operations are actually done simultaneously. There is a different kind of processing that shares many similarities with these and that is sometimes also characterizesd as SIMD: pipelining.

Image a car being put together on an assembly line: as the frame comes down the line one worker puts on the wheels, another the doors, another puts on the steering wheel, et cetera. Thus, the final product, a car, is gradually being constructed; since more than one car is being worked on simultaneously, this is a form of parallelism. And while it is possible for one worker to go through all these steps until the car is finished, it is more efficient to let each worker specialize in just one of the partial assembly operations.

We can do a similar story for computations in a CPU. Let's say we're dealing with floating point numbers of the form $a.b \times 10^c$. Now if we add 5.8×10^1 and 3.2×10^2, we

1. first bring them on the same power of ten: $0.58 \times 10^2 + 3.2 \times 10^2$,
2. do the addition: 3.88×10^2,
3. round to get rid of that last decimal place: 3.9×10^2

So now we can apply the assembly line principle to arithmetic: we can let the processor do each piece in sequence, but a long time ago it was recognized that operations can be split up like that, letting the sub-operations take place in different parts of the processor. The processor can now work on multiple operations at the same time: we start the first operation, and while it is under way we can start a second one, et cetera. In the context of computer arithmetic we call this assembly line the *pipeline*. Figure 1.4 illustrates some

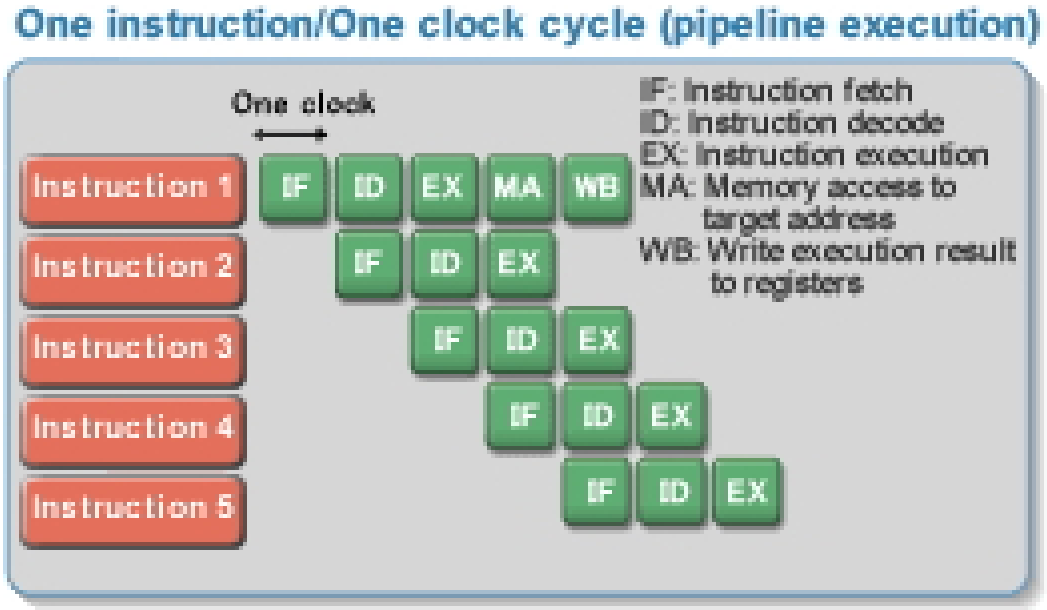


Figure 1.4: Pipelined computer instructions

actual pipelining of computer instructions.

If the pipeline has four stages, after filling the pipeline there will be four operations partially completed at any time. Thus, the pipeline operation is roughly equivalent to, in this example, a fourfold parallelism.

You would hope that this corresponds to a fourfold speedup; the following exercise lets you analyze this precisely.

Exercise 1.4. Assume that all the sub-operations take the same amount of time t. If there are s sub-operations (and assume $s > 1$), how much time does it take for one full calculation? And how much time for two? Recognize that the time for two operations is less than twice the time for a single operation, since the second is started while the first is still in progress. How much time does it take to do n operations? How much time would n operations take if the processor was not pipelined? What is the asymptotic improvement in speed of a pipelined processor over a non-pipelined one?

Around the 1970s this was the definition of a supercomputer: a machine with a single processor that could do floating point operations several times faster than other processors, as long as these operations were delivered as a stream of identical operations. This type of supercomputer essentially died out in the 1990s, but by that time micro-processors had become so sophisticated that they started to include pipelined arithmetic. So the idea of pipelining lives on.

Pipelining has similarities with array operations as described above: they both apply to sequences of identical operations, and they both apply the same operation to all operands. Because of this, pipelining is sometimes also considered *SIMD*.

1.2.1.3 GPUs

Graphics has always been an important application of computers, since everyone likes to look at pictures. With computer games, the demand for very fast generation of graphics has become even bigger. Since graphics processing is often relatively simple and structured, with for instance the same blur operation executed on each pixel, or the same rendering on each polygon, people have made specialized processors for doing just graphics. These can be cheaper than regular processors, since they only have to do graphics-type of operations, and they take the load of the main CPU of your computer.

Wait. Did we just say 'the same operation on each pixel/polygon'? That sounds a lot like SIMD, and in fact it is something very close to it.

Starting from the realization that graphics processing has a lot in common with traditional parallelism, people have tried to use **GPU!**s (**GPU!**s) for SIMD-type numerical computations. Doing so was cumbersome, until NVIDIA came out with the ***CUDA! (CUDA!)*** language. **CUDA!** is a way of explicitly doing data parallel programming: you write piece of code called a *kernel*, which applies to a single data element. You then indicate a two-dimensional or three-dimensional grid of points on which the kernel will be applied.

In pseudo-CUDA, a kernel definition for the game of life and its invocation would look like:

```
kerneldef life_step( board ):
    i = my_i_number()
    j = my_j_number()
    board[i,j] = life_evaluation( board[i-1:i+1,j-1:j+1] )

for t in [0:final_time]:
    <<N,N>>life_step( board )
```

where the `<<N,N>>` notation means that the processors should arrange themselves in an $N \times N$ grid. Every processor has a way of telling its own coordinates in that grid.

There are aspects to CUDA that make it different from SIMD, namely its threading, and for this reason NVIDIA uses the term *Single Instruction Multiple Thread (SIMT)*. We won't go into that here. The main purpose of this section was to remark on the similarities between GPU programming and SIMD array programming.

1.2.2 Loop-based parallelism

The above strategies of parallel programming were all based on assigning certain board locations to certain processors. Since the locations on the board can be updated independently, the processors can then all work in parallel.

There is a slightly different way of looking at this. Rather than going back to basics and reasoning about the problem abstractly, you can take the code of the basic, sequential, implementation of Life. You know that the two inner loops have independent iterations, so is there a way to tell that to the compiler, and let the compiler decide how to execute this?

The popular *OpenMP* system lets the programmer supply this information in comments:

```
def life_generation( board,tmp ):
    # OMP parallel for
    for i in [0:N-1]:
        for j in [0:N-1]:
            tmp[i,j] = board[i,j]
    # OMP parallel for
    for i in [0:N-1]:
        for j in [0:N-1]:
            board[i,j] = life_evaluation( tmp[i-1:i+1,j-1:j+1] )
```

The comments here state that both the `for i` loops are parallel, and therefore their iterations can be executed by whatever parallel resources are available.

In fact, all N^2 iterations of the `i,j` loop nest are independent, which we express as

```
def life_generation( board,tmp ):
    # OMP parallel for collapse(2)
    for i in [0:N-1]:
        for j in [0:N-1]:
            tmp[i,j] = board[i,j]
```

This approach of annotating the loops of a naively written sequential inplementation is a good way of getting started with parallelism. However, the structure of the resulting parallel execution may not be optimally suited to a given computer architecture. In the next section we will look at different ways of getting task parallelism, and why they may computationally be preferable.

1.2.3 Coarse-grained data parallelism

So far we have looked at implications of the fact that each cell in a step of Life can be updated independently. This view leads to tiny grains of computing, which are a good match to the innermost components of a processor core. However, if you look at parallelism on the level of the cores of a processor there are disadvantages to assigning small-grained computations randomly to the cores. (Most of these have something to do with the way memory is designed. For a detailed discussion see section HPSC-**??**.) Therefore, we are motivated to look at computations in larger chunks than a single cell update.

For instance, we can divide the Life board in lines or square patches, and formulate the algorithm in terms of operations on such larger units. This is called *coarse-grained parallelism*, and we will look at several variants of it.

1.2.3.1 Shared memory parallelism

In the approaches to parallelism mentioned so far we have implicitly assumed that a processing element can actually get hold of any data element it needs. Or look at it this way: a program has a set of instructions and so far we have assumed that any processor can execute any instruction.

This is certainly the case with multicore processors, where all cores can equally easily read any element from memory. We call this *shared memory*; see figure 1.5.

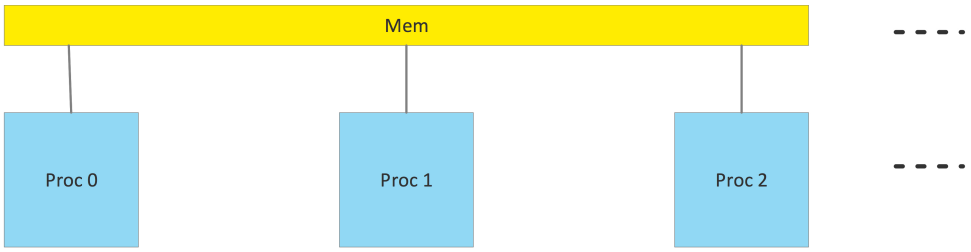


Figure 1.5: Illustration of shared memory: all processors access the same memory

In the CUDA example each processing element essentially reasoned ‘this is my number, and therefore I will work on this element of the array’. In other words, each processing elements assumes that it can work on any data element, and this works because a GPU has a form of shared memory.

While it is convenient to program this way, it is not possible to make arbitrarily large computers with shared memory. The shared memory approaches discussed so far are limited by the amount of memory you can put in a single PC, at the moment about 1 terabytes (which costs a lot of money!), or the processing power that you can associate with shared memory, at the moment around 48 cores.

If you need more processing power, you need to look at clusters, and ‘distributed memory programming’.

1.2.3.2 Distributed memory parallelism

Clusters, also called *distributed memory* computers, can be thought of as a large number of PCs with network cabling between them. This design can be scaled up to a much larger number of processors than

shared memory. In the context of cluster, each of these PCs is called a *node*. The network can be *Ethernet* or something more sophisticated like *Infiniband*.

Since all nodes work together, a cluster is in some sense one large computer. Since the nodes are also to an extent independent, this type of parallelism is called *Multiple Instruction Multiple Data (MIMD)*: each node has its own data, and executes its own program. However, most of the time the nodes will all execute the same program, so this model is often called *Single Program Multiple Data (SPMD)*; see figure 1.6. The advantage of this design is that tieing together thousands of processors allows you to run very large

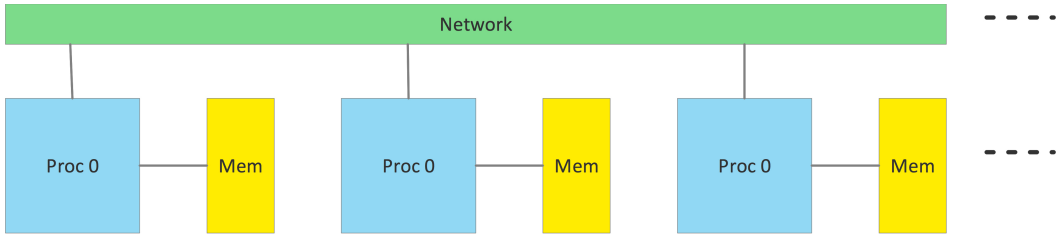


Figure 1.6: Illustration of distributed memory: every processor has its own memory and is connected to others through a network

problems. For instance, the almost 13 thousand processors of the Stampede supercomputer[1] (figure 1.7)

Figure 1.7: The Stampede supercomputer at the Texas Advanced Supercomputing Center

have almost 200 Terabyte of memory. Parallel programming on such a machine is a little harder that what

1. Stampede has more than 6400 nodes, each with 2 Intel Sandy Bridge processors. There is also an Intel Xeon Phi co-processors, but we don't count that for the moment.

we discussed above. First of all we have to worry about how to partition the problem over this *distributed memory*. But more importantly, our above assumption that each processing element can get hold of every data element no longer holds.

It is clear that each cluster node can access its local problem data without any problem, but this is not true for the 'remote' data on other nodes. In the former case the program simply reads the memory location; in the latter case accessing data is only possible because there is a network between the processors (see the orange cabling in the Stampede picture). Accessing data over the network probably involves an Operating System call and accessing the network card, both of which are slow operations.

1.2.3.3 Distributed memory programming

By far the most popular way for programming distributed memory machines is by using the Message Passing Interface (MPI) library. This library adds functionality to an otherwise normal C or Fortran program for exchanging data with other processors. The name derives from the fact that the technical term for exchanging data between distributed memory nodes is *message passing*.

Let's explore how you would program with MPI. We start with the case that each processor stores the cells of a single line of the Life board, and that processor p stores line p. In that case, to update that line it needs the lines above and below it, which come from processors $p-1$ and $p+1$ respectively. In MPI terms, the processor needs to receive a message from each of these processors, containing the state of their line.

Let's build up the basic structure of an MPI program. Througout this example, keep in mind that we are working in SPMD mode: all processes execute the same program. As illustrated in figure 1.8 a process

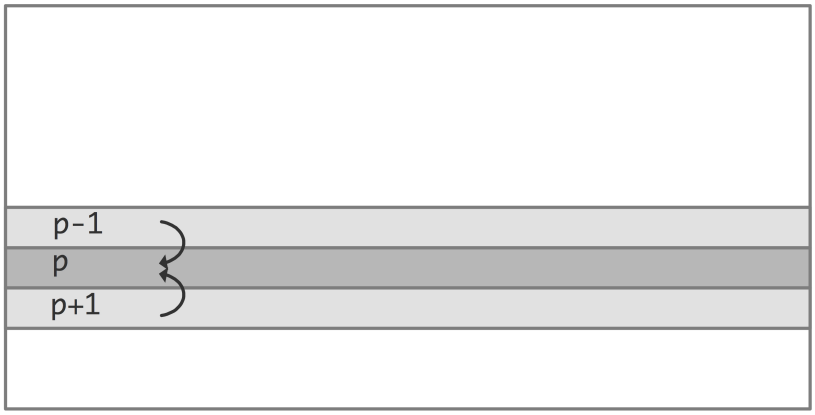


Figure 1.8: Processor p receives a line of data from $p-1$ and $p+1$

needs to get data from its neighbours. The first step is for each process to find out what its number is, so that it can name its neigbours.

```
p = my_processor_number()
```

Then the process can actually receive data from those neighbours (we ignore complications from the first and last line of the board here).

```
high_line = MPI_Receive(from=p-1,cells=N)
low_line = MPI_Receive(from=p+1,cells=N)
```

With this, it is possible to update the data stored on this process:

```
tmp_line = my_line.copy()
my_line = life_line_update(high_line,tmp_line,low_line,N)
```

Unfortunately, there is more to MPI than that. The most common way of using the library is through *two-sided communication*, where for each receive action there is a corresponding send action: a process can not just receive data from its neighbours, the neighbours have to send the data.

But now we recall the SPMD nature of the computation: if your neighbours send to you, you are someone else's neighbour and need to send to them. So the program code will contain both send and receive calls.

The following code is closer to the truth.

```
p = my_processor_number()
# send my data
my_line.MPI_Send(to=p-1,cells=N)
my_line.MPI_Send(to=p+1,cells=N)
# get data from neighbours
high_line = MPI_Receive(from=p-1,cells=N)
low_line = MPI_Receive(from=p+1,cells=N)
tmp_line = my_line.copy()
# do the local computation
my_line = life_line_update(high_line,tmp_line,low_line,N)
```

Since this is a general tutorial, and not a course in MPI programming, we'll leave the example phrased in pseudo-MPI, ignoring many details. However, this code is still not entirely correct conceptually. Let's fix that.

Conceptually, a process would send a message, which disappears somewhere in the network, and goes about its business. The receiving process would at some point issue a receive call, get the data from the network, and do something with it. This idealized behaviour is illustrated in the left half of figure 2.4. Practice is different.

Suppose a process sends a large message, something that takes a great deal of memory. Since the only memory in the system is on the processors, the message has to stay in the memory of one process, until it is copied to the other. We call this behaviour *blocking communication*: a send call will wait until the receiving processor is indeed doing a receive call. That is, the sending code is blocked until its message is received.

But this is a problem: if every process p starts sending to $p-1$, everyone is waiting for someone else to do a receive, and no one is actually doing a receive. This sort of situation is called *deadlock*.

Exercise 1.5. Do you now see why the code fragment leads to deadlock? Can you come up with a clever rearrangement of the sends and receives so that there is no deadlock?

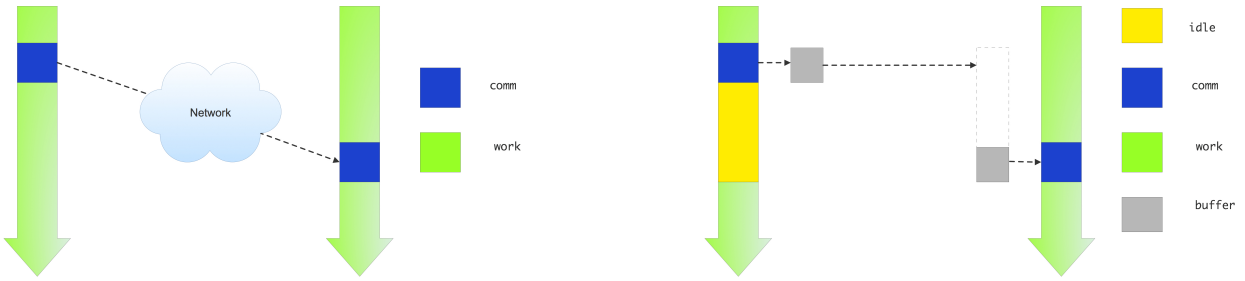


Figure 1.9: Illustration of 'ideal' and 'blocking' send

1.2.3.4 Task scheduling

All parallel realizations of Life you have seen so far were based on taking a single time step, and applying parallel computing to the updates in that time step. This was based on the fact that the points in the new timestep can be computed independently. But the outer iteration has to be done in that order. Right?

Well...

Let's suppose you want to compute the board *two* timesteps from now, without explicitly computing the next timestep. Would that be possible?

Exercise 1.6. Life expresses the value in `i,j` at time $t+1$ as a simple function of the 3×3 patch `i-1:i+1,j-1:j+1` at time t. Convince yourself that the value in `i,j` at $t+2$ can be computed as a function of a 5×5 patch at t.
Can you formulate rules for this update over two timesteps? Are these rules as elegant as the old ones, just expressed in a count of live and dead cells? If you would code the new rules as a case statement, how many clauses would there be? Let's not persue this further...

This has taught us something about dependence and independence. We can update the value at `i,j` based on 3×3 current points. Likewise, we can compute the value at `i,j` two steps away if we know 5×5 current points, et cetera.

The conclusion is that you do not need to finish a whole time step before you can start the next: for each point update only certain other points are needed, and not the whole board. If multiple processors are updating the board, they do not need to be working on the same timestep. This is sometimes called *asynchronous computing*. It means that processors do not have to synchronize what time step they are working on: within restrictions they can be working on different time steps.

Exercise 1.7. Just how independent can processors be? If processor i,j is working on time t, can processor $i+1,j$ be working on $t+2$? Can you give a formal description of how far out of step processor i,j and i',j' can be?

The previous sections were supposed to be about task parallelism, but we didn't actually define the concept of task. Informally, a processor receiving border information and then updating its local data sounds like something that could be called a task. To make it a little more formal, we define a task as some operations done on the same processor, plus a list of other tasks that have to be finished before this task can be finished.

This concept of computing is also known as *dataflow*: data flows as output of one operation to another; an operation can start executing when all its inputs are available. Another concept connected to this defini-

tion of tasks is that of a Directed Acyclic Graph (DAG): the dependencies between tasks form a graph, and you can not have cycles in this graph, otherwise you could never get started...

You can interpret the MPI examples in terms of tasks. The local computation of a task can start when data from the neighbouring tasks is available, and a task finds out about that by the messages from those neighbours coming in. However, this view does not add much information.

On the other hand, if you have shared memory, and tasks that do not all take the same amount of running time, the task view can be productive. In this case, we adopt a *master-worker model*: there is one master process that keeps a list of tasks, and there are a number of worker processors that can execute the tasks. The master executes the following program:

```
while there_are_tasks_left():
    for r in running_tasks:
        if r.finished():
            for t in scheduled_tasks:
                t.mark_input(r)
    t = find_available_task()
    p = find_available_processor()
    schedule(t,p)
```

1. The master finds which running tasks are finished;
2. For each scheduled task, if it needs the data of a finished task, mark that the data is available;
3. Find a task that can now execute, find a processor for it, and execute it there.

The master-worker model assumes that in general there are more available tasks than processors. In the Game of Life we can easily get this situation if you divide the board in more subdomains than there are processing elements. (Why would you do that? This mostly makes sense if you think about the memory hierarchy and cache sizes; see section HPSC-**??**.) So with $N \times N$ subdomains and T time step, we define the queue of tasks:

```
for t in [0:T]:
  for i in [0:N]:
    for j in [0:N]:
      task( id=[t+1,i,j],
        prereqs=[ [t,i,j],[t,i-1,j],[t,i+1,j] # et cetera
                ] )
```

Exercise 1.8. Argue that this model mostly makes sense on shared memory. Hint: if you would execute this model on distributed memory, how much data needs to be moved in general when you start a task?

1.3 Advanced topics

1.3.1 Data partitioning

The previous sections approached parallelization of the Game of Life by taking the sequential implementation and taking the basic loop structure. For instance, in section 1.2.3.3 we assigned a number of lines to each processor. This corresponds to a *one-dimensional partitioning* of the data. Sometimes, however, it is a good idea to use a two-dimensional one instead. (See figure 1.10 for an illustration of the basic idea.) In

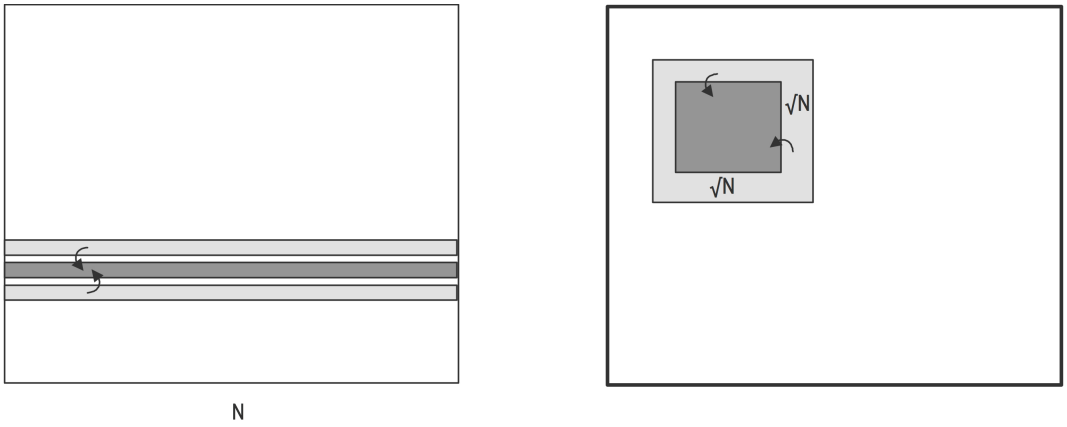


Figure 1.10: One-dimensional and two-dimensional distribution communication

this section you'll get the flavour of the argument.

Suppose each processor stores one line of the Life board. As you saw in the previous section, to update that line it needs to receive two lines worth of data, and this takes time. In fact, receiving one item of data from another node is much slower than reading one item from local memory. If we inventory the cost of one timestep in the distributed case, that comes down to

1. Receiving $2N$ Life cells from other processors[2]; and
2. Adding $8N$ values together to get the counts.

For most architectures, the cost of sending and receiving data will far outweigh the computation.

Let us now assume that we have N processors, each storing a $\sqrt{N} \times \sqrt{N}$ part of the Life board. We sometimes call this the processor's *subdomain*. To update this, a processor now needs to receive data from the lines above, under, and to the left and right of its part (we are ignoring the edge of the board here). That means four messages, each of size $\sqrt{N} + 2$. On the other hand, the update takes $8N$ operations. For large enough N, the communication, which is slow, will be outweighed by the computation, which is much faster.

Our analysis here was very simple, based on having exactly N processors. However, in many cases a more refined analysis gives the same conclusion: a two-dimensional distribution is to be prefered over a one-dimensional one; see for instance section HPSC-**??** for the analysis of the matrix-vector product algorithm.

2. For now we only count the transmission cost per item; there is also a one-time cost for each transmission, called the *latency*. For large enough messages we can ignore this; for details see HPSC-**??**.

Let's do just a little analysis on the following scenario:

- You have a parallel machine where each processor has an amount M of memory to store the Life board.
- You can buy extra processors for this machine, thereby expanding both the processing power (in operations per second) and the total memory.
- As you buy more processors, you can store a larger Life board: we're assuming that the amount M of memory is kept constant. (This strategy of scaling up the problem as you scale up the computer is called *weak scaling*. The scenario where you only increase the number of processors, keeping the problem fixed and therefore putting less and less Life cells on each processor, is called *strong scaling*.)

Let P be the number of processors, and N the size of the board. In terms of the amount of memory M you then have:

$$M = N^2/P.$$

Let's now consider a one-dimensional distribution. (Left half of figure 1.11.) Every processor but the first

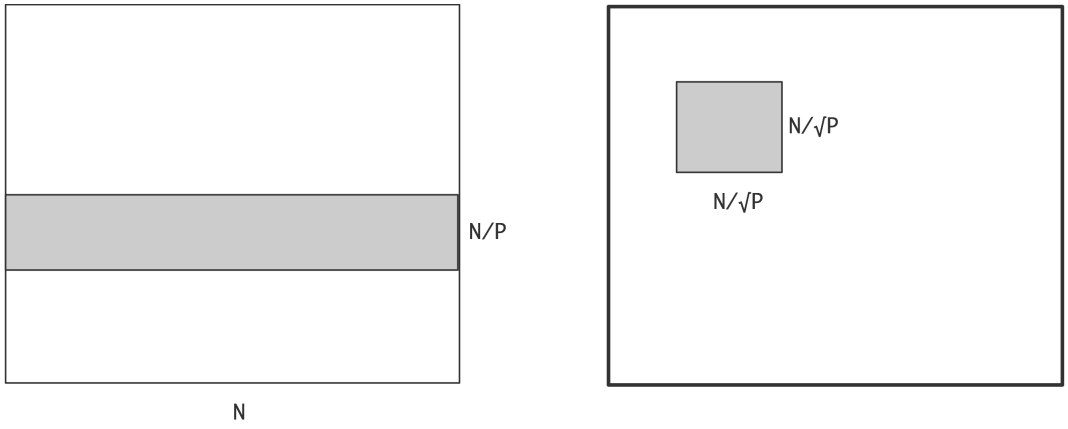


Figure 1.11: One-dimensional and two-dimensional distribution of a Life board

and last one needs to communicate two whole lines, meaning $2N$ elements. If you express this in terms of M you find a formula that contains the variable P. This means that as you buy more processors, and can store a larger problem, the amount of communication becomes a function of the number of processors.

Exercise 1.9. Show that the amount of communication goes up with the number of processors. On the other hand, show that the amount of work stays constant, and that it corresponds to a perfect distribution of the work over the processors.

Now consider a two-dimensional distribution. (Right half of figure 1.11.) Every processor that is not on the edge of the board will communicate with eight others. With the four 'corner' processors only a single item is exchanged.

Exercise 1.10. What is the amount of data exchanged with the processors left/right and top/bottom? Show that, expressed in terms of M, this formula does not contain the variable P. Show that, again, the work is constant in N and P.

The previous two exercises demonstrate an important point. Both the one and two-dimensional distribution lead to a perfect parallelization of the work. On the other hand, with the one-dimensional distribution the communication cost goes up with the number of processors, so the algorithm becomes less and less efficient.

1.3.2 Combining work, minimizing communication

In most of the above discussion we have considered the parallel update of the Life board as one bulk operation that is executed in sequence: you do all communication for one update step, and then the communication for the next, et cetera.

Now, the time for a communication between two processes has two components: there is a startup time (known as 'latency'), and then there is a time per item communicated. This is usually rendered as

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

where the α is the startup time and β the per-item time.

Exercise 1.11. Show that sending two messages of length n takes longer than one message of length $2n$, in other words $T(2n) < 2T(n)$. For what value of n is the overhead 50%? 10%?

If the ratio between α and β is large there is clearly an incentive to combine messages. For the naive parallelization strategies considered above there is no easy way to do this. However, there is a way to communicate only once every *two* updates. This communication will be larger, but there wil be savings in the startup cost.

First we must observe that to update a single Life cell by one time step we need the eight cells around it. So to update a cell by two time steps we need those eight cells plus the ones around it. This is illustrated in figure 1.12. If a processor has the responsibility for updating a subsection of the board, it needs the *halo region* around it. For a single update, this is a halo of width one, and for two updates this is a halo of width two.

Let's analyze the cost of this scheme. In the one-step-at-a-time implementation a processor does the following:

1. Receives four messages of length $\sqrt{N}$ and four of length 1; and
2. Then updates the part of the board it owns to the next time step.

In order to update its subdomain two timesteps, the following is needed:

1. Receive four messages of size $2\sqrt{N}$ and four of size 2;
2. Compute the updated values at time $t + 1$ of the subdomain plus a locally stored border of thickness 1 around it;
3. Update precisely the owned subdomain to its state at $t + 2$.

So now you send slightly more data, and you compute a little more, but you save half the latency cost.

1.3.3 Load balancing

The basic motivation of parallel computing is to be able to compute faster. Ideally, having p processors would make your computation p times faster (see section HPSC-**??** for definition and discussion of

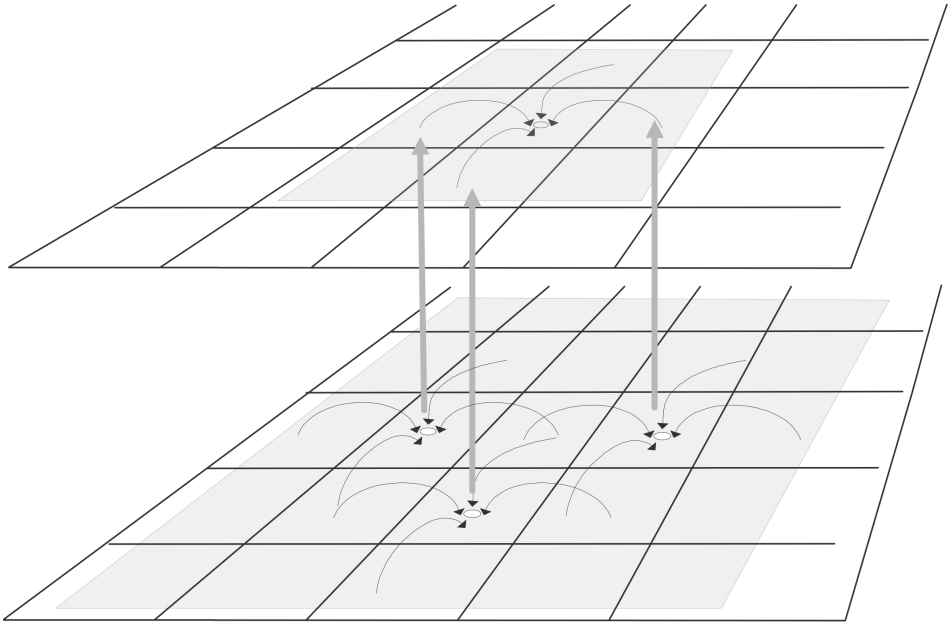


Figure 1.12: Two steps of Life updates

speedup), but practice doesn't always live up to that ideal. There are many reasons for this, but one is that the work may not be evenly divided between the processors. If some processors have more work than others, they will still be computing while the others have finished and are sitting idle. This is called *load imbalance*.

Exercise 1.12. Compute the speedup from using p processors if one processor has a fraction ϵ more work than the others; the others are assumed to be perfectly balanced. Also compute the speedup from the case where one processor having ϵ less work than all others. Which of the two scenarios is worse?

Clearly, there is a strong incentive for having a well-balanced load between the processors. Unfortunately, sometimes the workload changes during the run of a program, and you want to rebalance it. Doing so can be tricky, since it requires problem data to be moved, and processor have to reallocate and rearrange their data structures.

PART I

MPI

Chapter 2

MPI tutorial

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.

2.1 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.

2.1.1 History

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

2.1.2 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 2.1. The user types the command `mpirun` and supplies

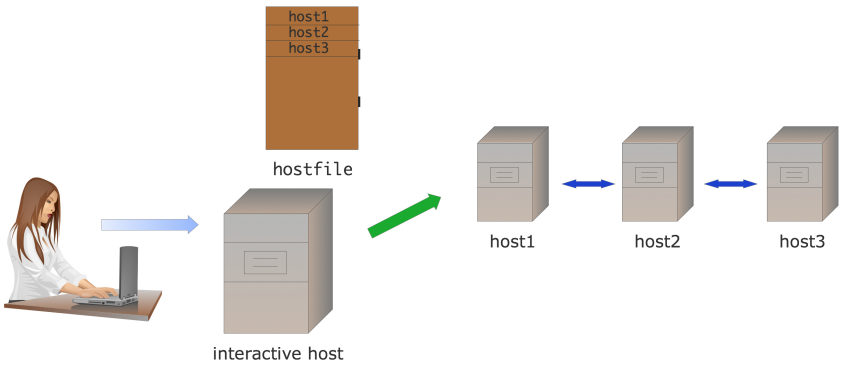


Figure 2.1: Interactive MPI setup

- 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 2.2) 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`

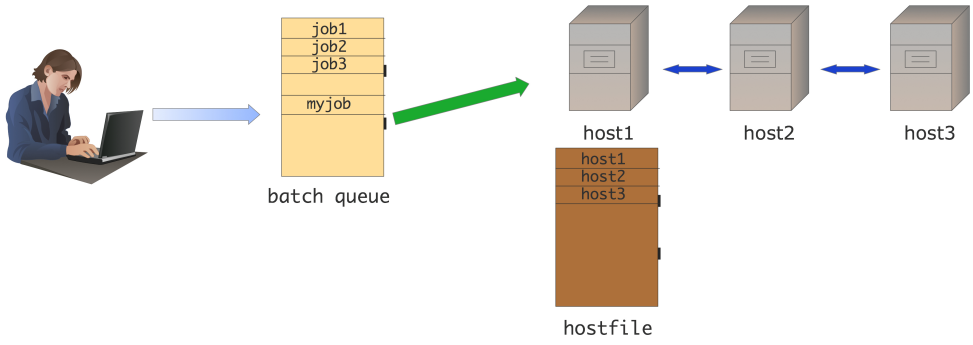


Figure 2.2: Batch MPI setup

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

You see that in both scenarios the parallel program is started by the `mpirun` command, which only supports an 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').

2.1.3 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.

2.2 Basic concepts

2.2.1 Initialization / finalization

The reference for the commands introduced here can be found in section 3.1.1.

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 **??** 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 2.4.1).

2.2.2 Communicators

Before we can discusss any further MPI routines we need to take a first look at an important concept: that of the *communicator*. A communicator stands for a group of processes. Thus, almost all MPI routines have a communicator argument: you can never ask how many processes there are, or send data to process 5, you have to ask how many processes in a given communicator, or send data to process 5 in a communicator.

There are several reasons for having communicators. One is that sometimes you may want to divide your processes in two groups, each of which engages in a different activity. Another reason is for ease of writing software libraries. If a software library that uses MPI starts by creating its own communicator, even if it contains the same processes as the communicator in the calling program, it can safely use that and never run the risk of confusion between messages in the library code and messages in the user code. We will discuss this in more detail later on.

For most of your MPI programming, the only communicator you will use is `MPI_COMM_WORLD` which contains all your processes. In section 2.6 you will learn of the mechanisms for creating other communicators from it.

2.2.3 Distinguishing between processes

The reference for the commands introduced here can be found in section 3.1.2.

In the SPMD model you run the same executable on each of a set of processors; see section HPSC-**??**. So how can you do anything useful if all processors run the same code? Here is where your first two MPI routines come in, which query the `MPI_COMM_WORLD` communicator and each process' place in it.

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:

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

2.3 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.

2.3.1 Blocking communication

The reference for the commands introduced here can be found in section 3.3.

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 2.3. Of course, if the receiving process gets

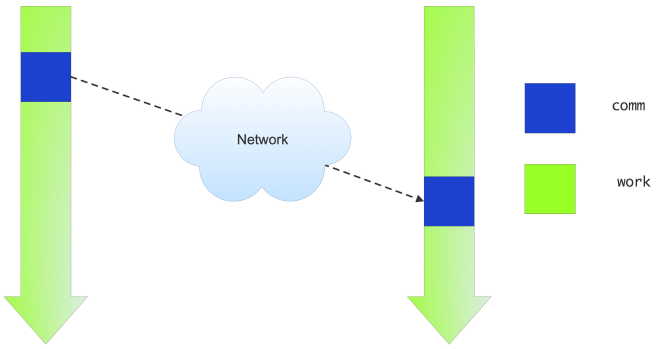


Figure 2.3: Illustration of an ideal send-receive interaction

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-**??**.

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 2.4. This is known as

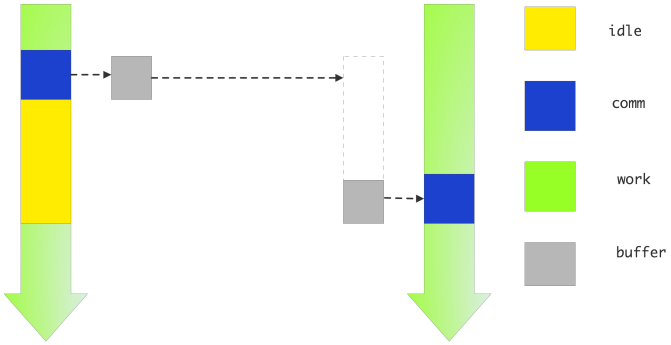


Figure 2.4: Illustration of a blocking communication: the sending processor is idle until the receiving processor issues and finishes the receive call

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-**??**.

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:

```
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*.

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

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

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

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 serialization*: only one processor is active at any time, so what should have been a parallel operation becomes a sequential

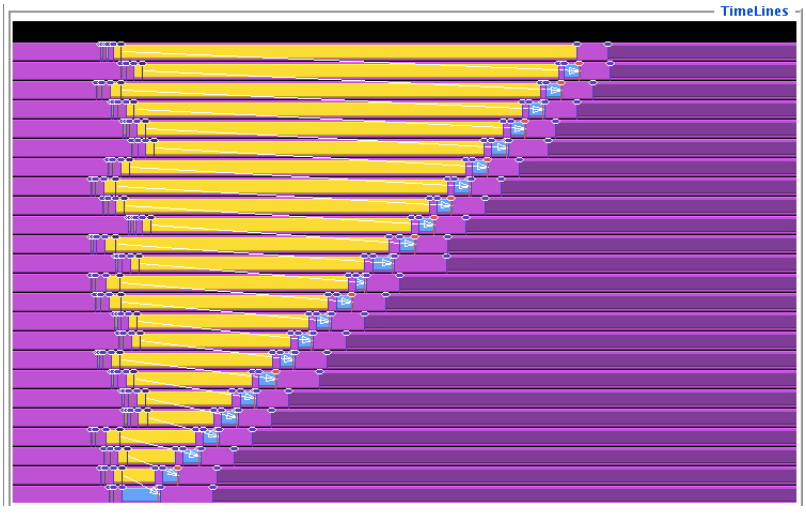


Figure 2.5: Trace of a simple send-recv code

one. This is illustrated in figure 2.5.

Exercise 2.1. Modify your earlier code, run it and reproduce the trace output of figure 2.5.

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

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.

2.3.1.1 *Deadlock-free blocking communication*

The reference for the commands introduced here can be found in section 3.4.

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

```
MPI_Sendrecv( /* 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 2.3. Take your code from exercise 2.1 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?

2.3.1.2 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 2.3.2):

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

2.3.1.3 Speculative messaging

The reference for the commands introduced here can be found in section 3.3.1.

In some applications it makes sense that a message can come from one of a number of processes. In this case, it is possible to specify `MPI_ANY_SOURCE` as the source. To find out where the message actually came from, you would use the `MPI_SOURCE` field of the status object that is delivered by `MPI_Recv` or the `MPI_Wait...` call after an `MPI_Irecv`.

There are various scenarios where receiving from 'any source' makes sense. One is that of the *master-worker model*. The master task would first send data to the worker tasks, then issues a blocking wait for the data of whichever process finishes first.

2.3.2 Non-blocking communication

The reference for the commands introduced here can be found in section 3.5.

In the previous section you saw that blocking communication makes programming tricky if you want to avoid deadlock and performance problems. The main advantage of these routines is that you have full control about where the data is: if the send call returns the data has been successfully received, and the send buffer can be used for other purposes or de-allocated.

By constrast, 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 2.6.

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
....
```

There are several wait calls:

- `MPI_Wait` waits for a a single request. If you are indeed waiting for a single nonblocking communication to complete, this is the right routine. If you are waiting for multiple requests you could call this routine in a loop.

  ```
  for (p=0; p<nrequests ; p++)
    MPI_Wait(request[p],&(status[p]));
  ```

  However, this would be inefficient if the first request is fulfilled much later than the others: your waiting process would have lots of idle time. In that case, use one of the following routines.
- `MPI_Waitall` allows you to wait for a number of requests, and it does not matter in what sequence they are satisfied. Using this routine is easier to code than the loop above, and it could be more efficient.
- The 'waitall' routine is good if you have need all nonblocking communications to be finished before you can proceed with the rest of the program. However, sometimes it is possible to take action as each request is satisfied. In that case you could use `MPI_Waitany` and write:

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

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

- `MPI_Waitsome` is very much like `Waitany`, except that it returns multiple numbers, if multiple requests are satisfied. Now the status argument is an array of `MPI_Status` objects.

Figure 2.7 shows the trace of a non-blocking execution using `MPI_Waitall`.

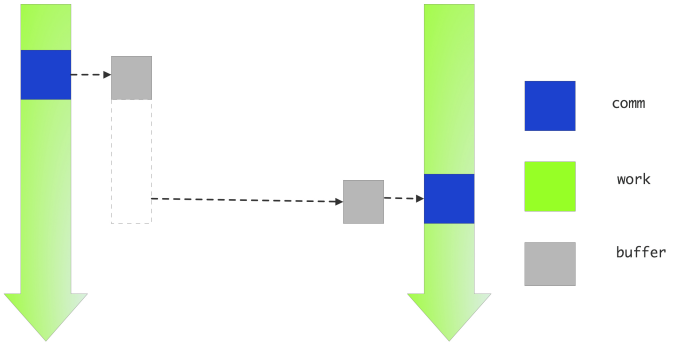


Figure 2.6: Illustration of a non-blocking communication: the sending processor immediately continues execution after issueing the send call

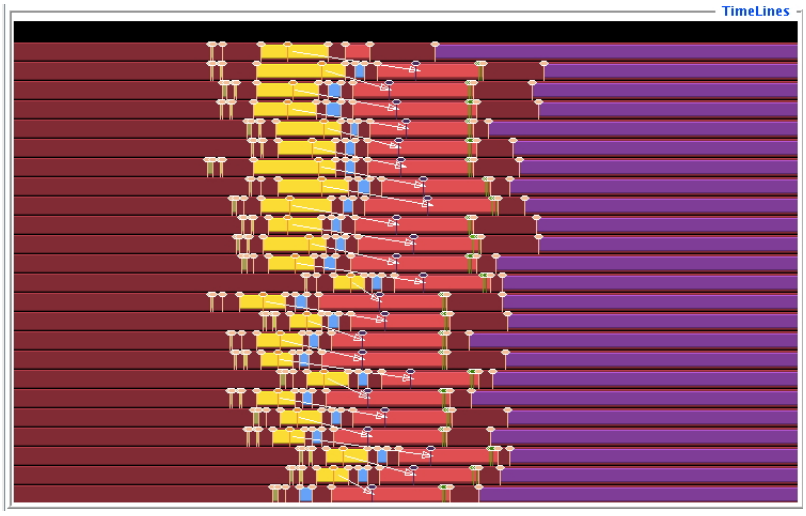


Figure 2.7: A trace of a nonblocking send between neighbouring processors

Exercise 2.5. Read section HPSC-**??** and give pseudo-code for the distributed sparse matrix-vector product using the above idiom for using `MPI_Test...` calls. Discuss the advan-

tages and disadvantages of this approach. The answer is not going to be black and white: discuss when you expect which approach to be preferable.

The `MPI_Wait...` routines are blocking. Thus, they are a good solution if the receiving process can not do anything until the data (or at least *some* data) is actually received. The `MPI_Test....` calls are themselves non-blocking: they test for whether one or more requests have been fullfilled, but otherwise immediately return. This can be used in the *master-worker model*: the master process creates tasks, and sends them to whichever worker process has finished its work, but while it waits for the workers it can itself do useful work. Pseudo-code:

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

2.3.2.1 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.

2.3.2.2 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`).

2.3.3 One-sided communication

The reference for the commands introduced here can be found in section 3.6.

Above, you saw point-to-point operations of the two-sided type: they require the co-operation of a sender and receiver. This co-operation could be loose: you can post a receive with `MPI_ANY_SOURCE` as sender, but there had to be both a send and receive call. In this section, you will see one-sided communication routines where a process can do a 'put' or 'get' operation, writing data to or reading it from another processor, without that other processor's involvement.

In one-sided MPI operations, also known as Remote Direct Memory Access (RDMA) or Remote Memory Access (RMA) operations, there are still two processes involved: the *origin*, which is the process that

originates the transfer, whether this is a 'put' or a 'get', and the *target* whose memory is being accessed. Unlike with two-sided operations, the target does not perform an action that is the counterpart of the action on the origin.

That does not mean that the origin can access arbitrary data on the target at arbitrary times. First of all, one-sided communication in MPI is limited to accessing only a specifically declared memory area on the target: the target declares an area of user-space memory that is accessible to other processes. This is known as a *window*. Windows limit how origin processes can access the target's memory: you can only 'get' data from a window or 'put' it into a window; all the other memory is not reachable from other processes.

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

Within one-sided communication, MPI has two modes: active RMA and passive RMA. In *active RMA*, or *active target synchronization*, the target sets boundaries on the time period (the 'epoch') during which its window can be accessed. The main advantage of this mode is that the origin program can perform many small transfers, which are aggregated behind the scenes. Active RMA acts much like asynchronous transfer with a concluding `Waitall`.

In *passive RMA*, or *passive target synchronization*, the target process puts no limitation on when its window can be accessed. (PGAS languages such as UPC are based on this model: data is simply read or written at will.) While intuitively it is attractive to be able to write to and read from a target at arbitrary time, there are problems. For instance, it requires a remote agent on the target, which may interfere with execution of the main thread, or conversely it may not be activated at the optimal time. Passive RMA is also very hard to debug and can lead to strange deadlocks.

2.3.3.1 Windows

The reference for the commands introduced here can be found in section 3.6.1.

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

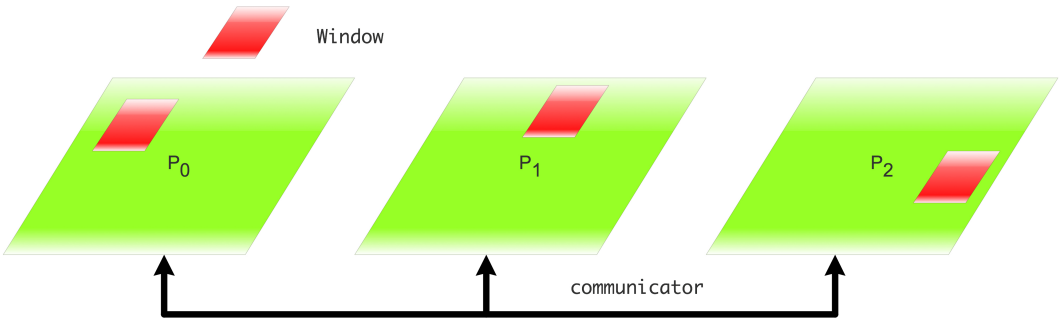


Figure 2.8: Collective definition of a window for one-sided data access

a memory area. Routine for creating and releasing windows are collective, so each process *has* to call them; see figure 2.8.

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

(For the `info` parameter you can often use `MPI_INFO_NULL`.) While the creation of a window is collective, each processor can specify its own window size, including zero, and even the type of the elements in it.

2.3.3.2 Active target synchronization: epochs

The reference for the commands introduced here can be found in section 3.6.3.

There are two mechanisms for *active target synchronization*, that is, one-sided communications where both sides are involved to the extent that they declare the communication epoch. In this section we look at the first mechanism, which is to use a *fence* operation:

```
MPI_Win_fence (int assert, MPI_Win win)
```

This operation is collective on the communicator of the window. It is comparable to `MPI_Wait` calls for non-blocking communication.

Unlike with wait calls, you always need two fences: one before and one after the 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.

Fences are, together with other window calls, collective operations. That means they imply some amount of synchronization between processes. Consider:

```
MPI_Win_fence( ... win ... ); // start an epoch
if (mytid==0) // do lots of work
else // do almost nothing
MPI_Win_fence( ... win ... ); // end the epoch
```

and assume that all processes execute the first fence more or less at the same time. The zero process does work before it can do the second fence call, but all other processes can call it immediately. However, they can not finish that second fence call until all one-sided communication is finished, which means they wait for the zero process.

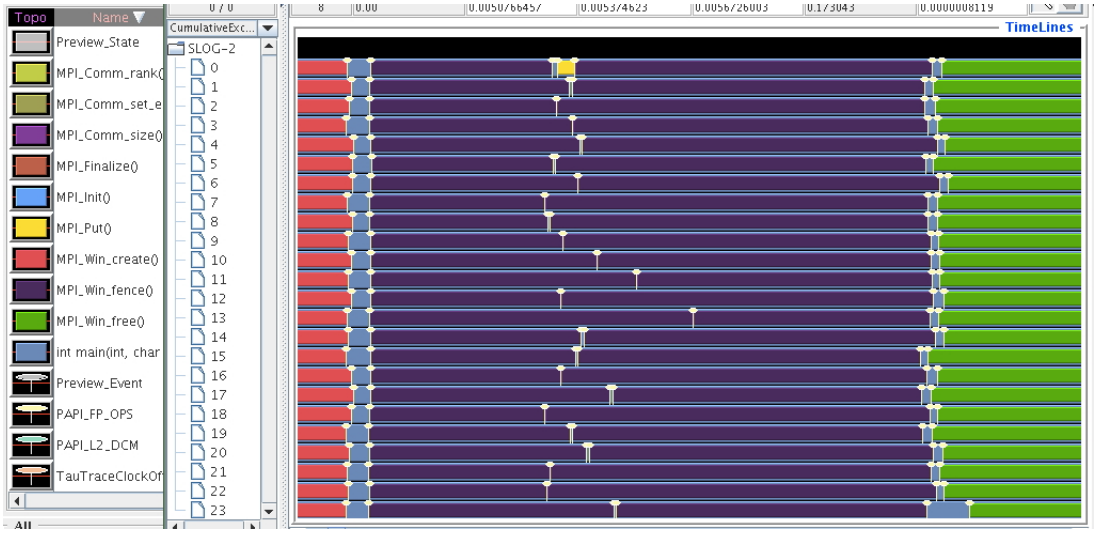


Figure 2.9: A trace of a one-sided communication epoch where process zero only originates a one-sided transfer

As a further restriction, you can not mix `Get` with `Put` or `Accumulate` calls in a single epoch. Hence, we can characterize an epoch as an *access epoch* on the origin, and as an *exposure epoch* on the target.

Assertions are an integer parameter: you can 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.

2.3.3.3 Put, get, accumulate

The reference for the commands introduced here can be found in section 3.6.2.

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)
```

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)
```

Exercise 2.6. 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 `MPI_Reduce`, the order in which the operands are accumulated is undefined. The same predefined operators are available, but no user-defined ones. There is one extra operator: `MPI_REPLACE`, this has the effect that only the last result to arrive is retained.

2.3.3.4 *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);
  }
```

2.3.3.5 *More active target synchronization*

The reference for the commands introduced here can be found in section 3.6.5.

There is a more fine-grained ways of doing *active target synchronization*. While fences corresponded to a global synchronization of one-sided calls, the `MPI_Win_start`, `MPI_Win_complete`, `MPI_`

`Win_post`, `Win_wait` routines are suitable, and possibly more efficient, if only a small number of processor pairs is involved. Which routines 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( /* group of target processes */ )
// access operations
MPI_Win_complete()
```

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

Both pairs of operations declare a *group of processors*; see section 2.6.3 for how to get such a group from a communicator. On an origin processor you would specify a group that includes the targets you will interact with, on a target processor you specify a group that includes the possible origins.

2.3.3.6 Passive target synchronization

The reference for the commands introduced here can be found in section 3.6.6.

In *passive target synchronization* only the origin is actively involved: the target makes no calls whatsoever. This means that the origin process remotely locks the window on the target.

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);
}
```

The two lock types are:

- `MPI_LOCK_SHARED` which should be used for `Get` calls: since multiple processors are allowed to read from a window in the same epoch, the lock can be shared.
- `MPI_LOCK_EXCLUSIVE` which should be used for `Put` and `Accumulate` calls: since only one processor is allowed to write to a window during one epoch, the lock should be exclusive.

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

2.3.3.7 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` if there is no special memory area; SGI is an example where such memory does exist.

2.3.3.8 Implementation

You may wonder how one-sided communication is realized[1]. Can a processor somehow get at another processor's data? Unfortunately, no.

Active target synchronization is implemented in terms of two-sided communication. Imagine that the first fence operation does nothing, unless it concludes prior one-sided operations. The Put and Get calls do nothing involving communication, except for marking with what processors they exchange data. The concluding fence is where everything happens: first a global operation determines which targets need to issue send or receive calls, then the actual sends and receive are executed.

Exercise 2.7. Assume that only Get operations are performed during an epoch. Sketch how these are translated to send/receive pairs. The problem here is how the senders find out that they need to send. Show that you can solve this with an `MPI_Scatter_reduce` call.

The previous paragraph noted that a collective operation was necessary to determine the two-sided traffic. Since collective operations induce some amount of synchronization, you may want to limit this.

Exercise 2.8. Argue that the mechanism with window post/wait/start/complete operations still needs a collective, but that this is less burdensome.

Passive target synchronization needs another mechanism entirely. Here the target process needs to have a background task (process, thread, daemon,...) running that listens for requests to lock the window. This can potentially be expensive.

2.4 Collectives

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.

The collective operations discussed in this section are:

- Broadcast, reduce, and scan: in these operations a single data item is sent from or collected on a 'root' process.

1. For more on this subject, see [4].

- Gather and scatter: here the root has an array from which to send, or in which to collect, the data from the other processors.
- All-to-all, which lets all processors communicate with all others.
- Barrier, which synchronizes all processes, and which separates events before it from those after it.

There are several variants of most collectives. For instance, the gather and reduce calls have a *root of the collective* where information is collected. There is a corresponding 'all' variant, where the result is not left just on the root but everywhere. There are also 'v' variants where the amount of data coming from or going to each processor is variable.

In addition to these collective operations, there are operations that are said to be 'collective on their communicator', but which do not involve data movement. Collective then means that all processors must call this routine; not to do so is an error that will probably manifest itself in 'hanging' code. One such example is `MPI_Win_fence`.

2.4.1 Rooted collectives: broadcast, reduce

The reference for the commands introduced here can be found in section 3.7.1.

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 2.10. Typically, it will be the root of a broadcast

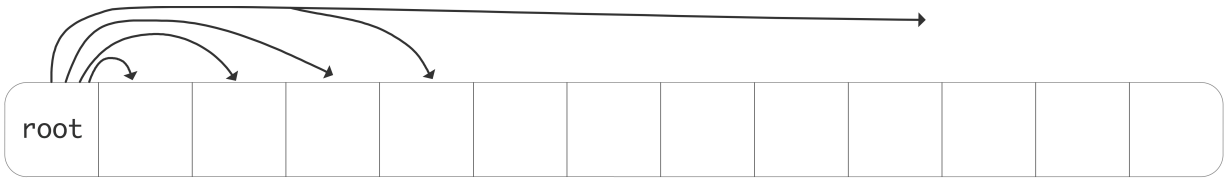


Figure 2.10: A simple broadcast

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 2.4.7.)

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 2.10 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 2.11. 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-**??**; see also [1].

The reverse of a broadcast is a reduction:

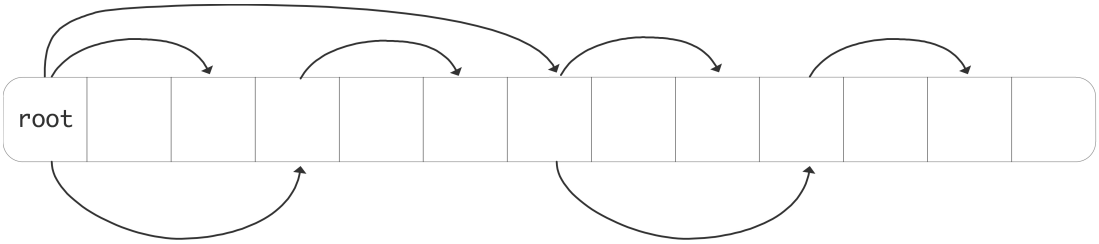


Figure 2.11: A tree-based broadcast

```
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; section 2.4.2.1.

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

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

If all processors need the result, you could then do a broadcast, but it is more efficient to use `MPI_Allreduce`; see section 2.4.6.

2.4.2 Scan operations

The reference for the commands introduced here can be found in section 3.7.6.

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.

Often, the more useful variant is the *exclusive scan* `MPI_Exscan`

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

with the same prototype.

Exercise 2.9. 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`. Are there operators where that is not the case?

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 2.10. 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

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

This operator is not communitative, and it needs to be declared as such with `MPI_Op_create`.

2.4.2.1 User-defined reductions

2.4.3 Gather and scatter

The reference for the commands introduced here can be found in section 3.7.2.

In the `MPI_Scatter` operation, the root spreads information to all other processes. The difference with a broadcast is that it involves individual information from/to every process. Thus, the gather operation typically has an array of items, one coming from each sending process, and scatter has an array, with an individual item for each receiving process; see figure 2.12.

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.

2.4.4 Variable-size-input collectives

The reference for the commands introduced here can be found in section 3.7.5.

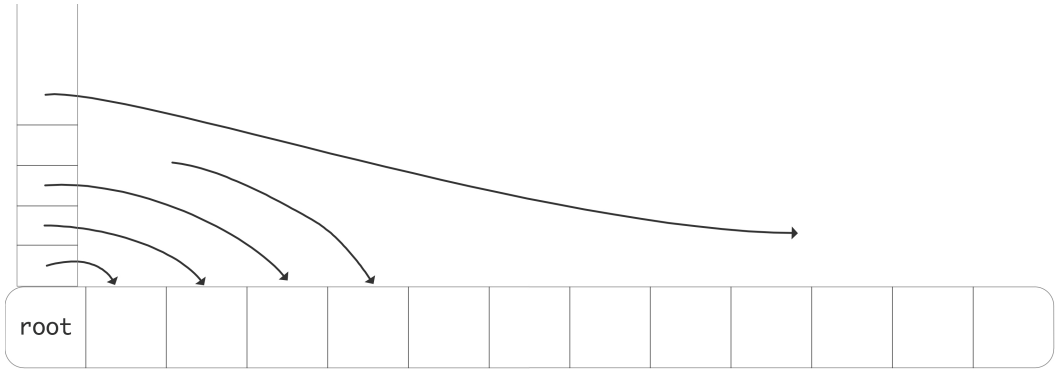


Figure 2.12: A scatter operation

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

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

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

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

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

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

and you have just seen how to construct that information.

2.4.5 Reduce-scatter

The reference for the commands introduced here can be found in section 3.7.3.

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

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

One important example of this command is the *sparse matrix-vector product*; see HPSC-**??** for background information. Each process contains one or more matrix rows, so by looking at indices the process can decide what other processes it needs data from. The problem is for a process to find out what other processes it needs to send data to.

Using `MPI_Reduce_scatter` the process goes as follows:

- Each process creates an array of ones and zeros, describing who it needs data from.
- The reduce part of the reduce-scatter yields an array of requester counts; after the scatter each process knows how many processes request data from it.
- Next, the sender processes need to find out what elements are requested from it. For this, each process sends out arrays of indices.
- The big trick is that each process now knows how many of these requests will be coming in, so it can post precisely that many `MPI_Irecv` calls, with a source of `MPI_ANY_SOURCE`.

2.4.6 'All'-type collectives

The reference for the commands introduced here can be found in section 3.7.4.

In many applications the result of a collective is needed on all processes. For instance, if x, y are distributed vector objects, and you want to compute

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

you need the inner product value on all processors. You could do this by writing a reduction followed by a broadcast, but more efficient algorithms exist. Surprisingly, an 'all-gather' operation takes as long as a rooted gather (see HPSC-**??** for details).

Thus, MPI has the following operations:

- `MPI_Allreduce` is equivalent to a `MPI_Reduce` followed by a broadcast.
- `MPI_Allgather` is equivalent to a `MPI_Gather` followed by a broadcast.
- `MPI_Allgatherv` is equivalent to an `MPI_Gatherv` followed by a broadcast.
- `MPI_Alltoall`, `MPI_Alltoallv`.

The 'v' variants are discussed in section 2.4.4.

2.4.7 Non-blocking collectives

MPI version 3 has non-blocking collectives.

2.4.8 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.

2.4.9 Collectives and synchronization

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

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

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

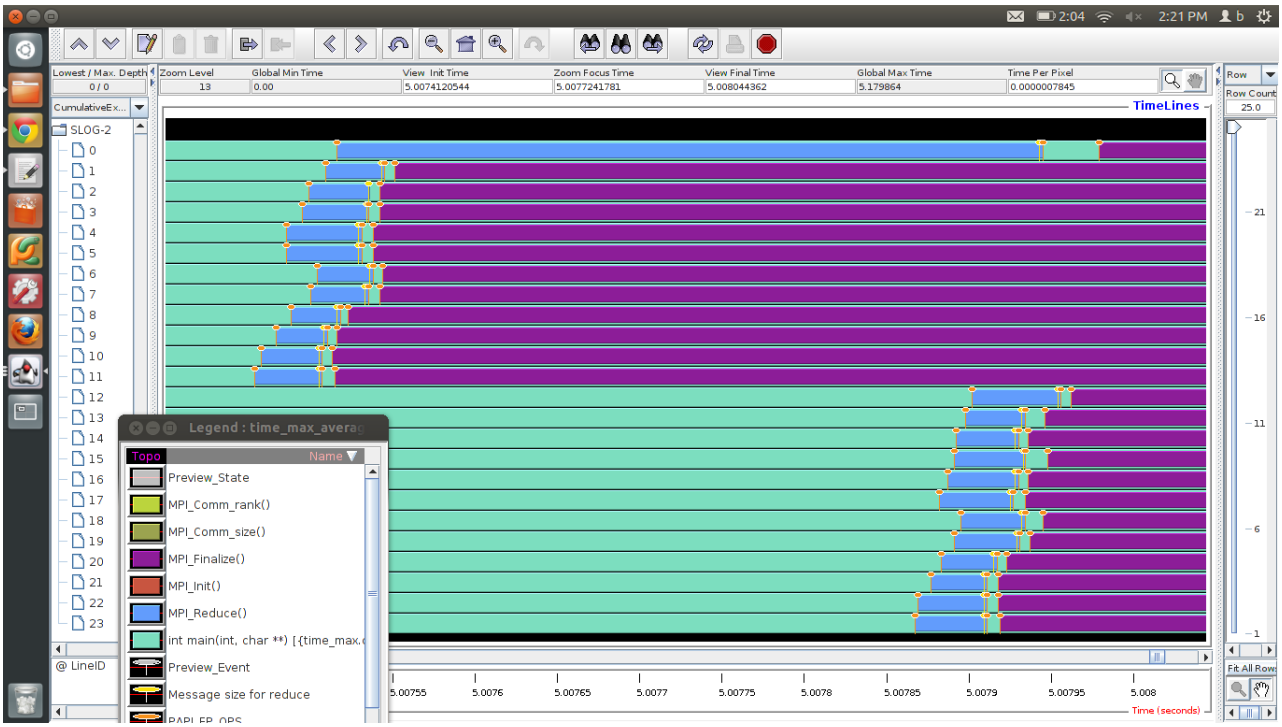


Figure 2.13: Trace of a reduction operation between two dual-socket 12-core nodes

operation the root may have to wait for other processors. This is illustrated in figure 2.13, which gives a TAU trace of a reduction operation on two nodes, with two six-core sockets (processors) each. We see that[2]:

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

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

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

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

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

Consider a specific scenario:

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

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

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

However, it is equally possible to have this execution:

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

2.5 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.

2.5.1 Elementary data types

The reference for the commands introduced here can be found in section 3.2.1.

MPI has a number of elementary data types, corresponding to the simple data types of programming languages. The names are made to resemble the types of C and Fortran, for instance `MPI_FLOAT` and `MPI_DOUBLE` versus `MPI_REAL` and `MPI_DOUBLE_PRECISION`.

2.5.2 Derived datatypes

The reference for the commands introduced here can be found in section 3.2.2.

MPI allows you to create your own data types, somewhat analogous to defining structures in a programming language. MPI data types are mostly of use if you want to send multiple items in one message.

There are two problems with using only elementary datatypes as you have seen so far.

- MPI communication routines can only send multiples of a single data type: it is not possible to send items of different types, even if they are contiguous in memory. It would be possible to use the `MPI_BYTE` data type, but this is not advisable.
- It is also ordinarily not possible to send items of one type if they are not contiguous in memory. You could of course send a contiguous memory area that contains the items you want to send, but that is wasteful of bandwidth.

With MPI data types you can solve these problems in several ways.

- You can create a new *contiguous data type* consisting of an array of elements of another data type. There is no essential difference between sending one element of such a type and multiple elements of the component type.
- You can create a *vector data type* consisting of regularly spaced blocks of elements of a component type. This is a first solution to the problem of sending non-contiguous data.
- For not regularly spaced data, there is the *indexed data type*, where you specify an array of index locations for blocks of elements of a component type. The blocks can each be of a different size.
- The *struct data type* can accomodate multiple data types.

And you can combine these mechanisms to get irregularly spaced heterogeneous data, et cetera.

2.5.2.1 Datatype signatures

With the primitive types you have seen so far, it pretty much went without saying that if the sender sends an array of doubles, the receiver had to declare the datatype also as doubles. With derived types that is no longer the case: the sender and receiver can declare a different datatype for the send and receive buffer, as long as these have the same *datatype signature*.

The signature of a datatype is the internal representation of that datatype. For instance, if the sender declares a datatype consisting of two doubles, and it sends four elements of that type, the receiver can receive it as two elements of a type consisting of four doubles.

You can also look at the signature as the form ‘under the hood’ in which MPI sends the data.

2.5.2.2 Basic calls

The reference for the commands introduced here can be found in section 3.2.2.1.

New MPI data types are created by

- `MPI_Type_contiguous`
- `MPI_Type_vector`
- `MPI_Type_struct`

- `MPI_Type_indexed`
- `MPI_Type_hindexed`

It is necessary to call `MPI_Type_commit` which makes MPI do the indexing calculations for the data type. When you no longer need the data type, you call `MPI_Type_free`.

2.5.2.3 Contiguous type

The reference for the commands introduced here can be found in section 3.2.2.2.

The simplest derived type is the 'contiguous' type, constructed with `MPI_Type_contiguous` A contigous type describes an array of items of an elementary or earlier defined type. There is no difference between sending one item of a contiguous type and multiple items of the constituent type. This is illustrated in figure 2.14.

Figure 2.14: A contiguous datatype is built up out of elements of a constituent type

2.5.2.4 Vector type

The reference for the commands introduced here can be found in section 3.2.2.3.

The simplest non-contiguous datatype is the 'vector' type, constructed with `MPI_Type_vector`. A vector type describes a series of blocks, all of equal size, spaced with a constant stride. This is illustrated in figure 2.15.

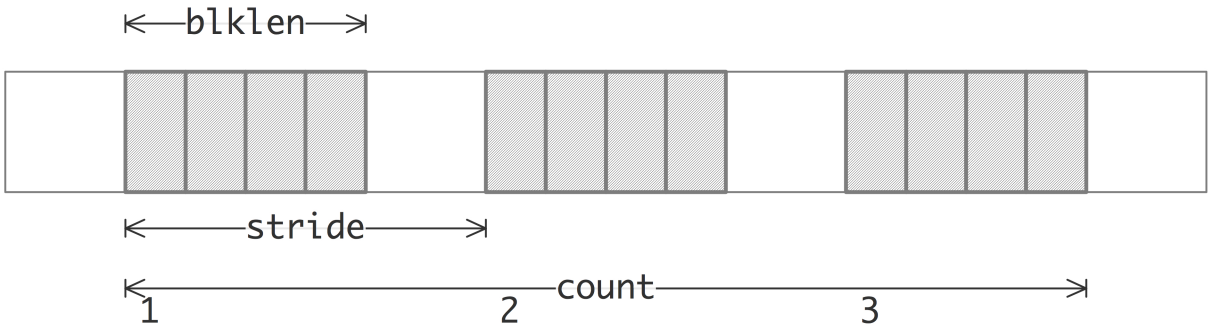


Figure 2.15: A vector datatype is built up out of strided blocks of elements of a constituent type

As an example of this datatype, consider the example of transposing a matrix, for instance to convert between C and Fortran arrays (see section HPSC-**??**). Suppose that a processor has a matrix stored in C, row-major, layout, and it needs to send a column to another processor. If the matrix is declared as

```
int M,N; double mat[M][N]
```

then a column has M blocks of one element, spaced N locations apart. In other words:

```
MPI_Datatype MPI_column;
MPI_Type_vector(
    /* count= */ M, /* blocklength= */ 1, /* stride= */ N,
    MPI_DOUBLE, &MPI_column );
```

Sending the first column is easy:

```
MPI_Send( mat, 1,MPI_column, ... );
```

The second column is just a little trickier: you now need to pick out elements with the same stride, but starting at `A[0][1]`.

```
MPI_Send( &(mat[0][1]), 1,MPI_column, ... );
```

You can make this marginally more efficient (and harder to read) by replacing the index expression by `mat+1`.

Exercise 2.11. Suppose you have a matrix of size $4N \times 4N$, and you want to send the elements `A[4*i][4*j]` with $i, j = 0, \ldots, N-1$. How would you send these elements with a single transfer?

2.5.2.5 Indexed type

The reference for the commands introduced here can be found in section 3.2.2.4.

The indexed datatype, constructed with `MPI_Type_indexed` can send arbitrarily located elements from an array of a single datatype. You need to supply an array of index locations, plus an array of blocklengths with a separate blocklength for each index. The total number of elements sent is the sum of the blocklengths.

2.5.2.6 Struct type

The reference for the commands introduced here can be found in section 3.2.2.5.

The structure type, created with `MPI_Type_create_struct`, can contain multiple data types. The specification contains a 'count' parameter that specifies how many blocks there are in a single structure. For instance,

```
struct {
 int i;
 float x,y;
} point;
```

has two blocks, one of a single integer, and one of two floats. This is illustrated in figure 2.16.

The structure type is very similar in functionality to `MPI_Type_hindexed`, which uses byte-based indexing. The structure-based type is probably cleaner in use.

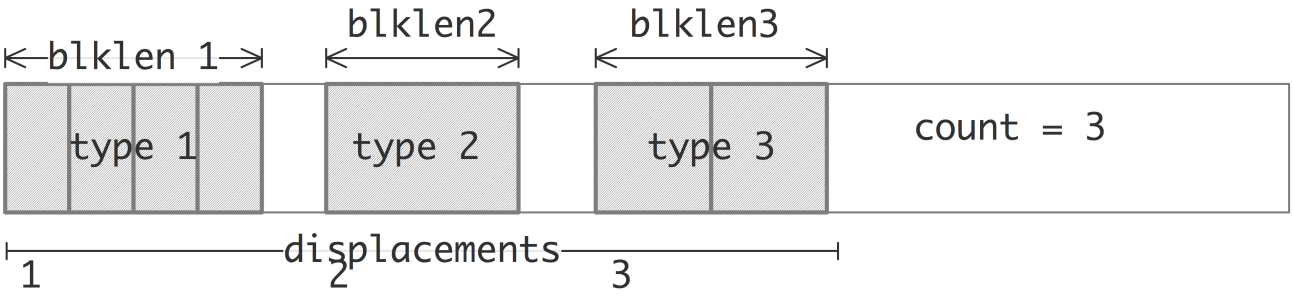


Figure 2.16: The elements of an MPI Struct datatype

2.5.3 Packing

The reference for the commands introduced here can be found in section 3.2.3.

One of the reasons for derived datatypes is dealing with non-contiguous data. In older communication libraries this could only be done by *packing* data from its original containers into a buffer, and likewise unpacking it at the receiver into its destination data structures.

MPI offers this packing facility, partly for compatibility with such libraries, but also for reasons of flexibility. Unlike with derived datatypes, which transfers data atomically, packing routines add data sequentially to the buffer and unpacking takes them sequentially.

This means that one could pack an integer describing how many floating point numbers are in the rest of the packed message. Correspondingly, the unpack routine could then investigate the first integer and based on it unpack the right number of floating point numbers.

MPI offers the following:

- The `MPI_Pack` command adds data to a send buffer;
- the `MPI_Unpack` command retrieves data from a receive buffer;
- the buffer is sent with a datatype of `MPI_PACKED`.

2.6 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.

2.6.1 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.

2.6.2 Creating new communicators

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

2.6.2.1 Duplicating communicators

The reference for the commands introduced here can be found in section 3.9.1.

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.

2.6.2.2 Splitting a communicator

The reference for the commands introduced here can be found in section 3.9.2.

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*.

2.6.2.3 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.

2.6.3 Intra-communicators

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

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)
```

2.6.4 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, t
```

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 2.8.4).

2.7 Hybrid programming: MPI and threads

The reference for the commands introduced here can be found in section 3.12.

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.

2.8 Leftover topics

2.8.1 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.

2.8.1.1 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. Therefore, there is a special value `MPI_STATUS_IGNORE` that you can supply instead of a status object, which tells MPI that the status does not have to be reported. For routines such as `MPI_Waitany` where an array of statuses is needed, you can supply `MPI_STATUSES_IGNORE`.

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.

2.8.1.2 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.

2.8.2 Semantics

MPI processes are only synchronized to a certain extent, so you may wonder what guarantees there are that running a code twice will give the same result. You need to consider two cases: first of all, if the two runs are on different numbers of processors there are already numerical problems; see HPSC-**??**.

Let us then limit ourselves to two runs on the same set of processors. In that case, MPI is deterministic as long as you do not use wildcards such as `MPI_ANY_SOURCE`. Formally, MPI messages are 'non-overtaking': two messages between the same sender-receiver pair will arrive in sequence.

2.8.3 Error handling

The reference for the commands introduced here can be found in section 3.10.

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 [3]:

3. The routine `MPI_Errhandler_set` is deprecated.

```
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();
}
```

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/O*MPI!I/O: if an output file has the wrong permissions, code can possibly progress without writing data, or writing to a temporary file.

2.8.4 Fault tolerance

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

First of all, one of the possible MPI error return codes (section **??**) 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.

2.8.5 Timing

The reference for the commands introduced here can be found in section 3.11.1.

Timing of parallel programs is tricky. On each node you can use a timer, typically based on some **OS!** (**OS!**) call. MPI supplies its own routine `MPI_Wtime` which gives *wall clock time*. Normally you don't worry about the starting point for this timer: you call it before and after an event and subtract the values.

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

If you execute this on a single processor you get fairly reliable timings, except that you would need to subtract the overhead for the timer. This is the usual way to measure timer overhead:

```
t = MPI_Wtime();
// absolutely nothing here
t = MPI_Wtime()-t;
```

However, if you try to time a parallel application you will most likely get different times for each process, so you would have to take the average or maximum. Another solution is to synchronize the processors by using a *barrier* :

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

Exercise 2.12. This scheme also has some overhead associated with it. How would you measure that?

Now suppose you want to measure the time for a single send. It is not possible to start a clock on the sender and do the second measurement on the receiver, because the two clocks need not be synchronized. Usually a *ping-pong* is done:

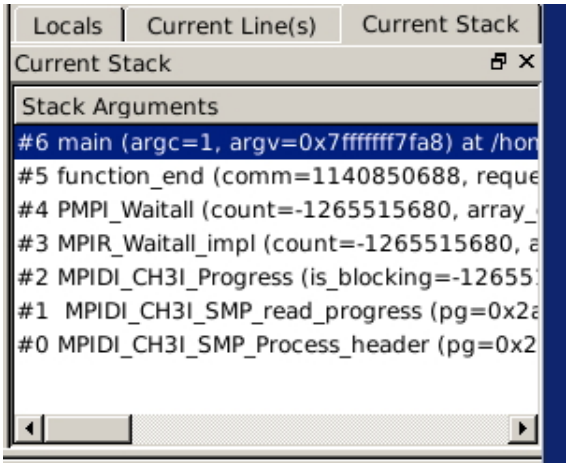


Figure 2.17: A stack trace, showing the `PMPI` calls.

```
if ( proc_source ) {
  MPI_Send( /* to target */ );
  MPI_Recv( /* from target */ );
else if ( proc_target ) {
  MPI_Recv( /* from source */ );
  MPI_Send( /* to source */ );
}
```

Exercise 2.13. Why is it generally not a good idea to use processes 0 and 1 for the source and target processor? Can you come up with a better guess?

Exercise 2.14. Take the pingpong program of section 3.11.1 and modify it to use longer messages. How does the timing increase with message size?

Exercise 2.15. Take the pingpong program of section 3.11.1 and modify it to let half the processors be source and the other half the targets. Does the pingpong time increase?

No matter what sort of timing you are doing, it is good to know the accuracy of your timer. The routine `MPI_Wtick` gives the smallest possible timer increment. If you find that your timing result is too close to this 'tick', you need to find a better timer (for CPU measurements there are cycle-accurate timers), or you need to increase your running time, for instance by increasing the amount of data.

2.8.6 Profiling

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

MPI allows you to write your own profiling interface. To make this possible, every routine `MPI_Something` calls a routine `PMPI_Something` that does the actual work. You can now write your `MPI_...` routine which calls `PMPI_...`, and inserting your own profiling calls. As you can see in figure 2.17, normally only the `PMPI` routines show up in the stack trace.

Does the standard mandate this?

2.8.7 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.

2.8.7.1 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 7.2 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.

2.8.7.2 Large scale debugging

Check out *ddt* or *TotalView*.

2.8.7.3 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.

2.8.8 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.

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

2.8.9 The origin of one-sided communication in ShMem

The *Cray T3E* had a library called *shmem* which offered a type of shared memory. Rather than having a true global address space it worked by supporting variables that were guaranteed to be identical between processors, and indeed, were guaranteed to occupy the same location in memory. Variables could be declared to be shared a 'symmetric' pragma or directive; their values could be retrieved or set by `shmem_get` and `shmem_put` calls.

2.9 Review questions

If you answer that a statement is false, give a one-line explanation.

1. True or false: `mpicc` is a compiler.
2. True or false: `mpirun` can only be used for interactive parallel runs.
3. What is the function of a hostfile?
4. True or false: in each communicator, processes are numbered consecutively from zero.
5. If a processors issues a send to another processor, and that other processors issues a receive from the first *at exactly the same time*, the send and receive call return immediately
6. Describe a deadlock scenario involving three processors.
7. True or false: a message sent with `MPI_Isend` from one processor can be received with an `MPI_Recv` call on another processor.
8. True or false: a message sent with `MPI_Send` from one processor can be received with an `MPI_Irecv` on another processor.
9. Why does the `MPI_Irecv` call not have an `MPI_Status` argument?
10. Give a simple model for the time a send operation takes.
11. Give a simple model for the time a broadcast of a single scalar takes.
12. What is the relation between the concepts of 'origin', 'target' and 'window' in one-sided communication.
13. What are the three routines for one-sided data transfer?
14. Give an example of a collective call with and without a root processor.
15. Give two examples of derived datatypes.
16. Give an example where the sender uses a different type to send than the receiver uses in the corresponding receive call. Name the types involved.

2.10 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 3

MPI Reference

This section gives reference information and illustrative examples of the use of MPI. While the code snippets given here should be enough, full programs can be found in the repository for this book https://bitbucket.org/VictorEijkhout/parallel-computing-book.

3.1 Basics

3.1.1 MPI setup

This reference section gives the syntax for routines introduced in section 2.2.1.

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

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

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

```
use mpi
```

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

Every MPI program has to start with

```
MPI_Init(&argc,&argv);
```

where `argc` and `argv` are the arguments of a C language main program:

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

The regular way to conclude an MPI program is through

```
MPI_Finalize();
```

but an abnormal end to a run can be forced by

```
MPI_Abort(comm,value);
```

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

The commandline arguments can only be guaranteed to be passed correctly to process zero. Here is a fragment of code that shows use of commandline arguments. The program `examples/mpi/c/init.c` takes a single integer commandline argument. If the user forgets to specify an argument of specifies `-h`, a usage message is printed and the program aborts, otherwise the parameter is broadcast to all processes.

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

3.1.2 MPI ranks and communicator sizes

This reference section gives the syntax for routines introduced in section 2.2.3.

Every MPI process has its own local storage. So if you pretend that all these small arrays are really together one big array, you need to know where each piece fits in the whole. For this you need to know at the very least how many processes there are and what the rank of a process is.

The following example creates a distributed array that will contains the values of a function at equidistant points in the interval $[0,1]$. The code implements this as follows:

1. each process has an array of 10 points,
2. so the distributed array has a length of 10 times the number of processes;
3. of this array, each process has 10 points, starting at 10 times the process id.
4. To fill in values in the local array, the process iterates only over its local points,
5. but it needs to calculate the global coordinate of those points.

```
// local.cxx
int nlocalpoints = 10,
```

```
  ntotal_points = ntids*nlocalpoints,
  my_global_start = mytid*nlocalpoints;
double stepsize = 1./(ntotal_points-1);
array = new double[nlocalpoints];
for (int i=0; i<nlocalpoints; i++)
  array[i] = f( (i+my_global_start)*stepsize );
```

3.1.3 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.

3.2 Data types

3.2.1 Elementary types

This reference section gives the syntax for routines introduced in section 2.5.1.

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))

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

3.2.2 Derived datatypes

This reference section gives the syntax for routines introduced in section 2.5.2.

The space taken by a derived type is not immediately obvious from its definition since padding maybe applied. The actual size can be retrieved with `MPI_Type_extent`:

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```

See the example in section 3.2.2.5

3.2.2.1 Type create and release calls

This reference section gives the syntax for routines introduced in section 2.5.2.2.

A derived type needs to be committed with `MPI_Type_commit`:

```
int MPI_Type_commit (MPI_datatype *datatype)
```

The commit call is typically used to find an efficient 'flat' representation of recursively defined datatypes.

When you no longer need the derived type, its space can be released with `MPI_Type_free`:

```
int MPI_Type_free (MPI_datatype *datatype)
```

After the type free call

- The definition of the datatype identifier will be changed to `MPI_DATATYPE_NULL`.
- Any communication using this data type, that was already started, will be completed succesfully.
- Datatypes that are defined in terms of this data type will still be usable.

3.2.2.2 Contiguous type

This reference section gives the syntax for routines introduced in section 2.5.2.3.

A contiguous datatype, created with a call to `MPI_Type_contiguous`,

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

consists of a number of elements of a datatype, contiguous in memory. Sending one element of a contiguous type is fully equivalent to sending a number of elements of the constituent type.

```
MPI_Datatype newvectortype;
if (mytid==sender) {
  MPI_Type_contiguous(count,MPI_DOUBLE,&newvectortype);
  MPI_Type_commit(&newvectortype);
```

```
  MPI_Send(source,1,newvectortype,the_other,0,comm);
  MPI_Type_free(&newvectortype);
} else if (mytid==receiver) {
  MPI_Status recv_status;
  int recv_count;
  MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
    &recv_status);
  MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
  ASSERT(count==recv_count);
}
```

3.2.2.3 Vector type

This reference section gives the syntax for routines introduced in section 2.5.2.4.

The `MPI_Type_vector` type can be used to create a type of regularly spaced blocks of data. All block lengths need to be the same, and the vector type is built out of a single constituent type.

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

In this example a vector type is created only on the sender, in order to send a strided subset of an array; the receiver receives the data as a contiguous block.

```
// vector.c
source = (double*) malloc(stride*count*sizeof(double));
target = (double*) malloc(count*sizeof(double));
MPI_Datatype newvectortype;
if (mytid==sender) {
  MPI_Type_vector(count,1,stride,MPI_DOUBLE,&newvectortype);
  MPI_Type_commit(&newvectortype);
  MPI_Send(source,1,newvectortype,the_other,0,comm);
  MPI_Type_free(&newvectortype);
} else if (mytid==receiver) {
  MPI_Status recv_status;
  int recv_count;
  MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
    &recv_status);
  MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
  ASSERT(recv_count==count);
}
```

3.2.2.4 Indexed data

This reference section gives the syntax for routines introduced in section 2.5.2.5.

The indexed datatype is similar to the vector type, in the sense that it consists of a series of blocks of items, all of the same type. However, where the vector type was described by a single stride and blocklength, with `MPI_Type_indexed` you can specify the location and length of each block.

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

The following example picks items that are on prime number-indexed locations.

```
// indexed.c
indices = (int*) malloc(count*sizeof(double));
blocklengths = (int*) malloc(count*sizeof(double));
source = (int*) malloc(totalcount*sizeof(double));
target = (int*) malloc(count*sizeof(double));
MPI_Datatype newvectortype;
if (mytid==sender) {
  MPI_Type_indexed(count,blocklengths,indices,MPI_INT,&newvectortype);
  MPI_Type_commit(&newvectortype);
  MPI_Send(source,1,newvectortype,the_other,0,comm);
  MPI_Type_free(&newvectortype);
} else if (mytid==receiver) {
  MPI_Status recv_status;
  int recv_count;
  MPI_Recv(target,count,MPI_INT,the_other,0,comm,
    &recv_status);
  MPI_Get_count(&recv_status,MPI_INT,&recv_count);
  ASSERT(recv_count==count);
}
```

You can also `MPI_Type_create_hindexed` which describes blocks of a single old type, but with indix locations in bytes, rather than in multiples of the old type.

```
int MPI_Type_create_hindexed
 (int count, int blocklens[], MPI_Aint indices[],
  MPI_Datatype old_type,MPI_Datatype *newtype)
```

You can use this to pick all occurrences of a single component out of an array of structures. However, you need to be very careful with the index calculation. Use pointer arithmetic, as in the example in section 3.2.2.5. Another use of this function is in sending an `stl<vector>`, that is, a vector object from the *C++ standard library*, if the component type is a pointer. No further explanation here.

3.2.2.5 Structure data

This reference section gives the syntax for routines introduced in section 2.5.2.6.

The `MPI_Type_create_struct` routine creates a type consisting of blocks of multiple datatypes, much like `MPI_Type_indexed` makes an array of blocks of a single type.

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

`count` The number of blocks in this datatype. The `blocklengths`, `displacements`, `types` arguments have to be at least of this length.

`blocklengths` array containing the lengths of the blocks of each datatype.

`displacements` array describing the relative location of the blocks of each datatype.

`types` array containing the datatypes; each block in the new type is of a single datatype; there can be multiple blocks consisting of the same type.

In this example, unlike the previous ones, both sender and receiver create the structure type. With structures it is no longer possible to send as a derived type and receive as a array of a simple type. (It would be possible to send as one structure type and receive as another, as long as they have the same *datatype signature*.)

```
// struct.c
struct object {
  char c;
  double x[2];
  int i;
};
MPI_Datatype newstructuretype;
int structlen = 3;
int blocklengths[structlen]; MPI_Datatype types[structlen];
MPI_Aint displacements[structlen];
// where are the components relative to the structure?
blocklengths[0] = 1; types[0] = MPI_CHAR;
displacements[0] = (size_t)&(myobject.c) - (size_t)&myobject;
blocklengths[1] = 2; types[1] = MPI_DOUBLE;
displacements[1] = (size_t)&(myobject.x[0]) - (size_t)&myobject;
blocklengths[2] = 1; types[2] = MPI_INT;
displacements[2] = (size_t)&(myobject.i) - (size_t)&myobject;
MPI_Type_create_struct(structlen,blocklengths,displacements,types,&newstructu
MPI_Type_commit(&newstructuretype);
{
  MPI_Aint typesize;
  MPI_Type_extent(newstructuretype,&typesize);
  if (mytid==0) printf("Type extent: %d bytes\n",typesize);
}
if (mytid==sender) {
```

```
  MPI_Send(&myobject,1,newstructuretype,the_other,0,comm);
} else if (mytid==receiver) {
  MPI_Recv(&myobject,1,newstructuretype,the_other,0,comm,MPI_STATUS_IGNORE);
}
MPI_Type_free(&newstructuretype);
```

Note the `displacement` calculations in this example, which involve some not so elegant pointer arithmetic. It would have been incorrect to write

```
displacement[0] = 0;
displacement[1] = displacement[0] + sizeof(char);
```

since you do not know the way the *compiler* lays out the structure in memory[1]. The space that MPI takes for a structure type can be queried with `MPI_Type_extent`.

(There is a deprecated function `MPI_Type_struct` with the same functionality.)

3.2.3 Packed data

This reference section gives the syntax for routines introduced in section 2.5.3.

With `MPI_PACK` data elements can be added to a buffer one at a time. The `position` parameter is updated each time by the packing routine.

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

Conversely, `MPI_UNPACK` retrieves one element from the buffer at a time. You need to specify the MPI datatype.

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

A packed buffer is sent of received with a datatype of `MPI_PACKED`. The sending routine uses the `position` parameter to specify how much data is sent, but the receiving routine does not know this value a priori, so has to specify an upper bound.

```
// pack.c
if (mytid==sender) {
  MPI_Pack(&nsends,1,MPI_INT,buffer,buflen,&position,comm);
  for (i=0; i<nsends; i++) {
```

1. Homework question: what does the language standard say about this?

```
      double value = rand()/(double)RAND_MAX;
      MPI_Pack(&value,1,MPI_DOUBLE,buffer,buflen,&position,comm);
    }
    MPI_Pack(&nsends,1,MPI_INT,buffer,buflen,&position,comm);
    MPI_Send(buffer,position,MPI_PACKED,other,0,comm);
  } else if (mytid==receiver) {
    int irecv_value;
    double xrecv_value;
    MPI_Recv(buffer,buflen,MPI_PACKED,other,0,comm,MPI_STATUS_IGNORE);
    MPI_Unpack(buffer,buflen,&position,&nsends,1,MPI_INT,comm);
    for (i=0; i<nsends; i++) {
      MPI_Unpack(buffer,buflen,&position,&xrecv_value,1,MPI_DOUBLE,comm);
    }
    MPI_Unpack(buffer,buflen,&position,&irecv_value,1,MPI_INT,comm);
    ASSERT(irecv_value==nsends);
  }
```

3.3 Blocking communication

This reference section gives the syntax for routines introduced in section 2.3.1.

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

The basic blocking receive command is

```
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`; see section 2.3.1.3.

The following code is guaranteed to block, since a `MPI_Recv` always blocks:

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

On the other hand, if we put the send call before the receive, code may not block for small messages that fall under the *eager limit*. In this example we send gradually larger messages. From the screen output you can see what the largest message was that fell under the eager limit; after that the code hangs because of a deadlock.

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

If you want a code to behave the same for all message sizes, you force the send call to be blocking by using `MPI_Ssend`:

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

3.3.1 Receive status

This reference section gives the syntax for routines introduced in section 2.3.1.3.

In section 2.3.1.3 we mentioned the master-worker model as one opportunity for inspecting the `MPI_SOURCE` field of the `MPI_Status` object. Here is a small example: the tasks perform a variable amount of work (modeled here by a random wait) before sending a message to the master. The master waits for any source, and inspects the status field to report where the message comes from.

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

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

  for (int p=0; p<ntids-1; p++) {
    ierr = MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm,
      &status); CHK(ierr);
    int sender = status.MPI_SOURCE;
    printf("Message from sender=%d: %d\n",
     sender,recv_buffer[p]);
  }
} else {
  float randomfraction = (rand() / (double)RAND_MAX);
  int randomwait = (int) ( ntids * randomfraction );
  printf("process %d waits for %e/%d=%d\n",
   mytid,randomfraction,ntids,randomwait);
  sleep(randomwait);
  ierr = MPI_Send(&randomwait,1,MPI_INT, ntids-1,0,comm); CHK(ierr);
}
```

3.4 Deadlock-free blocking messages

This reference section gives the syntax for routines introduced in section 2.3.1.1.

If messsages are send roughly in pairs, the `MPI_Sendrecv` call is an easy way to prevent deadlock. Here you specify both the target of a send and the source of a receive, which can be same in case of a pairwise exchange of data, but they need not be the same.

```
int MPI_Sendrecv(
  void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
  void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag,
```

```
  MPI_Comm comm, MPI_Status *status)
```

As an example we set up a ring of three processors: each process sends to its right neighbour, and receives from its left neighbour.

```
// sendrecv.c
right = (mytid+1)%3; left = (mytid+2)%3;
MPI_Sendrecv( &my_data,1,MPI_INTEGER, right,0,
&other_data,1,MPI_INTEGER, left,0,
comm,MPI_STATUS_IGNORE);
```

3.5 Non-blocking communication

This reference section gives the syntax for routines introduced in section 2.3.2.

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

Here is a simple code that does a non-blocking exchange between two processors:

```
// irecvnonblock.c
MPI_Request request[2]
MPI_Status status[2];
other = 1-mytid;
MPI_Irecv(&recvbuf,1,MPI_INT,other,0,comm,&(request[0]));
```

```
MPI_Isend(&sendbuf,1,MPI_INT,other,0,comm,&(request[1]));
MPI_Waitall(2,request,status);
```

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`.

The above fragment is unrealistically simple. In a more general scenario we have to manage send and receive buffers: we need as many buffers as there are simultaneous non-blocking sends and receives.

```
// irecvloop.c
MPI_Request requests =
  (MPI_Request*) malloc( 2*ntids*sizeof(MPI_Request) );
recv_buffers = (int*) malloc( ntids*sizeof(int) );
send_buffers = (int*) malloc( ntids*sizeof(int) );
for (int p=0; p<ntids; p++) {
  int left_p = (p-1) % ntids,
    right_p = (p+1) % ntids;
  send_buffer[p] = ntids-p;
  MPI_Isend(sendbuffer+p,1,MPI_INT, right_p,0, requests+2*p);
  MPI_Irecv(recvbuffer+p,1,MPI_INT, left_p,0, requests+2*p+1);
}
MPI_Waitall(2*ntids,requests,MPI_STATUSES_IGNORE);
```

Instead of waiting for all messages, we can wait for any message to come with `MPI_Waitany`, and process the receive data as it comes in.

```
// irecv_source.c
if (mytid==ntids-1) {
  int *recv_buffer;
  MPI_Request *request;
  recv_buffer = (int*) malloc((ntids-1)*sizeof(int));
  request = (MPI_Request*) malloc((ntids-1)*sizeof(MPI_Request));

  for (int p=0; p<ntids-1; p++) {
    ierr = MPI_Irecv(recv_buffer+p,1,MPI_INT, p,0,comm,
      request+p); CHK(ierr);
  }
  for (int p=0; p<ntids-1; p++) {
    int index,sender;
    MPI_Waitany(ntids-1,request,&index,MPI_STATUS_IGNORE);
    printf("Message from %d: %d\n",index,recv_buffer[index]);
  }
```

Note the `MPI_STATUS_IGNORE` parameter: we know everything about the incoming message, so we do not need to query a status object. Contrast this with the example in section 3.3.1.

3.6 One-sided communication

This reference section gives the syntax for routines introduced in section 2.3.3.

3.6.1 Windows and epochs

This reference section gives the syntax for routines introduced in section 2.3.3.1.

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

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

The `MPI_Info` parameter can be used to pass implementation-dependent information:

```
MPI_Info info;
MPI_Info_create(&info);
MPI_Info_set(info,"no_locks","true");
MPI_Win_create( ... info ... &win);
MPI_Info_free(&info);
```

It is always valid to use `MPI_INFO_NULL`.

3.6.2 Remote memory access

This reference section gives the syntax for routines introduced in section 2.3.3.3.

```
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)
```

The `MPI_Get` call is very similar;

```
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
            origin_datatype, int target_rank, MPI_Aint target_disp,
            int target_count, MPI_Datatype target_datatype, MPI_Win
            win)
```

Here is a single put operation. Note that the window create and window fence calls are collective, so they have to be performed on all processors of the communicator that was used in the create call.

```
// putfence.c
MPI_Win the_window;
MPI_Win_create(&other_number,1*sizeof(int),sizeof(int),
               MPI_INFO_NULL,comm,&the_window);
MPI_Win_fence(0,the_window);
```

```
if (mytid==0) {
  MPI_Put( /* data on origin: */   &my_number, 1,MPI_INT,
       /* data on target: */   other,0,    1,MPI_INT,
       the_window);
}
MPI_Win_fence(0,the_window);
MPI_Win_free(&the_window);
```

Very similar, a get operation.

```
// getfence.c
MPI_Win_create(&other_number,2*sizeof(int),sizeof(int),
               MPI_INFO_NULL,comm,&the_window);
MPI_Win_fence(0,the_window);
if (mytid==0) {
  MPI_Get( /* data on origin: */   &my_number, 1,MPI_INT,
       /* data on target: */   other,1,    1,MPI_INT,
       the_window);
}
MPI_Win_fence(0,the_window);
```

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)
```

3.6.3 Active target synchronization

This reference section gives the syntax for routines introduced in section 2.3.3.2.

```
MPI_Win_fence (int assert, MPI_Win win)
```

3.6.4 Assertions

The `MPI_Win_fence` call, as well `MPI_Win_start` and such, take an argument through which assertions can be passed about the activity before, after, and during the epoch. The value zero is always allowed, by you can make your program more efficient by specifying one or more of the following:

- `MPI_MODE_NOCHECK`: this is used with `MPI_Win_start` and `MPI_Win_post`; it indicates that when the origin 'start' call is made, the target 'post' call has already been issued. This is comparable to using `MPI_Rsend`.

- `MPI_MODE_NOSTORE`: this is used to specify that the local window was not updated in the preceding epoch.
- `MPI_MODE_NOPUT`: this is used to specify that a local window will not be used as target in this epoch.
- `MPI_MODE_NOPRECEDE`: this states that the `MPI_Win_fence` call does not conclude a sequence of RMA operations. If this assertion is made on any process in a window group, it must be specified by all.
- `MPI_MODE_NOSUCCEED`: this states that the `MPI_Win_fence` call is not the start of a sequence of local RMA calls. If any process in a window group specifies this, all process must do so.

3.6.5 More active target synchronization

This reference section gives the syntax for routines introduced in section 2.3.3.5.

The 'fence' mechanism (section 3.6.3) uses a global synchronization on the communicator of the window, which may lead to performance inefficiencies if processors are not in step which each other. There is a mechanism that is more fine-grained, by using synchronization only on a processor *group*. This takes four different calls, two for starting and two for ending the epoch, separately for target and origin.

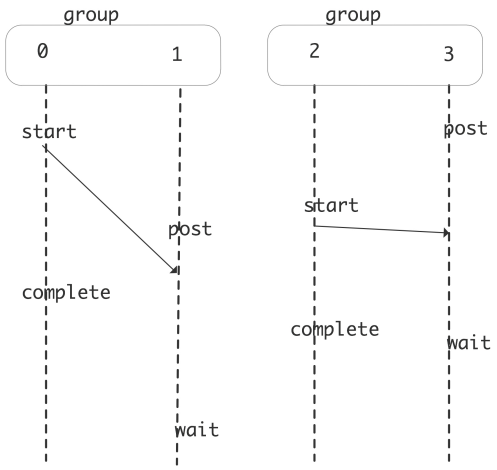


Figure 3.1: Window locking calls in fine-grained active target synchronization

You start and complete an *exposure epoch* with :

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_wait(MPI_Win win)
```

In other words, this turns your window into the *target* for a remote access.

You start and complete an *access epoch* with :

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_complete(MPI_Win win)
```

In other words, these calls border the access to a remote window, with the current processor being the *origin* of the remote access.

In the following snippet a single processor puts data on one other. Note that they both have their own definition of the group, and that the receiving process only does the post and wait calls.

```
// postwaitwin.c
if (mytid==origin) {
  MPI_Group_incl(all_group,1,&target,&two_group);
  // access
  MPI_Win_start(two_group,0,the_window);
  MPI_Put( /* data on origin: */   &my_number, 1,MPI_INT,
           /* data on target: */   target,0,   1,MPI_INT,
       the_window);
  MPI_Win_complete(the_window);
}

if (mytid==target) {
  MPI_Group_incl(all_group,1,&origin,&two_group);
  // exposure
  MPI_Win_post(two_group,0,the_window);
  MPI_Win_wait(the_window);
}
```

3.6.6 Passive target synchronization

This reference section gives the syntax for routines introduced in section 2.3.3.6.

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

3.7 Collectives

3.7.1 Rooted collectives

This reference section gives the syntax for routines introduced in section 2.4.1.

The `MPI_Bcast` call has a single data argument. Its value on the root processor is copied to all other processors, where any previous value is overwritten.

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

There is an example in section 3.1.1.

The `MPI_Reduce` call combines the values from the individual processors. In order not to overwrite the input value on the root, this call has two data arguments, a send buffer and a receive buffer.

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

On processes that are not the root, the receive buffer is ignored.

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

On the root, you need two buffers, which could be a significant memory demand in the case of a large array to be reduced. Therefore, you can specify `MPI_IN_PLACE` as the send buffer on the root. The reduction call then uses the value in the receive buffer as the root's contribution to the operation.

```
// reduceinplace.c
float myrandom,result,*sendbuf,*recvbuf;
myrandom = (float) rand()/(float)RAND_MAX;
int target_proc = ntids-1;
// add all the random variables together
if (mytid==target_proc) {
  sendbuf = (float*)MPI_IN_PLACE; recvbuf = &result;
} else {
  sendbuf = &myrandom;    recvbuf = NULL;
}
MPI_Reduce(sendbuf,recvbuf,1,MPI_FLOAT,MPI_SUM,
           target_proc,comm);
// the result should be approx ntids/2:
if (mytid==target_proc)
  printf("Result %6.3f compared to ntids/2=%5.2f\n",
         result,ntids/2.);
```

3.7.2 Gather and scatter

This reference section gives the syntax for routines introduced in section 2.4.3.

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

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

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

Here is a small example:

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

if (mytid==root)
  localsizes = (int*) malloc( (ntids+1)*sizeof(int) );

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

This will also be the basis of a more elaborate example in section 3.7.5.

The `MPI_IN_PLACE` option can be used for the send buffer on the root; the data for the root is then assumed to be already in the correct location in the receive buffer.

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

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

3.7.3 Reduce-scatter

This reference section gives the syntax for routines introduced in section 2.4.5.

The `MPI_Reduce_scatter` command is equivalent to a reduction on an array of data, followed by a scatter of that data to the individual processes.

To be precise, there is an array `recvcounts` where `recvcounts[i]` gives the number of elements that ultimate wind up on process `i`. The result is equivalent to doing a reduction with a length equal to the sum of the `recvcounts[i]` values, followed by a scatter where process `i` receives `recvcounts[i]` values. (Since the amount of data to be scattered depends on the process, this is in fact equivalent to `MPI_Scatterv` rather than a regular scatter.)

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

For instance, if all `recvcounts[i]` values are 1, the sendbuffer has one element for each process, and the receive buffer has length 1.

An important application of this is establishing an irregular communication pattern. Assume that each process knows which other processes it wants to communicate with; the problem is to let the other processes know about this. The solution is to use `MPI_Reduce_scatter` to find out how many processes want to communicate with you, and then wait for precisely that many messages with a source value of `MPI_ANY_SOURCE`.

```
// reducescatter.c
// record what processes you will communicate with
int *recv_requests;
// find how many procs want to communicate with you
MPI_Reduce_scatter
  (recv_requests,&nsend_requests,counts,MPI_INT,
  MPI_SUM,comm);
// send a msg to the selected processes
for (i=0; i<ntids; i++)
  if (recv_requests[i]>0)
    MPI_Isend(&msg,1,MPI_INT, /*to:*/ i,0,comm,
      mpi_requests+irequest++);
// do as many receives as you know are coming in
for (i=0; i<nsend_requests; i++)
  MPI_Irecv(&msg,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,comm,
      mpi_requests+irequest++);
MPI_Waitall(irequest,mpi_requests,MPI_STATUSES_IGNORE);
```

3.7.4 'All'-type collectives

This reference section gives the syntax for routines introduced in section 2.4.6.

The following collectives construct a result on all processes: `MPI_Allgather`

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

`MPI_Allreduce`

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

`MPI_Alltoall`

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

Each processor has a contribution in their send buffer; the global result is returned in each processor's receive buffer.

```
// allreduce.c
float myrandom,sumrandom;
myrandom = (float) rand()/(float)RAND_MAX;
// add the random variables together
MPI_Allreduce(&myrandom,&sumrandom,
1,MPI_FLOAT,MPI_SUM,comm);
// the result should be approx ntids/2:
if (mytid==ntids-1)
  printf("Result %6.9f compared to .5\n",sumrandom/ntids);
```

If a large amount of data is being communicated, it may be wasteful to have both a (large) send and receive buffer. This problem can be circumvented by using `MPI_IN_PLACE` as the specification of the send buffer. The send data is then assumed to be in the receive buffer. After the reduction it is, of course, overwritten.

```
// allreduceinplace.c
int nrandoms = 500000;
float *myrandoms;
myrandoms = (float*) malloc(nrandoms*sizeof(float));
for (int irand=0; irand<nrandoms; irand++)
  myrandoms[irand] = (float) rand()/(float)RAND_MAX;
// add all the random variables together
MPI_Allreduce(MPI_IN_PLACE,myrandoms,
```

```
nrandoms,MPI_FLOAT,MPI_SUM,comm);
// the result should be approx ntids/2:
if (mytid==ntids-1) {
  float sum=0.;
  for (int i=0; i<nrandoms; i++) sum += myrandoms[i];
  sum /= nrandoms*ntids;
  printf("Result %6.9f compared to .5\n",sum);
}
```

3.7.5 Variable-size-input collectives

This reference section gives the syntax for routines introduced in section 2.4.4.

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

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

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

```
int MPI_Gatherv
  (void *sendbuf, int sendcnt, MPI_Datatype sendtype,
   void *recvbuf, int *recvcnts, int *displs, MPI_Datatype recvtype,
   int root, MPI_Comm comm)
```

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

`MPI_Alltoallv`.

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

For example, in an `MPI_Gatherv` call each process has an individual number of items to contribute. To gather this, the root process needs to find these individual amounts with an `MPI_Gather` call, and

locally construct the offsets array. Note how the offsets array has size `ntids+1`: the final offset value is automatically the total size of all incoming data.

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

// the root process decides how much data will be coming:
// allocate arrays to contain size and offset information
if (mytid==root) {
  localsizes = (int*) malloc( (ntids+1)*sizeof(int) );
  offsets = (int*) malloc( ntids*sizeof(int) );
}
// everyone contributes their info
MPI_Gather(&localsize,1,MPI_INT,
     localsizes,1,MPI_INT,root,comm);
// the root constructs the offsets array
if (mytid==root) {
  offsets[0] = 0;
  for (i=0; i<ntids; i++)
    offsets[i+1] = offsets[i]+localsizes[i];
  alldata = (int*) malloc( offsets[ntids]*sizeof(int) );
}
// everyone contributes their data
MPI_Gatherv(localdata,localsize,MPI_INT,
      alldata,localsizes,offsets,MPI_INT,root,comm);
```

3.7.6 Scan

This reference section gives the syntax for routines introduced in section 2.4.2.

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.

Scan operations are often useful in index calculations. Suppose that every processor has part of a long array, and it knows only how many element it has. The following bit computes the global index of its first element.

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

3.8 Cancelling messages

In section 2.3.1.3 we showed a master-worker example where the master accepts in arbitrary order the messages from the workers. Here we will show a slightly more complicated example, where only the result of the first task to complete is needed. Thus, we issue an `MPI_Recv` with `MPI_ANY_SOURCE` as source. When a result comes, we broadcast its source to all processes. All the other workers then use this information to cancel their message with an `MPI_Cancel` operation.

```
// cancel.c
if (mytid==ntids-1) {
  MPI_Status status;
  ierr = MPI_Recv(dummy,0,MPI_INT, MPI_ANY_SOURCE,0,comm,
                  &status); CHK(ierr);
  first_tid = status.MPI_SOURCE;
  ierr = MPI_Bcast(&first_tid,1,MPI_INT, ntids-1,comm); CHK(ierr);
  printf("first msg came from %d\n",first_tid);
} else {
  float randomfraction = (rand() / (double)RAND_MAX);
  int randomwait = (int) ( ntids * randomfraction );
  MPI_Request request;
  printf("process %d waits for %e/%d=%d\n",
   mytid,randomfraction,ntids,randomwait);
  sleep(randomwait);
  ierr = MPI_Isend(dummy,0,MPI_INT, ntids-1,0,comm,
                   &request); CHK(ierr);
  ierr = MPI_Bcast(&first_tid,1,MPI_INT, ntids-1,comm
                  ); CHK(ierr);
  if (mytid!=first_tid) {
    ierr = MPI_Cancel(&request); CHK(ierr);
  }
}
```

3.9 Communicators

3.9.1 Communicator duplication

This reference section gives the syntax for routines introduced in section 2.6.2.1.

In section 2.8.2 it was explained that MPI messages are non-overtaking. This may lead to confusing situations, witness the following snippet:

```
// commdup_wrong.cxx
class library {
private:
  MPI_Comm comm;
  int mytid,ntids,other;
  MPI_Request *request;
public:
  library(MPI_Comm incomm) {
    comm = incomm;
    MPI_Comm_rank(comm,&mytid);
    other = 1-mytid;
    request = new MPI_Request[2];
  };
  int communication_start();
  int communication_end();
};
  ierr = MPI_Isend(&sdata,1,MPI_INT,other,1,comm,&(request[0])); CHK(ierr);
  my_library.communication_start();
  ierr = MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,comm,&(request[1])); CH
  ierr = MPI_Waitall(2,request,status); CHK(ierr);
  my_library.communication_end();
```

This models a main program that does a simple message exchange, and it makes two calls to library routines. Unbeknown to the user, the library also issues send and receive calls, and they turn out to interfere:

```
int library::communication_start() {
int sdata=6,rdata, ierr;
ierr = MPI_Isend(&sdata,1,MPI_INT,other,2,comm,&(request[0])); CHK(ierr);
ierr = MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,comm,&(request[1])); CHK(
return 0;
}

int library::communication_end() {
MPI_Status status[2];
int ierr;
ierr = MPI_Waitall(2,request,status); CHK(ierr);
return 0;
}
```

Here

- The main program does a send,
- the library call `function_start` does a send and a receive; because the receive can match either send, it is paired with the first one;
- the main program does a receive, which will be paired with the send of the library call;
- both the main program and the library do a wait call, and in both cases all requests are succesfully fulfilled, just not the way you intended.

The solution is to give the library a separate communicator with `MPI_Comm_dup`. Newly created communicators should be released again with `MPI_Comm_free`.

```
// commdup_right.cxx
class library {
private:
  MPI_Comm comm;
  int mytid,ntids,other;
  MPI_Request *request;
public:
  library(MPI_Comm incomm) {
    MPI_Comm_dup(incomm,&comm);
    MPI_Comm_rank(comm,&mytid);
    other = 1-mytid;
    request = new MPI_Request[2];
  };
  ~library() {
    MPI_Comm_free(&comm);
  }
  int communication_start();
  int communication_end();
};
```

3.9.2 Splitting communicators

This reference section gives the syntax for routines introduced in section 2.6.2.2.

The command `MPI_Comm_split` takes a communicator, and divides it into a number of disjoint communicators. It does this by assigning processes to the same subcommunicator if they have the same user-specified 'colour' value.

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

The ranking of processes in the new communicator is determined by a 'key' value. Most of the time, there is no reason to use a relative ranking that is different from the global ranking, so the `MPI_Comm_rank` value of the global communicator is a good choice.

```
// mvp2d.cxx
row_number = ntids % ntids_i;
col_number = ntids / ntids_j;
MPI_Comm_split(global_comm,row_number,mytid,&row_comm);
MPI_Comm_split(global_comm,col_number,mytid,&col_comm);
```

3.10 Error handling

This reference section gives the syntax for routines introduced in section 2.8.3.

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.

3.11 More utility stuff

3.11.1 Timing

This reference section gives the syntax for routines introduced in section 2.8.5.

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.

```
// pingpong.c
int src = 0,tgt = ntids/2;
double t, send=1.1,recv;
if (mytid==src) {
  t = MPI_Wtime();
  for (int n=0; n<NEXPERIMENTS; n++) {
    MPI_Send(&send,1,MPI_DOUBLE,tgt,0,comm);
    MPI_Recv(&recv,1,MPI_DOUBLE,tgt,0,comm,MPI_STATUS_IGNORE);
  }
  t = MPI_Wtime()-t; t /= NEXPERIMENTS;
  printf("Time for pingpong: %e\n",t);
} else if (mytid==tgt) {
  for (int n=0; n<NEXPERIMENTS; n++) {
    MPI_Recv(&recv,1,MPI_DOUBLE,src,0,comm,MPI_STATUS_IGNORE);
    MPI_Send(&recv,1,MPI_DOUBLE,src,0,comm);
  }
}
```

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);
```

3.12 Multi-threading

This reference section gives the syntax for routines introduced in section 2.7.

```
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.

PART II

OPENMP

Chapter 4

OpenMP tutorial

4.1 Basics

4.1.1 OpenMP code structure

The reference for the commands introduced here can be found in section 5.1.1.

4.1.2 Stuff

4.1.2.1 Critical sections

There are two pragmas for critical sections: and . The second one is more limited but has performance advantages.

A `critical` section works by acquiring a lock, which carries a substantial overhead. Furthermore, if your code has multiple critical sections, they are all mutually exclusive: if a thread is in one critical section, the other ones are all blocked.

On the other hand, the syntax for `atomic` sections is limited, but such sections are not exclusive and they can be more efficient, since they assume that there is a hardware mechanism for making them critical.

The problem with `critical` sections being mutually exclusive can be mitigated by naming them:

```
#pragma omp critical (optional_name_in_parens)
```

Chapter 5

OmpMP Reference

This section gives reference information and illustrative examples of the use of MPI. While the code snippets given here should be enough, full programs can be found in the repository for this book https://bitbucket.org/VictorEijkhout/parallel-computing-book.

5.1 Basics

5.1.1 OpenMP setup

This reference section gives the syntax for routines introduced in section 4.1.1.

If you use OMP commands in a program file, be sure to include the proper header file *omp.h*.

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

PART III

THE REST

Chapter 6

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 7

Support libraries

ParaMesh

Global Arrays

PETSc

Hdf5 and Silo

PART IV

TUTORIALS

here are some tutorials

7.1 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 7.1.6.

7.1.1 A simple example

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

7.1.1.1 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.

7.1.1.2 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.

7.1.2 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.

7.1.2.1 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.

7.1.2.2 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.

7.1.3 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}}
```

7.1.4 Miscellania

7.1.4.1 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.

7.1.4.2 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`.

7.1.4.3 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`.

7.1.4.4 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
```

7.1.5 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.)

7.1.6 A Makefile for LaTeX

7.2 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`.

7.2.1 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.

7.2.2 Finding errors

Let us now consider some programs with errors.

7.2.2.1 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.
```

7.2.2.2 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.

7.2.3 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?

7.2.4 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 201
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.

7.2.5 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?

7.2.6 Further reading

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

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

7.3 Tracing

7.3.1 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 $@ $^
```

PART V

PROJECTS, INDEX

Chapter 8

Class projects

8.1 A style guide for project submissions

Here are some guidelines for how to submit assignments and projects. As a general rule, consider programming as an experimental science, and your writeup as a report on some tests you have done: explain the problem you're addressing, your strategy, your results.

Structure of your writeup Most of the exercises in this book test whether you are able to code the solution to a certain problem. That does not mean that turning in the code is sufficient, nor code plus sample output. Turn in a writeup in pdf form that was generated from a text processing program such as Word or (preferably) LaTeX (for a tutorial, see HPSC-**??**). Your writeup should have

- The relevant fragments of your code,
- an explanation of your algorithms or solution strategy,
- a discussion of what you observed,
- graphs of runtimes and TAU plots; see 7.3.

Observe, measure, hypothesize, deduce In most applications of computing machinery we care about the efficiency with which we find the solution. Thus, make sure that you do measurements. In general, make observations that allow you to judge whether your program behaves the way you would expect it to.

Quite often your program will display unexpected behaviour. It is important to observe this, and hypothesize what the reason might be for your observed behaviour.

Including code If you include code samples in your writeup, make sure they look good. For starters, use a mono-spaced font. In LaTeX, you can use the `verbatim` environment or the `verbatiminput` command. In that section option the source is included automatically, rather than cut and pasted. This is to be preferred, since your writeup will stay current after you edit the source file.

Including whole source files makes for a long and boring writeup. The code samples in this book were generated as follows. In the source files, the relevant snippet was marked as

```
... boring stuff
#pragma samplex
  .. interesting! ..
#pragma end
... more boring stuff
```

The files were then processed with the following command line (actually, included in a makefile, which requires doubling the dollar signs):

```
for f in *.{c,cxx,h} ; do
  cat $x | awk 'BEGIN {f=0}
                    /#pragma end/ {f=0}
                    f==1 {print $0 > file}
                    /pragma/ {f=1; file=$2 }
                  '
done
```

which gives (in this example) a file `samplex`. Other solutions are of course possible.

Code formatting Code without proper indentation is very hard to read. Fortunately, most editors have some knowledge of the syntax of the most popular languages. The *emacs* editor will, most of the time, automatically activate the appropriate mode based on the file extension. If this does not happen, you can activate a mode by `ESC x fortran-mode` et cetera, or by putting the string `-*- fortran -*-` in a comment on the first line of your file.

The *vi* editor also has syntax support: use the commands `:synxtax on` to get syntax colouring, and `:set cindent` to get automatic indentation while you're entering text. Some of the more common questions are addressed in http://stackoverflow.com/questions/97694/auto-indent-spaces-with-c-i

Running your code A single run doesn't prove anything. For a good report, you need to run your code for more than one input dataset (if available) and in more than one processor configuration. When you choose problem sizes, be aware that an average processor can do a billion operations per second: you need to make your problem large enough for the timings to rise above the level of random variations and startup phenomena.

When you run a code in parallel, beware that on clusters the behaviour of a parallel code will always be different between one node and multiple nodes. On a single node the MPI implementation is likely optimized to use the shared memory. This means that results obtained from a single node run will be unrepresentative. In fact, in timing and scaling tests you will often see a drop in (relative) performance going from one node to two. Therefore you need to run your code in a variety of scenarios, using more than one node.

Repository organization If you submit your work through a repository, make sure you organize your submissions in subdirectories, and that you give a clear name to all files.

8.2 Warmpup exercises

We start with some simple exercises.

8.2.1 Hello world

The exercises in this section are about the routines introduced in section 2.2.3; for the reference information see section **??**.

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

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

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

8.2.2 Collectives

It is a good idea to be able to collect statistics, so before we do anything interesting, we will look at MPI collectives; section 2.4.

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

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

and measures how long the sleep actually was:

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

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

Exercise 8.1. Change this program to compute the average jitter by changing the reduction operator.

Exercise 8.2. Now compute the standard deviation

$$\sigma = \sqrt{\frac{\sum_i (x_i - m)^2}{n}}$$

where m is the average value you computed in the previous exercise.

- Solve this exercise twice: once by following the reduce by a broadcast operation and once by using an `Allreduce`.
- Run your code both on a single cluster node and on multiple nodes, and inspect the TAU trace. Some MPI implementations are optimized for shared memory, so the trace on a single node may not look as expected.
- Can you see from the trace how the allreduce is implemented?

Exercise 8.3. Finally, use a gather call to collect all the values on processor zero, and print them out. Is there any process that behaves very differently from the others?

8.2.3 Linear arrays of processors

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

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

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

```
// linear-serial.c
int my_number = mytid, other_number=-1.;
```

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

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

```
int *gather_buffer=NULL;
if (mytid==0) {
  gather_buffer = (int*) malloc(ntids*sizeof(int));
  if (!gather_buffer) MPI_Abort(comm,1);
}
MPI_Gather(&other_number,1,MPI_INT,
           gather_buffer,1,MPI_INT, 0,comm);
if (mytid==0) {
  int i,error=0;
  for (i=0; i<ntids; i++)
    if (gather_buffer[i]!=i-1) {
```

```
        printf("Processor %d was incorrect: %d should be %d\n",
               i,gather_buffer[i],i-1);
        error =1;
      }
    if (!error) printf("Success!\n");
    free(gather_buffer);
  }
```

8.2.3.1 Coding with blocking calls

Passing data to a neighbouring processor should be a very parallel operation. However, if we code this naively, with `MPI_Send` and `MPI_Recv`, we get an unexpected serial behaviour, as was explained in section 2.3.1.

```
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 2.3.1.2 for the explanation why.)

Exercise 8.4. Compile and run this code, and generate a TAU trace file. Confirm that the execution is serial. Does replacing the `Ssend` by `Send` change this?

Let's clean up the code a little.

Exercise 8.5. First write this code more elegantly by using `MPI_PROC_NULL`.

8.2.3.2 A better blocking solution

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

Exercise 8.6. Rewrite the code using `MPI_Sendrecv`. Confirm with a TAU trace that execution is no longer serial.

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

8.2.3.3 Non-blocking calls

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

Exercise 8.7. Implement a fully parallel version by using `MPI_Isend` and `MPI_Irecv`.

8.2.3.4 One-sided communication

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

Exercise 8.8. Write two versions of the code: one using `MPI_Put` and one with `MPI_Get`. Make TAU traces.

Investigate blocking behaviour through TAU visualizations.

Exercise 8.9. If you transfer a large amount of data, and the target processor is occupied, can you see any effect on the origin? Are the fences synchronized?

8.3 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.

In this set of exercises you are going to take an example program `mandel_main.cxx` and extend it to use a variety of MPI programming constructs. This program has been set up as a *master-worker* model: there is one master processor (for a change this is the last processor, rather than zero) which gives out work to, and accepts results from, the worker processors. It then takes the resuls and construct an image file from them.

8.3.1 Invocation

The `mandel_main` program is called as

```
mpirun -np 123 mandel_main steps 456 iters 789
```

where the `steps` parameter indicates how many steps in x, y direction there are in the image, and `iters` gives the maximum number of iterations in the `belong` test.

If you forget the parameter, you can call the program with

```
mandel_serial -h
```

and it will print out the usage information.

8.3.2 Tools

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

```
// mandel.h
class circle {
 public :
  circle(int pxls,int bound,int bs);
  void next_coordinate(struct coordinate& xy);
  int is_valid_coordinate(struct coordinate xy);
  void invalid_coordinate(struct coordinate& xy);
```

and a global routine that tests whether a coordinate is in the set, at least up to an iteration bound. It returns zero if the series from the given starting point has not diverged, or the iteration number in which it diverged if it did so.

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

In the former case, the point could be in the Mandelbrot set, and we colour it black, in the latter case we give it a colour depending on the iteration number.

```
if (iteration==0)
  memset(colour,0,3*sizeof(float));
else {
  float rfloat = ((float) iteration) / workcircle->infty;
  colour[0] = rfloat;
  colour[1] = MAX((float)0,(float)(1-2*rfloat));
  colour[2] = MAX((float)0,(float)(2*(rfloat-.5)));
}
```

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

```
void queue::wait_for_work(MPI_Comm comm,circle *workcircle) {
```

```
MPI_Status status; int ntids;
MPI_Comm_size(comm,&ntids);
int stop = 0;

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

  MPI_Recv(&xy,1,coordinate_type,ntids-1,0, comm,&status);
  stop = !workcircle->is_valid_coordinate(xy);
  if (stop) 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:

```
// mandel_serial.cxx
class serialqueue : public queue {
private :
  int free_processor;
public :
  serialqueue(MPI_Comm queue_comm,circle *workcircle)
    : queue(queue_comm,workcircle) {
    free_processor=0;
  };
  /**
      The 'addtask' routine adds a task to the queue. In this
      simple case it immediately sends the task to a worker
      and waits for the result, which is added to the image.

      This routine is only called with valid coordinates;
      the calling environment will stop the process once
      an invalid coordinate is encountered.
  */
  int addtask(struct coordinate xy) {
    MPI_Status status; int contribution, err;

    err = MPI_Send(&xy,1,coordinate_type,
     free_processor,0,comm); CHK(err);
```

```
    err = MPI_Recv(&contribution,1,MPI_INT,
     free_processor,0,comm, &status); CHK(err);

    coordinate_to_image(xy,contribution);
    total_tasks++;
    free_processor++;
    if (free_processor==ntids-1)
      // wrap around to the first again
      free_processor = 0;
    return 0;
  };
```

Exercise 8.10. Explain why this solution is very inefficient. Make a trace of its execution that bears this out.

8.3.3 Bulk task scheduling

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

Exercise 8.11. Code a solution where you give a task to all worker processes using non-blocking sends and receives, and then wait for these tasks with `MPI_Waitall` to finish before you give a new round of data to all workers. Make a trace of the execution of this and report on the total time.
You can do this by writing a new class that inherits from `queue`, and that provides its own `addtask` method:

```
// mandel_bulk.cxx
class bulkqueue : public queue {
public :
  bulkqueue(MPI_Comm queue_comm,circle *workcircle)
    : queue(queue_comm,workcircle) {
```

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

8.3.4 Collective task scheduling

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

Exercise 8.12. Code a solution which uses scatter to distribute data to the worker tasks, and gather to collect the results. Is this solution more or less efficient than the previous?

8.3.5 Asynchronous task scheduling

At the start of section 8.3.3 we said that bulk scheduling mostly makes sense if all tasks take similar time to complete. In the Mandelbrot case this is clearly not the case.

Exercise 8.13. Code a fully dynamic solution that uses `MPI_Probe` or `MPI_Waitany`. Make an execution trace and report on the total running time.

8.3.6 One-sided solution

Let us reason about whether it is possible (or advisable) to code a one-sided solution to computing the Mandelbrot set. With active target synchronization you could have an exposure window on the host to which the worker tasks would write. To prevent conflicts you would allocate an array and have each worker write to a separate location in it. The problem here is that the workers may not be sufficiently synchronized because of the differing time for computation.

Consider then passive target synchronization. Now the worker tasks could write to the window on the master whenever they have something to report; by locking the window they prevent other tasks from interfering. After a worker writes a result, it can get new data from an array of all coordinates on the master.

It is hard to get results into the image as they become available. For this, the master would continuously have to scan the results array. Therefore, constructing the image is easiest done when all tasks are concluded.

8.4 Data parallel grids

8.4.1 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`.

8.4.1.1 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

$$g'_{ij} = 1/4 \cdot (g_{i+1,j} + g_{i-1,j} + g_{i,j+1} + g_{i,j-1}). \tag{8.1}$$

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

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

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

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

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

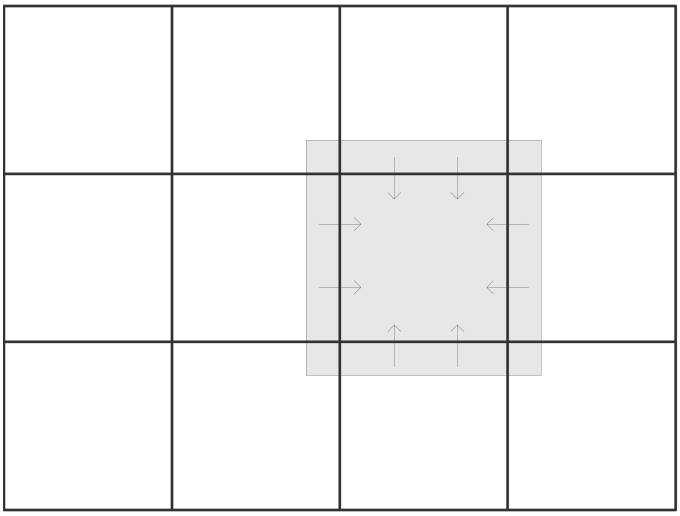


Figure 8.1: A grid divided over processors, with the 'ghost' region indicated

From formula (8.1) 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 8.1. These elements surrounding the processor's own part are called the *halo* or *ghost region* of that processor.

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

8.4.1.2 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-**??** and HPSC-**??**.) Note how the `INDEX` macro is used to index in a $\texttt{ilength} \times \texttt{jlength}$ target array `values`, while reading from a $(\texttt{ilength}+2) \times (\texttt{jlength}+2)$ 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.;
      }
    }
```

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.

[4] R. Thakur, W. Gropp, and B. Toonen. Optimizing the synchronization operations in MPI one-sided communication. *Int'l Journal of High Performance Computing Applications*, 19:119–128, 2005.

Chapter 9

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