Parallel Computing for Science and Engineering

Victor Eijkhout

1st edition 2015

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

1 **Introduction to parallel programming** *8*
1.1 *Introduction* *8*
1.2 *Parallel variants* *11*
1.3 *Advanced topics* *23*

I MPI
29
2 **MPI tutorial** *30*
2.1 *Distributed memory and message passing* *30*
2.2 *Basic concepts* *32*
2.3 *Point-to-point communication* *34*
2.4 *Collectives* *50*
2.5 *Data types* *57*
2.6 *Communicators* *61*
2.7 *Synchronization* *65*
2.8 *Hybrid programming: MPI and threads* *65*
2.9 *Leftover topics* *66*
2.10 *Review questions* *72*
2.11 *Literature* *73*
3 **MPI Reference** *74*
3.1 *Basics* *74*
3.2 *Data types* *76*
3.3 *Blocking communication* *82*
3.4 *Deadlock-free blocking messages* *86*
3.5 *Non-blocking communication* *86*
3.6 *One-sided communication* *91*
3.7 *Collectives* *96*
3.8 *Cancelling messages* *103*
3.9 *Communicators* *104*
3.10 *Leftover topics* *107*
3.11 *Error handling* *108*
3.12 *More utility stuff* *108*
3.13 *Multi-threading* *109*

II OpenMP
111
4 **OpenMP tutorial** *112*
4.1 *Basics* *112*
4.2 *Creating parallelism* *117*
4.3 *Work sharing* *119*
4.4 *Controlling thread data* *126*
4.5 *Reductions* *129*
4.6 *Synchronization* *131*
4.7 *Tasks* *136*
4.8 *Stuff* *138*
4.9 *Performance* *143*
5 **OmpMP Reference** *144*
5.1 *Basics* *144*
5.2 *Thread stuff* *145*
5.3 *Parallel regions* *145*
5.4 *Worksharing* *146*
5.5 *Controlling thread data* *149*
5.6 *Synchronization* *150*
5.7 *Internal control variables* *152*
5.8 *Tasks* *153*
5.9 *Stuff* *153*

III The Rest
155
6 **Hybrid computing** *156*
6.1 *Discussion* *156*
6.2 *Hybrid MPI-plus-threads execution* *156*
7 **Support libraries** *157*

IV Tutorials
159
7.1 *Debugging* *161*
7.2 *Tracing* *170*

V Projects, index
171
8 **Class projects** *172*
8.1 *A Style Guide to Project Submissions* *172*
8.2 *Warmup Exercises* *174*
8.3 *Mandelbrot set* *178*

8.4 *Data parallel grids* *184*
9 **Ascii table** *187*
10 **Index and list of acronyms** *190*

Chapter 1

Introduction to parallel programming

1.1 Introduction

There are many ways 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*[1], 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[2]. 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:

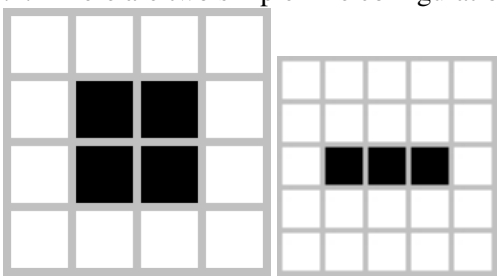
- If a cell is alive, and it has fewer than two live neighbours, it dies of loneliness.

1. Martin Gardner, 'Mathematical Games – the fantastic combinations of John Conway's new solitaire game Life', Scientific American 223, October 1970, pp 120–123.
2. Victor Eijkhout, with Robert van de Geijn and Edmond Chow, Introduction to High Performance Scientific Computing, 2011.

- 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. If you search online you can find some great animations.

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[3]:

```
def life_evaluation( cells ):
  # cells is a 3x3 array
  count = 0
  for i in [0,1,2]:
    for j in [0,1,2]:
      if i!=1 and j!=1:
        count += cells[i,j]
  return life_count_evaluation( cells[1,1],count )
def life_count_evaluation( cell,count )
  if count<2:
    return 0 # loneliness
```

3. We use a quasi-python syntax here, except that in arrays we let the upper bound be inclusive.

```
  elif count>3:
    return 0 # overcrowding
  elif cell==1 and (count==2 or count==3):
    return 1 # live on
  elif cell==0 and count==3:
    return 1 # spontaneous generation
  else:
    return 0 # no change in dead cells
```

The driver code would then be something like:

```
# create an initial board; we omit this code
life_board.create(final_time,N,N)

# now iterate a number of steps on the whole board
for t in [0:final_time-1]:
    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: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 creates a board for each time step, 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-1]:
    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.

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

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.

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 language.

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 except in limited cases, 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 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.

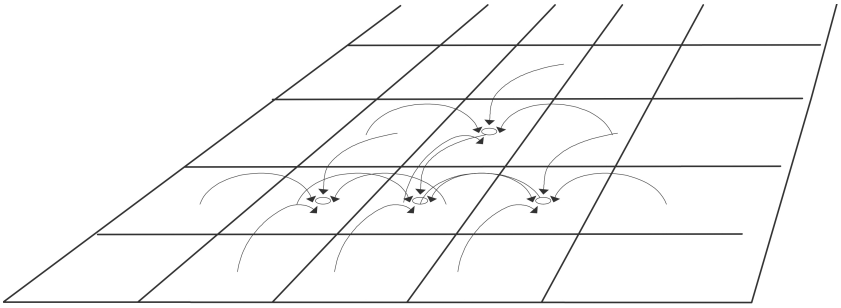


Figure 1.1: Illustration of data parallelism: all points of the board get the same update treatment

The fine-grained data parallel model of computing is known as Single Instruction Multiple Data (SIMD): the same instruction is performed on multiple data elements. 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. Thus, 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, up to 2^{16} elements. These days, SIMD still exists, but in slightly different guises and on much smaller scale; 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 1.2 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 executed with these vector instructions.

First of all the code needs to have the right structure. The original code does not have a lot of parallelism in the inner loop, where it can be exploited with vector instruction:

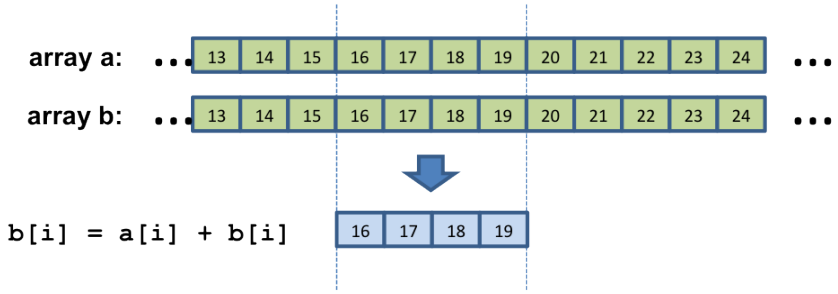


Figure 1.2: Four-wide vector instructions work on four operand pairs at the same time

```
for i in [0:N]:
  for j in [0:N]:
    count = 0
    for ii in {-1,0,+1}:
      for jj in {-1,0,+1}:
        if ii!=0 and jj!=0:
          count += board[i+ii,j+jj]
```

Instead, we have to exchange loops as:

```
for i in [0:N]:
  for j in [0:N]:
    count[j] = 0
  for ii in {-1,0,+1}:
    for jj in {-1,0,+1}:
      if ii!=0 and jj!=0:
        for j in [0:N]:
          count[j] += board[i+ii,j+jj]
```

Note that the `count` variable now has become an array. This is one of the reasons that compilers are unable to make this transformation.

Regular programming languages have no way of saying ‘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 cannot 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]
```

Here we perform half the number of iterations, but each new iteration comprises two old ones. In this version the compiler will have no trouble concluding that there are two operations that can be done simultaneously. This transformation of a loop is called *loop unrolling*, in this case, 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. In this section we will look at *pipelining*, which is a different way of dealing with identical instructions.

Imagine 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 to 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*.

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 off 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 Graphics Processing Units (GPUs) for SIMD-type numerical computations. Doing so was cumbersome, until *NVIDIA* came out with the *Compute-Unified Device Architecture (CUDA)* language. CUDA is a way of explicitly doing data parallel programming: you write a 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. Since the locations can be updated independently, the *iterations* of the loop are *independent* and can be executed in any order, or in fact simultaneously. So is there a way to tell the compiler that the iterations are independent, and let the compiler decide how to execute them?

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]
```

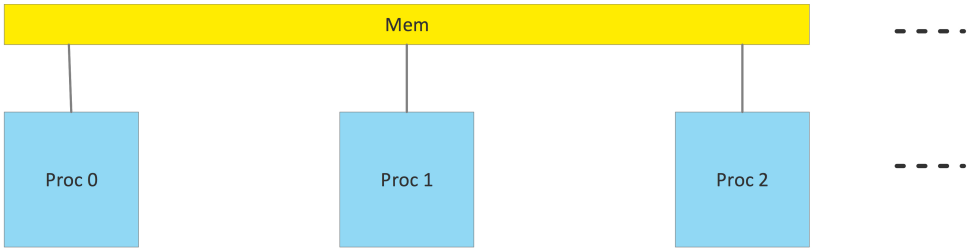


Figure 1.3: Illustration of shared memory: all processors access the same memory

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 structured: moving such small amounts of work around can be more costly than executing them (for a detailed discussion see section HPSC-1.3). 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 1.3.

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 element 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

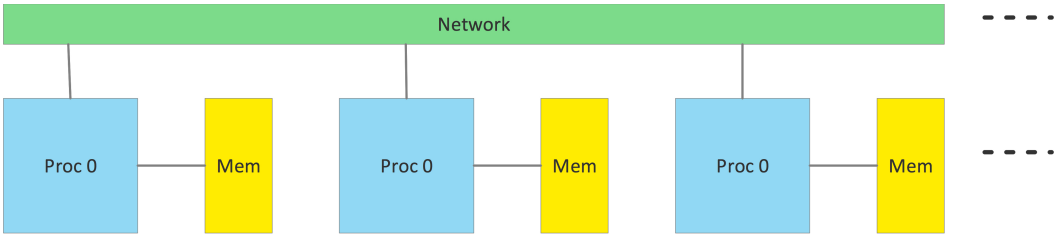


Figure 1.4: Illustration of distributed memory: every processor has its own memory and is connected to others through a network

put in a single PC, at the moment about 1 terabyte (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 a 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 1.4. The advantage of this design is that tying together thousands of processors allows you to run very large problems. For instance, the almost 13 thousand processors of the Stampede supercomputer[4] (figure 1.5) have almost 200 terabytes of memory. Parallel programming on such a machine is a little harder than what 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: in the Stampede picture you can see yellow cabling connecting the nodes in each cabinet, and orange cabling overhead that connects the cabinets. Accessing data over the network probably involves an operating system call and accessing the network card, both of which are slow operations.

4. Stampede has more than 6400 nodes, each with 2 Intel Sandy Bridge processors. Each node also has an Intel Xeon Phi co-processor, but we don't count those for the moment.

Figure 1.5: The Stampede supercomputer at the Texas Advanced Supercomputing Center

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 1.6 a process 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)
```

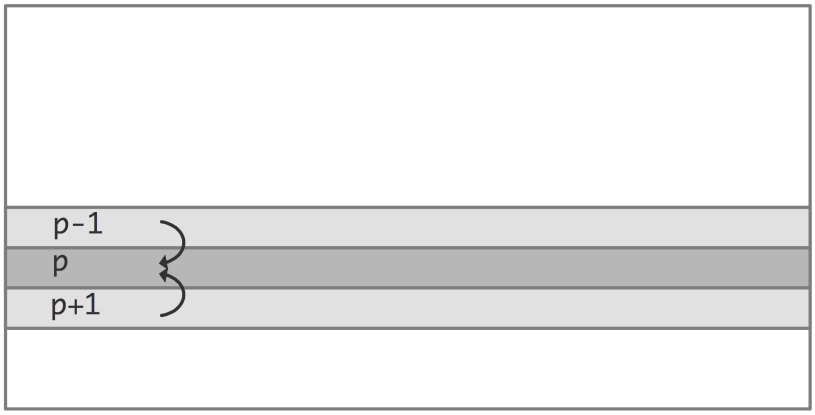


Figure 1.6: Processor p receives a line of data from $p-1$ and $p+1$

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)
```

(We omit the code for `life_line_update`, which computes the updated cell values on a single line.) 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 cannot 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

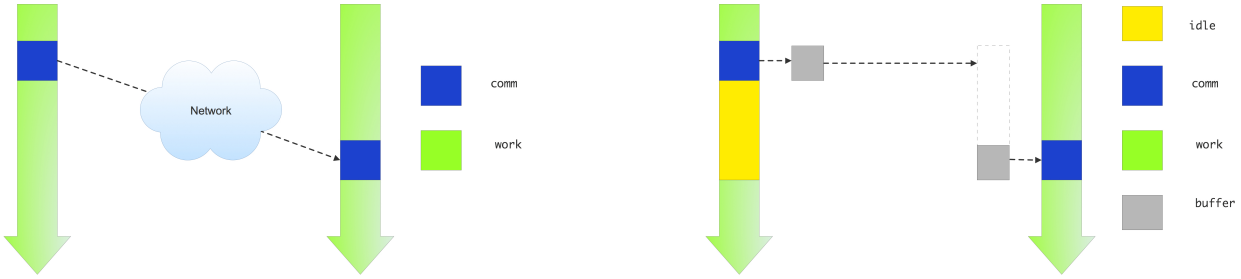


Figure 1.7: Illustration of 'ideal' and 'blocking' send

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 1.7. 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 processor, 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?

Finding a 'clever rearrangement' is usually not the best way to solve deadlock problems. A common solution is to use *non-blocking communication* calls. Here the send or receive instruction only indicates to the system the buffer with send data, or in which to receive data. You then need a second call to ensure that the operation is actually completed.

In pseudo-code:

```
send( buffer1, to=neighbour1, result=request1 );
send( buffer2, to=neighbour2, result=request2 );
// maybe execute some other code
wait( request1 ); wait( request2 ); // make sure the operations are done
```

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 time step 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 exercise makes an important point about dependence and independence. If the value at `i,j` depends on 3×3 previous points, and each of these have a similar dependence, we can compute the value at `i,j` if we know 5×5 points two steps away, 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 definition of tasks is that of a Directed Acyclic Graph (DAG): the dependencies between tasks form a graph, and you cannot 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:

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 pseudo-code for this is:

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

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 parts 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-1.3.) So with $N \times N$ divisions of the board and T time steps, 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 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 1.8 for an illustration of the basic idea.) In 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[5]; and

5. 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-1.3.2.

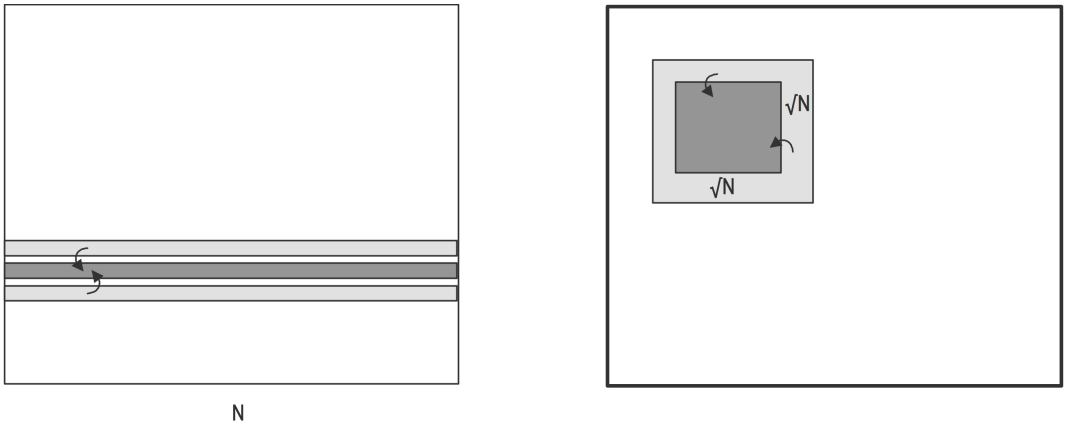


Figure 1.8: One-dimensional and two-dimensional distribution communication

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. In practice you will have fewer processors, and each processor will have a subdomain rather than a single point. However, 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-6.2.2.3 for the analysis of the matrix-vector product algorithm.

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 per processor 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:

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

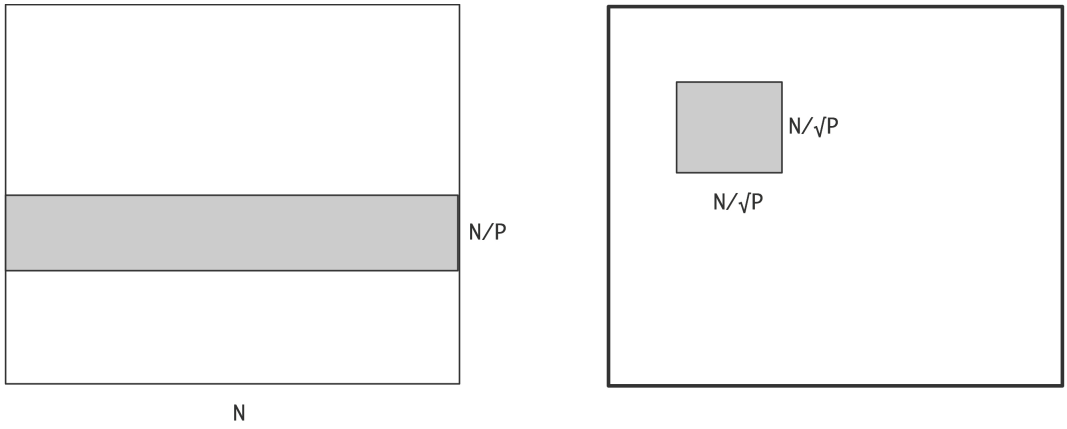


Figure 1.9: One-dimensional and two-dimensional distribution of a Life board

Let's now consider a one-dimensional distribution. (Left half of figure 1.9.) Every processor but the first 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 1.9.) 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: different parallelization strategies can have different overhead and therefore different efficiencies. Both the one and two-dimensional distribution lead to a perfect parallelization of the work. On the other hand, with the two-dimension distribution the communication is constant, while 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

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

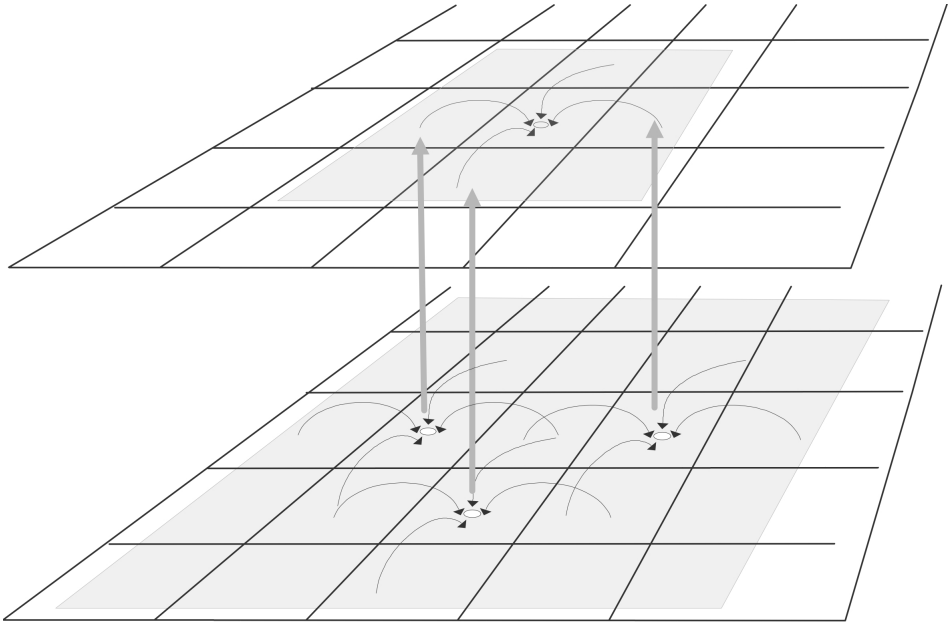


Figure 1.10: Two steps of Life updates

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 will 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 them. This is illustrated in figure 1.10. 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. We assume that the board is square of size $N \times N$, and that there are $P \times P$ processors, so that each processor is computing an $(N/P) \times (N/P)$ part of the board.

In the one-step-at-a-time implementation a processor does the following:

1. Receives four messages of length 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 $2N$ and four of size 4;

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. Since communication latency can be quite high, this scheme can be faster overall.

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-2.2 for definition and discussion of 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 has 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. How to do this depends on what sort of parallelism you are dealing with.

In section 1.2.3.4 you saw that in shared memory it can make sense to divide the work in more units than there are processors. Statistically, this evens out the work balance. On the other hand, in distributed memory (section 1.2.3.3) such dynamic assignment is not possible, so you have to be careful in dividing up the work. 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 processors have to reallocate and rearrange their data structures. This is a very advanced topic, and not at all simple to do.

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

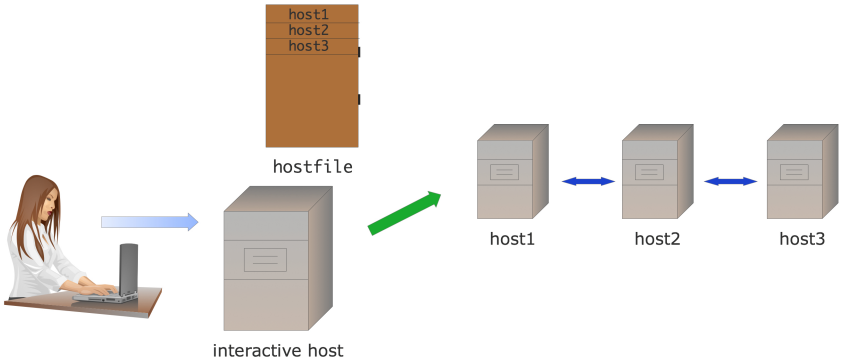


Figure 2.1: Interactive MPI setup

- 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`

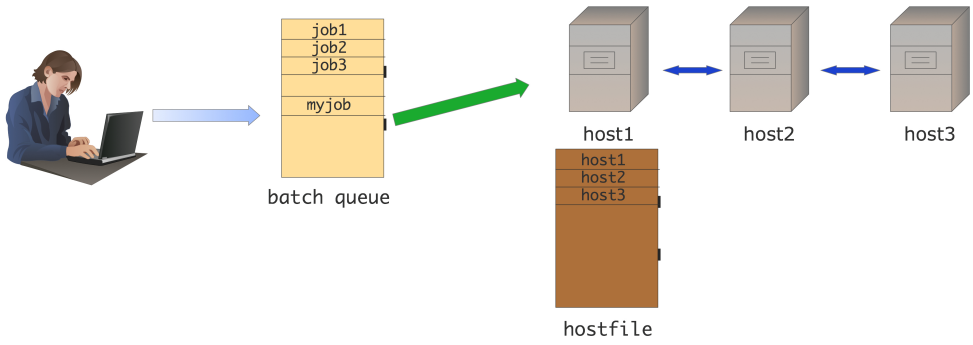


Figure 2.2: Batch MPI setup

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

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

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

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. The calls for this are `MPI_Init` and `MPI_Finalize`. 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. For Fortran see the reference section 3.1.1. (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 2.2.4.

In the SPMD model you run the same executable on each of a set of processors; see section HPSC-2.3.2. So how can you do anything useful if all processors run the same code? Here is where your first two MPI routines come in, 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();
```

2.2.4 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 );
```

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 Concepts: blocking vs non-blocking, synchronous vs asynchronous

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

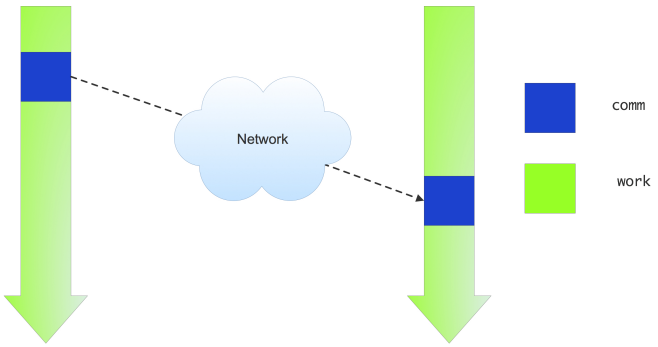


Figure 2.3: Illustration of an ideal send-receive interaction

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

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 a local data structure until the receiving process has indicated that it is ready to receive it. This is pictured in figure 2.4, left picture. This

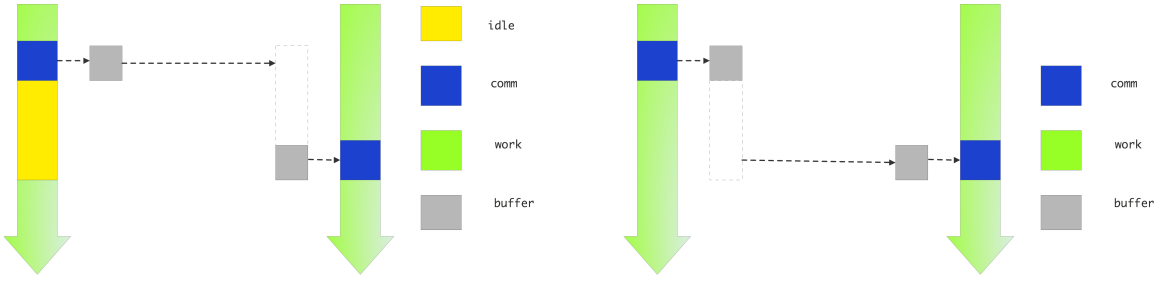


Figure 2.4: Left: illustration of a blocking communication: the sending processor is idle until the receiving processor issues and finishes the receive call. Right: illustration of a non-blocking communication: the sending processor immediately continues execution after issueing the send call

is known as *blocking* communication: a process that issues a send or receive call will then block until the corresponding receive or send call is successfully concluded. However, this can be inefficient; it is better

if the data is set aside for the system to transfer it, while the user program continues. This is known as *non-blocking* communication, illustrated in the right figure.

It is easiest to think of blocking as a form of synchronization with the other process, but that is not quite true. Synchronization is a concept in itself, and we talk about *synchronous* communication if there is actual coordination going on with the other process, and *asynchronous* communication if there is not. Blocking then only refers to the program waiting until the user data is safe to reuse; in the synchronous case a blocking call means that the data is indeed transferred, in the asynchronous case it only means that the data has been transferred to some system buffer. The four possible cases are illustrated in figure 2.5.

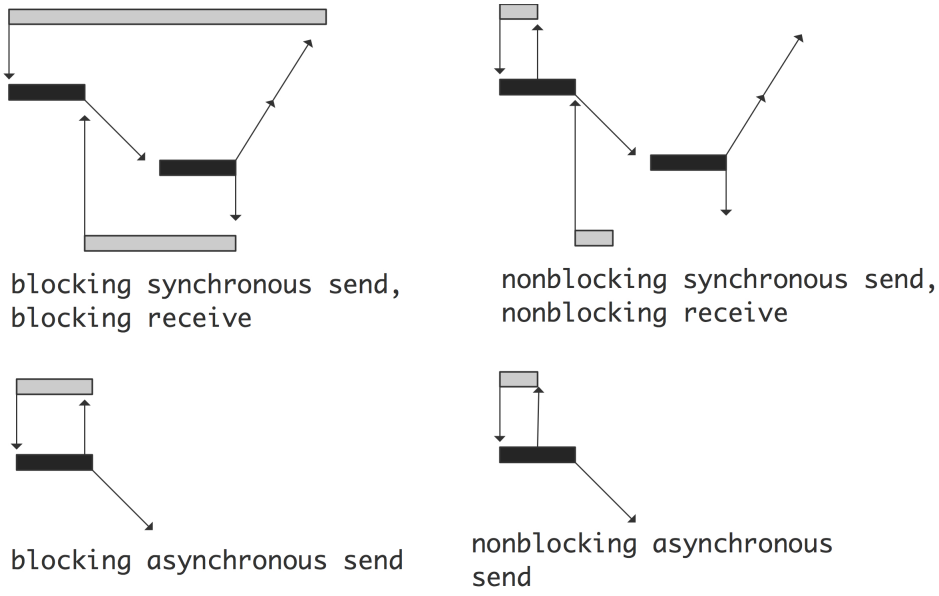


Figure 2.5: Blocking and synchronicity

2.3.2 Blocking communication

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

Above we signalled a first problem with blocking communication: if your processes are not perfectly synchronized your performance may degrade because processes spend time waiting for each other; see HPSC-2.10.1. But there is a more insidious, more serious problem. Suppose two process need to exchange data, and consider the following pseudo-code, which purports to exchange data between processes 0 and 1:

```
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 one. This is illustrated in figure 2.6.

Exercise 2.1. Modify your earlier code, run it and reproduce the trace output of figure 2.6.

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);
```

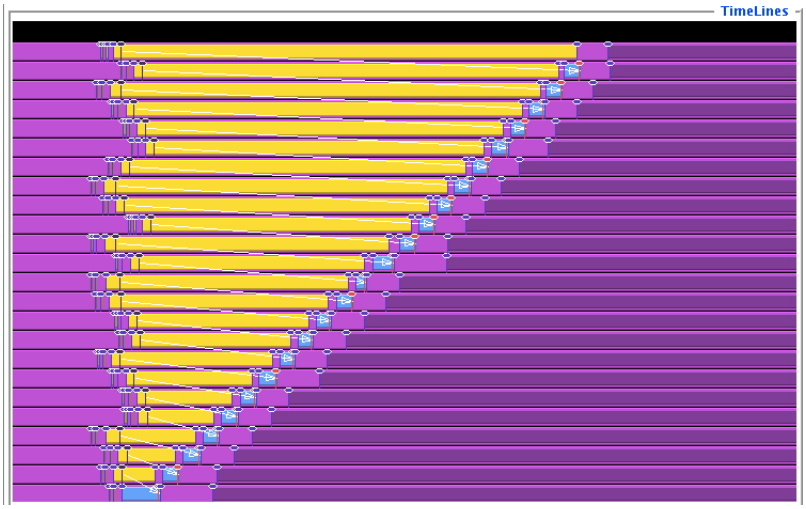


Figure 2.6: Trace of a simple send-recv code

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.

2.3.2.1 *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`

```
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; see figure 2.7. 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.

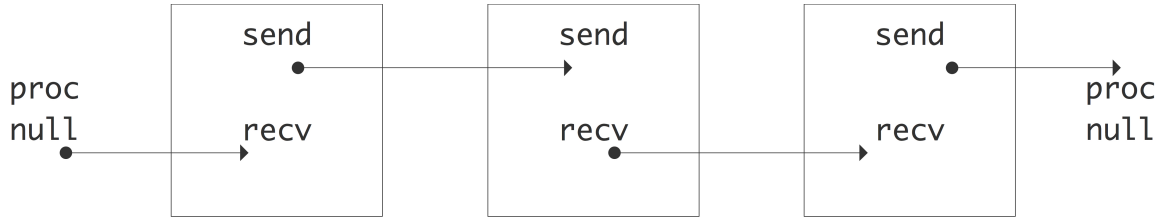


Figure 2.7: An MPI Sendrecv call

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?

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

2.3.2.2 Wildcards in the receive call

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.

2.3.3 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 **??**.

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

Exercise 2.4. Read section HPSC-6.5 and give pseudo-code for the distributed sparse matrix-vector product using the above idiom for using `MPI_Test...` calls. Discuss the advantages and disadvantages of this approach. The answer is not going to be black and white: discuss when you expect which approach to be preferable.

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

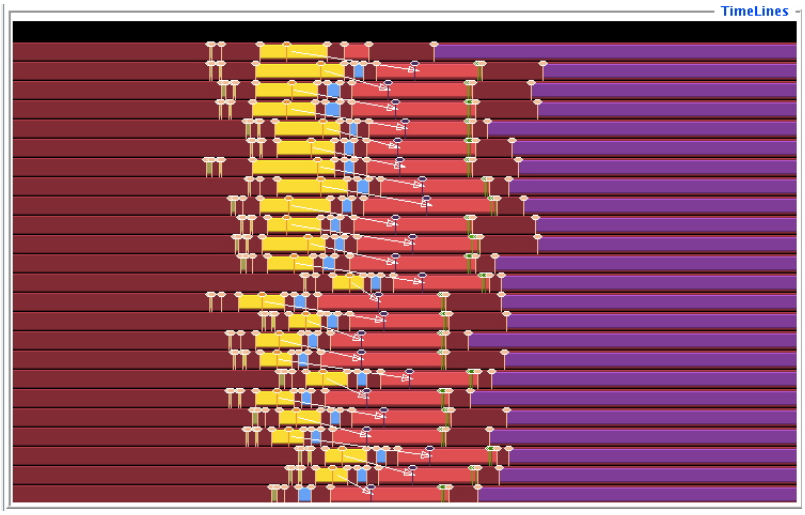


Figure 2.8: A trace of a nonblocking send between neighbouring processors

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
}
```

2.3.3.1 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.

2.3.3.2 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`).

2.3.4 Buffered communication

The reference for the commands introduced here can be found in section 3.5.1.

By now you have probably got the notion that managing buffer space in MPI is important: data has to be somewhere, either in user-allocated arrays or in system buffers. Buffered sends are yet another way of managing buffer space.

1. You allocate your own buffer space, and you attach it to your process;
2. You use the `MPI_Bsend` call for sending;
3. You detach the buffer when you're done with the buffered sends.

There can be only one buffer per process; its size should be enough for all outstanding `MPI_Bsend` calls that are simultaneously outstanding, plus `MPI_BSEND_OVERHEAD`.

2.3.5 Persistent communication

The reference for the commands introduced here can be found in section 3.5.2.

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

To do this, MPI has *persistent communication*:

- You describe the communication with `MPI_Send_init`, which has the same calling sequence as `MPI_Isend`, or `MPI_Recv_init`, which has the same calling sequence as `MPI_Irecv`.
- The actual communication is performed by calling `MPI_Start`, for a single request, or `MPI_Startall` for an array or requests.
- Completion of the communication is confirmed with `MPI_Wait` or similar routines as you have seen in the explanation of non-blocking communication.
- The wait call does not release the request object: that is done with `MPI_Request_free`.

2.3.6 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.

2.3.6.1 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

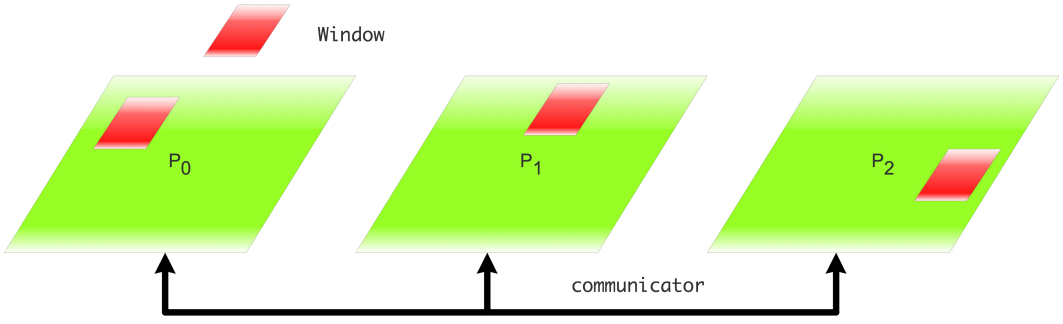


Figure 2.9: Collective definition of a window for one-sided data access

a memory area. Routine for creating and releasing windows are collective, so each process *has* to call them; see figure 2.9.

```
MPI_Info info;
MPI_Win window;
MPI_Win_create( /* memory area */, 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.

The MPI specification allows that the memory of a window can be separate from the regular program memory. The routine `MPI_Alloc_mem` can return a pointer to such priviliged memory.

```
MPI_Info info ;
int error ;
error = MPI_Info_create ( & info ) ;
error = MPI_Info_set ( info , "no_locks" , "true" ) ;
/* Use the info object*/
error = MPI_Info_free ( info ) ;
```

2.3.6.2 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.

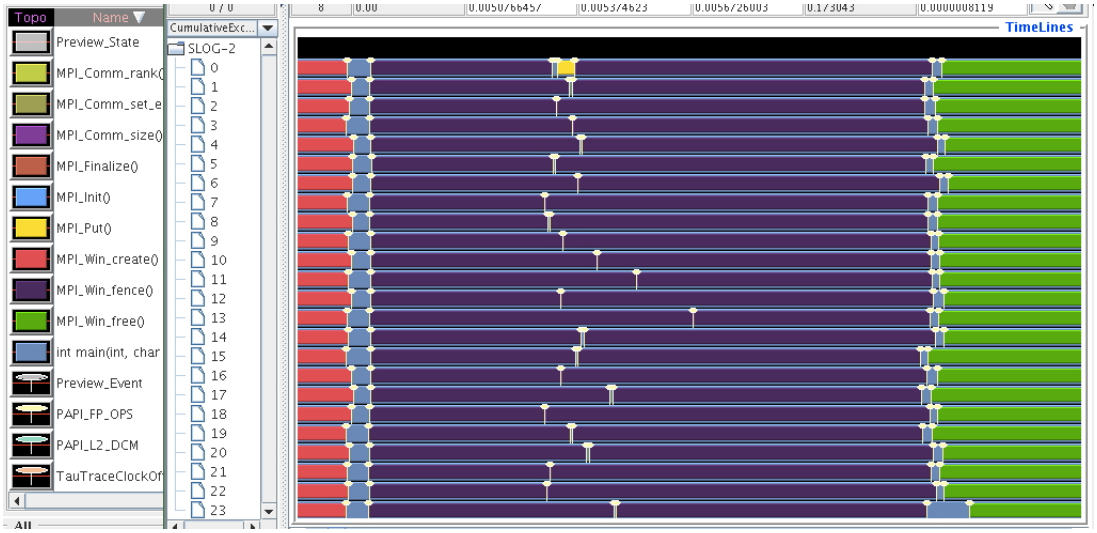


Figure 2.10: A trace of a one-sided communication epoch where process zero only originates a one-sided transfer

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.

2.3.6.3 *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 2.5. 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.

2.3.6.4 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);
}
```

2.3.6.5 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.

2.3.6.6 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_EXCLUSIVE, 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*.

The above mechanism is of limited use. Suppose processor zero has a data structure `work_table` with items that need to be processed. A counter `first_work` keeps track of the lowest numbered item that still needs processing. You can imagine the following *master-worker* scenario:

- Each process connects to the master,
- inspects the `first_work` variable,
- retrieves the corresponding work item, and
- increments the `first_work` variable.

It is important here to avoid a *race condition* (see section HPSC-2.6.1.5) that would result from a second process reading the `first_work` variable before the first process could have updated it. Therefore, the reading and updating needs to be an *atomic operation*.

Unfortunately, you can not have a put and get call in the same access epoch. For this reason, MPI version 3 has added certain atomic operations, such as `MPI_Fetch_and_op`.

2.3.6.7 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.

2.3.6.8 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.

1. For more on this subject, see [6].

Exercise 2.6. 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 2.7. 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.

2.3.6.9 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.

2.3.7 Remaining topics in point-to-point communication

2.3.7.1 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 2.3.3):

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

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.
- 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);
```

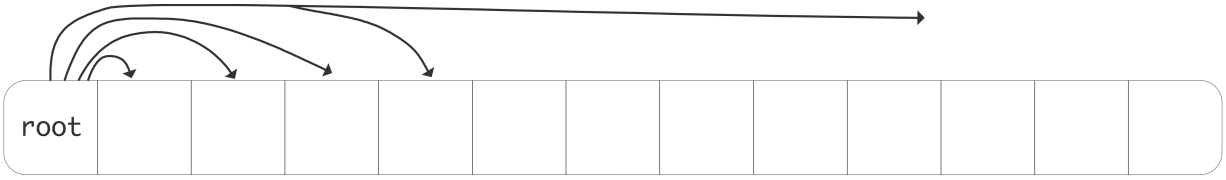


Figure 2.11: A simple broadcast

The root is the process that is sending its data; see figure 2.11. Typically, it will be the root of a 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 2.4.8.)

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 2.11 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 2.12. How does the communication time now depend on the number of processors?

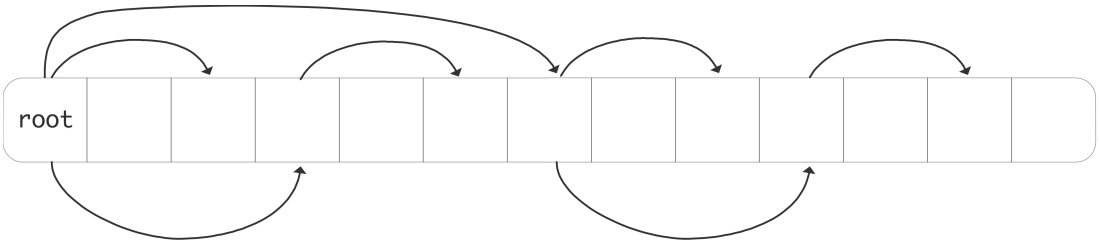


Figure 2.12: A tree-based broadcast

The theory of the complexity of collectives is described in more detail in HPSC-6.1; see also [1].

The reverse of a broadcast is a reduction:

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

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

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 2.4.7.

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`; see section **??**

2.4.3 User-defined reductions

The reference for the commands introduced here can be found in section 3.7.7.

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

2.4.4 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

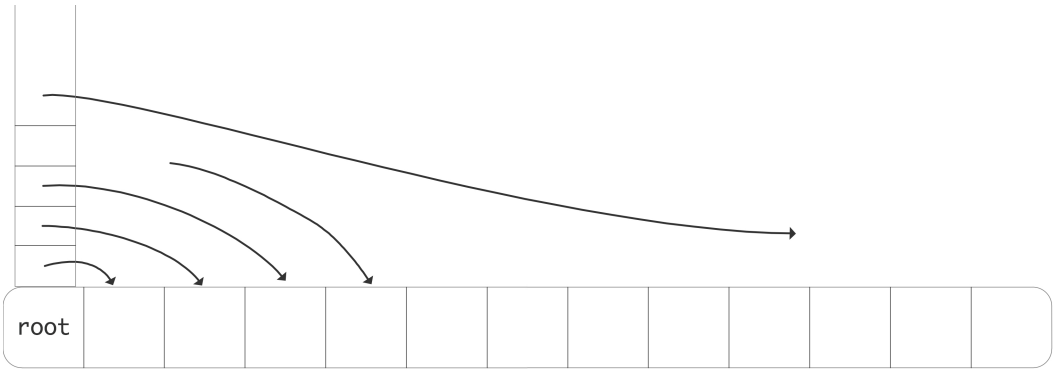


Figure 2.13: A scatter operation

individual item for each receiving process; see figure 2.13.

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.

2.4.5 Variable-size-input collectives

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

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.

2.4.6 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-6.1.

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-6.5.1 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`.

2.4.7 '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-6.1 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 2.4.5.

2.4.8 Non-blocking collectives

The reference for the commands introduced here can be found in section 3.7.8.

Above you have seen how the 'Isend' and 'Irecv' routines can overlap communication with computation. This is not possible with the collectives you have seen so far: they act like blocking sends or receives. However, there are also *non-blocking collectives*. These have roughly the same calling sequence as their blocking counterparts, except that they output an `MPI_Request`. You can then use an `MPI_Wait` call to make sure the collective has completed.

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

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

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

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

This can also be used for 3D FFT operations [3]. Occasionally, a non-blocking collective can be used for non-obvious purposes, such as the `MPI_Ibarrier` in [4].

2.4.9 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.

2.4.10 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

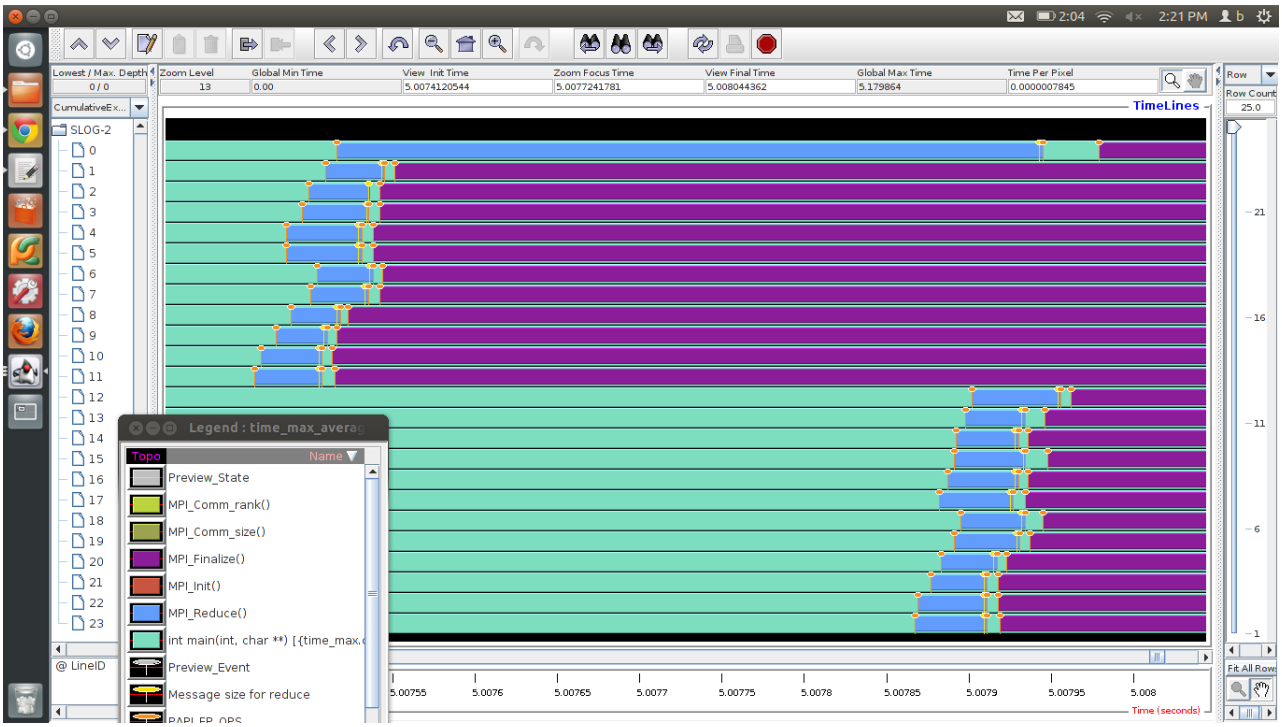


Figure 2.14: Trace of a reduction operation between two dual-socket 12-core nodes

operation the root may have to wait for other processors. This is illustrated in figure 2.14, 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 2.15.

Figure 2.15: 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 2.16.

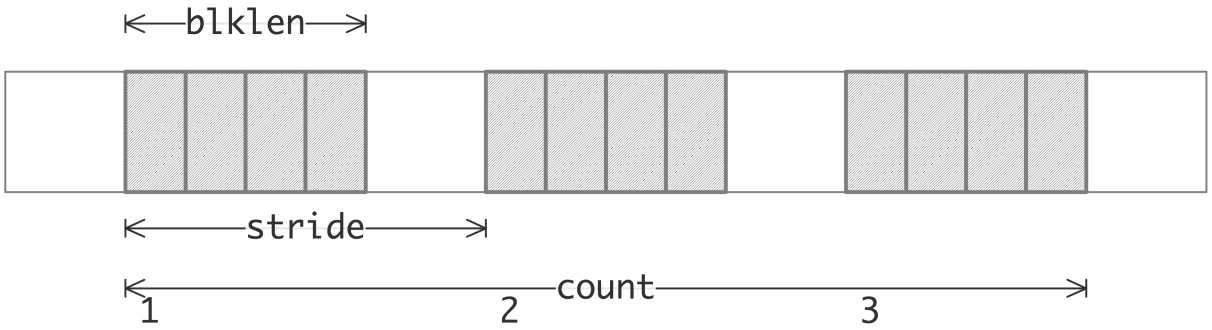


Figure 2.16: A vector datatype is built up out of strided blocks of elements of a constituent type

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-34.2). 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 2.17.

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.

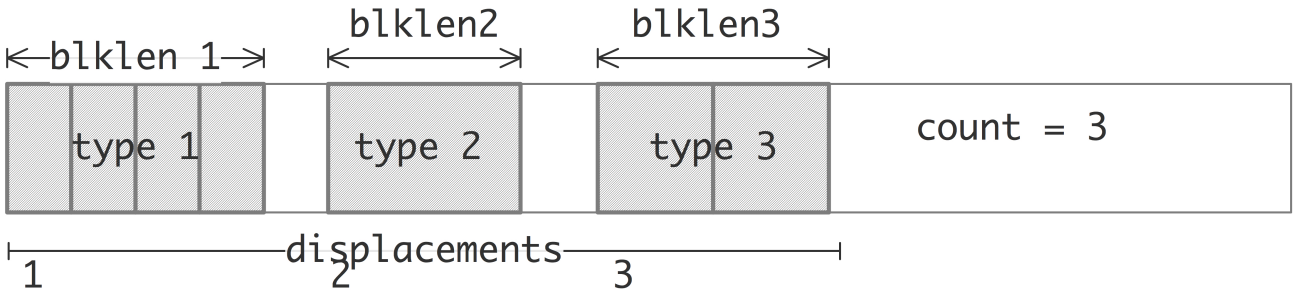


Figure 2.17: The elements of an MPI Struct datatype

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,
```

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 2.9.4).

2.6.5 Process topologies

The reference for the commands introduced here can be found in section 3.9.3.

In the communicators you have seen so far, processes are linearly ordered. In some circumstances the problem you are coding has some structure, and expressing the program in terms of that structure would be convenient. For this purpose, MPI can define a virtual *topology*. There are two types:

- regular, Cartesian, grids; and
- general graphs.

2.6.5.1 Cartesian grid topology

The reference for the commands introduced here can be found in section 3.9.3.1.

A *Cartesian grid* is a structure, typically in 2 or 3 dimensions, of points that have two neighbours in each of the dimensions. Thus, if a Cartesian grid has sizes $K \times M \times N$, its points have coordinates (k, m, n) with $0 \leq k < K$ et cetera. Most points have six neighbours $(k \pm 1, m, n)$, $(k, m \pm 1, n)$, $(k, m, n \pm 1)$; the exception are the edge points. A grid where edge processors are connected through *wraparound connections* is called a *periodic grid*.

The most common use of Cartesian coordinates is to find the rank of process by referring to it in grid terms. For instance, one could ask 'what are my neighbours offset by $(1, 0, 0)$, $(-1, 0, 0)$, $(0, 1, 0)$ et cetera'.

2.7 Synchronization

MPI programs conform to the SPMD model, and this means that events in one process can be unrelated in time to events in another process. Any *synchronization* that happens is induced by communication and other MPI mechanisms. By synchronization here we mean any sort of temporal ordering of events in different processes.

You have already seen some mechanisms.

1. In blocking communication, the receive call does not return until the send call has completed.
2. In non-blocking communication, the wait on a receive request will not return until the send has been completed.
3. In one-sided communication, the fence mechanism impose a certain ordering on events.

Another synchronization mechanism is induced by the *barrier* mechanism. However, while an `MPI_Barrier` call guarantees that all processes have reached a certain location in their source, this does not necessarily imply anything about message traffic. Consider this example

Proc 0	Proc 1	Proc 2
Isend to 1	Irecv from any source	
Barrier	Barrier	Barrier
Wait for send request	wait for recv request	Isend to 1
	(another wildcard recv)	wait for send request

The unexpected behaviour here is that the (first) receive on process 1 can be matched with the send on process 2: the barrier on process 1 only guarantees that the receive instruction was performed, not the actual transfer. For that you need the `MPI_Wait` call, which is after the barrier.

2.8 Hybrid programming: MPI and threads

The reference for the commands introduced here can be found in section 3.13.

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.

2.9 Leftover topics

2.9.1 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.

2.9.1.1 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.

2.9.1.2 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.

2.9.2 Error handling

The reference for the commands introduced here can be found in section 3.11.

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]:

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

Another simple possibility is to specify

3. The routine `MPI_Errhandler_set` is deprecated.

```
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.

2.9.3 Fortran issues

The reference for the commands introduced here can be found in section 3.10.2.

- Fortran routines have the same prototype as C routines except for the addition of an integer error parameter.
- The call for `MPI_Init in Fortran` does not have the commandline arguments; they need to be handled separately.
- The routine `MPI_Sizeof` is only available in Fortran, it provides the functionality of the C/C++ operator `sizeof`.

2.9.4 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.

2.9.5 Timing

The reference for the commands introduced here can be found in section 3.12.1.

Timing of parallel programs is tricky. On each node you can use a timer, typically based on some Operating System (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:

```
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 3.12.1 and modify it to use longer messages. How does the timing increase with message size?

Exercise 2.15. Take the pingpong program of section 3.12.1 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.

2.9.6 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 2.18, normally only the `PMPI` routines show up in the stack trace.

Does the standard mandate this?

2.9.7 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.

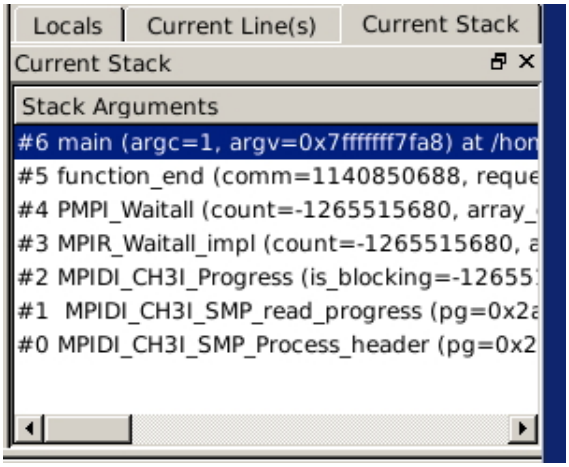


Figure 2.18: A stack trace, showing the `PMPI` calls.

2.9.7.1 *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 7.1 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`.

2.9.7.2 *Large scale debugging*

Check out *ddt* or *TotalView*.

2.9.7.3 *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.

2.9.8 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.

2.9.9 Determinism

MPI processes are only synchronized to a certain extent, so you may wonder what guarantees there are that running a code twice will give the same result. You need to consider two cases: first of all, if the two runs are on different numbers of processors there are already numerical problems; see HPSC-3.3.7.

Let us then limit ourselves to two runs on the same set of processors. In that case, MPI is deterministic as long as you do not use wildcards such as `MPI_ANY_SOURCE`. Formally, MPI messages are 'non-overtaking': two messages between the same sender-receiver pair will arrive in sequence. Actually, they may not arrive in sequence: they are *matched* in sequence in the user program. If the second message is much smaller than the first, it may actually arrive earlier in the lower transport layer.

2.9.10 Progress

Non-blocking communication implies that messages make *progress* while computation is going on. However, communication of this sort can typically not be off-loaded to the network card, so it has to be done by a process. This requires a separate thread of execution, with obvious performance problems. Therefore, in practice overlap may not actually happen, and for the message to make progress it is necessary for the MPI library to become active occasionally. For instance, people have inserted dummy `MPI_Probe` calls.

A similar problem arises with passive target synchronization: it is possible that the origin process may hang until the target process makes an MPI call.

2.10 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. Describe a deadlock scenario involving three processors.
6. True or false: a message sent with `MPI_Isend` from one processor can be received with an `MPI_Recv` call on another processor.
7. True or false: a message sent with `MPI_Send` from one processor can be received with an `MPI_Irecv` on another processor.
8. Why does the `MPI_Irecv` call not have an `MPI_Status` argument?

9. What is the relation between the concepts of ‘origin’, ‘target’, ‘fence’, and ‘window’ in one-sided communication.
10. What are the three routines for one-sided data transfer?
11. Give an example of a collective call with and without a root processor.
12. Given a distributed array, meaning that every processor has

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

    give the approximate MPI-based code that computes the maximum value in the array, and leaves the result on every processor.
13. Give two examples of derived datatypes.
14. Give a practical example where the sender uses a different type to send than the receiver uses in the corresponding receive call. Name the types involved.

2.11 Literature

Online resources:

- MPI 1 Complete reference:
  http://www.netlib.org/utk/papers/mpi-book/mpi-book.html
- Official MPI documents:
  http://www.mpi-forum.org/docs/
- List of all MPI routines:
  http://www.mcs.anl.gov/research/projects/mpi/www/www3/

Tutorial books on MPI:

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

Chapter 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 corresponding Fortran calls are

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

Note that the `MPI_Init` call is one of the few that differs between C and Fortran: the C routine takes the commandline arguments, which Fortran lacks.

Any *commandline argument* to the program 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
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);
```

For Fortran:

```
// init.F90
integer :: input_argument;
character(len=20) :: argstring
if (mytid==0) then
  if ( command_argument_count()==0 ) then
     ! the program is called without parameter
     print *,"Usage: init [0-9]+\n"
     return
```

```
  else
     call get_command_argument(1,argstring) ! test for "-h"
     print *,argstring
     select case(adjustl(argstring))
     case("-h","--help")
        print *,"Usage: init [0-9]+\n"
        return
     case default
        ! parse input argument
        read(argstring,'(i5)') input_argument
     end select
  end if
end if
call MPI_Bcast(input_argument,1,MPI_INTEGER,0,comm,err)
```

3.1.2 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.

```
// contiguous.c
MPI_Datatype newvectortype;
if (mytid==sender) {
  MPI_Type_contiguous(count,MPI_DOUBLE,&newvectortype);
  MPI_Type_commit(&newvectortype);
  MPI_Send(source,1,newvectortype,receiver,0,comm);
  MPI_Type_free(&newvectortype);
} else if (mytid==receiver) {
  MPI_Status recv_status;
  int recv_count;
  MPI_Recv(target,count,MPI_DOUBLE,sender,0,comm,
    &recv_status);
  MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
  ASSERT(count==recv_count);
}
```

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(int));
blocklengths = (int*) malloc(count*sizeof(int));
source = (int*) malloc(totalcount*sizeof(int));
target = (int*) malloc(count*sizeof(int));
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,&newstruct
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`.

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

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

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++) {
    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);
}
```

You can precompute the size of the required buffer as follows:

```
int MPI_Pack_size(
  int incount, MPI_Datatype datatype,
  MPI_Comm comm, int *size);
```

Add one time `MPI_BSEND_OVERHEAD`.

3.3 Blocking communication

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

The basic send command is `MPI_Send`:

```
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 `MPI_Recv`:

```
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 received message can be determined from the status object. See section 3.3.1 for more about the status object.

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);
}

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

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 2.3.2.2.

Any time you receive data, there can be an `MPI_Status` object describing the data that was received. This can be necessary if you use `MPI_ANY_SOURCE` or `MPI_ANY_TAG` in the description of the receive message. If you are not interested in the status information, you can use the values `MPI_STATUS_IGNORE` or (for `MPI_Waitall`, `MPI_Waitany`) `MPI_STATUSES_IGNORE`.

A receive call has a `count` parameter, but this describes the length of the buffer, not the amount of data expected. That quantity can be retrieved with `MPI_Get_count`.

```
// C:
int MPI_Get_count(MPI_Status *status,MPI_Datatype datatype,
    int *count)
! Fortran:
MPI_Get_count(INTEGER status(MPI_STATUS_SIZE),INTEGER datatype,
    INTEGER count,INTEGER ierror)
```

The status object is returned when the message is received. Thus, with `MPI_Recv` it is returned explicitly, but with `MPI_Irecv` it is returned from the `MPI_Wait...` call.

Fortran note In Fortran the `MPI_Status` object needs to be explicitly created:

```
integer status(MPI_STATUS_SIZE)
```

In section 2.3.2.2 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++) {
    err = MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm,
      &status); CHK(err);
    int sender = status.MPI_SOURCE;
    printf("Message from sender=%d: %d\n",
     sender,recv_buffer[p]);
  }
} else {
  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);
  err = MPI_Send(&randomwait,1,MPI_INT, ntids-1,0,comm); CHK(err);
}
```

3.4 Deadlock-free blocking messages

This reference section gives the syntax for routines introduced in section 2.3.2.1.

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. To swap equal-sized buffers you can use `MPI_Sendrecv_replace`.

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

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 2.3.3.

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 an `MPI_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

Fortran note The request parameter is an integer.

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.

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

```
// irecv_source.F90
if (mytid==ntids-1) then
   do p=1,ntids-1
      print *,"post"
      call MPI_Irecv(recv_buffer(p),1,MPI_INTEGER,p-1,0,comm,&
           requests(p),err)
   end do
   do p=1,ntids-1
      call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE,err)
      write(*,'("Message from",i3,":",i5)') index,recv_buffer(index)
   end do
```

3.5.1 Buffered communication

This reference section gives the syntax for routines introduced in section 2.3.4.

`MPI_Buffer_attach`

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

where the size is indicated in bytes. The possible error codes are

- `MPI_SUCCESS` the routine completed successfully.

- `MPI_ERR_BUFFER` The buffer pointer is invalid; this typically means that you have supplied a null pointer.
- `MPI_ERR_INTERN` An internal error in MPI has been detected.

The buffer is detached with `MPI_Buffer_detach`:

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

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

You can compute the needed size of the buffer with `MPI_Pack_size`; see section 3.2.3.

`MPI_Bsend`

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

The asynchronous version is `MPI_Ibsend`.

You can force delivery by

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

3.5.2 Persistent communication

This reference section gives the syntax for routines introduced in section 2.3.5.

The calls `MPI_Send_init` and `MPI_Recv_init` for creating a persistent communication have the same syntax as those for non-blocking sends and receives. The difference is that they do not start an actual communication, they only create the request object.

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

Given these request object, a communication (both send and receive) is then started with `MPI_Start` for a single request or `MPI_Start_all` for multiple requests, given in an array.

```
int MPI_Start(MPI_Request *request)
int MPI_Startall(int count, MPI_Request *array_of_requests)
```

These are equivalent to starting an `Isend` or `Isend`; correspondingly, it is necessary to issue an `MPI_Wait...` call (section 3.5) to determine their completion.

After a request object has been used, possibly multiple times, it can be freed; see 3.5.3.

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

```
// persist.c
if (mytid==src) {
  MPI_Send_init(send,s,MPI_DOUBLE,tgt,0,comm,requests+0);
  MPI_Recv_init(recv,s,MPI_DOUBLE,tgt,0,comm,requests+1);
  printf("Size %d\n",s);
  t[cnt] = MPI_Wtime();
  for (int n=0; n<NEXPERIMENTS; n++) {
MPI_Startall(2,requests);
MPI_Waitall(2,requests,MPI_STATUSES_IGNORE);
  }
  t[cnt] = MPI_Wtime()-t[cnt];
  MPI_Request_free(requests+0); MPI_Request_free(requests+1);
} else if (mytid==tgt) {
  for (int n=0; n<NEXPERIMENTS; n++) {
MPI_Recv(recv,s,MPI_DOUBLE,src,0,comm,MPI_STATUS_IGNORE);
MPI_Send(recv,s,MPI_DOUBLE,src,0,comm);
  }
}
```

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

3.5.3 About `MPI_Request`

An `MPI_Request` object is not actually an object, unlike `MPI_Status`. Instead it is an (opaque) pointer. This meeans that when you call, for instance, `MPI_Irecv`, MPI will allocate an actual request object, and return its address in the `MPI_Request` variable.

Correspondingly, calls to `MPI_Wait...` or `MPI_Test` free this object. If your application is such that you do not use 'wait' call, you can free the request object explicitly with `MPI_Request_free`.

```
int MPI_Request_free(MPI_Request *request)
```

You can inspect the status of a request without freeing the request object with `MPI_Request_get_status`:

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

```
);
```

3.6 One-sided communication

This reference section gives the syntax for routines introduced in section 2.3.6.

3.6.1 Windows and epochs

This reference section gives the syntax for routines introduced in section 2.3.6.1.

C syntax for `MPI_Win_create`

```
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` or `static const`.

The size parameter is measured in bytes. In C this is easily done with the `sizeof` operator; for doing this calculation in Fortran, see section 3.10.2.3.

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`.

`MPI_Alloc_mem`

```
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
```

3.6.2 Remote memory access

This reference section gives the syntax for routines introduced in section 2.3.6.3.

C interface to :

```
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)
```

Fortran interface

```
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
    target_disp, target_count, target_datatype, win, ierror)

<type> :: origin_addr(*)
INTEGER(KIND=MPI_ADDRESS_KIND) :: target_disp
INTEGER :: origin_count, origin_datatype,
    target_rank, target_count, target_datatype,
    win, ierror
```

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(&window_data,2*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,1,    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 2.3.6.2.

```
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 2.3.6.5.

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.

You start and complete an *exposure epoch* with :

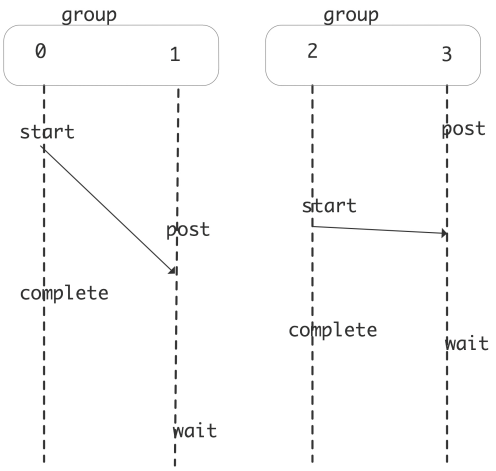


Figure 3.1: Window locking calls in fine-grained active target synchronization

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

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

You start and complete an *access epoch* with :

```
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 2.3.6.6.

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

The call atomically retrieves an item from the window indicated, and replaces the item on the target by doing an accumulate on it with the data on the origin.

```
int MPI_Fetch_and_op(const void *origin_addr, void *result_addr,
        MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,
        MPI_Op op, MPI_Win win)
```

```
// passive.cxx
if (mytid==repository) {
  // Processor zero creates a table of inputs
  // and associates that with the window
  inputs = new float[ninputs];
  for (int i=0; i<ninputs; i++)
    inputs[i] = .01 * rand() / (float)RAND_MAX ;
  MPI_Win_create(inputs,ninputs*sizeof(float),sizeof(float),
   MPI_INFO_NULL,comm,&the_window);
} else {
  // for all other processors the window is null
  MPI_Win_create(NULL,0,sizeof(int),
   MPI_INFO_NULL,comm,&the_window);
}
for (int i=0; i<ninputs; i++)
  myjobs[i] = 0;
if (mytid!=repository) {
  float contribution=(float)mytid,table_element;
  int loc=0;
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE,repository,0,the_window);
    // read the table element by getting the result from adding zero
    err = MPI_Fetch_and_op(&contribution,&table_element,MPI_FLOAT,
                  repository,loc,MPI_SUM,the_window); CHK(err);
    MPI_Win_unlock(repository,the_window);
}
```

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 mynumber,result,*sendbuf,*recvbuf;
mynumber = (float) mytid;
int target_proc = ntids-1;
// add all the random variables together
if (mytid==target_proc) {
```

```
  sendbuf = (float*)MPI_IN_PLACE; recvbuf = &result;
  result = mynumber;
} else {
  sendbuf = &mynumber;    recvbuf = NULL;
}
MPI_Reduce(sendbuf,recvbuf,1,MPI_FLOAT,MPI_SUM,
           target_proc,comm);
// the result should be ntids*(ntids-1)/2:
if (mytid==target_proc)
  printf("Result %6.3f compared to n(n-1)/2=%5.2f\n",
         result,ntids*(ntids-1)/2.);
```

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

```
// reduceinplace.F90
 call random_number(mynumber)
 target_proc = ntids-1;
 ! add all the random variables together
 if (mytid.eq.target_proc) then
    result = mytid
    call MPI_Reduce(MPI_IN_PLACE,result,1,MPI_REAL,MPI_SUM,&
         target_proc,comm,err)
 else
    mynumber = mytid
    call MPI_Reduce(mynumber,result,1,MPI_REAL,MPI_SUM,&
         target_proc,comm,err)
 end if
 ! the result should be ntids*(ntids-1)/2:
 if (mytid.eq.target_proc) then
   write(*,'("Result ",f5.2," compared to n(n-1)/2=",f5.2)') &
        result,ntids*(ntids-1)/2.
end if
```

3.7.2 Gather and scatter

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

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 2.4.6.

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 2.4.7.

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 2.4.5.

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);
```

3.7.7 User-defined reductions

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

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

3.7.8 Non-blocking collectives

This reference section gives the syntax for routines introduced in section 2.4.8.

The same calling sequence as the blocking counterpart, except for the addition of an `MPI_Request` parameter. For instance `MPI_Ibcast`:

```
int MPI_Ibcast(
  void *buffer, int count, MPI_Datatype datatype,
  int root, MPI_Comm comm,
  MPI_Request *request)
```

3.8 Cancelling messages

In section 2.3.2.2 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 2.9.9 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])); C
  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.F90
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);
```

3.9.3 Process topologies

This reference section gives the syntax for routines introduced in section 2.6.5.

3.9.3.1 Cartesian grid topology

This reference section gives the syntax for routines introduced in section 2.6.5.1.

The cartesian topology is specified by giving `MPI_Cart_create` the sizes of the processor grid along each axis, and whether the grid is periodic along that axis.

```
int MPI_Cart_create(
  MPI_Comm comm_old, int ndims, int *dims, int *periods,
  int reorder, MPI_Comm *comm_cart)
```

Each point in this new communicator has a coordinate and a rank. They can be queried with `MPI_Cart_coord` and `MPI_Cart_rank` respectively.

```
int MPI_Cart_coords(
  MPI_Comm comm, int rank, int maxdims,
  int *coords);
int MPI_Cart_rank(
  MPI_Comm comm, init *coords,
  int *rank);
```

Note that these routines can give the coordinates for any rank, not just for the current process.

```
// cart.c
MPI_Comm comm2d;
ndim = 2; periodic[0] = periodic[1] = 0;
```

```
dimensions[0] = idim; dimensions[1] = jdim;
MPI_Cart_create(comm,ndim,dimensions,periodic,1,&comm2d);
MPI_Cart_coords(comm2d,mytid,ndim,coord_2d);
MPI_Cart_rank(comm2d,coord_2d,&rank_2d);
printf("I am %d: (%d,%d); originally %d\n",rank_2d,coord_2d[0],coord_2d[1],m
```

The `reorder` parameter to `MPI_Cart_create` indicates whether processes can have a rank in the new communicator that is different from in the old one.

Strangely enough you can only shift in one direction, you can not specify a shift vector.

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int displ, int *source,
                   int *dest)
```

If you specify a processor outside the grid the result is `MPI_PROC_NULL`.

3.10 Leftover topics

3.10.1 32-bit size issues

The `size` parameter in MPI routines is defined as an `int`, meaning that it is limited to 32-bit quantities. There are ways around this, such as sending a number of `MPI_Type_contiguous` blocks that add up to more than 2^{31}.

3.10.2 Fortran issues

This reference section gives the syntax for routines introduced in section 2.9.3.

3.10.2.1 Data types

The equivalent of `MPI_Aint in Fortran` is

```
integer(kind=MPI_ADDRESS_KIND) :: winsize
```

3.10.2.2 Type issues

Fortran90 is a strongly typed language, so it is not possible to pass argument by reference to their address, as C/C++ do with the `void*` type for send and receive buffers. In Fortran this is solved by having separate routines for each datatype, and providing an `Interface` block in the MPI module. If you manage to request a version that does not exist, the compiler will display a message like

```
There is no matching specific
subroutine for this generic subroutine call [MPI_Send]
```

3.10.2.3 Byte calculations

Fortran lacks a `sizeof` operator to query the sizes of datatypes. Since sometimes exact byte counts are necessary, for instance in one-sided communication, Fortran can use the `MPI_Sizeof` routine:

```
call MPI_Sizeof(windowdata,window_element_size,ierr)
window_size = window_element_size*500
call MPI_Win_create( windowdata,window_size,window_element_size,... );
```

3.11 Error handling

This reference section gives the syntax for routines introduced in section 2.9.2.

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

3.12 More utility stuff

3.12.1 Timing

This reference section gives the syntax for routines introduced in section 2.9.5.

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

```
// C
double MPI_Wtime(void);
! F
DOUBLE PRECISION MPI_WTIME()
```

which gives the number of seconds from a certain point in the past. (Note the absence of the error parameter in the fortran call.)

```
// 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);
```

3.13 Multi-threading

This reference section gives the syntax for routines introduced in section 2.8.

```
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 The OpenMP model

OpenMP is based on on two concepts: the use of of *threads* and the *fork/join model* of parallelism. For now you can think of a thread as a sort of process: the computer executes a sequence of instructions. The fork/join model says that a thread can split itself ('fork') into a number of threads that are identical copies. At some point these copies go away and the original thread is left ('join'), but while the *team of threads* exists, you have parallelism available to you. The part of the execution between fork and join is known as a *parallel region*.

Figure 4.1 gives a simple picture of this: a thread forks into a team of threads, and these threads themselves can fork again.

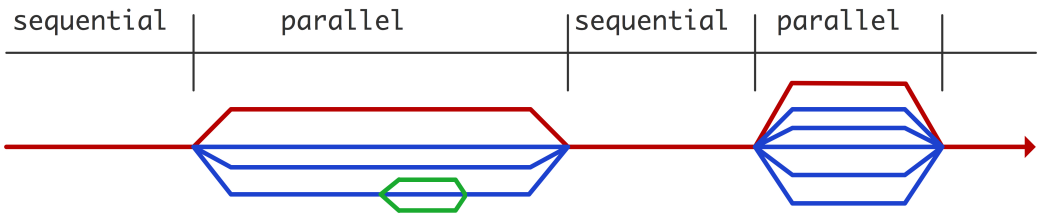


Figure 4.1: Thread creation and deletion during parallel execution

The threads that are forked are all copies of the *master thread*: they have access to all that was computed so far; this is their *shared data*. Of course, if the threads were completely identical the parallelism would be pointless, so they also have private data, and they can identify themselves: they know their thread number. This allows you to do meaningful parallel computations with threads.

This brings us to the third important concept: that of *work sharing* constructs. In a team of threads, initially there will be replicated execution; a work sharing construct divides available parallelism over the threads.

So there you have it: OpenMP uses teams of threads, and inside a parallel region the work is distributed over the threads with a work sharing construct. Threads can access shared data, and they have some private data.

For more on threads, see HPSC-2.6.1.

4.1.2 OpenMP language constructs

Of course, OpenMP is not magic, so you have to tell it when something can be done in parallel. This is mostly done through *directives*, and to a lesser extent through library calls.

The first thing we want to do is create a parallel region. Here is a very simple example:

```
// hello.c
#pragma omp parallel
  {
    printf("Hello world!\n");
  }
```

or in Fortran

```
// hello.F90
!$omp parallel
  print *,"Hello world!"
!$omp end parallel
```

It prints out the 'hello world' message once for each thread. How many threads there are can be determined in a number of ways; we will get to that later.

This code corresponds to the model we just discussed:

- Immediately preceding the parallel block, one thread will be executing the code. In the main program this is the *initial thread*.
- At the start of the block, a new *team of threads* is created, and the thread that was active before the block becomes the *master thread* of that team.
- Each thread in the team then executes the body of the block; it will have access to all variables of the surrounding environment.

4.1.3 Directives

The reference for the commands introduced here can be found in section 5.1.2.

Directives look like *cpp directives*, but they are actually handled by the compiler, not the preprocessor. In C/C++, a directive is followed by a single command or a block in curly braces; in Fortran there is a matching end directive for every opening directive.

4.1.3.1 Parallel regions

The reference for the commands introduced here can be found in section 5.2.1.

The simplest way to create a parallel region is through the `parallel` pragma. In C this is followed by a block, in Fortran you conclude the block with `end parallel`.

```
#pragma omp parallel
{
  // this is executed by all threads
}
```

It would be pointless to have the block be executed identically by all threads, so you will probably use the function `omp_get_thread_num`, to find out which thread you are, and execute work that is individual to that thread. There is also a function `omp_get_num_threads` to find out the total number of threads. Both these functions give a number relative to the current team; recall from figure 4.1 that new teams can be created recursively.

Exercise 4.1. Take the 'hello world' program above, and modify it so that you get multiple messages to you screen, saying

```
Hello from thread 0 out of 4!
Hello from thread 2 out of 4!
```

and so on. What happens if you set your number of threads larger than the available cores on your computer?

Exercise 4.2. What happens if you call `omp_get_thread_num` and `omp_get_num_threads` outside a parallel region?

Exercise 4.3. Test nested parallelism. First of all, set `OMP_NESTED` to `TRUE`. Now write an OpenMP program as follows:

1. Write a subprogram that contains a parallel region.
2. Write a main program with a parallel region; call the subprogram both inside and outside the parallel region.
3. Insert print statements
   (a) in the main program outside the parallel region,
   (b) in the parallel region in the main program,
   (c) in the subprogram outside the parallel region,
   (d) in the parallel region inside the subprogram.

Run your program and count how many print statements of each type you get.

4.1.3.2 *Thread data*

A thread has access to variables of two kinds. Variables on the *heap* are typically created by a call to `malloc` (in C) or `new` (in C++). Threads that are spawned have the same type of access to them.

There are also variables on the *stack*, which have a local *scope*. Thread access to such variables is more complicated, since each thread has its own *context*. Most relevant to our story, the context can contain local copies of variables, that is, each thread has a variable, say `x`, but they are local to each thread context. Thus, one thread setting the value of its instance of `x` has no influence on what another thread sees as the value of its instance of `x`.

To use the technical term, there are

- *shared variables*, where each thread refers to the same data item, and

- *private variables*, where each thread has its own instance.

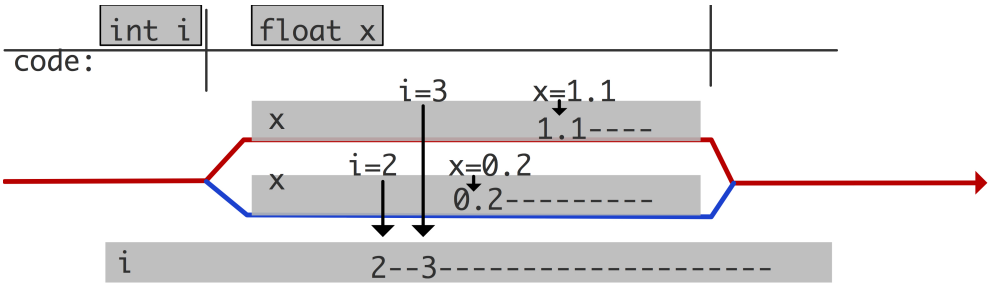


Figure 4.2: Locality of variables in threads

This is illustrated in figure 4.2.

When a new *thread team* is created, there are various mechanisms for indicating which variables are shared and which are private.

- By default, any variable declared in the surrounding scope is shared. Thus, in the following code segment, each thread sets the value of `x`, and its value after the parallel region is indeterminate because you do not know in what sequence the threads performed the update.

```
int x;
#omp parallel
{
  x = omp_get_thread_num();
}
// what is the value of x?
```

- Any variable declared inside the parallel region is private. In the following code segment the variable `x` is local to the thread: the value that is set by one thread does not interfere with what other threads do.

```
#omp parallel
{
  double x;
  x = // some dynamic value
  ... = .... x ....
}
```

  Since the parallel region is also a *lexical scope*, the variable `x` does not exist after the parallel region. (In Fortran this mechanism of local allocation does not exist, so this question does not come up.)
- In a parallel loop, the loop variable is private, even though it may have been declared outside the parallel construct.
- A variable declared in the surrounding scope can explicitly be declared private or shared by adding a `private` or `shared` clause respectively to the OpenMP directive. In the following fragment, each For the behaviour of shared variables, the default, see the point above.

- Unless otherwise stated, private variables are undefined at the start of the parallel region, and disappear at the end; shared variables become undefined after the parallel region.
- Data allocated on the *heap* is shared[1].

4.1.4 OpenMP code structure

The reference for the commands introduced here can be found in section 5.1.3.

OpenMP is largely based on directives: instructions to the compiler and runtime that are hidden in comment-like structures. In Fortran, the directives are actually in a comment:

```
!$OMP construct
```

while in C they are hidden in a *pragma*: something that is normally meant for the *C preprocessor*:

```
#pragma omp construct
```

After the directive, in C you usually find a 'structured block': a single statement or a block in curly braces. In Fortran, the directive is explicitly closed with

```
!$OMP END construct
```

Above, you saw the `parallel` directive: when a thread encounters it, it spawns new threads to form a *thread team*. This team stays active during the execution of the *structured block* that follows, after which only the master thread remains. However, in addition to this lexical description of thread activity, the thread team also stays intact in any code that is dynamically encountered. For instance, any subroutines called the OpenMP *construct* are also executed in parallel.

4.1.5 Some externalities

4.1.5.1 Compiling

Your file or Fortran module needs to contain

```
#include "omp.h"
```

in C, and

```
use omp_lib
```

for Fortran.

OpenMP is handled by extensions to your regular compiler, typically by adding an option to your commandline:

1. However, note that the C and Fortran standards do not actually define a heap. Therefore it is wise to spell out your intentions.

```
# gcc
gcc -o foo foo.c -fopenmp
# Intel compiler
icc -o foo foo.c -openmp
```

If you have separate compile and link stages, you need that option in both.

When you use the openmp compiler option, a *cpp* variable `_OPENMP` will be defined. Thus, you can have conditional compilation by writing

```
#ifdef _OPENMP
   ...
#else
   ...
#endif
```

4.1.5.2 *Running an OpenMP program*

You run an OpenMP program by invoking it the regular way (for instance `./a.out`), but its behaviour is influenced by some *OpenMP environment variables*. The most important one is `OMP_NUM_THREADS`:

```
export OMP_NUM_THREADS=8
```

which sets the number of threads that a program will use. See section 5.7 for a list of all environment variables.

4.2 Creating parallelism

The *fork/join model* of OpenMP means that you need some way of indicating where an activity can be forked for independent execution. There are two ways of doing this:

1. You can declare a parallel region and split one thread into a whole team of threads. We will discuss this next in section 4.2.1. The division of the work over the threads is controlled by *work sharing construct* (section 4.3).
2. Alternatively, you can use tasks and indicating one parallel activity at a time. You will see this in section 4.7.

Note that OpenMP only indicates how much parallelism is present; whether independent activities are in fact executed in parallel is a runtime decision. The factors influencing this are discussed in section 4.2.2.

4.2.1 Parallel regions

The reference for the commands introduced here can be found in section 5.3.

The simplest way to create parallelism in OpenMP is to use the `parallel` pragma. A block preceded by the `omp parallel` pragma is executed by a newly created team of threads. This is an instance of the *Single Program Multiple Data (SPMD)* model: all threads execute the same segment of code. Since it would not be interesting to have all threads do the same calculations, you can use the function `omp_get_thread_num` to distinguish between them.

For instance, if you program computes

```
result = f(x)+g(x)+h(x)
```

you could parallelize this as

```
double result = 0;
#pragma omp parallel
{
  int num = omp_get_thread_num();
  if (num==0)      result += f(x);
  else if (num==1) result += g(x);
  else if (num==2) result += h(x);
}
```

This example shows how the three functions are computed in parallel, but other than that **there are many things wrong with this example**. The rest of this section will be about explaining what is wrong here, and what can be done about it.

4.2.2 Just how many threads?

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

Declaring a parallel region tells OpenMP that a team of threads can be created. The actual size of the team depends on various factors (see section 5.7 for variables and functions mentioned in this section).

- The *environment variable* `OMP_NUM_THREADS` limits the number of threads that can be created.
- If you don't set this variable, you can also set this limit dynamically with the *library routine* `omp_set_num_threads`. This routine takes precedence over the aforementioned environment variable if both are specified.
- A limit on the number of threads can also be set as a clause on a parallel region.

If you specify a greater amount of parallelism than the hardware supports, the runtime system will probably ignore your specification and choose a lower value. To ask how much parallelism is actually used in your parallel region, use `omp_get_num_threads`. To query these hardware limits, use `omp_get_num_procs`.

Another limit on the number of threads is imposed when you use nested parallel regions. This can arise if you have a parallel region in a subprogram which is sometimes called sequentially, sometimes in parallel. The variable `OMP_NESTED` controls whether the inner region will create a team of more than one thread.

4.2.3 Thread data in parallel regions

OpenMP is a programming system for shared memory. This means that you often want all threads to see the same data; we call this 'shared' data. However, sometimes you want a thread to have 'private' data, for instance for intermediate results in a computation.

Private variables act as if they are created at the start of the parallel region. If a variable by that name already exists in the surrounding scope, that variable will become inaccessible. Here is a clear example of a private variable, created inside the parallel region:

```
int x = 5;
#pragma omp parallel
{
  int x = 6;
}
// x is still 5
```

This example is pretty much the same:

```
int x = 5;
#pragma omp parallel private(x)
{
  x = 6;
}
// x is still 5
```

We will discuss all this in detail in section 4.4.

4.3 Work sharing

The reference for the commands introduced here can be found in section 5.4.

The declaration of a *parallel region* establishes a team of threads. This offers the possibility of parallelism, but to actually get meaningful parallel activity you need something more. OpenMP uses the concept of a *work sharing construct*: a way of dividing parallelizable work over a team of threads. The work sharing constructs are:

- `for/do` The threads divide up the loop iterations among themselves; see 4.3.1.
- `sections` The threads divide a fixed number of sections between themselves; see section 4.3.2.
- `single` The section is executed by a single thread; section 4.3.3.
- `task`
- `workshare` Can parallelize Fortran array syntax.

4.3.1 Loop parallelism

The reference for the commands introduced here can be found in section 5.4.1.

The parallel execution of a loop can be handled a number of different ways. For instance, you can create a parallel region around the loop, and adjust the loop bounds:

```
#pragma omp parallel
{
  int threadnum = omp_get_thread_num(),
    numthreads = omp_get_num_threads();
  int low = N*threadnum/numthreads,
    high = N*(threadnum+1)/numthreads;
  for (i=low; i<high; i++)
    // do something with i
}
```

A more natural option is to use the `parallel for` pragma:

```
#pragma omp parallel
#pragma omp for
for (i=0; i<N; i++) {
  // do something with i
}
```

This has several advantages. For one, you don't have to calculate the loop bounds for the threads yourself, but you can also tell OpenMP to assign the loop iterations according to different schedules (section **??**).

Figure 4.3 shows the execution on four threads of

```
#pragma omp parallel
{
  code1();
#pragma omp for
  for (i=1; i<=4*N; i++) {
    code2();
  }
  code3();
}
```

The code before and after the loop is executed identically in each thread; the loop iterations are spread over the four threads.

Instead of having two pragmas, one for the parallel region and one for worksharing over the loop iterations, you can combine the two:

```
#pragma omp parallel for
  for (i=0; .....
```

The loop has to satisfy a number of basic constraints, such as that you can not jump out of it.

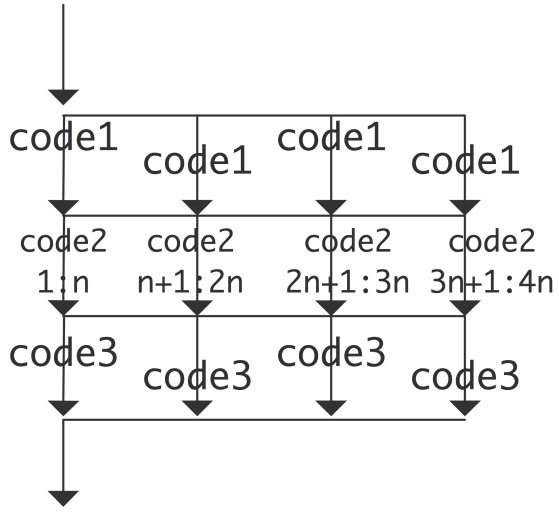


Figure 4.3: Execution of parallel code inside and outside a loop

4.3.1.1 *Loop schedules*

The reference for the commands introduced here can be found in section 5.4.1.1.

Usually you will have many more iterations in a loop than there are threads. Thus, there are several ways you can assign your loop iterations to the threads. OpenMP lets you specify this with the `schedule` clause.

```
#pragma omp for schedule(....)
```

The first distinction we now have to make is between static and dynamic schedules. With static schedules, the iterations are assigned purely based on the number of iterations and the number of threads (and the `chunk` parameter; see later). In dynamic schedules, on the other hand, iterations are assigned to threads that are unoccupied. Dynamic schedules are a good idea if iterations take an unpredictable amount of time, so that *load balancing* is needed.

The default static schedule is to assign one consecutive block of iterations to each thread. If you want different sized blocks you can defined a `chunk` size:

```
#omp pragma for schedule(static[,chunk])
```

(where the square brackets indicate an optional argument). With static scheduling, the compiler will split up the loop iterations at compile time, so, provided the iterations take roughly the same amount of time, this is the most efficient at runtime.

The choice of a chunk size is often a balance between the low overhead of having only a few chunks, versus the load balancing effect of having smaller chunks.

Exercise 4.4. Why is a chunk size of 1 typically a bad idea? (Hint: think about cache lines, and read HPSC-1.4.1.2.)

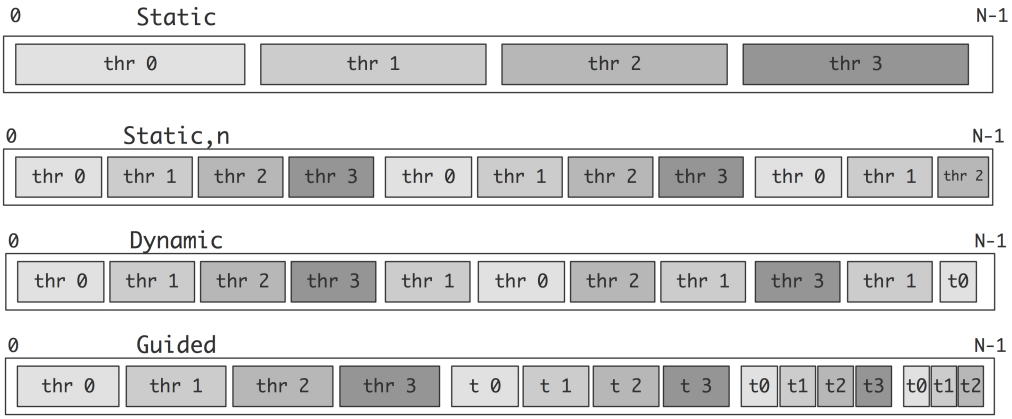


Figure 4.4: Illustration of the loop scheduling strategies

In dynamic scheduling OpenMP will put blocks of iterations (the default chunk size is 1) in a task queue, and the threads take one of these tasks whenever they are finished with the previous.

```
#omp pragma for schedule(static[,chunk])
```

While this schedule may give good load balancing if the iterations take very differing amounts of time to execute, it does carry runtime overhead for managing the queue of iteration tasks.

Finally, there is the `guided` schedule, which gradually decreases the chunk size. The thinking here is that large chunks carry the least overhead, but smaller chunks are better for load balancing. The various schedules are illustrated in figure 4.4.

If you don't want to decide on a schedule in your code, you can specify the `runtime` schedule. The actual schedule will then at runtime be read from the `OMP_SCHEDULE` environment variable. You can even just leave it to the runtime library by specifying `auto`

Exercise 4.5. Program the *LU factorization* algorithm without pivoting.

```
for k=1,n:
  A[k,k] = 1./A[k,k]
  for i=k+1,n:
    A[i,k] = A[i,k]/A[k,k]
    for j=k+1,n:
      A[i,j] = A[i,j] - A[i,k]*A[k,j]
```

1. Argue that it is not possible to parallelize the outer loop.
2. Argue that it is possible to parallelize both the i and j loops.
3. Parallelize the algorithm by focusing on the i loop. Why is the algorithm as given here best for a matrix on row-storage? What would you do if the matrix was on column storage?
4. Argue that with the default schedule, if a row is updated by one thread in one iteration, it may very well be updated by another thread in another. Can you find

a way to schedule loop iterations so that this does not happen? What practical reason is there for doing so?

4.3.1.2 *Reductions*

There is an extended discussion of reductions in section 4.5

4.3.1.3 `nowait`

The implicit barrier at the end of a work sharing construct can be cancelled with a `nowait` clause. This has the effect that threads that are finished can continue with the next code in the parallel region:

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i=0; i<N; i++) { ... }
  // more parallel code
}
```

In the following example, threads that are finished with the first loop can start on the second. Note that this requires both loops to have the same schedule.

```
#pragma omp parallel
{
  x = local_computation()
#pragma omp for nowait
  for (i=0; i<N; i++) {
    x[i] = ...
  }
#pragma omp for
  for (i=0; i<N; i++) {
    y[i] = ... x[i] ...
  }
}
```

4.3.1.4 *While loops*

OpenMP can only handle 'for' loops: *while loops* can not be parallelized. So you have to find a way around that. While loops are for instance used to search through data:

```
while ( a[i]!=0 && i<imax ) {
 i++; }
// now i is the first index for which \n{a[i]} is zero.
```

We replace the while loop by a for loop that examines all locations:

```
result = -1;
#omp parallel for
for (i=0; i<imax; i++) {
  if (a[i]!=0 && result<0) result = i;
}
```

Exercise 4.6. Show that this code has a race condition.

You can fix the race condition by making the condition into a critical section; section 4.6.2.1. In this particular example, with a very small amount of work per iteration, that is likely to be inefficient in this case (why?). A more efficient solution uses the `lastprivate` pragma:

```
result = -1;
#omp parallel for lastprivate(result)
for (i=0; i<imax; i++) {
  if (a[i]!=0) result = i;
}
```

You have now solved a slightly different problem: the result variable contains the *last* location where `a[i]` is zero.

4.3.2 Sections

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

A parallel loop is an example of independent work units that are numbered. If you have a pre-determined number of independent work units, the `sections` is more appropriate. In a `sections` construct can be any number of `section` constructs. These need to be independent, and they can be execute by any available thread in the current team, including having multiple sections done by the same thread.

```
#pragma omp sections
{
#pragma omp section
  // one calculation
#pragma omp section
  // another calculation
}
```

This construct can be used to divide large blocks of independent work. Suppose that in the following line, both `f(x)` and `g(x)` are big calculations:

```
y = f(x) + g(x)
```

You could then write

```
double y1,y2;
#pragma omp sections
```

```
{
#pragma omp section
  y1 = f(x)
#pragma omp section
  y2 = g(x)
}
y = y1+y2;
```

Instead of using two temporaries, you could also use a critical section; see section 4.6.2.1. However, the best solution is have a `reduction` clause on the `sections` directive:

```
  y = f(x) + g(x)
```

You could then write

```
y = 0;
#pragma omp sections reduction(+:y)
{
#pragma omp section
  y += f(x)
#pragma omp section
  y += g(x)
}
```

4.3.3 Single/master

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

`master` `single`

The `single` and `master` pragma limit the execution of a block to a single thread. This can for instance be used to print tracing information or doing *I/O* operations.

```
#pragma omp parallel
{
#pragma omp single
  printf("We are starting this section!\n");
  // parallel stuff
}
```

Another use of `single` is to perform initializations in a parallel region:

```
#pragma omp parallel
{
  int a;
  #pragma omp single
```

```
    a = f(); // some computation
  #pragma omp sections
    // various different computations using a
}
```

The point of the single directive in this last example is that the computation needs to be done only once, because of the shared memory. Since it's a work sharing construct there is an *implicit barrier* after it, which guarantees that all threads have the correct value in their local memory (see section 4.8.4.

Exercise 4.7. What is the difference between this approach and how the same computation would be parallelized in MPI?

The `master` directive, also enforces execution on a single thread, specifically the master thread of the team, but it does not have the synchronization through the implicit barrier.

Exercise 4.8. Modify the above code to read:

```
#pragma omp parallel
{
  int a;
  #pragma omp master
    a = f(); // some computation
  #pragma omp sections
    // various different computations using a
}
```

This code is no longer correct. Explain.

4.3.4 Fortran array syntax parallelization

The reference for the commands introduced here can be found in section 5.4.4.

The `parallel do` directive is used to parallelize loops, and this applies to both C and Fortran. However, Fortran also has implied loops in its *array syntax*. To parallelize array syntax you can use the `workshare` directive.

4.4 Controlling thread data

The reference for the commands introduced here can be found in section 5.5.

In a parallel region there are two types of data: private and shared. In this sections we will see the various way you can control what category your data falls under; for private data items we also discuss how their values relate to shared data.

4.4.1 Shared data

In a parallel region, any data declared outside it will be shared: any thread using a variable `x` will the same memory location associated with that variable.

Example:

```
  int x = 5;
#pragma omp parallel
  {
    x = x+1;
    printf("shared: x is %d\n",x);
  }
```

All threads increment the same variable, so after the loop it will have a value of five plus the number of threads; or maybe less because of the data races involved.

Sometimes this global update is what you want; in other cases the variable is intended only for intermediate results in a computation. In that case there are various ways of creating data that is local to a thread, and therefore invisible to other threads.

4.4.2 Private data

The reference for the commands introduced here can be found in section 5.5.2.

In the C/C++ language it is possible to declare variables inside a *lexical scope*; roughly: inside curly braces. This concept extends to OpenMP parallel regions and directives: any variable declared in a block following an OpenMP directive will be local to the executing thread.

Example:

```
  int x = 5;
#pragma omp parallel
  {
    int x; x = 3;
    printf("local: x is %d\n",x);
  }
```

After the parallel region, the outer variable `x` will still have the value `5`.

The Fortran language does not have this concept of scope, so you have to use a `private` clause:

```
!$OMP parallel private(x)
```

The `private` directive declares data to have a separate copy in the memory of each thread. Such private variables are initialized as they would be in a main program. Any computed value goes away at the end of the parallel region. (However, see below.) Thus, you should not rely on any initial value, or on the value of the outer variable after the region.

```
  int x = 5;
#pragma omp parallel private(x)
  {
    x = x+1; // dangerous
    printf("private: x is %d\n",x);
  }
```

```
printf("after: x is %d\n",x); // also dangerous
```

Private arrays are tricky.

- In C, if an array is statically defined, e.g.,
  ```
  double a[2][5];
  ```
  declaring `private(a)` will indeed put a copy on in each thread. Note that this will lead to an explosion in memory use; in fact, for large arrays you may experience *stack overflow*.
- Dynamically allocated arrays, e.g.,
  ```
  double *a; a = (double*) malloc( some_size );
  ```
  can not be made private with `private(a)`: this only gives each thread a private pointer, but these pointers all point to the same memory location.

4.4.3 Default

As remarked, most data in a parallel section is shared. This default behaviour can be controlled by adding a `default` clause:

```
#pragma omp parallel default(shared) private(x)
{ ... }
#pragma omp parallel default(private) shared(matrix)
{ ... }
```

and if you want to play it safe:

```
#pragma omp parallel default(none) private(x) shared(matrix)
{ ... }
```

Setting `default(none)` is useful for debugging. If your code behaves differently in parallel from sequential there is probably a data race. Specifying the status of every variable is a good way to debug this.

4.4.4 First and last private

You can force initialization with `firstprivate`.

```
  int t=2;
#pragma omp parallel firstprivate(t)
  {
    t += f( omp_get_thread_num() );
    g(t);
  }
```

The variable `t` behaves like a private variable, except that it is initialized to the outside value.

It is possible to preserve a private variable from the last iteration with `lastprivate`:

```
#pragma omp parallel for \
        lastprivate(tmp)
for (i=0; i<N; i+) {
  tmp = ......
  x[i] = .... tmp ....
}
..... tmp ....
```

4.4.5 Persistent data

Most data in OpenMP parallel regions is either inherited from the master thread and therefore shared, or temporary within the scope of the region and fully private. There is also a mechanism for data that is private to a thread, but not limited in lifetime to one parallel region. The `threadprivate` pragma is used to declare that each thread is to have a private copy of a variable:

```
#pragma omp threadprivate(var)
```

This variable will then have the lifetime of an ordinary variable, but inside a parallel region the private copy is used.

4.5 Reductions

The reference for the commands introduced here can be found in section 5.4.1.3.

Parallel tasks often produce some quantity that needs to be summed or otherwise combined. In section 4.2.1 you saw an example, and it was stated that the solution given there was not very good.

The problem in that example was the race condition involving the `result` variable. The simplest solution is to eliminate the race condition by declaring a *critical section*:

```
double result = 0;
#pragma omp parallel
{
  double local_result;
  int num = omp_get_thread_num();
  if (num==0)      local_result = f(x);
  else if (num==1) local_result = g(x);
  else if (num==2) local_result = h(x);
#pragma omp critical
  result += local_result;
}
```

This is a good solution if the amount of serialization in the critical section is small compared to computing the functions f, g, h. On the other hand, you may not want to do that in a loop:

```
double result = 0;
#pragma omp parallel
{
  double local_result;
#pragma omp for
  for (i=0; i<N; i++) {
    local_result = f(x,i);
#pragma omp critical
    result += local_result;
  } // end of for loop
}
```

Exercise 4.9. Can you think of a small modification of this code, that still uses a critical section, that is more efficient? Time both codes.

The easiest way to effect a reduction is of course to the use the `reduction` clause. Adding this to an `omp for` loop has the following effect:

- OpenMP will make a copy of the reduction variable per thread, initialized to the identity of the reduction operator, for instance 1 for multiplication.
- Each thread will then reduce into its local variable;
- At the end of the loop, the local results are combined, again using the reduction operator, into the global variable.

This is one of those cases where the parallel execution can have a slightly different value from the one that is computed sequentially, because floating point operations are not associative. See HPSC-3.3.7 for more explanation.

If your code can not be easily structure as a reduction, you can realize the above scheme by hand by 'duplicating' the global variable and gather the contributions later. This example presumes three threads, and gives each a location of their own to store the result computed on that thread:

```
double result,local_results[3];
#pragma omp parallel
{
  int num = omp_get_thread_num();
  if (num==0)      local_results[num] = f(x)
  else if (num==1) local_results[num] = g(x)
  else if (num==2) local_results[num] = h(x)
}
result = local_results[0]+local_results[1]+local_results[2]
```

While this code is correct, it may be inefficient because of a phenomemon called *false sharing*. Even though the threads write to separate variables, those variables are likely to be on the same *cacheline* (see HPSC-1.4.1.2 for an explanation). This means that the cores will be wasting a lot of time and bandwidth updating each other's copy of this cacheline.

False sharing can be prevent by giving each thread its own cacheline:

```
double result,local_results[3][8];
#pragma omp parallel
{
  int num = omp_get_thread_num();
  if (num==0)       local_results[num][1] = f(x)
// et cetera
}
```

A more elegant solution gives each thread a true local variable, and uses a critical section to sum these, at the very end:

```
double result = 0;
#pragma omp parallel
{
  double local_result;
  local_result = .....
#pragam omp critical
  result += local_result;
}
```

4.6 Synchronization

In the constructs for declaring parallel regions above, you had little control over in what order threads executed the work they were assigned. This section will discuss *synchronization* constructs: ways of telling threads to bring a certain order to the sequence in which they do things.

4.6.1 Barrier

The reference for the commands introduced here can be found in section 5.6.1.1.

A barrier defines a point in the code where all active threads will stop until all threads have arrived at that point. With this, you can guarantee that certain calculations are finished. For instance, in this code snippet, computation of `y` can not proceed until another thread has computed its value of `x`.

```
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  x[mytid] = some_calculation();
  y[mytid] = x[mytid]+x[mytid+1];
}
```

This can be guaranteed with a `barrier` pragma:

```
#omp parallel
{
  int mytid = omp_get_thread_num();
  x[mytid] = some_calculation();
#pragma omp barrier
  y[mytid] = x[mytid]+x[mytid+1];
}
```

4.6.2 Mutual exclusion

Sometimes it is necessary to let only one thread execute a piece of code. Such a piece of code is called a *critical section*, and OpenMP has several mechanisms for realizing this.

The most common use of critical sections is to update a variable. Since updating involves reading the old value, and writing back the new, this has the possibility for a *race condition*: another thread reads the current value before the first can update it; the second thread the updates to the wrong value.

Critical sections are an easy way to turn an existing code into a correct parallel code. However, there are disadvantages to this, and sometimes a more drastic rewrite is called for.

4.6.2.1 `critical` and `atomic`

The reference for the commands introduced here can be found in section 5.6.2.

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

The typical application of a critical section is to update a variable:

```
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  double tmp = some_function(mytid);
#pragma omp critical
  sum += tmp;
}
```

Exercise 4.10. Consider a loop where each iteration updates a variable.

```
#pragma omp parallel for shared(result)
  for ( i ) {
      result += some_function_of(i);
  }
```

Discuss qualitatively the difference between:

- turning the update statement into a critical section, versus

- letting the threads accumulate into a private variable `tmp` as above, and summing these after the loop.

Do an Ahmdal-style quantitative analysis of the first case, assuming that you do n iterations on p threads, and each iteration has a critical section that takes a fraction f.

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

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

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

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

4.6.3 Locks

OpenMP also has the traditional mechanism of a *lock*. A lock is somewhat similar to a critical section: it guarantees that some instructions can only be performed by one process at a time. However, a critical section is indeed about code; a lock is about data. With a lock you make sure that some data elements can only be touched by one process at a time.

One simple example of the use of locks is generation of a *histogram*. A histogram consists of a number of bins, that get updated depending on some data. Here is the basic structure of such a code:

```
int count[100];
float x = some_function();
int ix = (int)x;
if (ix>=100)
  error();
else
  count[ix]++;
```

It would be possible to guard the last line:

```
#pragma omp critical
  count[ix]++;
```

but that is unnecessarily restrictive. If there are enough bins in the histogram, and if the `some_function` takes enough time, there are unlikely to be conflicting writes. The solution then is to create an array of locks, with one lock for each `count` location.

4.6.4 Example: Fibonacci computation

The *Fibonacci sequence* is recursively defined as

$$F(0) = 1, \qquad F(1) = 1, \qquad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2.$$

We start by sketching the basic single-threaded solution. The naive code looks like:

```
int main() {
  value = new int[nmax+1];
  value[0] = 1;
  value[1] = 1;
  fib(10);
}

int fib(int n) {
  int i, j, result;
  if (n>=2) {
    i=fib(n-1); j=fib(n-2);
    value[n] = i+j;
  }
  return value[n];
}
```

Howver, this is inefficienty, since most intermediate values will be computed more than once. We solve this by keeping track of which results are known:

```
  ...
  done = new int[nmax+1];
  for (i=0; i<=nmax; i++)
    done[i] = 0;
  done[0] = 1;
  done[1] = 1;
  ...
int fib(int n) {
  int i, j;
  if (!done[n]) {
    i = fib(n-1); j = fib(n-2);
    value[n] = i+j; done[n] = 1;
  }
  return value[n];
}
```

The OpenMP parallel solution calls for two different ideas. First of all, we parallelize the recursion by using tasks (section 4.7:

```
int fib(int n) {
  int i, j;
  if (n>=2) {
#pragma omp task shared(i) firstprivate(n)
    i=fib(n-1);
```

```
#pragma omp task shared(j) firstprivate(n)
    j=fib(n-2);
#pragma omp taskwait
    value[n] = i+j;
  }
  return value[n];
}
```

This computes the right solution, but, as in the naive single-threaded solution, it recomputes many of the intermediate values.

A naive addition of the `done` array leads to data races, and probably an incorrect solution:

```
int fib(int n) {
  int i, j, result;
  if (!done[n]) {
#pragma omp task shared(i) firstprivate(n)
    i=fib(n-1);
#pragma omp task shared(i) firstprivate(n)
    j=fib(n-2);
#pragma omp taskwait
    value[n] = i+j;
    done[n] = 1;
  }
  return value[n];
}
```

For instance, there is no guarantee that the `done` array is updated later than the `value` array, so a thread can think that `done[n-1]` is true, but `value[n-1]` does not have the right value yet.

One solution to this problem is to use a lock, and make sure that, for a given index `n`, the values `done[n]` and `value[n]` are never touched by more than one thread at a time:

```
int fib(int n)
{
  int i, j;
  omp_set_lock( &(dolock[n]) );
  if (!done[n]) {
#pragma omp task shared(i) firstprivate(n)
    i = fib(n-1);
#pragma omp task shared(j) firstprivate(n)
    j = fib(n-2);
#pragma omp taskwait
    value[n] = i+j;
    done[n] = 1;
  }
```

```
  omp_unset_lock( &(dolock[n]) );
  return value[n];
}
```

This solution is correct, optimally efficient in the sense that it does not recompute anything, and it uses tasks to obtain a parallel execution.

However, the efficiency of this solution is only up to a constant. A lock is still being set, even if a value is already computed and therefore will only be read. This can be solved with a complicated use of critical sections, but we will forego this.

4.7 Tasks

The reference for the commands introduced here can be found in section 5.8.

Tasks are a mechanism that OpenMP uses under the cover: if you specify something as being parallel, OpenMP will create a 'block of work': a section of code plus the data environment in which it occurred. This block is set aside for execution at some later point.

Let's look at a simple example using the `task` directive.

Code	Execution
`x = f();`	the variable `x` gets a value
`#pragma omp task` `{ y = g(x); }`	a task is created with the current value of `x`
`z = h();`	the variable `z` gets a value

The thread that executes this code segment creates a task, which will later be executed, probably by a different thread. The exact execution of the task is up to a *task scheduler*, which operates invisible to the user.

Even though the above segment looks like a linear set of statements, it is impossible to say when the code after the `task` directive will be executed. This means that the following code is incorrect:

```
  x = f();
#pragma omp task
  { y = g(x); }
  z = h(y);
```

When the statement computing `z` is executed, the task computing `y` has only been scheduled; it has not necessarily been executed yet.

In order to have a guarantee that a task is finished, you need the `taskwait` directive. The following creates two tasks, which can be executed in parallel, and then waits for the results:

Code	Execution
`x = f();`	the variable `x` gets a value

`#pragma omp task`
`{ y1 = g1(x); }`
`#pragma omp task`
`{ y2 = g2(x); }`
	two tasks are created with the current value of `x`

`#pragma omp taskwait`	the thread waits until the tasks are finished
`z = h(y1)+h(y2);`	the variable `z` is computed using the task results

The `task` pragma is followed by a structured block. Each time the structured block is encountered, a new task is generated. On the other hand `taskwait` is a standalone directive; the code that follows is just code, it is not a structured block belonging to the directive.

Another aspect of the distinction between generating tasks and executing them: usually the tasks are generated by one thread, but executed by many threads. Thus, the typical idiom is:

```
#pragma omp parallel
#pragma omp single
{
  // code that generates tasks
}
```

This makes it possible to execute loops in parallel that do not have the right kind of iteration structure for a `omp parallel for`. As an example, you could traverse and process a linked list:

```
#pragma omp parallel
#pragma omp single
{
  while (!tail(p)) {
    p = p->next();
#pragma omp task
    process(p)
  }
#pragma omp taskwait
}
```

You can indicate task dependencies in several ways:

1. Using the 'task wait' directive you can explicitly indicate the *join* of the *forked* tasks. The instruction after the wait directive will therefore be dependent on the spawned tasks.
2. The `taskgroup` directive, followed by a structured block, ensures completion of all tasks created in the block, even if recursively created.
3. Each OpenMP task can have a `depend` clause, indicating what *data dependency* of the task. By indicating what data is produced or absorbed by the tasks, the scheduler can construct the dependency graph for you.

4.8 Stuff

4.8.1 Timing

The reference for the commands introduced here can be found in section 5.9.1.

OpenMP has a wall clock timer routine `omp_get_wtime` with resolution `omp_get_wtick`.

Exercise 4.11. Use the timing routines to demonstrate speedup from using multiple threads.

- Write a code segment that takes a measurable amount of time, that is, it should take a multiple of the tick time.
- Write a parallel loop and measure the speedup. You can for instance do this

```
for (int use_threads=1; use_threads<=nthreads; use_threads++) {
#pragma omp parallel for num_threads(use_threads)
    for (int i=0; i<nthreads; i++) {
        .....
    }
    if (use_threads==1)
      time1 = tend-tstart;
    else // compute speedup
```

- In order to prevent the compiler from optimizing your loop away, let the body compute a result and use a reduction to preserve these results.

4.8.2 Dependency analysis

Whether loop iterations can be executed independently depends on relations between data accessed in the iterations. Iterations are of course independent if a data item is read in two different iterations, but if the same item is read in one iteration and written in another, or written in two different iterations, we need to do further analysis.

Analysis of *data dependencies* can be performed by a compiler, but compilers take, of necessity, a conservative approach. This means that iterations may be independent, but can not be recognized as such by a compiler. Therefore, OpenMP shifts this responsibility to the programmer.

The three types of dependencies are:

- flow dependencies, or 'read-after-write';
- anti dependencies, or 'write-after-read'; and
- output dependencies, or 'write-after-write'.

```
for (i) {
  y[i] = t;
  x[i+1] = y[i+1];
  t = x[i];
}
```

4.8.2.1 Flow dependencies

Flow dependencies, or read-afer-write, are not a problem if the read and write occur in the same loop iteration:

```
for (i=0; i<N; i++) {
  x[i] = .... ;
  .... = ... x[i] ... ;
}
```

On the other hand, if the read happens in a later iteration, there is no simple way to parallelize the loop:

```
for (i=0; i<N; i++) {
  .... = ... x[i] ... ;
  x[i+1] = .... ;
}
```

This usually requires rewriting the code.

4.8.2.2 Anti dependencies

The simplest case of write-after-read is a reduction:

```
for (i=0; i<N; i++) {
  t = t + .....
}
```

This can be dealt with by explicit declaring the loop to be a reduction, or to use any of the other strategies in section 4.5.

If the read and write are on an array the situation is more complicated. The iterations in this fragment

```
for (i=0; i<N; i++) {
  x[i] = ... x[i+1] ... ;
}
```

can not be executed in arbitrary order as such. However, conceptually there is no dependency. We can solve this by introducing a temporary array:

```
for (i=0; i<N; i++)
  xtmp[i] = x[i];
for (i=0; i<N; i++) {
  x[i] = ... xtmp[i+1] ... ;
}
```

This is an example of a transformation that a compiler is unlikely to perform, since it can greatly affect the memory demands of the program. Thus, this is left to the programmer.

4.8.2.3 Output dependencies

The case of write-after-write does not occur by itself: if a variable is written twice in sequence without an intervening read, the first write can be removed without changing the meaning of the program. Thus, this case reduces to a flow dependency.

4.8.3 Thread safety

With OpenMP it is relatively easy to take existing code and make it parallel by introducing parallel sections. If you're careful to declare the appropriate variables shared and private, this may work fine. However, your code may include calls to library routines that include a *race condition*; such code is said not to be *thread-safe*.

For example a routine

```
static int isave;
int next_one() {
 int i = isave;
 isave += 1;
 return i;
}

...
for ( .... ) {
  int ivalue = next_one();
}
```

has a clear race condition, as the iterations of the loop may get different `next_one` values, as they are supposed to, or not. This can be solved by using an `critical` pragma for the `next_one` call; another solution is to use an `threadprivate` declaration for `isave`. This is for instance the right solution if the `next_one` routine implements a *random number generator*.

4.8.4 Relaxed memory model

The reference for the commands introduced here can be found in section 5.9.3.

4.8.4.1 Thread synchronization

Let's do a *producer-consumer* model[2]. This can be implemented with sections, where one section, the producer, sets a flag when data is available, and the other, the consumer, waits until the flag is set.

```
#pragma omp parallel sections
{
  // the producer
  #pragma omp section
```

2. This example is from Intel's excellent OMP course by Tim Mattson

```
  {
    ... do some producing work ...
    flag = 1;
  }
  // the consumer
  #pragma omp section
  {
    while (flag==0) { }
    ... do some consuming work ...
  }
}
```

One reason this doesn't work, is that the compiler will see that the flag is never used in the producing section, and that is never changed in the consuming section, so it may optimize these statements, to the point of optimizing them away.

The producer then needs to do:

```
... do some producing work ...
#pragma omp flush
#pragma atomic write
  flag = 1;
#pragma omp flush(flag)
```

and the consumer does:

```
#pragma omp flush(flag)
while (flag==0) {
  #pragma omp flush(flag)
}
#pragma omp flush
```

This code strictly speaking has a *race condition* on the `flag` variable. It is better to use an `atomic` pragma here: the producer has

```
#pragma atomic write
  flag = 1;
```

and the consumer:

```
while (1) {
  #pragma omp flush(flag)
  #pragma omp atomic read
    flag_read = flag
  if (flag_read==1) break;
}
```

4.8.5 Accelerators

In OpenMP 4.0 there is support for offloading work to an *accelerator* or *co-processor*:

```
#pragma omp target [clauses]
```

with clauses such as

- `data`: place data
- `update`: make data consistent between host and device

4.8.6 SIMD

OpenMP 4.0 has a way of indicating that a loop should not be arbitrarily divided over threads, but should be executed over simd lanes:

```
#pragma omp simd [clauses]
```

4.8.7 Overhead costs

Code parallelization ideally divides the running time of your program by the number of parallel processing entities. In practice, the following factors counteract this.

4.8.7.1 Amdahl effects

Any code will have parts that are not parallelizable. Amdahl's law (see HPSC-2.2.3) quantizes the effect this has on parallel speedup. In an OpenMP code, the sections that are executed by a single thread will play the role of the sequential part.

4.8.7.2 Thread overhead

At the start of an OpenMP program, a pool of threads is created. This incurs a one-time overhead that will probably be amortized over the total runtime.

Work sharing constructs act as if they create a new team of threads every time. In practice, the program probably keeps a pool of threads around that are dormant in between parallel sections. This means that there is no thread creation overhead associated with the start of a parallel section.

4.8.7.3 Load balance

On the other hand, at the end of a work sharing construct there is a barrier, so an unbalanced load distribution will decrease the parallel efficiency. If loop iterations are not uniform in their running time, it may pay off to use dynamic rather than static scheduling.

On the other other hand, dynamic scheduling has overhead of its own, since it involves the operating system.

4.8.7.4 *Synchronization*

Various synchronization constructs, such as critical sections, as well as dynamic loop scheduling, are realized through *operating system* functions. These are often quite costly, taking many thousands of cycles. Thus, the *cost of a critical sections* goes far beyond the Amdahl cost of the loss of paralellism. Critical sections should be used only if the parallel work far outweighs it.

4.9 Performance

See the EPCC benchmark suite [5].

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.

The definitive information on OpenMP can be found on http://openmp.org/; more tutorials can be found at http://openmp.org/wp/resources/ where the one by Tim Mattson is particularly recommended.

5.1 Basics

5.1.1 OpenMP setup

If you use OMP library routines, you have to make them known to the compiler. In C/C++ you include the header file *omp.h*:

```
#include "omp.h"
```

In Fortran you use a module:

```
use omp_lib
```

5.1.2 Directives

This reference section gives the syntax for routines introduced in section 4.1.3.

Directives in C/C++ are case-sensitive. Directives can be broken over multiple lines by escaping the line end.

Directives in Fortran start with a *sentinel*, commonly `!$omp`. If you break a directive over more than one line, all but the last line need to have a continuation character, and each line needs to have the sentinel:

```
!$OMP parallel do &
!%OMP   copyin(x),copyout(y)
```

The directives are case-insensitive. In *Fortran fixed-form source* files, `c$omp` and `*$omp` are allowed too.

5.1.3 Code and execution structure

This reference section gives the syntax for routines introduced in section 4.1.4.

Here are a couple of important concepts:

Definition 1

structured block *An OpenMP directive is followed by an* structured block*; in C this is a single statement, a compound statement, or a block in braces; In Fortran it is delimited by the directive and its matching* `end` *directive.*

A structured block can not be jumped into, so it can not start with a labeled statement, or contain a jump statement leaving the block.

construct *An OpenMP* construct *is the section of code starting with a directive and spanning the following structured block, plus in Fortran the end-directive. This is a lexical concept: it contains the statements directly enclosed, and not any subroutines called from them.*

region of code *A* region of code *is defined as all statements that are dynamically encountered while executing the code of an OpenMP construct. This is a dynamic concept: unlike a 'construct', it does include any subroutines that are called from the code in the structured block.*

5.2 Thread stuff

5.2.1 Creating parallel threads

This reference section gives the syntax for routines introduced in section 4.1.3.1.

`parallel`

```
// hellocount.c
#pragma omp parallel
  {
    int mythread,nthreads;
    nthreads = omp_get_num_threads();
    mythread = omp_get_thread_num();
    printf("Hello from %d out of %d\n",mythread,nthreads);
  }
```

If the structure block consists only of a `for`/`do` or `sections` region, you can use the combined directives `parallel for`, `parallel do`, `parallel sections`.

5.3 Parallel regions

This reference section gives the syntax for routines introduced in section 4.2.1.

The `parallel` pragma creates a team of threads from the current thread, and makes each thread execute the following block.

To test whether you are in a parallel region, use `omp_in_parallel`.

```
int omp_in_parallel() // C
LOGICAL omp_in_parallel() ! F
```

The number of threads used can differ between parallel regions. This is known as *dynamic mode*. You can set this with `omp_set_dynamic` and query with `omp_get_dynamic`.

If a region should be executed serially under certain conditions, the `if` clause can be used:

```
#pragma omp parallel if (n>1000)
#pragma omp for
for (i=0; i<n; i++) {
   ...
}
```

5.4 Worksharing

This reference section gives the syntax for routines introduced in section 4.3.

The OpenMP *worksharing constructs* serve to distribute work over threads.

5.4.1 Loop parallelism

This reference section gives the syntax for routines introduced in section 4.3.1.

The `for` (C) and `do` (Fortran) directives tell OpenMP to distribute the iterations of a loop over the threads of a team. These pragmas do not create a team of threads: they take the current team of threads and divide the loop iterations over them. This means that the `omp for` or `omp do` directive needs to be inside a parallel region. It is also possible to have a combined `omp parallel for` or `omp parallel do` directive.

There are some restrictions on the loop: basically, OpenMP needs to be able to determine in advance how many iterations there will be.

- The loop can not contains `break`, `return`, `exit` statements, or `goto` to a label outside the loop.
- The `continue` (C) or `cycle` (F) statement is allowed.
- The index update has to be an increment (or decrement) by a fixed amount.
- The loop index variable is automatically private, and not changes to it inside the loop are allowed.

5.4.1.1 Schedules

This reference section gives the syntax for routines introduced in section 4.3.1.1.

The schedule can be declared explicitly, set at runtime through the `OMP_SCHEDULE` environment variable, or left up to the runtime system by specifying `auto`. Especially in the last two cases you may want to enquire what schedule is currently being used with `omp_get_schedule`.

```
int omp_get_schedule(omp_sched_t * kind, int * modifier );
```

Its mirror call is `omp_set_schedule`, which sets the value that is used when schedule value `runtime` is used. It is in effect equivalent to setting the environment variable `OMP_SCHEDULE`.

```
void omp_set_schedule (omp_sched_t kind, int modifier);
```

Here are the various schedules you can set with the `schedule` clause:

affinity Set by using value `omp_sched_affinity`
auto The schedule is left up to the implementation. Set by using value `omp_sched_auto`
dynamic value: 2. The modifier parameter is the *chunk* size; default 1. Set by using value `omp_sched_dynamic`
guided Value: 3. The modifier parameter is the `chunk` size. Set by using value `omp_sched_guided`
runtime Use the value of the `OMP_SCHEDULE` environment variable. Set by using value `omp_sched_runtime`
static value: 1. The modifier parameter is the *chunk* size. Set by using value `omp_sched_static`

5.4.1.2 Ordered loop iterations

If loop iterations contain code that needs to be executed in the sequence of iterates, the `ordered` clause can be used:

```
#pragma omp for ordered private(t)
for (i=0; i<N; i++) {
  t = F(i) // some expensive calculation
#pragma omp ordered
  a[i+1] = a[i] + t
}
```

Note the separate `ordered` construct inside the loop. Without that, nothing will actually be ordered.

5.4.1.3 Reductions

This reference section gives the syntax for routines introduced in section 4.5.

Many loops are used to compute a reduction: the per-iteration results are combined into a final result, such as a sum or product. You can solve this by using partial results in each thread (or by using an atomic update), but the easiest way is to use the `reduction` clause.

```
  int sum;
#pragma omp parallel for reduction(+:sum)
  for (i=0; i<N; i++)
    sum  = sum + f(i);
```

Exercise 5.1. Write a program to test the fact that the partial results are initialized to the unit of the reduction operator.

Arithmetic reductions: $+, *, -, \max, \min$

Logical operator reductions: `&,&&,|,||,^`

Fortran additionally has max and min.

5.4.2 Master and single

The `master` and `single` constructs are quite similar. They both indicate that a structured block is to be executed by only a single thread, and that all other threads that encounter the block skip it. However, the `master` pragma assigns the execution of the block to a specific thread: the master thread. Therefore, it is technically not a worksharing construct, and thus there is no barrier at the end of it.

The `single` pragma does have a barrier, which can be removed with `nowait`.

```
$! omp parallel
..
$! omp single
..
$! omp end single nowait
..
$! omp end parallel
```

5.4.3 Sections

```
#pragma omp parallel
{
#pragma omp sections
  {
#pragma omp section
    do_thing_A();
#pragma omp section
    do_thing_B();
#pragma omp section
    do_thing_C();
  }
}
```

5.4.4 Fortran `workshare`

This reference section gives the syntax for routines introduced in section 4.3.4.

The `workshare` directive exists only in Fortran. It can be used to parallelize the implied loops in *array syntax*, as well as *forall* loops.

5.5 Controlling thread data

This reference section gives the syntax for routines introduced in section 4.4.

5.5.1 Shared data

Data that existed in the master thread of a team is shared between the team. This is default behaviour. The clause `shared` can be used for completeness, for instance if `default(none)` is declared.

While shared data is readable and writable by every thread, their view of data may not always be consistent. Therefore, reads should be preceded by a `flush` command. Fortunately, in many cases this is done by default; see section 5.9.3.

5.5.2 Private data

This reference section gives the syntax for routines introduced in section 4.4.2.

Data that is declared private with the `private` directive is put on a separate *stack per thread*. The OpenMP standard does not dictate the size of these stacks, but beware of *stack overflow*. A typical default is a few megabyte; you can control it with the environment variable `OMP_STACKSIZE`

`firstprivate` `lastprivate` `copyin`

5.5.3 Defaults

You can add clauses to an `omp parallel` pragma to specify explicitly what variables are private and what variables shared. Note the cases with default behaviour:

- Loop variables in an `omp for` are private;
- Local variables in the parallel region are private.

You can alter this default behaviour with the `default` clause:

- The `none` option is good for debugging, because it forces you to specify for each variable in the parallel region whether it's private or shared.
- The `shared` clause means that any private variables need to be declared explicitly.
- The `private` clause means that any shared variables need to be declared explicitly. This value is not available in C.

5.6 Synchronization

- `atomic` Update of a single memory location. Only certain specified syntax pattterns are supported. This was added in order to be able to use hardware support for atomic updates.
- `barrier`: section 5.6.1
- `critical`: section 5.6.2
- `ordered`
- locks: section 5.6.3
- `flush`: section 5.9.3
- `nowait`

5.6.1 Barriers

A barrier is a location in the code that needs to be reached by all threads before any of them can continue. There is a directive for declaring an explicit barrier, but there are also implicit barriers associated with certain other directives.

5.6.1.1 Explicit barriers

This reference section gives the syntax for routines introduced in section 4.6.1.

You saw an example above where explicit barriers are needed. You can sometimes replace an explicit barrier by an implicit one:

```
#pragma omp parallel
{
  // do something
#pragma omp barrier
  // do something else
}
```

is equivalent to

```
#pragma omp parallel
{
  // do something
}
#pragma omp parallel
  // do something else
}
```

but the second code has more overhead in creating the team of threads a second time.

5.6.1.2 Implicit barriers

At the end of a parallel region the team of threads is dissolved and only the master thread continues. Therefore, there is an *implicit barrier at the end of a parallel region*.

There is some *barrier behaviour* associated with `omp for` loops and other *worksharing constructs* (see section **??**). For instance, there is an *implicit barrier* at the end of the loop. This barrier behaviour can be cancelled with the `nowait` clause.

You will often see the idiom

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i=0; i<N; i++)
    a[i] = // some expression
#pragma omp for
  for (i=0; i<N; i++)
    b[i] = ..... a[i] .....
```

Here the `nowait` clause implies that threads can start on the second loop while other threads are still working on the first. Since the two loops use the same schedule here, an iteration that uses `a[i]` can indeed rely on it that that value has been computed.

5.6.2 Critical sections

This reference section gives the syntax for routines introduced in section 4.6.2.1.

The pragmas `critical` and `atomic` are two ways to indicate that a section of code can only be executed by one thread at a time.

```
#pragma omp critical [(name)] new-line
    structured-block
```

Not required to be in a parallel region?

5.6.3 Locks

Create/destroy:

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

Set and release:

```
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
```

Since the set call is blocking, there is also

```
omp_test_lock();
```

Unsetting a lock needs to be done by the thread that set it.

Lock operations implicitly have a `flush`.

5.7 Internal control variables

OpenMP has a number of settings that can be set through *environment variables*, and both queried and set through *library routines*. These settings are called *Internal Control Variables (ICVs)*: an OpenMP implementation behaves as if there is an internal variable storing this setting.

First, there are 4 ICVs that behave as if each thread has its own copy of them. The default is implementation-defined unless otherwise noted.

- It may be possible to adjust dynamically the number of threads for a parallel region. Variable: `OMP_DYNAMIC`; routines: `omp_set_dynamic`, `omp_get_dynamic`.
- If a code contains *nested parallel regions*, the inner regions may create new teams, or they may be executed by the single thread that encounters them. Variable: `OMP_NESTED`; routines `omp_set_nested`, `omp_get_nested`. Allowed values are `TRUE` and `FALSE`; the default is false.
- The number of threads used for an encountered parallel region can be controlled. Variable: `OMP_NUM_THREADS`; routines `omp_set_num_threads`, `omp_get_max_threads`.
- The schedule for a parallel loop can be set. Variable: `OMP_SCHEDULE`; routines `omp_set_schedule`, `omp_get_schedule`.

Non-obvious syntax:

```
export OMP_SCHEDULE="static,100"
```

Other settings:

- `omp_get_num_threads`: query the number of threads active at the current place in the code; this can be lower than what was set with `omp_set_num_threads`. For a meaningful answer, this should be done in a parallel region.
- `omp_get_thread_num`
- `omp_in_parallel`: test if you are in a parallel region
- `omp_get_num_procs`: query the physical number of cores available.

Other environment variables:

- `OMP_STACKSIZE` controls the amount of space that is allocated as per-thread *stack*; the space for private variables.
- `OMP_WAIT_POLICY` determines the behaviour of threads that wait, for instance for *critical section*:
  - `ACTIVE` puts the thread in a *spin-lock*, where it actively checks whether it can continue;
  - `PASSIVE` puts the thread to sleep until the Operating System (OS) wakes it up.

  The 'active' strategy uses CPU while the thread is waiting; on the other hand, activating it after the wait is instantaneous. With the 'passive' strategy, the thread does not use any CPU while waiting, but activating it again is expensive. Thus, the passive strategy only makes sense if threads will be waiting for a (relatively) long time.
- `OMP_PROC_BIND` with values `TRUE` and `FALSE` can bind threads to a processor. On the one hand, doing so can minimize data movement; on the other hand, it may increase load imbalance.

5.8 Tasks

This reference section gives the syntax for routines introduced in section 4.7.

OpenMP v4 has a `depend` clause.

```
#pragma omp task depend(dependency-type: list)
```

5.9 Stuff

5.9.1 Timing

This reference section gives the syntax for routines introduced in section 4.8.1.

To do *OpenMP timing* you can use any system utility; however there is a dedicated routine `omp_get_wtime` that express the time since some starting point as a double:

```
double omp_get_wtime(void);
```

The starting point is arbitrary and is different for each program run; however, in one run it is identical for all threads.

To measure a time difference:

```
double tstart,tend,duration;
tstart = omp_get_wtime();
// do stuff
tend = omp_get_wtime();
duration = tend-tstart;
```

The timer resolution is given by:

```
double omp_get_wtick(void);
```

5.9.2 Affinity

For performance it can be a good idea to bind threads to specific processors or cores. OpenMP (as of *version 3.1*) has a mechanism for *thread affinity*: `OMP_PROC_BIND`

```
export OMP_PROC_BIND=true
```

Apart from this, compilers can have proprietary mechanism; e.g., for the intel compiler the variable is

```
export KMP_AFFINITY=compact,0
```

for the sun compiler:

```
export SUNW_MP_PROCBIND=TRUE
```

for gcc (pre-openmp 3.1)

```
export GOMP_CPU_AFFINITY=0-63
```

5.9.3 Relaxed memory model

This reference section gives the syntax for routines introduced in section 4.8.4.

`flush`

- There is an implicit flush of all variables at the start and end of a *parallel region*.
- There is a flush at each barrier, whether explicit or implicit, such as at the end of a *work sharing*.
- At entry and exit of a *critical section*
- When a *lock* is set or unset.

PART III

THE REST

Chapter 6

Hybrid computing

6.1 Discussion

Hybrid computing decreases the number of messages.

On the other hand it makes the run more synchronous.

New version of Amdahl: sections that MPI-parallel but not OpenMP-parallel.

Allows overdecomposition on the node.

6.2 Hybrid MPI-plus-threads execution

In hybrid execution, the main question is whether all threads are allowed to make MPI calls. To determine this, replace the `MPI_Init` call by `MPI_Init_thread`:

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

Here the `required` and `provided` parameters can take the following values:

`MPI_THREAD_SINGLE` Only a single thread will execute.

`MPI_THREAD_FUNNELLED` The program may use multiple threads, but only the mean thread will make MPI calls.

`MPI_THREAD_SERIAL` The program may use multiple threads, all of which may make MPI calls, but there will never be simultaneous MPI calls in more than one thread.

`MPI_THREAD_MULTIPLE` Multiple threads may MPI calls, without restrictions.

The *mpirun* program usually propagates *environment variables*, so the value of `OMP_NUM_THREADS` when you call `mpirun` will be seen by each MPI process.

- It is possible to use blocking sends in threads, and let the threads block. This does away with the need for polling.
- You can not send to a thread number.

Chapter 7

Support libraries

ParaMesh

Global Arrays

PETSc

Hdf5 and Silo

PART IV

TUTORIALS

here are some tutorials

7.1 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`.

7.1.1 Step 0: compiling for debug

You often need to recompile your code before you can debug it. A first reason for this is that the binary code typically knows nothing about what variable names corresponded to what memory locations, or what lines in the source to what instructions. In order to make the binary executable know this, you have to include the *symbol table* in it, which is done by adding the `-g` option to the compiler line.

Usually, you also need to lower the *compiler optimization level*: a production code will often be compiled with flags such as `-O2` or `-Xhost` that try to make the code as fast as possible, but for debugging you need to replace this by `-O0` ('oh-zero'). The reason is that higher levels will reorganize your code, making it hard to relate the execution to the source[1].

7.1.2 Invoking `gdb`

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

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

tutorials/gdb/c/hello.c

1. Typically, actual code motion is done by `-O3`, but at level `-O2` the compiler will inline functions and make other simplifications.

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

Program exited normally.
(gdb) quit
%%
```

Important note: the program was compiled with the *debug flag* `-g`. This causes the *symbol table* (that is, the translation from machine address to program variables) and other debug information to be included in the binary. This will make your binary larger than strictly necessary, but it will also make it slower, for instance because the compiler will not perform certain optimizations[2].

To illustrate the presence of the symbol table do

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

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

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

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

tutorials/gdb/c/say.c

```
%% cc -o say -g say.c
%% ./say 2
```

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

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

Program exited normally.
```

7.1.3 Finding errors

Let us now consider some programs with errors.

7.1.3.1 C programs

tutorials/gdb/c/square.c

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

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

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

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

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

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

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

There is still an error in our program:

```
(gdb) run
50000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x000000010000f000
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.
```

7.1.3.2 Fortran programs

Compile and run the following program:

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

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

Program received signal EXC_BAD_INSTRUCTION, Illegal instruction/operand.
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.

7.1.4 Memory debugging with Valgrind

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

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

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

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

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

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

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

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

7.1.5 Stepping through a program

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

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

```
%% ./roots
sum: nan
```

Start it in gdb as follows:

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

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

Here you have done the following:

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

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

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

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

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

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

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

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

7.1.6 Inspecting values

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

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

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

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

or set the condition when you define the breakpoint:

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

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

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

7.1.7 Parallel debugging

Debugging parallel programs is harder than than sequential programs, because every sequential bug may show up, plus a number of new types, caused by the interaction of the various processes.

Here are a few possible parallel bugs:

- Processes can *deadlock* because they are waiting for a message that never comes. This typically happens with blocking send/receive calls due to an error in program logic.
- If an incoming message is unexpectedly larger than anticipated, a memory error can occur.

- A collective call will hang if somehow one of the processes does not call the routine.

There are few low-budget solutions to parallel debugging. The main one is to create an xterm for each process. We will describe this next. There are also commercial packages such as *DDT* and *TotalView*, that offer a GUI. They are very convenient but also expensive. The *Eclipse* project has a parallel package, *Eclipse PTP*, that includes a graphic debugger.

7.1.7.1 MPI debugging with gdb

You can not run parallel programs in gdb, but you can start multiple gdb processes that behave just like MPI processes! The command

```
mpirun -np <NP> xterm -e gdb ./program
```

create a number of `xterm` windows, each of which execute the commandline `gdb ./program`. And because these xterms have been started with `mpirun`, they actually form a communicator.

7.1.7.2 Full-screen parallel debugging with DDT

In this tutorial you will run and diagnose a few incorrect MPI programs using DDT. You can start a session with `ddt yourprogram &`, or use `File > New Session > Run` to specify a program name, and possibly parameters. In both cases you get a dialog where you can specify program parameters. It is also important to check the following:

- You can specify the number of cores here;
- It is usually a good idea to turn on memory checking;
- Make sure you specify the right MPI.

When DDT opens on your main program, it halts at the `MPI_Init` statement, and need to press the forward arrow, top left of the main window.

7.1.7.2.1 ***Problem1*** This program has every process independently generate random numbers, and if the number meets a certain condition, stops execution. There is no problem with this code as such, so let's suppose you simply want to monitor its execution.

- Compile `abort.c`. Don't forget about the `-g -O0` flags; if you use the makefile they are included automatically.
- Run the program with DDT, you'll see that it concludes succesfully.
- Set a breakpoint at the Finalize statement in the subroutine, by clicking to the left of the line number. Now if you run the program you'll get a message that all processes are stopped at a breakpoint. Pause the execution.
- The 'Stacks' tab will tell you that all processes are the same point in the code, but they are not in fact in the same iteration.
- You can for instance use the 'Input/Output' tabs to see what every process has been doing.
- Alternatively, use the variables pane on the right to examine the `it` variable. You can do that for individual processes, but you can also control click on the `it` variable and choose `View as Array`. Set up the display as a one-dimensional array and check the iteration numbers.

- Activate the barrier statement and rerun the code. Make sure you have no breakpoints. Reason that the code will not complete, but just hang.
- Hit the general Pause button. Now what difference do you see in the 'Stacks' tab?

7.1.7.2.2 ***Problem2*** Compile `problem1.c` and run it in DDT. You'll get a dialog warning about an abort.

- Pause the program in the dialog. Notice that only the root process is paused. If you want to inspect other processes, press the general pause button. Do this.
- In the bottom panel click on `Stacks`. This gives you the 'call stack', which tells you what the processes were doing when you paused them. Where is the root process in the execution? Where are the others?
- From the call stack it is clear what the error was. Fix it and rerun with `File > Restart Session`.

7.1.7.2.3 ***Problem2***

7.1.8 Further reading

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

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

7.2 Tracing

7.2.1 TAU profiling and tracing

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

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

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

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

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

To use TAU, do `module load tau`. You have to set the environment variable `TAU_TRACE` to 1; it's advisable to set `TRACEDIR` to some directory for all the TAU output. Likewise set `TAU_PROFILE` to 1 and set `PROFILEDIR`.

PART V

PROJECTS, INDEX

Chapter 8

Class projects

8.1 A Style Guide to Project Submissions

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

Structure of your writeup Most of the 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-35). 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 7.2.

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 Warmup 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 Trace output

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

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

8.2.3 Collectives

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

For each exercise, submit code, a TAU trace, and an analysis of what you see in the traces. Submit your work by leaving a code, graphics, and a writeup in your repository.

8.2.4 Linear arrays of processors

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

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

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

```
// 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);
}
```

8.2.4.1 *Coding with blocking calls*

Passing data to a neighbouring processor should be a very parallel operation. However, if we code this naively, with `MPI_Send` and `MPI_Recv`, we get an unexpected serial behaviour, as was explained in section 2.3.2.

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

```
MPI_Recv( /* data: */ &other_number,1,MPI_INT,
          /* from: */ mytid-1, 0, comm, &status);
```

(Note that this uses an `Ssend`; see section 2.3.7.1 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`.

8.2.4.2 A better blocking solution

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

Exercise 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.

8.2.4.3 Non-blocking calls

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

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

8.2.4.4 One-sided communication

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

Exercise 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.

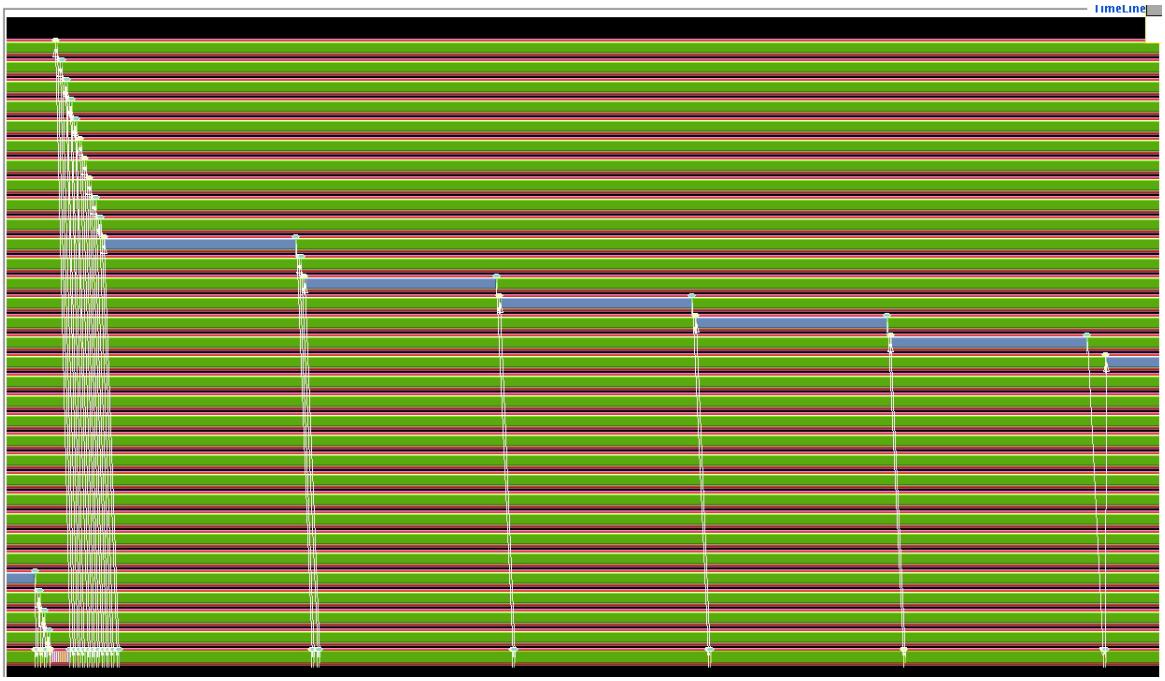


Figure 8.1: Trace of a serial Mandelbrot calculation

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.

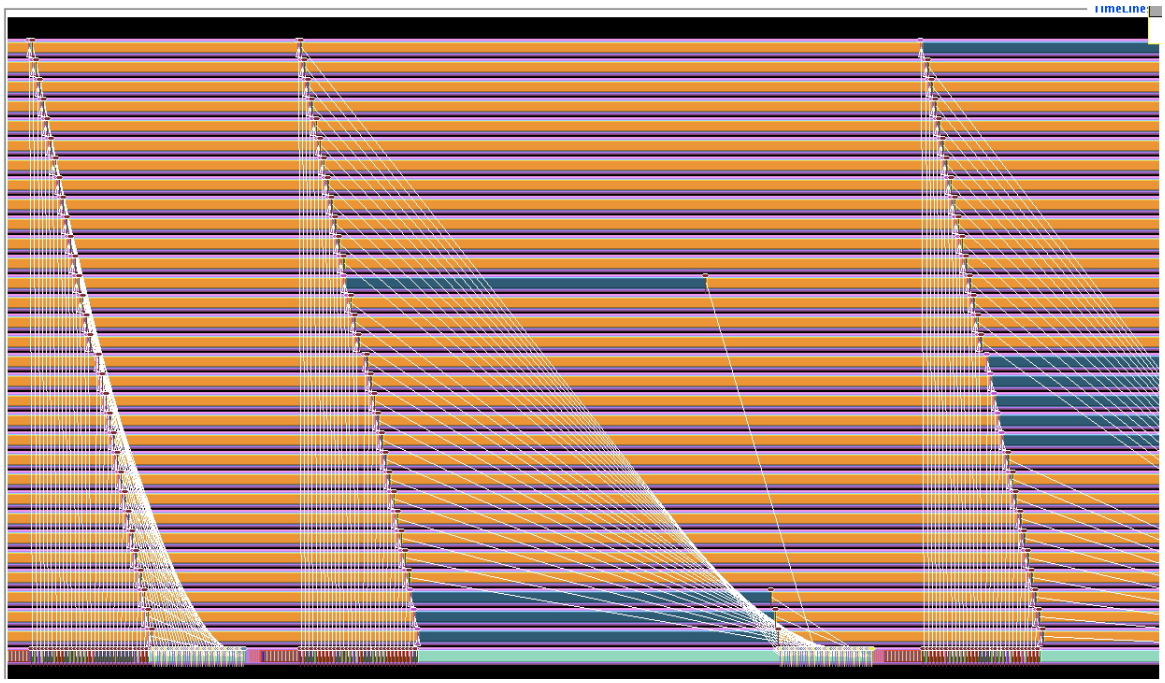


Figure 8.2: Trace of a bulk Mandelbrot calculation

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

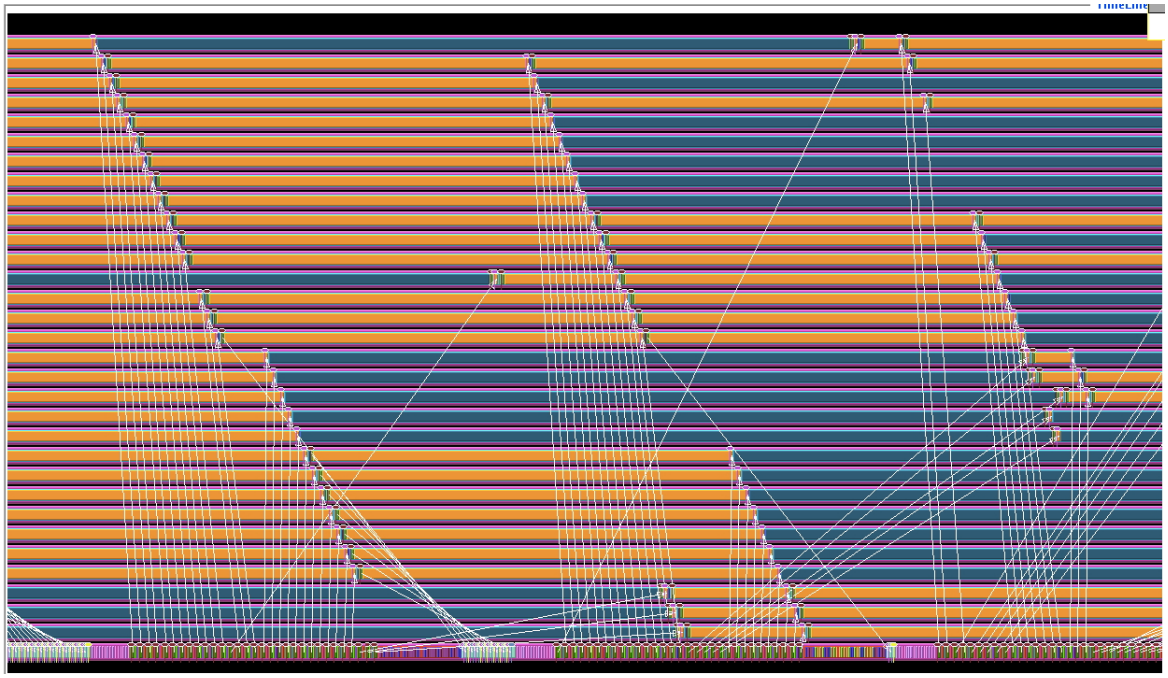


Figure 8.3: Trace of an asynchronous Mandelbrot calculation

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.

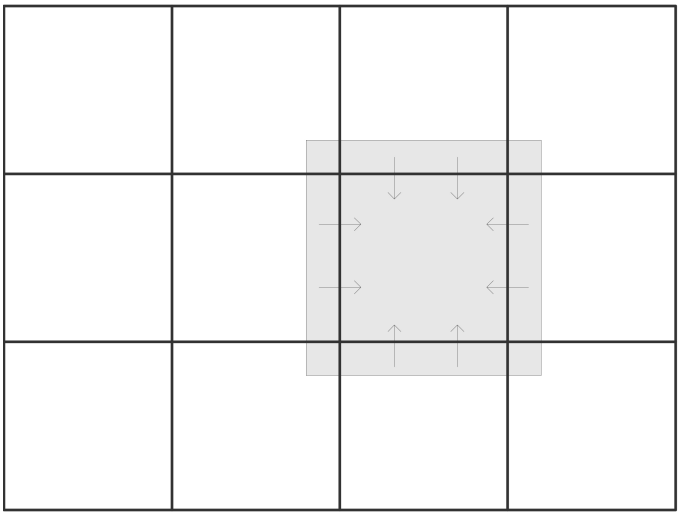


Figure 8.4: A grid divided over processors, with the 'ghost' region indicated

8.4 Data parallel grids

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

8.4.1 Description of the problem

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

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

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-6.5.) 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

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

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 8.4. 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.

8.4.2 Code basics

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

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

From the processor parameters we make a processor grid object:

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

and from the numerical parameters we make a number grid:

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

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

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

To set random values:

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

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

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

Next we need to look at some data structure details.

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

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

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

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

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

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

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

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

A good example of the use of `INDEX` is in the `number_grid::relax` routine: this takes points from the `shadow` array and averages them into a point of the `values` array. (To understand the reason for this particular averaging, see HPSC-4.2.3 and HPSC-5.5.3.) 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.;
  }
}
```

Chapter 9

Ascii table

Because every programmer needs one.

ASCII CONTROL CODES

dec CHAR hex oct

b7 b6 b5 / BITS b4 b3 b2 b1	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
	CONTROL		SYMBOLS NUMBERS		UPPERCASE		LOWERCASE	
0 0 0 0	0 0 NUL 0	16 10 DLE 20	32 20 SP 40	48 30 0 60	64 40 @ 100	80 50 P 120	96 60 ‘ 140	112 70 p 160
0 0 0 1	1 1 SOH 1	17 11 DC1 21	33 21 ! 41	49 31 1 61	65 41 A 101	81 51 Q 121	97 61 a 141	113 71 q 161
0 0 1 0	2 2 STX 2	18 12 DC2 22	34 22 ” 42	50 32 2 62	66 42 B 102	82 52 R 122	98 62 b 142	114 72 r 162
0 0 1 1	3 3 ETX 3	19 13 DC3 23	35 23 # 43	51 33 3 63	67 43 C 103	83 53 S 123	99 63 c 143	115 73 s 163
0 1 0 0	4 4 EOT 4	20 14 DC4 24	36 24 $ 44	52 34 4 64	68 44 D 104	84 54 T 124	100 64 d 144	116 74 t 164
0 1 0 1	5 5 ENQ 5	21 15 NAK 25	37 25 % 45	53 35 5 65	69 45 E 105	85 55 U 125	101 65 e 145	117 75 u 165
0 1 1 0	6 6 ACK 6	22 16 SYN 26	38 26 & 46	54 36 6 66	70 46 F 106	86 56 V 126	102 66 f 146	118 76 v 166
0 1 1 1	7 7 BEL 7	23 17 ETB 27	39 27 ’ 47	55 37 7 67	71 47 G 107	87 57 W 127	103 67 g 147	119 77 w 167
1 0 0 0	8 8 BS 10	24 18 CAN 30	40 28 (50	56 38 8 70	72 48 H 110	88 58 X 130	104 68 h 150	120 78 x 170
1 0 0 1	9 9 HT 11	25 19 EM 31	41 29) 51	57 39 9 71	73 49 I 111	89 59 Y 131	105 69 i 151	121 79 y 171
1 0 1 0	10 A LF 12	26 1A SUB 32	42 2A * 52	58 3A : 72	74 4A J 112	90 5A Z 132	106 6A j 152	122 7A z 172
1 0 1 1	11 B VT 13	27 1B ESC 33	43 2B + 53	59 3B ; 73	75 4B K 113	91 5B [133	107 6B k 153	123 7B { 173
1 1 0 0	12 C FF 14	28 1C FS 34	44 2C , 54	60 3C < 74	76 4C L 114	92 5C \ 134	108 6C l 154	124 7C | 174
1 1 0 1	13 D CR 15	29 1D GS 35	45 2D – 55	61 3D = 75	77 4D M 115	93 5D] 135	109 6D m 155	125 7D } 175
1 1 1 0	14 E SO 16	30 1E RS 36	46 2E . 56	62 3E > 76	78 4E N 116	94 5E ^ 136	110 6E n 156	126 7E ~ 176
1 1 1 1	15 F SI 17	31 1F US 37	47 2F / 57	63 3F ? 77	79 4F O 117	95 5F _ 137	111 6F o 157	127 7F DEL 177

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] Torsten Hoefler, Prabhanjan Kambadur, Richard L. Graham, Galen Shipman, and Andrew Lumsdaine. A case for standard non-blocking collective operations. In *Proceedings, Euro PVM/MPI*, Paris, France, October 2007.

[4] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. *SIGPLAN Not.*, 45(5):159–168, January 2010.

[5] Fiona Reid, Mark Bull, and Nicola McDonnell. *A Microbenchmark Suite for OpenMP Tasks*, volume 7312, pages 271–274. SpringerLink, 2012.

[6] 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 10

Index and list of acronyms

AVX Advanced Vector Extensions
BSP Bulk Synchronous Parallel
CAF Co-array Fortran
CUDA Compute-Unified Device Architecture
DAG Directed Acyclic Graph
DSP Digital Signal Processing
FPU Floating Point Unit
FFT Fast Fourier Transform
FSA Finite State Automaton
GPU Graphics Processing Unit
HPC High-Performance Computing
HPF High Performance Fortran
ICV Internal Control Variable
MIC Many Integrated Cores
MIMD Multiple Instruction Multiple Data
MPI Message Passing Interface
MTA Multi-Threaded Architecture
NUMA Non-Uniform Memory Access
OS Operating System
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

_OPENMP, 117

active target synchronization, 43, 44, 47
anti dependency, *see* data dependencies
argc, 74
argv, 74
array processors, 12
asynchronous computing, 22
atomic operation, 48
atomic operations, 48

barrier, 69
    in MPI, 65
batch
    job, 31
    scheduler, 31
Beowulf cluster, 30
breakpoint, 166
buffers, 35

C
    preprocessor, 116
C++, 72
    standard library, 80
      vector, 80
C99, 77
cacheline, 130
choice, 76
chunk, 147
chunk, 147
clusters, 18
coarse-grained parallelism, 17
collective
    root of the, 50
collectives, 50–57
    non-blocking, 55
commandline argument, 75
communication
    asynchronous, **35–36**
    blocking, 21, **35–39**
    non-blocking, 21, **35–36**, **39–42**
    one-sided, 42–49
    one-sided, implementation of, 48–49
    overlap with computation, 41
    persistent, **42**, 89–90
    synchronous, **35–36**
    two-sided, 20, 36–42
communicator, 33, 61–65
compiler, 81
    optimization level, 161
Compute-Unified Device Architecture (CUDA), 15
construct, 116, 145
context, 114
contiguous
    data type, 58
core dump, 161
cpp, 117
Cray
    T3E, 49
critical section
    flush at, 154
critical section, 109, 129, 132, 152
critical sections
    cost of a , 143

data dependencies, 138–140
data dependency, 137
data parallelism, 11
dataflow, 22
datatype, 57–61
    derived, 58–60, 77–81
    elementary, 57, 76–77
    signature, 58, 80

ddd, 161
DDT, 161, 168–169
ddt, 71
deadlock, 21, 36, 49, 167
debug flag, 162
debugger, 161
debugging, 161–169
    parallel, 167
dense linear algebra, 63
directives, 113, **144**
    cpp, 113
distributed memory, 18
distributed shared memory, 43
dynamic mode, 146

eager limit, 83
Eclipse, 168
    PTP, 168
emacs, 173
environment variables, 156
epoch, 44
    access, 45, 47, 94
    exposure, 45, 47, 93
Ethernet, 18
ethernet, 32

false sharing, 130
fence, 44
Fibonacci sequence, 133–136
fine-grained parallelism, 11
flow dependency, *see* data dependencies
fork/join model, **112**, 117, 137
Fortran, 33
    1-based indexing, 88
    array syntax, 126, 149
    fixed-form source, 144
    forall loops, 149
    Fortran90, 74

gdb, 161–168
ghost region, 184
GNU, 161
    gdb, *see* gdb
grid
    Cartesian, 65
    periodic, 65
group, 93
group of
    processors, 47

halo, 184
halo region, 26
handshake, 49
heap, 114, 116
histogram, 133

I/O
    in OpenMP, 125
ibrun, 31
implicit barrier, 151
    after single directive, 126
independent iterations, 16
indexed
    data type, 58
Infiniband, 18
inner product, 51
instrumentation, 170
Internal Control Variable (ICV), **152**

kernel, 15

latency, 23
lexical scope, 115, 127
load balancing, 121
load imbalance, 27
lock, 133
    flush at, 154
loop unrolling, 14
LU factorization, 122

`malloc`, 114
Mandelbrot set, 178
master-worker, 48, 178
master-worker model, 22, 39, 41
matching, 72
matrix
    sparse, 55
matrix-vector product
    sparse, 54

memory
distributed, 18
shared, 17
message passing, 19
MPI
2.2, 72
Fortran issues, 68, 107–108
I/O, 68
semantics, 72
mpi.h, 74
MPI_Abort, 33, 75
MPI_Accumulate, 46, 93
MPI_Aint
in Fortran, 107
MPI_Aint, 77
MPI_Allgather, 55, 99
MPI_Allgatherv, 55, 101
MPI_Alloc_mem, 44, 91
MPI_Allreduce, 52, 55, 99
MPI_Alltoall, 55, 100
MPI_Alltoallv, 55, 101
MPI_ANY_SOURCE, 39, 54, 72, 84, 99, 103
MPI_ANY_TAG, 84
MPI_Barrier, 65, 69
MPI_Bcast, 96
MPI_BOTTOM, 77
MPI_Bsend, 42, 89
MPI_Bsend_init, 90
MPI_BSEND_OVERHEAD, 42, 82
MPI_Buffer_attach, 88
MPI_Buffer_detach, 89
MPI_Cancel, 103
MPI_Cart_coord, 106
MPI_Cart_create, 106
MPI_Cart_rank, 106
MPI_Comm_create, 63
MPI_Comm_dup, 62, 105
MPI_Comm_free, 62, 105
MPI_Comm_group, 63
MPI_COMM_NULL, 62
MPI_Comm_rank, 33
MPI_COMM_SELF, 62
MPI_Comm_set_errhandler, 67
MPI_Comm_set_name, 63
MPI_Comm_size, 33
MPI_Comm_split, 62, 106
MPI_COMM_WORLD, 33, 62
MPI_Datatype, 98
MPI_DATATYPE_NULL, 77
MPI_ERR_BUFFER, 89
MPI_ERR_INTERN, 89
MPI_ERROR, 68
MPI_Error_string, 68
MPI_ERRORS_ARE_FATAL, 67
MPI_ERRORS_RETURN, 67, 108
MPI_Exscan, 52, 102
MPI_Fetch_and_op, 48, 95
MPI_Finalize, 32, 75
MPI_Gather, 54, 55, 97
MPI_Gatherv, 55, 101
MPI_Get, 46, 92
MPI_Get_count, 66, 67, 85
MPI_Group_difference, 63
MPI_Group_excl, 63
MPI_Group_incl, 63
MPI_Ibarrier, 55
MPI_Ibcast, 103
MPI_Ibsend, 89
MPI_IN_PLACE, 96, 100
MPI_Info, 91
MPI_INFO_NULL, 44, 91
MPI_Init
in Fortran, 68
MPI_Init, 32, 74
MPI_Init_thread, 66, 156
MPI_Iprobe, 67
MPI_Irecv, 39
MPI_Isend, 39
MPI_LOCK_EXCLUSIVE, 48
MPI_LOCK_SHARED, 48
MPI_MAX, 51
MPI_MODE_NOCHECK, 93
MPI_MODE_NOPRECEDE, 45, 93
MPI_MODE_NOPUT, 45, 93
MPI_MODE_NOSTORE, 45, 93
MPI_MODE_NOSUCCEED, 45, 93

MPI_Op, 108
MPI_Op_create, 52
MPI_PACK, 81
MPI_Pack, 61
MPI_Pack_size, 89
MPI_PACKED, 61, 82
MPI_Probe, 67
MPI_PROC_NULL, 38, 107, 177
MPI_PROD, 51
MPI_Put, 46, 91
MPI_Recv, 83
MPI_Recv_init, 42, 89
MPI_Reduce, 54, 55, 96
MPI_Reduce_scatter, 54, 98
MPI_REPLACE, 46
MPI_Request, 55, 86, 103
MPI_Request_free, 42, 90
MPI_Request_get_status, 90
MPI_Rsend, 49, 93
MPI_Rsend_init, 90
MPI_Scan, 52, 102
MPI_Scatter, 53, 98
MPI_Scatter_reduce, 49
MPI_Scatterv, 98, 101
MPI_Send, 83
MPI_Send_init, 42, 89
MPI_Sendrecv, 38, 86, 177
MPI_Sendrecv_replace, 39, 86
MPI_Sizeof, 68, 108
MPI_SOURCE, 39, 85
MPI_Ssend, 49
MPI_Ssend_init, 90
MPI_Start, 42, 89
MPI_Start_all, 89
MPI_Startall, 42
MPI_Status, 66, 84–86
MPI_STATUS_IGNORE, 66, 84, 88
MPI_STATUSES_IGNORE, 66, 84, 87
MPI_SUM, 51, 54
MPI_Test, 90
MPI_THREAD_FUNNELED, 109
MPI_THREAD_FUNNELLED, 156
MPI_THREAD_MULTIPLE, 109, 156
MPI_THREAD_SERIAL, 156
MPI_THREAD_SERIALIZED, 109
MPI_THREAD_SINGLE, 109, 156
MPI_Type_commit, 59, 77
MPI_Type_contiguous, 58, 59, 77, 107
MPI_Type_create_hindexed, 80
MPI_Type_create_struct, 60, 80
MPI_Type_extent, 77, 81
MPI_Type_free, 59, 77
MPI_Type_hindexed, 59, 60
MPI_Type_indexed, 59, 60, 79
MPI_Type_struct, 58, 81
MPI_Type_vector, 58, 59, 78
MPI_UNPACK, 82
MPI_Unpack, 61
MPI_Wait, 40, 42, 55
MPI_Wait..., 40, 90
MPI_Waitall, 40, 84
MPI_Waitany, 40, 66, 84, 87
MPI_Waitsome, 40
MPI_Win_complete, 94
MPI_Win_create, 91
MPI_Win_fence, 44, 93, 177
MPI_Win_lock, 95
MPI_Win_post, 93
MPI_Win_start, 93, 94
MPI_Win_unlock, 95
MPI_Win_wait, 93
MPI_Wtick, 70, 109
MPI_Wtime, 69, 108
MPI_WTIME_IS_GLOBAL, 109
mpif.h, 74
mpirun, 31, 32, 62
    and environment variables, 156
Multiple Instruction Multiple Data (MIMD), 18

new, 114
node, 18
NVIDIA, 15

omp
    atomic, 132, 141, **151**
    barrier, 131
    copyin, 149

critical, 132, 140, **151**
default, 128
do, 146
firstprivate, 149
flush, 149, 154
for, 146
barrier behaviour, 151
if, 146
lastprivate, 124, 149
master, 125, 148
ordered, 147
parallel, 113, 116, 118, 145, 146
parallel do, 145
parallel for, 120, 145
parallel sections, 145
private, 115, 127, 149
section, 124
sections, 124
shared, 115
single, 125, 148
task, 136, 137
taskgroup, 137
taskwait, 136, 137
threadprivate, 129, 140
workshare, 126, 149
omp clause
default, 149
none, 149
private, 149
shared, 149
depend, 137, 153
firstprivate, 128
lastprivate, 128
nowait, 123, 151
private, 127
reduction, 147
schedule
auto, 122
chunk, 121
guided, 122
runtime, 122
shared, 149
omp.h, 144
OMP_DYNAMIC, 152
omp_get_dynamic, 146, 152
omp_get_max_threads, 152
omp_get_nested, 152
omp_get_num_procs, 118, 152
omp_get_num_threads, 114, 118, 152
omp_get_schedule, 147, 152
omp_get_thread_num, 114, 118, 152
omp_get_wtick, 138
omp_get_wtime, 138, 153
omp_in_parallel, 146, 152
OMP_NESTED, 118, 152
OMP_NUM_THREADS, 117, 118, 152, 156
OMP_PROC_BIND, 152, 153
omp_sched_affinity, 147
omp_sched_auto, 147
omp_sched_dynamic, 147
omp_sched_guided, 147
omp_sched_runtime, 147
omp_sched_static, 147
OMP_SCHEDULE, 122, 147, 152
omp_set_dynamic, 146, 152
omp_set_nested, 152
omp_set_num_threads, 118, 152
omp_set_schedule, 147, 152
OMP_STACKSIZE, 149, 152
OMP_WAIT_POLICY, 152
one-dimensional partitioning, 23
OpenMP, 16
accelerator support in, 142
co-processor support in, 142
environment variables, 118
environment variables, 117, **152**
library routines, 118
library routines, **152**
version 3.1
thread affinity, 153
operating system, 143
origin, 42, 47, 94
output dependency, *see* data dependencies

packing, 61
parallel region
barrier at the end of, 150

parallel region, 112, 119
    flush at, 154
parallel regions
    nested, 152
passive target synchronization, 43, 47
persistent communication, *see* communication, persistent
ping-pong, 70
pipeline, 14
pipelining, 14
`PMPI_...`, 70
pragma, 116
private variables, 115
producer-consumer, 140
progress, 72
purify, 165

race condition, 48, 132, 140, 141
random number generator, 140
`reduction`, 130
region of code, 145
RMA
    active, 43
    passive, 43

scaling
    strong, 24
    weak, 24
scan
    exclusive, 52
    inclusive, 52
schedule
    clause, 147
`schedule`, 121
scope, 114
segmentation fault, 163
segmented scan, 52
sentinel, **144**
sequential implementation, 10
serialization
    unexpected, 37
shared data, 112
shared variables, 114
shmem, 49
Single Instruction Multiple Data (SIMD), 15
Single Instruction Multiple Thread (SIMT), 16
Single Program Multiple Data (SPMD), 18, 118
`sizeof`, 68, 91
spin-lock, 152
ssh, 31
stack, 114, 152
    overflow, 128, 149
    per thread, 149
struct
    data type, 58
structured block, 116, 145
subdomain, 24
symbol table, 161, 162
synchronization
    in MPI, 65
    in OpenMP, 131–136

target, 42, 47, 94
    active synchronization, *see* active target synchronization
    passive synchronization, *see* passive target synchronization
task
    scheduler, 136
TAU, 170
thread
    affinity, 153
    team, 116
        and variable access, 115
thread-safe, 66, 140
threads, 112
    initial, 113
    master, 112, 113
    team of, 112, 113
timing
    MPI, 69–70
    OpenMP, 153
topology, 64
TotalView, 71, 161, 168

valgrind, 71, 165–166
vector
    data type, 58

vi, 173
virtual shared memory, 43

wall clock, 108
wall clock time, 69
while loops, 123
window, 43–45
work sharing, 112
work sharing construct, 117, 119
workshare
    flush after, 154
worksharing constructs, **146–149**
    implied barriers at, 151
wraparound connections, 65