Parallel Programming with MPI*

David G. Robertson
Department of Physics and Astronomy
Otterbein College
Westerville, OH 43081
drobertson@otterbein.edu

Keywords: parallel programming, message passing, MPI,
domain decomposition, waves, oscillations

January 1, 2008

*Work supported by the National Science Foundation under grant CCLI DUE 0618252

Abstract

This module introduces some basic concepts and techniques of parallel computing in the context of simple physical systems. It focuses on the distributed memory model of parallel computation and uses MPI (Message Passing Interface) as the programming environment. The physical systems studied illustrate naturally the idea of domain decomposition. Besides learning the basics of MPI programming, students can explore a range of issues related to performance in parallel computation.

This module is designed for students with fairly sophisticated programming skills in C, C++ or Fortran. The students should ideally have taken (or be taking) an upper-level course in classical mechanics, as familiarity with systems of coupled oscillators may be helpful. Students with only introductory physics should also be fine, however, especially if they have had some exposure to differential equations.

Access to a distributed-memory parallel computing system is also required. This can be a dedicated system with specialized hardware, often accessible via regional or national supercomputing centers, but a simple "network of workstations" can also be used. This is a collection of computers connected using ordinary ethernet, with the MPI library and associated low-level drivers installed. Virtually all such systems will use the Unix operating system and some sort of batch processing; these tools will need to be mastered.

Finally, a tool for visualization will be very useful. `gnuplot` is a freely available program that allows the necessary visualization, and its use is described below. Other tools such as Vtk, Matlab and Mathematica could also be used if desired.

Contents

1 Overview

This module introduces basic concepts of distributed memory parallel computing in the context of simple physical systems. The emphasis is on the basic ideas and techniques of parallel computing rather than the solution of a sophisticated or timely scientific problem. The main problem considered is the solution of the equations of motion for a set of coupled oscillators in one and two dimensions, or equivalently, the wave equation in one and two dimensions.

Writing a parallel application involves decomposing a problem into tasks, which are then assigned to processors. In many cases, the tasks are not independent of one another, with the result that some sort of communication between processors is necessary. This might involve sending data from processor to processor, synchronization, etc. This inter-processor communication is the new and interesting feature of parallel programming, and standard programming models and tools have been developed to enable it. One of these is the Message Passing Interface (MPI) [1, 2], which is introduced here. MPI is a library of functions that can be called from a C, C++ or Fortran program to allow data transfer, collective operations among processors, and more.

The principal goals of this module are

- to familiarize students with the basic ideas and techniques of parallel computing;
- to introduce them to the MPI standard library for inter-processor communication; and
- to have them use the basic MPI tools to write a parallel code for solving the equations of motion for a coupled set of oscillators.

This last example may seem simple, but in fact it contains most of the features that make parallel programming subtle and interesting.

2 What You Will Need

It will be essential that the students are familiar with programming in a high level language like C, C++ or Fortran. (Full MPI implementations exist for these languages.) In particular, students using C will need to be familiar

with the basics of pointers. They will also ideally have taken (or be taking) an upper-level course in classical mechanics.

You will of course need access to a parallel computing environment with MPI. If you do not have a local "Beowulf cluster," you may be able to access such a system at a nearby regional or national supercomputing center. Since nearly all such machines use the Unix operating system, students will probably need to be familiar with this. In addition, if it is a public computer there will certainly be a batch processing system in place that will need to be understood.

A way of visualizing the oscillator chain will also be very helpful, both to make the results visible and as an aid in debugging. Especially if students pursue higher dimensional applications – the 2d case is both reasonably accessible and highly instructive – then in my experience a good visualization tool will be absolutely essential. `gnuplot` [3] is a freely available program that allows the necessary visualization, though at a fairly basic level; some examples involving gnuplot are given below. However, other tools, such as Matlab or Mathematica, could also be used.

3 Parallel Computing Basics

Parallel computing refers to a general process whereby a computational task is broken into pieces which can be worked on concurrently. A parallel computer is basically a set of processors that are wired together to enable this. By working on different parts of the problem simultaneously, a parallel computation can be performed in less time than a "serial" one, running on a single processor. Alternatively, for the same computing time a larger problem can be solved. Essentially all modern high-performance computing is parallel computing, because the price-to-performance ratio for parallel systems is so low. Historically, high-performance computing meant the development of very sophisticated, specialized processors and memory to enable the fastest possible calculation. Eventually, however, many of the ideas and developments in this area were carried over to the commodity microprocessor world, with the result that these processors became much more powerful. At present it is simply enormously cheaper to make a fast computer by connecting together a (possibly large) number of commodity CPUs than by working to engineer a single super-fast processor. As of this writing, the largest parallel computers in the world have several tens of thousands of processors.

Writing a parallel application involves decomposing a problem into tasks, which are assigned to different processors. In the ideal case these tasks are independent of one another, so the problem is "perfectly parallel, " or "embarrassingly parallel." As a simple example, say we have a model of some physical system that depends on a number of parameters. For example, this could be a model of fluid behavior in which quantities like the fluid density, temperature, etc. can be varied. We would like to calculate the results of the model for a range of possible densities and temperatures, to make plots indicating the range of variation. Ordinarily we would achieve this by running the program many times with different parameter values (or writing the program so that it changes the parameter values and recalculates the model, outputting the results to a file, let's say).

With a parallel computer, however, we could run the model simultaneously, with different parameter values, on as many processors as are available. A key point here is (the assumption) that the individual runs are independent. So there is no reason not to just run the code on a multitude of processors, and we expect that if there are N processors then the job can be finished in $1/N$ of the serial time. This represents the maximum possible speedup from parallel computation.

In many cases, however, the tasks are *not* completely independent, with the result that some sort of communication between processors is necessary. This might involve, for example, sending data from processor to processor, synchronization of processors, an so on. This inter-processor communication is at the heart of parallel programming, and is what makes parallel computing interesting. In recent years, standards for programming parallel computers have become well established. One such standard is the Message Passing Interface (MPI), a library of functions that can be called from C, C++ or Fortran programs to enable communication between processors in a parallel computer [1].

As a specific example where data exchange is required, let's consider the problem of calculating the propagation of a wave in some region. The physics is described by the wave equation, a partial differential equation in space and time. In two dimensions,

$$\frac{1}{v^2}\frac{\partial^2 D}{\partial t^2} = \frac{\partial^2 D}{\partial x^2} + \frac{\partial^2 D}{\partial y^2} \tag{1}$$

where $D = D(x, y, t)$ is the wave displacement. and v is the wave speed. Our task is to (approximately) solve this equation given some suitable initial

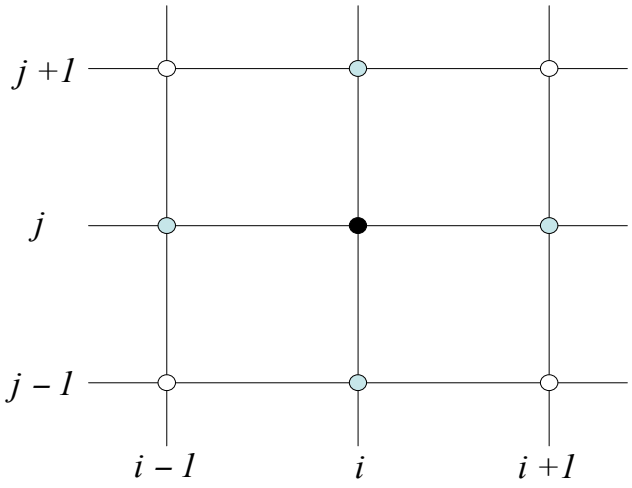

Figure 1: Part of the uniform grid introduced to discretize the wave equation. The black dot is the (i, j) grid point; the shaded dots are the nearest-neighbor points involved in the approximate second derivatives.

and boundary conditions. Note that an exceedingly large number of scientific problems are framed in terms of partial differential equations, so our considerations here will have wide applicability.[1]

Now, on a computer we cannot calculate at an infinite number of space points, so to begin with let us make a grid of discrete points at which we will evaluate the displacement. For simplicity we may imagine this is a uniform grid, i.e., that the distance between points is the same in both the x and y directions and is unchanging over the entire region of interest. (Depending on the specific problem under consideration, it could be advantageous to use a non-uniform grid. The basic ideas we are discussing would be unchanged however; only the details would be altered.) We can label the grid points with integer indices i and j, for the x and y directions, respectively, and call the wave displacement at the i, j grid point

$$D_{ij}(t) \ .$$

[1] A partial list from physics: diffusion and heat flow, electromagnetic potentials and fields, quantum wavefunctions, fluid flow, etc.

Thus we have a set of functions, one for each value of i and j, that depend on t.

Next, we must approximate the space derivatives with differences. To stay focused on the issues that arise when solving this problem on a parallel computer let us skip over the details of this process for now, and merely observe that derivatives are "non-local"; that is, the derivative of a function at a point involves not only the value of the function at that point, but also in the neighborhood of the point. After all, graphically the derivative represents the slope of the (tangent to the) function at the point of interest. So it expresses "where the function is going" in the neighborhood of that point.

The upshot is that an approximate partial derivative of D at grid point i, j with respect to x, say, will involve not only the value of D_{ij} itself but also those of its neighbors $D_{i+1,j}$ and/or $D_{i-1,j}$. Similarly, the y derivative will involve $D_{i,j+1}$ and $D_{i,j-1}$. Second derivatives are of course derivatives of derivatives, and so also involve neighboring grid points, perhaps even farther-flung neighbors than the nearest ones. It turns out that one possible approximation to the derivatives in eq. (1) can be written

$$\frac{\partial^2 D}{\partial x^2} + \frac{\partial^2 D}{\partial y^2} \approx \frac{1}{h^2} \left[D_{i+1,j} + D_{i-1,j} + D_{i,j+1} + D_{i,j-1} - 4D_{ij} \right] \ , \qquad (2)$$

where h is the spacing between the grid points. The interested reader may consult sect. 3.2 below for details of how this is obtained, but for now observe that the expression involves the nearest neighbors, as expected, and in a nice, symmetric way.

Using eq. (2), the wave equation takes the form of a set of coupled equations for the displacements on the grid:

$$\frac{d^2}{dt^2} D_{ij} = \frac{v^2}{h^2} \left[D_{i+1,j} + D_{i-1,j} + D_{i,j+1} + D_{i,j-1} - 4D_{ij} \right] \ . \qquad (3)$$

We would now also interpret the time derivatives as differences, allowing us to calculate the wave displacements at a series of grid points in time, or "time steps." The details of this will also be given below; the main thing to notice right now is that as we step forward in time, what happens to the displacement at the ij site depends on the displacement there as well as at the displacements at the nearest neighbor sites. This sort of coupling between grid points is very characteristic of differential equations and hence is exceedingly common in computer simulations of physical systems.

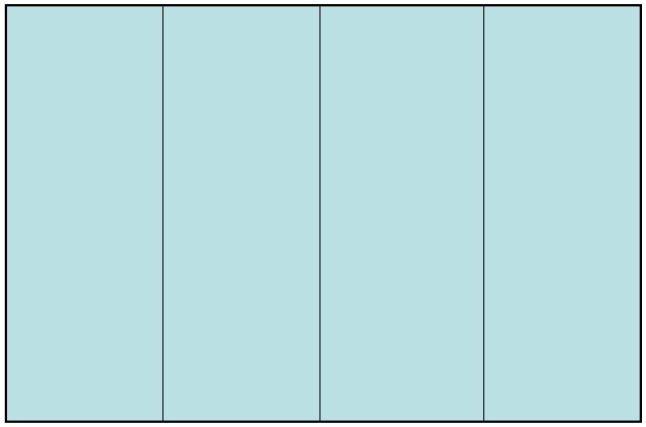

Figure 2: Decomposition of a planar domain by strips. Separate processors are assigned to the four rectangles and work concurrently.

So the typical type of problem we have in mind is this: given some initial state of the wave, and suitable boundary conditions,[2] we use eq. (3) to calculate the wave state (displacements) at a later time. The general structure of this evolution is such that each point is coupled to its nearest neighbors. Now let us think about how we might break this problem into pieces and implement it on a parallel computer.

The most natural approach would be to divide the physical domain into pieces, or sub-domains, and assign each sub-domain to a different processor. This is known in the trade as "domain decomposition." One possibility, decomposition by strips, is shown in fig. 2; other choices are possible. Now imagine the problem as seen by each processor. It has a set of grid points to update using the discrete wave equation, which couples nearest neighbors. For most of the points in the sub-domain this is no problem, since their neighbors are also in the sub-domain and hence are known to the processor.

[2]The need for boundary conditions can be seen from the difference equation (3). If we consider a grid point on the boundary of the domain then it is missing a neighbor on one side; how then do we apply eq. (3) for these points? The answer is that we must specify explicitly what happens at the edge points, either by giving the displacements there or by specifying the derivative in a direction normal to the boundary. In the language of continuum partial differential equations, these correspond to Dirichlet and Neumann boundary conditions, respectively.

However, points along an edge that is adjacent to another sub-domain are a problem: to update these we need the displacements at the neighbor points, but one of the neighbor points is in a different sub-domain and hence stored in the memory of a different processor. Before we can update the edge displacements we will need to get the neighbor values from the other processor, and vice versa, of course.[3] Hence the need for data exchange, or "message passing," in this general class of problems.

Now, for the parallel version to actually be faster than a serial program, the time spent updating the displacement values – which is done concurrently by the processors – must be sufficiently large compared to the time spent sending data around. Stated differently, if no communication were required then N processors could update the same number of points in $1/N$ the time of a single processor – this is the perfectly parallel case. The communication overhead reduces this speedup, but as long as it is small compared to the computation time there will still *be* a speedup. For our particular problem it is noteworthy that the number of points to update (the computational load) scales like the area of the grid, while the number of edge values to send/receive only scales like the grid perimeter. So as the number of grid points increases we expect the computational load to grow faster than the communication overhead and to eventually dominate. The exact location of the "break-even" point, where the computation becomes significant enough to provide a net speedup with the parallel algorithm, depends on many factors and is very difficult to predict.

Note that for some problems the ratio of computation to communication may be so low that it is just not worth parallelizing. This is basically a sign that the individual tasks are not sufficiently independent.

On the subject of speedup, you might wonder if it would be better to decompose the domain in a different way, e.g. by "checkerboarding" as in fig. 3. Without worrying too much about the details, it is clear that this will require more data exchanges (there are more interfaces between sub-domains) but that each data exchange will involve a smaller amount of data. So which is best? Is it better to send relatively more small messages, or fewer but larger messages? It depends on the computer! Typically the time needed to send a message can be thought of as consisting of two parts: a general overhead or "latency" that reflects the time needed to set up the message,

[3]If the edge points are an actual physical boundary of the domain, then we already have boundary conditions that tell us what to do and nothing new is required.

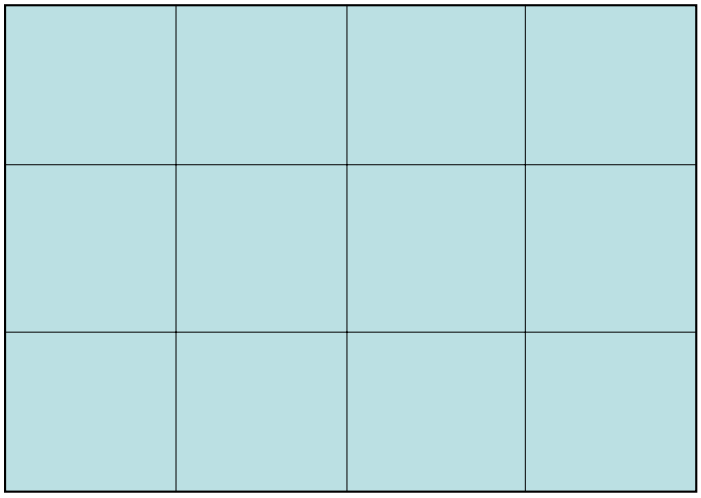

Figure 3: Checkerboard decomposition of a planar domain.

and the time to actually send it, which scales like the size of the message. Thus

$$T_{msg} = T_{lat} + mT_d \tag{4}$$

where m is the number of data values to be sent. T_{lat} and T_d are constants that depend, in general, on the details of the machine hardware and the software that performs the message passing. For some systems the latency may be so high that it pays to send fewer messages, at the cost of making them larger. For other systems the opposite may be the case. Part of the art of parallel programming is in tailoring algorithms for maximum speed and efficiency on different computing hardware.

3.1 Taxonomy of Parallel Computers

There are many possible arrangements for parallel computing hardware. They can generally be characterized as possessing "distributed memory", "shared memory", or some combination thereof.

The distributed memory model (fig. 4) is a very general one. In such systems each physical processor is "isolated" from the others, in the sense that its own memory is accessible only to it. As an extreme example, a set of networked PCs is a distributed memory system. Each CPU has access

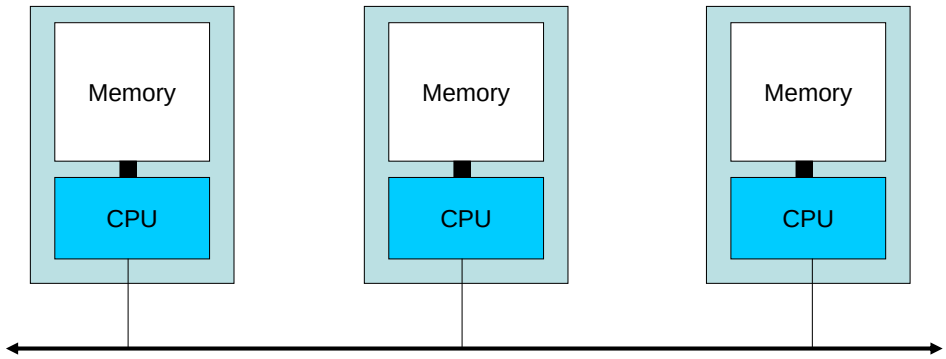

Figure 4: Distributed memory architecture for parallel computers.

to a set of memory. It can read from and write to this memory, but other processors cannot access this memory directly. If some processor needs data that resides in the memory of another CPU, then that data must be explicitly sent from one to the other. This is the natural arena for the message passing paradigm we have been discussing.

In a shared memory architecture (fig. 5), all processors have direct access to a shared pool of memory. Any CPU can read from or write to any of the shared memory. This simplifies significantly the problem of programming, as messages to not need to be sent. In our diffusion example, we could implement the same domain decomposition as before, with the same result, that each processor (CPU) needs data from its neighbor CPUs to proceed. In this case, however, each CPU can simply read the needed data from the shared memory, and use it to update the value of all points assigned to it. The only real programming issue that arises is one of scheduling: we don't want any processor to update its edge points before its neighbors have had a chance to use the current values. So we need to insure that the processors are synchronized in some way.

Other examples could be discussed to show that programming in a shared memory environment is generally simpler than is message passing, but perhaps this will be intuitively clear. The real limitation of shared memory parallelism has to do with how the CPUs are physically connected to the memory. There is a bunch of circuitry called a "memory bus" that accomplishes this, but it is too difficult to separately wire each CPU to each memory

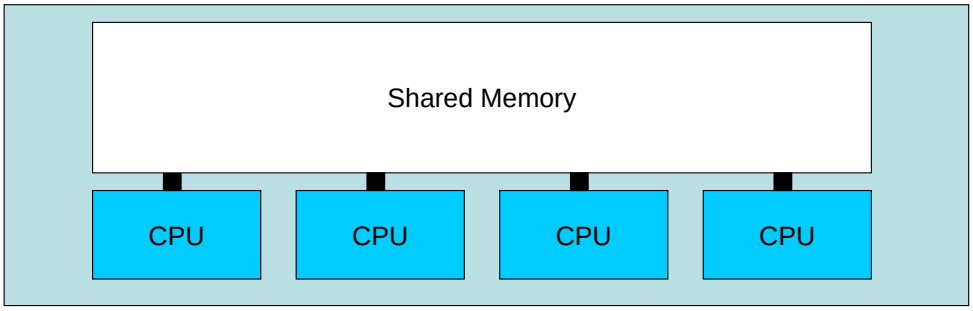

Figure 5: Shared memory architecture for parallel computers.

location, especially if there are very many CPUs. So in practice there is essentially a single "pipe" through which data moves between the CPUs and the memory. The problem is that as the number of CPUs grows this pipe is more and more easily saturated, with the result that CPUs spend more and more time waiting for data. The net effect is to limit the number of CPUs that can be practically used, and at present this effective limit is on the order of a few tens of processors. By contrast, as of this writing the largest distributed memory machines have several tens of thousands of processors.

Combinations or hybrid designs are also possible. A fairly common arrangement, for example, is to have a collection of "nodes", each of which is a multi-CPU computer. The nodes are then wired together to create the full parallel computer. Within each node the CPUs share a pool of memory, but between nodes data must be sent back and forth. It is even possible for the system memory to be physically distributed but "logically shared"; that is, the operating system can make it appear to the programmer that the memory is shared, while actually handling data exchanges behind the scenes. From a user's point of view this generally means that while programming appears to feature the relative simplicity of the shared memory model, some of the shared memory is "fast" (the memory that is physically attached to the CPU) while other memory is "slow" (memory that is attached to other CPUs and for which the system must move data through the machine). The reader will appreciate that this can make writing efficient, high-performance programs for such machines very subtle.

In this module we will adopt the message passing paradigm, as it allows

the most control over program execution and (most importantly) scaling to the largest number of processors. Our next task is to explore how such computers may be programmed. In recent years certain standards for parallel computing have arisen, notably (in the distributed-memory world) the Message Passing Interface (MPI). MPI is actually a library of functions that can be called from a C, C++ or Fortran program, and that enable the programmer to send and receive data between processors. It also provides a number of high level features including collective communication, user-defined datatypes, and virtual topologies of processors. In section 4 we will explore the basics of parallel programming using MPI.

3.2 Addendum: Numerical Derivatives and Boundary Conditions

Here we examine the numerical approximation of derivatives in more detail. In fact this is a large subject and there are many issues and options; we shall only touch on the basics. The interested reader is advised to consult a standard text on numerical analysis (for example [4, 5]) for additional discussion.

We assume a grid has been introduced, with discrete points separated by a spacing h. The function whose derivative we want is given at the grid points, which are labeled by an index i:

$$f(x) \to f_i \ .$$

To approximate the derivative we recall its definition,

$$\frac{df(x)}{dx} \equiv \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \ . \tag{5}$$

This suggests that if h is small but not zero, then we can take

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x)}{h} \ , \tag{6}$$

with the approximation (apparently) getting better as we make h smaller and smaller.[4] In the grid notation we would write

$$f_i' \approx \frac{f_{i+1} - f_i}{h} \ . \tag{7}$$

[4]On a computer, where real numbers are represented in a discrete fashion, problems will arise if h is made too small. The precise nature of the problem will depend on how

In fact, we could just as well take the difference "to the left," namely

$$f_i' \approx \frac{f_i - f_{i-1}}{h} \ . \tag{8}$$

There is no reason to expect this to be a worse approximation than the original "right" derivative.[5] A better approximation than either might be the *average* of the left and right derivatives. So we could choose instead

$$\begin{aligned} f_i' &\approx \frac{1}{2}\left[\frac{f_{i+1} - f_i}{h} + \frac{f_i - f_{i-1}}{h}\right] \\ &= \frac{f_{i+1} - f_{i-1}}{2h} \ , \end{aligned} \tag{9}$$

which has a pleasingly symmetric character. It is also more accurate than the others – the error introduced is of order h^2, compared to order h for either of the "asymmetric" differences.

To see why this is so, imagine that we know the function $f(x)$ and Taylor expand it about the point x:

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \mathcal{O}(h^3) \ . \tag{10}$$

We can solve this for $f'(x)$,

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h) \ , \tag{11}$$

showing that the approximation (7) above is accurate to $\mathcal{O}(h)$. A similar analysis shows the same result for expression (8), of course – we simply Taylor expand in the other direction. However, eq. (10) also implies

$$f(x+h) - f(x-h) = 2hf'(x) + \mathcal{O}(h^3) \tag{12}$$

we use eq. (6) and a detailed discussion of these issues would take us too far afield. For the moment just keep in mind that h cannot be made too small in practice. This limits the accuracy of our approximate derivatives, although more sophisticated schemes for calculating approximate derivatives can help fix this problem.

[5]Mathematically the two are identical in the limit $h \to 0$. Indeed, this requirement is part of what *defines* a function that can be differentiated at the point x. This is actually the source of the freedom we have in defining approximate derivatives: the only real requirement is that they reduce to the correct thing in the limit $h \to 0$. Anything that has this property will be an approximation to the derivative. However, some approximations are faster or more accurate, hence the extensive discussion of the issue in the literature.

(the $\mathcal{O}(h^2)$ terms cancel in the sum), so that

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2) \; . \tag{13}$$

Hence the symmetric approximation to the derivative is indeed more accurate for a fixed value of h.

Eq. (10) further implies that

$$f(x+h) + f(x-h) = 2f(x) + h^2 f''(x) + \mathcal{O}(h^4) \; . \tag{14}$$

This immediately gives an approximation to the second derivative,

$$f''(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} + \mathcal{O}(h^2) \; , \tag{15}$$

or, in discrete notation,

$$f_i'' \approx \frac{f_{i+1} + f_{i-1} - 2f_i}{h^2} \; . \tag{16}$$

This is one of many possible approximations to the second derivative, chosen here for its simplicity.

Let us also expand the discussion of boundary conditions a bit. As mentioned above, for points on the edge of the physical domain there are neighbors missing, and hence additional information is required in order to update their displacements. To resolve this we can imagine that there *are* neighbors and impose some (boundary) conditions that determine the new neighbors' displacements. Specifically, we could:

1. Specify that the missing neighbors are points with some given displacements. For example, we could specify that these points have zero displacement for all times; in the wave example this would indicated a boundary like a drumhead, where the medium is fixed. In the usual language of partial differential equations this would correspond to Dirichlet boundary conditions.

2. Specify that the normal derivative at the boundary has some prescribed value. To see how this solves the problem, consider the left-hand edge of the domain, consisting of points with $i = 1$. We imagine there is a mass with $i = 0$ so that the $i = 1$ mass does have a left neighbor and can

be updated. The problem is then how to determine the displacement of the $i = 0$ mass. Using the right derivative we have

$$f_0' \approx \frac{f_1 - f_0}{h} \tag{17}$$

which can be solved for f_0:

$$f_0 \approx f_1 - h f_0' \,. \tag{18}$$

So if we fix f_0' to some value then we can use this to determine f_0. The net result is again that the boundary condition fixes the values of the edge points. A precisely similar analysis can be used for the right hand end, of course. In the usual language of partial differential equations this would correspond to Neumann boundary conditions.

3. Use any convenient mixture of Dirichlet and Neumann conditions at one or both ends, provided all edge points are ultimately accounted for.

4 Introduction to MPI

4.1 History

As noted above, MPI is a library of functions (subroutines in Fortran) that can be called from a C, C++ or Fortran program to enable communication between processors [1]. It is actually defined in a language-independent way, so that in theory it may be implemented for any high-level language. In practice implementations exist for C, C++, Fortran 90 and Java. MPI version 1.2 includes the basic functionality and is fully implemented on many systems. MPI 2 includes many new features, e.g. parallel I/O and the ability to start and stop processes while the program is running, but it has yet to be fully implemented on many systems. (Subsets of it do commonly exist.) Since the availability of MPI 2 functionality is evolving, in this module we concentrate on the basics as realized in MPI 1.2.

4.2 Basic Concepts

The quickest way to become acquainted with MPI is to examine some simple programs. We can use these to highlight the basic structures and procedures that MPI uses.

Here is our first MPI program, written in C:

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

int main (int argc, char *argv[]) {
   int rank, size;
   MPI_Init (&argv, &argc);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   printf("Hello from processor %d of %d\n", rank, size);
   MPI Finalize ();
   return 0;
}
```

You might want to pause a moment and see if you can guess what this program will do before reading further.

The most important thing to keep in mind at the outset is this: *this code gets executed exactly as written on every processor.* So each line of the above program is executed separately on however many processors we start it running. Remember also that MPI assumes a distributed memory model – each processor has its own local memory. Thus each variable is declared separately by every processor and exists locally for that processor. Processors can modify their own copies without affecting the local variables of any other processors. In the present example, integer variables `rank` and `size` exists on each processor. (There is also an int `argc` and an array of pointers to char `argv` but we will not use these for the moment.)

With this in mind, let us walk through the above code. Note that there is an MPI header file, `mpi.h`, that must be included in any source file that uses MPI. The function `MPI_Init` is an initialization routine; it sets up the MPI environment and passes the values of `argc` and `argv`, if present, to all processors. This must always be the first MPI routine that is called by your program.

Next comes a function `MPI_Comm_rank`. This takes two arguments, an MPI "communicator" and a pointer to an `int`. More on communicators below; for the moment just note that this routine returns the "rank" of the processor. The rank is an integer running from 0 to $n-1$, where n is the total number of processors, and is assigned by the system at run time. You can think of

the rank as a unique ID number for processors. It is used often in MPI, e.g. to identify the source and destination processors for messages. Thus you will usually want to call this function early on in your MPI program.

In the present example each processor calls the routine, and each receives a value which is stored in the variable `rank`. Remember that these are all local variables: if there are 8 processors, there are 8 copies of `rank`, each (after the call) with a different integer value from 0 to 7. Note that in general you do not have any control over what actual CPU gets assigned any particular rank; this is decided by the system at run time.

By the way, we can now answer a question that may have been bothering you: If each processor executes the same code, how they can ever do anything different? The answer is that they can branch on the value of `rank`. As a crude example, you could do something like:

```
if (rank == 0) {
   ... do something ...
}
if (rank == 1) {
   ... do something else ...
}
...
```

Now, what is an "MPI communicator"? Basically it is a group of processors that can communicate with each other. Whenever you send or receive a message, it will be within a particular communicator, which is required as an argument of all send/receive functions. The call to `MPI_Init` automatically defines a default communicator called `MPI_COMM_WORLD`, which includes all the processors that are running. In many cases this is all you need, though MPI does allow you to define new communicators, i.e., subsets of processors that can only communicate among themselves.[6]

Next comes a function `MPI_Comm_size`, which again takes an MPI communicator and returns an integer. This function returns the total number of

[6]You might do this as a security measure, to help ensure that messages are being passed correctly. Technically the processor rank is defined relative to a particular communicator, or processor group; this is why the call to `MPI_Comm_rank` included the communicator as an argument. If we were to define another communicator then at least some of our processes would belong to two communicators, the new one plus `MPI_COMM_WORLD`. In this case each processor could, and generally would, have a different rank ID in each communicator.

processors in the communicator. If `MPI_COMM_WORLD` is supplied then this is the total number of processors. Each processor receives the same value from the function, but it is stored in a local copy of the variable `size`.

Next comes a standard C `printf` statement, which prints out the values of `rank` and `size`. Remember that each processor executes this statement, so if there are n processors then there will be n lines of output printed. Each contains the same text but the local values of `rank` and `size` are used. Of course, each processor has the same value of `size`.

If run on 4 processors, say, the output produced by the code might look something like this:

```
Hello from processor 1 of 4
Hello from processor 3 of 4
Hello from processor 0 of 4
Hello from processor 2 of 4
```

(Remember that ranks run from 0 to $n-1$!) An important feature to note is that there is generally *no synchronization between processors*. The output can appear in any order, and if the code is run several times then in general the output order will vary. The lesson here is that you should never assume that processors are synchronized in any way, unless you insure it with suitable code. MPI provides a number of functions for synchronizing processors, though we will not use them here.

The final act is a call to `MPI_Finalize`, which cleans up MPI and should be the last MPI routine called by your code.

4.3 Some Technical Points

In C, MPI uses a number of internally-defined data structures, e.g. communicators, data types, etc. MPI provides "handles," or names that can be used to refer to these. For example, there is a defined type called `MPI_Comm`, a communicator. If you wished to define a new communicator `foo`, you would first declare it as

```
MPI_Comm foo;
```

All such types, along with objects like `MPI_COMM_WORLD`, are defined in the header file `mpi.h`, which must be included in any MPI code. This file also contains prototypes for all MPI routines.

MPI functions conform to a standard presentation. In C, a function name has the format

```
MPI_Xxxxxx (...)
```

and returns an int indicating whether the function executed normally or not. As usual in C, this return value can be ignored, as we have done in the above sample code. Less slipshod programmers would use, e.g.,

```
err = MPI_Comm_rank (...);
```

and test the value of `err` to see if anything went wrong. If `err` is 0 then the function executed normally; if non-zero there was some error.

In Fortran, subroutines have an extra argument of type `INTEGER` which holds the return value.

4.4 Exercises

1. Compile and run the sample program from this section. This will likely involve a number of steps that are specific to your local computing environment, and it is essential that these basics be sorted out before proceeding further. Run the code several times with various numbers of processors and observe the results.

2. Modify the sample code so that only the processor whose rank is half the total number of processors actually prints the greeting. If the total number of processors is odd, then only the processor with the largest rank should print.

4.5 Sending Messages

Now let's send some messages. In MPI, a message consists of an array of elements of some type. The array can be of any length including one, corresponding to a single data item. The possible data types include all the basic types present in C or Fortran; in addition, user-defined types may be constructed and sent. Thus if your code uses, say, a struct to hold the Cartesian coordinates (x, y, z) of a point, you could define an MPI datatype that matches this and then send these structs directly in messages, rather than individual x, y and z values. We will not need this facility for the basic projects in this module, but it will come in handy for more advanced applications.

Now, when you call a function to send a message, you will specify the number of items to be sent and their type. MPI provides its own "handles," or names, for the basic types, that you use in the message passing functions. Thus, for example,

```
float x[10];
...
MPI_Send (..., 10, MPI_FLOAT, ...);
```

shows schematically how 10 `float` values in an array could be sent. The `MPI_FLOAT` is the MPI handle that indicates a `float`.

The most basic type of message passing is "point-to-point" communication. This is communication between two processors, one of which sends the message and the other receives. All such communication happens within a communicator, with the processor ranks identifying the source and destination of the message.

This may be contrasted with "collective" communication, which involves a group of processors. For example, we could send some data from one processor to all the others, or gather data from all processors onto one processor. Of course we could perform these tasks using point-to-point communications, but MPI provides collective routines that save coding and make errors less likely. We shall discuss these in more detail later.

In point-to-point communication one processor sends and the other receives. There are a variety of different send routines available in MPI, but for now we will use the "standard" one. Its C signature is

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

Let us examine the arguments in turn.

The first argument, `buf`, is a pointer to the beginning of the data to be sent. In other words it is the memory address where the message data starts. If we were sending the contents of an array `x[]` then for `buf` we could use `&x[0]`. Alternatively, since in C the name of an array is in most cases equivalent to a pointer to its first element, we could simply use `x`. The argument `count` specifies the number of data items to be sent, and `type` indicates their type. Note that `type` is a specially defined MPI type; here we would use, e.g., `MPI_FLOAT` for floats, etc.

So which data are actually sent? Answer: the first `count` consecutive items in memory starting at the address `buf`. So we will need to be aware of

how our data are laid out in the computer memory. Fortunately, languages have rules about this. Both C and Fortran guarantee that for one- and two-dimensional arrays (declared statically), the consecutive elements will be stored in consecutive memory locations. So if we wanted to send the first 10 elements of a 1d array of ints `x[]`, we would specify `x` for `buf`, 10 for `count`, and `MPI_INTEGER` for `type`. The function would then go to the address specified by `x`, which is the address of the first element of the array, and send the contents of the next 10 `int`-sized chunks of memory. Since the array elements are stored consecutively, this does just what we want.

Note that MPI assumes you know what you are doing! If you start in the wrong place, or run off the end of an array, for example by making `count` too large, MPI will happily gather up whatever bytes it finds and send them along. In general these bytes may contain anything at all, including data not interpretable as numbers or even random garbage. No error messages are generated; the only sign you may have of this type of error is downstream, when the corrupted data are used somewhere else.

It remains to specify how the elements of two-dimensional arrays are organized. In C, elements are stored in "row major" order, which means that if we regard the array as a 2d matrix, then reading along the rows steps us through consecutive memory locations. That is, we start with the (0,0) element, then (0,1), (0,2), ..., through the end of the first row, then (1,0), (1,1), (1,2), ..., and so on. Note that this assumes you have declared the array statically, as, e.g.,

```
float x[10][10];
```

If you have instead declared a pointer to pointer to float, and dynamically allocated storage for the individual rows, then these rows are not necessarily adjacent in memory. If you are doing this, however, then you presumably understand how the data *are* laid out in memory and you should be able to make the necessary adjustments!

In Fortran array elements are stored in "column major" order, so that reading along columns steps us through consecutive memory locations. Thus (1,1), (2,1), (3,1), ..., (1,2), (2,2), (3,2), ..., etc.

Next, `rank` is the rank of the destination processor, i.e., where the message should be sent. `tag` is just a numeric ID that can be attached to a message, which for example could be checked at the destination to make sure the correct message was received. Finally, `comm` is the MPI communicator within which the communication occurs. Often this is just `MPI_COMM_WORLD`.

So a complete call to `MPI_Send` might look like

```
float data[500];
 ...
MPI_Send (data, 100, MPI_REAL, 3, 123, MPI_COMM_WORLD);
```

This would send the first 100 elements of the array `data` to processor 3, with an ID tag of 123. The call

```
MPI_Send (&data[100], 100, MPI_REAL, 3, 123, MPI_COMM_WORLD);
```

would instead send 100 items starting from `data[100]`, that is, elements 100 to 199 of the array.

Remember that `MPI_Send` also returns an integer status code, which we are here ignoring.

Now, say we call `MPI_Send` in this way, and it returns. What has actually happened at that point? All that is *guaranteed* is that it is safe to overwrite the sent data without affecting the message. It is not known whether the message has been received at its destination, or even if it has actually been sent (the data may have been copied to a local system buffer and may be waiting to be sent on from there). `MPI_Send` is free to do either of these, and different MPI implementations can and will differ in their behavior. If you definitely need or want one of these behaviors then MPI provides additional sending routines that are specific in their criteria for completion. For example, the "synchronous send" routine, `MPI_Ssend`, definitely does not return until the message has actually reached its destination. The "buffered send," `MPI_Bsend`, returns after the data have been copied to a buffer – faster than the other sends, but more work to set up, since the user (you) has to create the buffers and tell MPI about them. We will not discuss all the possibilities in detail here; check the MPI documentation if you are interested.

One aspect of the ambiguity in `MPI_Send` is that certain bugs can manifest themselves differently on different systems. For example, say we send a message but, due to a bug, it is never received at its destination. (This could happen if the destination processor fails to do the necessary thing to receive the message; more on this in a moment.) On systems where `MPI_Send` is synchronous, i.e., it waits for receipt before returning, then the sending processor will hang. If it buffers, however, then the sending processor will proceed on its way, and the only indication of the bug may be garbled or useless results.

As an aside, in MPI parlance we have been discussing the "completion criteria" for the various send modes, that is, what has happened when the routine is "complete." All of the routines mentioned so far are "blocking," meaning the routine does not return until the completion criterion (receipt at destination, or copied to a buffer, or whatever) has been satisfied. For each of these send modes there is also a corresponding "non-blocking" version, which returns immediately but with no implication that the send is complete. You would then later test to see if the routine had completed, or else wait for it to complete.

Thus, for example, an ordinary (blocking) synchronous send sits and does not return until the message has been received at its destination (its completion criterion is met). This can be inefficient, however, since the sender accomplishes nothing useful while waiting. A non-blocking synchronous send, on the other hand, returns immediately. The sending processor can then do something else useful (one hopes!); later it can check whether the send is complete. If so, its completion criterion has been satisfied, in this case indicating that the message has been received. This allows you to maximize efficiency, by minimizing the time that processors are idle. Again, the MPI documentation can orient you on these advanced techniques.

Fortunately, receiving a message is relatively simple! There is one routine, with prototype

```
int MPI_Recv (void *buf, int count, MPI_Datatype type, \
              int source, int tag, MPI_Comm comm, \
              MPI_Status *stat)
```

Incoming data are placed in consecutive memory locations starting at `buf`. In this case `count` is actually the *maximum* number of data items to be received. You can send fewer without fear, but if you send more than `count` items the behavior is undefined. Expect the worst! `source` is the rank of the sending processor, and `type`, `tag` and `comm` are the same as for a send. Note that the tags must match between sender an d receiver for the message to get through. In addition, the communicators must match, and the source and destination ranks must be valid. Finally, there is a special status `struct` that will contain some details concerning the message. We'll discuss these in a moment.

The completion criterion for a receive is simple: the received data are ready for use. This means that the routine returns when the message has

arrived.[7]

Here is a sample program that sends a simple message from processor 1 to processor 0:

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

void main (int argc, char *argv[]) {
   int rank, i;
   float foo[100], bar[200];
   MPI_Status stat;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   if (rank == 1) {
      for (i=0; i<100; i++)
         foo[i] = i;
      MPI_Send (foo,100,MPI_FLOAT,0,42,MPI_COMM_WORLD);
   }
   else if (rank == 0) {
      MPI_Recv (bar,200,MPI_FLOAT,1,42,MPI_COMM_WORLD,&stat);
      printf("Processor %d, received data\n", rank);
      printf("Check: bar[10] = %f\n", bar[10]);
   }
   MPI_Finalize ();
}
```

Study this example carefully! Remember that the same code is executed on each processor. So each has a copy of the arrays `foo` and `bar`, as well as the two ints and `stat`. We begin by initializing MPI and determining our ranks. Then, if we are processor 1 we first fill up the array `foo` with data (just to have something to send), and send all 100 elements to processor 0. The tag is 42.

If we are not processor 1, however, we skip this branch and proceed. If we are processor 0 then we take the next branch, with a call to `MPI_Recv`. We

[7]There is also a non-blocking receive, which returns immediately. The receiving processor could then do some useful work, rather than waiting, and come back later to check if the receive is complete. When it is, the received data are ready for use.

allow up to 200 incoming floats, and store the data in the array `bar` (starting at its beginning). Status information is placed in `stat`. Processor 0 then prints a couple of lines of output, which includes one of the received data values as a check. The output would look like this:

```
Processor 0 received data
bar[10] = 10.00000
```

If we are neither processor 0 nor 1, we skip over everything and wait at `MPI_Finalize` for the others to finish.

As a simplifying feature, the receiving processor may use certain wildcards. To avoid having to specify the sending processor, use `MPI_ANY_SOURCE` in place of the source rank. To receive with any tag use `MPI_ANY_TAG` in place of the tag. In this case the actual source of the received message and the actual tag are stored in the status struct. You would access these as `stat.MPI_SOURCE` and `stat.MPI_TAG`, respectively. Thus the sample program above could be written:

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

void main (int argc, char *argv[]) {
   int rank, i;
   float foo[100], bar[200];
   MPI_Status stat;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   if (rank == 1) {
      for (i=0; i<100; i++)
         foo[i] = i;
      MPI_Send (foo,100,MPI_FLOAT,0,42,MPI_COMM_WORLD);
   }
   else if (rank == 0) {
      MPI_Recv (bar,200,MPI_FLOAT,MPI_ANY_SOURCE,\
                MPI_ANY_TAG,MPI_COMM_WORLD,&stat);
      printf("P %d, received data from P %d\n", \
             rank, stat.MPI_SOURCE);
      printf("The tag was %d\n", stat.MPI_TAG);
```

```
      printf("Check: bar[10] = %f\n", bar[10]);
   }
   MPI_Finalize ();
}
```

In this case the output would be

```
P 0 received data from P 1
The tag was 42
bar[10] = 10.00000
```

The status struct also contains the number of data items actually received. This is accessed using a function `MPI_Get_count`; see the MPI documentation for further details.

4.6 Exercises

1. Rewrite the sample program in this section so that a message is sent from processor 1 to processor 2. The message should be an array of 50 integers with any values you choose. Processor 2 should receive the message with both source and tag wildcarded. Processor 2 should then print the source, the tag, and one of the array elements received.

2. Write a program in which messages are exchanged between processor 0 and processor 2. Processor 0 should calculate the squares of the first 200 integers, then transfer this array (of integers) to processor 2. Processor 2 should then divide all even values in the array by 42.0, and send the resulting (real) array back to processor 0. Output whatever you feel is necessary to confirm that the data were transferred correctly. Do not use source and tag wildcarding!

3. Write a program that will run on four processors. P3 will receive messages from all the others, with the source rank wildcarded. Each other processor should send a different integer to P3 (your choice). When P3 receives the value from P0, it should print the square of that integer value. When P3 receives a value from P1, it prints 2 to the integer power passed. Finally, when P3 receives an integer from P2, it prints the sum of all integers up to and including the sent value.

4.7 Advanced Topic: Derived Datatypes

MPI provides a way of building new, more complicated, data types from existing ones. Thus we could for example define new types corresponding to sub-arrays, like columns (for C) or rows (for Fortran) where the data elements are not adjacent in memory, or types corresponding to C `structs` or Fortran common blocks. Derived types may be constructed from the basic types and from other derived types, so complicated hierarchies can be developed. We can then use these types in sends and receives to pass the entire data object at once. The obvious alternative, sending such data by means of repeated individual sends, is slow, clumsy and error prone.

The basic procedure consists of first *constructing* the new datatype and then *committing* it. After it has been committed it is available for use.

There are several "types" of derived datatype, corresponding to how the new type is specified. We shall here focus on the "vector" datatype, which is useful for data that is non-adjacent but distributed in a homogeneous pattern in memory. The function that constructs this datatype is

```
MPI_Type_vector (int count, int blocklength, int stride, \
                 MPI_Datatype oldtype, MPI_Datatype *newtype)
```

`newtype` is the newly constructed type; it is built out of items of type `oldtype`. The new type consists of a set of *blocks*, each with `blocklength` contiguous items of type `oldtype`. `stride` is the number of old data items from the beginning of one block to the beginning of the next block.

This is easiest to understand with a specific example. Fig. 6 shows a set of contiguous memory locations, each the size of an `oldtype`. (Thus, e.g., if `oldtype` is `float`, then in most cases the size of each element (rectangle) is 4 bytes.) We wish to define a datatype that consists of the shaded items. In this case we have 2 blocks, each of which has 3 items of type `oldtype`. So `count` is 2 and `blocklength` is 3. `stride` in this case would be 5; that is, the number of old data items from the start of one block to the beginning of the next.

To commit the new datatype, we call

```
MPI_Type_commit (MPI_Datatype *newtype)
```

After this the new type is ready for use.

Here is a C program that defines a type corresponding to a column of a 2d array (regarded as a matrix):

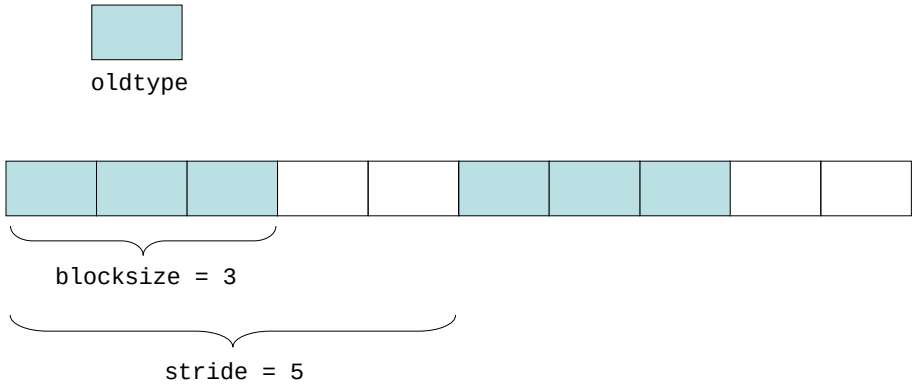

Figure 6: Derived datatype example. The shaded blocks represent `oldtyype` data items to be included in the new type.

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

#define NCOLS 8
#define NROWS 4

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

  int myrank, i, j;
  double x[NROWS][NCOLS];
  MPI_Status stat;
  MPI_Datatype coltype;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
  MPI_Type_vector (NROWS, 1, NCOLS, MPI_DOUBLE, &coltype);
  MPI_Type_commit (&coltype);

  /* Now test by sending some data: */
  if (myrank == 2) {
    for (i=0; i<NROWS; i++)
      for (j=0; j<NCOLS; j++)
        x[i][j] = i*10.0 + j;
    MPI_Send (&x[0][6],1,coltype,0,42,MPI_COMM_WORLD,&status);
```

```
  }
  else if (myrank == 0) {
    MPI_Recv (&x[0][3],1,coltype,2,42,MPI_COMM_WORLD,&status);
    for (i=0; i<NROWS; i++)
      printf("P%d: x[%d][3] = %lf\n", myrank, i, x[i][3]);
  }
  MPI_Finalize ();
  return 0;
}
```

This example will also repay careful study. Note that the `coltype` contains a total number of elements equal to the number of rows in the matrix; each block is a single element; and the stride between elements is equal to the number of columns. This is true for the elements that comprise *any* column of the matrix, so the new datatype can be used quite generally.

To test the type we have filled up the array `x` on processor 2 with data indicating the row and column. We then send the seventh column of `x` to processor 0. Note that as usual the first argument to `MPI_Send` is the starting address of the data to be sent, in this case the address of the top element of the column. Likewise, the receiving processor can place the incoming column anywhere it likes; in the example at hand, the incoming data becomes the fourth column of `x` on processor 0. Note that it is perfectly legal to set the starting address of the outgoing or incoming data to be anything; in this case the data will be taken or written according to the datatype specification, leading most likely to meaningless results. MPI assumes you know what you are doing!

The final act is for processor 0 to print its fourth column, to confirm that the data was transferred correctly. The output of the code is

```
P0: x[0][3] = 6.000000
P0: x[1][3] = 16.000000
P0: x[2][3] = 106.000000
P0: x[3][3] = 1006.000000
```

as expected.

5 Projects

5.1 Coupled Oscillator Chain

Consider a one-dimensional chain of N particles, each with mass m. The particles are connected by identical springs with spring constant k and have the same undisturbed (equilibrium) separation a. Let us denote the displacement from the equilibrium position of the ith mass by x_i, with $i = 1, \ldots, N$. Thus $x_i = 0$ for all i in equilibrium.

The particles are allowed only to move along the direction of their separation; we thus describe longitudinal oscillations of the system. The equation of motion for the ith mass is then

$$\begin{aligned} m\frac{d^2x_i}{dt^2} &= -k(x_i - x_{i+1}) - k(x_i - x_{i-1}) \\ &= -k(2x_i - x_{i+1} - x_{i-1}) \ , \end{aligned} \tag{19}$$

or

$$\frac{d^2x_i}{dt^2} = -\omega^2(2x_i - x_{i+1} - x_{i-1}) \tag{20}$$

where $\omega \equiv \sqrt{k/m}$. Our goal is to solve this set of equations approximately. Of course we must specify the initial displacement and velocity for each mass, say at $t = 0$. Notice that the system has the property that the behavior of each mass depends on its own displacement and those of its nearest neighbors. Indeed, the right hand side of eq. (20) has the same structure as the discrete wave equation in one dimension, so we can also think of this as a describing transverse waves on a string, for example.

Now imagine that we know all the displacements and velocities at some time t. How would we calculate the displacements and velocities at a slightly later time $t + \epsilon$? First recall that a second order differential equation can always be rewritten as a pair of first order equations. In the case at hand we can introduce the velocity of the ith mass and write eq. (20) as

$$\frac{dx_i}{dt} = v_i \tag{21}$$

$$\frac{dv_i}{dt} = a_i \ , \tag{22}$$

where

$$a_i = -\omega^2(2x_i - x_{i+1} - x_{i-1}) \ . \tag{23}$$

Now, if the time step ϵ is small, then the derivatives can be approximated by differences, "unpacking" the definition of the derivative as before. In the most straightforward approach, eq. (22) is approximated as

$$\frac{v_i(t+\epsilon) - v_i(t)}{\epsilon} = a_i(t) \ , \tag{24}$$

which gives

$$v_i(t+\epsilon) = v_i(t) + \epsilon a_i(t) \ . \tag{25}$$

Thus given the displacements x_i, x_{i-1} and x_{i+1} at any time, we can calculate the acceleration $a_i(t)$ (using eq. 23)) and then the velocity a short time later from eq. (25).

We can do the same thing to update the position, of course. Eq. (21) can be approximated for small ϵ as

$$x_i(t+\epsilon) = x_i(t) + \epsilon v_i(t) \ . \tag{26}$$

Since we know $v_i(t)$ we can calculate $x_i(t+\epsilon)$. So the final result is thus that given $x_i(t)$ and $v_i(t)$ we can determine $x_i(t+\epsilon)$ and $v_i(t+\epsilon)$ using the above procedure. These new values can then be taken as the starting values for the next step, of course, so the process can be iterated to give the (approximate) solution at any later time.

Clearly, to get accurate results we must take ϵ to be "small," but how small is "small"? The smaller the better, it seems, but we would really like to keep ϵ as large as we can get away with. One reason is that the smaller we make ϵ, the more time steps are required to get to the actual time of interest. More time steps means we have to wait longer for the answer.[8]

Now, the simple difference approximation we introduced above is actually rather inaccurate; that is, a fairly small ϵ is required, leading to more overall work, than is the case with other schemes. But as previously discussed, there is a great deal of freedom in how we approximate our derivatives. Let us pursue a simple modification to our approach that yields much more accurate results with almost no extra work [7].

[8]Other considerations come into play here as well, for example the "roundoff" error associated with computer arithmetic on real numbers and its potential interaction with the specific algorithm used to approximate the differential equations of interest. This is a long and technical story, and the interested reader is invited to consult a text on numerical analysis for further details [6].

Eq. (26) computes the change in x_i over the interval t to $t + \epsilon$ using the velocity at the *beginning* of the interval, but in general, of course, the velocity changes during the interval. If ϵ is small enough, then this change in v_i is also small and the approximation may be reasonable, but we can do better by using the velocity at some point *during* the interval to compute the change in x_i. The simplest choice is to use the velocity in the middle of the interval, and this is what we shall do. Hence we take

$$x_i(t + \epsilon) = x_i(t) + \epsilon v_i(t + \epsilon/2) \tag{27}$$

as the formula for updating x_i. How do we get $v_i(t + \epsilon/2)$? Well, the same remarks apply to the velocity: eq. (25) updates v_i using the acceleration at the beginning of a time step, but a better approach would be to use the acceleration at the middle of the step. But what if we had $v_i(t - \epsilon/2)$, say, and were stepping to $v_i(t + \epsilon/2)$? The middle of this step would be at time t, so

$$v_i(t + \epsilon/2) = v_i(t - \epsilon/2) + \epsilon a_i(t) \ . \tag{28}$$

But this works well for us, since $a_i(t)$ depends on the displacements (see eq. (23)), which are themselves known at time t. So the upshot is that we can use eqs. (27) and (28) to evolve the system, with the displacements and velocities "leapfrogging" each other as we proceed.

One problem remains: how do we get the whole thing started? Our initial conditions are given as $x_i(0)$ and $v_i(0)$, but we need $v_i(\epsilon/2)$ to begin the process. Here we can simply use

$$v_i(\epsilon/2) = v_i(0) + (\epsilon/2)a_i(0) \tag{29}$$

to set the needed velocities.

So the final algorithm for solving the set of difference equations of motion eqs. (28) and (27), is as follows. We specify some initial displacements $x_i(0)$ and velocities $v_i(0)$ at $t = 0$. We obtain $v_i(\epsilon/2)$ from eq. (29). Then, assuming we know $x_i(t)$ and $v_i(t + \epsilon/2)$ for some t we can update them by:

1. using eq. (27) to calculate the new displacements $x_i(t + \epsilon)$, and then

2. using eq. (28) to compute the new velocities $v_i(t + 3\epsilon/2)$.

We can iterate this for as long as we wish, with the results of each step becoming the initial values for the next step.

The only unresolved issue is what happens at the ends of the chain – the end masses $i = 1$ and $i = N$ have no neighbors to the left and right, respectively. Of course, we still have to specify some boundary conditions for the system. For now let us imagine that these end masses are connected by the same sort of springs to walls that cannot move. We can accomplish this by imagining that there are neighboring "masses" with $x_0(t) = x_{N+1}(t) = 0$ and $v_0(t) = v_{N+1}(t) = 0$ for all times. Later we can, e.g., introduce driving forces at one or both ends; this will amount to specifying the motion of the fake masses at $i = 0$ and/or $i = N + 1$.[9]

5.2 Exercises

1. To familiarize yourself with the equations of motion, write a serial (single-processor) code to simulate N coupled masses using the above algorithm. It is convenient to define $m = 1$ – it appears only in combination with k so there is no need for two variable values.

2. Run the code for $N = 2$. Choose k around 1 for simplicity and experiment with different time steps ϵ. One way to determine useful values of ϵ is to reduce ϵ until the solution does not change within some desired tolerance. For example, say we wish to calculate the mass displacements at $t = 10$, to a relative accuracy of 10^{-4}. We could evolve the system from $t = 0$ to $t = 10$ with decreasing values of ϵ, until the relative difference between the solution from one run to the next is less than 10^{-4}. Of course, if ϵ becomes too small then the increasing number of steps required will make roundoff error more and more important. Compare the "naive" (Euler) method we discussed above to the more sophisticated leapfrog (midpoint) method.

   Use `gnuplot` or another visualization tool to examine the oscillations of the chain. (It is easiest to make 2d plots showing the displacement of one mass versus time.)

3. In this case there are two normal modes of oscillation, with frequencies $\omega_1 = \sqrt{3k/m}$ and $\omega_2 = \sqrt{k/m}$. Guess, or derive for yourself (or consult a mechanics textbook [8] to find) the displacements for the two normal

[9] If we wished to make the ends completely free, i.e., have the end masses connected to nothing, we could simply rewrite the equations of motion for these so that they only have forces arising from the single neighbor.

modes. Set up initial conditions that match these displacements and check that the system executes simple harmonic motion. Determine the frequency from the simulation and check that it agrees with the exact formula.

4. Compare your numerical results with the exact solution for one or both of the normal modes and use this to further refine your value for ϵ.

5. Modify the code to calculate the total energy of the system (kinetic plus potential for each particle) at each time step. Is the energy approximately constant in time?

6. Modify the code so that there is a sinusoidal driving force at the left end; i.e., give the fake mass $i = 0$ a motion

$$x_0(t) = A \sin \tilde{\omega} t \tag{30}$$

   Start the system with $x_i = v_i = 0$ and examine the maximum steady-state displacement of either mass over time. Observe that as $\tilde{\omega}$ approaches one of the normal mode frequencies, the maximum steady-state displacement becomes large. Use this method of driving the system at a variable frequency to determine the normal mode frequencies and compare to the exact values.

7. The exact result for the normal mode frequencies with N masses is

$$\omega_n^2 = \frac{4k}{m} \sin^2 \left(\frac{n\pi}{2(N+1)} \right) ,$$

   where n is the mode number. Use the driving technique to determine several normal mode frequencies for $N = 10$ and confirm this result.

8. Run the simulation for a large ($N = 100$ or more) chain initially in equilibrium, but with the sinusoidal driving force. The disturbance created by the driving force will propagate down the chain; determine its (approximate) speed. Decide qualitatively how this propagation speed should depend on ω and confirm that this is the case. Does the propagation speed depend on $\tilde{\omega}$? If so, why?

5.3 Parallel Simulation

Once the use and meaning of he equations of motion are clear and you are getting reasonable results from the serial code, we can move on to an MPI version that runs on multiple processors for possibly large values of N. Let us first discuss the general structure of the parallel application.

The most natural way of dividing the problem up is to split the chain into pieces, assigning a subset of masses to each available processor. As in our earlier discussion, every mass in a subdomain except for the end masses can be updated independently of the other subdomains – the equations of motion relate the response of each mass to its own displacement and those of its immediate neighbors. So each processor can update most of its masses right away; only the two at the ends require information from neighboring domains.

In more detail, let's assume we have N_T total masses and we will run on M processors. Dividing the masses among the processors means each will handle $N = N_T/M$ masses; let us agree that this will always be a whole number, to keep things simple.[10] Furthermore, let's assert that processor 0 has the left-most group of N masses, processor 1 the next group, and so on.

Now, each processor will require two N-dimensional arrays to hold the displacements and velocities of the masses for which it is responsible; let's call these `x[]` and `v[]`. Since each processor executes the same code, each has its own (local) arrays with these names which can hold the displacements and velocities for that processor's subdomain.

Roughly speaking each iteration of the code will involve first sending and receiving the edge values as needed, and then updating the positions and velocities of all masses. The latter task is most easily accomplished, as in your serial code, by a `for` loop over the masses. To avoid having to treat the end masses separately, let us adopt the following trick: instead of having N-dimensional arrays, let us make them $N+2$ dimensional. The added elements will be used to hold the values for the neighbors of the end masses. Thus for a given processor, the array elements `x[1],..., x[N]` are the displacements of the masses assigned to that processor. The elements `x[0]` and `x[N+1]`

[10] Note that it would be a bad idea to give different numbers of masses to different processors since this would result in a load imbalance – processors with fewer masses would typically wind up waiting at the data exchange stage because other processors were still updating their sub-domains. For many applications load balancing can be a significant problem, but in this case we can easily arrange for perfect balance.

are available to hold the displacements of the masses immediately to the left and right of this subdomain, respectively. If one of these is a fixed end then `x[0]` or `x[N+1]` will just be zero, as appropriate, and unchanging. If not, then values received from neighboring processors via message passing will be stored there in the first (message passing) step of the algorithm. In the next step, a single loop updating elements 1 to N (just as in the serial code) then takes care of all of this processor's masses and we are ready for the next iteration.

5.4 Exercises

1. Now for the real problem! Write a parallel version of the chain solver to run on some convenient number of processors. (It is helpful if this number is not hard-wired into the code, so that it can be changed easily. To this end, recall that you can find out how many processors there are from within the code (while it is running) with a call to `MPI_Comm_size`.) Leave the driving force in place at the left hand edge.

   (Note that visualization will be an extremely useful debugging tool. Since the serial code was already working, the likeliest source of bugs is (surely!) the message passing. In many cases such errors will be manifested as odd behavior at the boundaries of the subdomains. Visualizing your solution as a disturbance propagates past such a boundary will quickly reveal whether the message passing is working correctly or not.)

2. Once you are convinced that your MPI code is working properly, measure the speedup relative to the serial code for a fixed problem size. How much of the theoretical speedup (factor of M for M processors) do you obtain? If possible, increase the number of processors to the point where the performance actually gets worse than that of the serial code. Why does this happen?

   For code timing, you can use

   ```
   MPI_Wtime ()
   ```

   This function takes no arguments and returns a `double`, the value of the system clock. To time a section of code, call it before and after and take the difference; this is the elapsed time:

```
double start, end;
...
start = MPI_Wtime ();
<... stuff to be timed ...>
end = MPI_Wtime ();
printf("Elapsed time: %lf\n", end-start);
```

If you do this on every processor then each will print its elapsed time, of course. This is a useful way of identifying load imbalances.

5.5 The Two Dimensional Case

A natural extension of the above program is to two space dimensions, i.e. a plane "mattress" of masses connected by springs, oscillating in the direction normal to the plane. The equations of motion for this case are the same as for the discrete wave equation, eq. (3).

Renaming $D_{ij} \to x_{ij}$ and introducing the velocities v_{ij} we have the kinematical equations for the "leapfrog" method,

$$x_{ij}(t+\epsilon) = x_{ij}(t) + \epsilon v_{ij}(t+\epsilon/2) \tag{31}$$

$$v_{ij}(t+\epsilon/2) = v_{ij}(t-\epsilon/2) + \epsilon a_{ij}(t) \ , \tag{32}$$

with the acceleration

$$a_{ij} = \omega^2 \left[x_{i+1,j} + x_{i-1,j} + x_{i,j+1} + x_{i,j-1} - 4x_{ij} \right] \ . \tag{33}$$

There are now options for decomposing the domain. We could divide it into strips that cross the entire plane in either direction, or "checkerboard." In either case the message passing becomes more involved, because each processor must now send strips of edge values, not just single values, to its neighbors.

As the simplest approach, let's divide the domain into strips that run horizontally (for C) or vertically (for Fortran). That is, in C the strips should cover the entire domain from left to right; they are then "stacked" vertically. The reasons for these particular orientations will become clear shortly.

Now, each processor will need 2d arrays to hold the displacements and velocities for its points. As before, let us give these arrays one extra row or column on every side. For sides that represent physical boundaries of the

system, these extra rows and columns can hold the fixed boundary values (Dirichlet). For rows and columns on edges adjacent to another subdomain, they serve to hold the neighboring processors' edge values, which are sent before the masses are updated.

To see the reason for partitioning the domain as described above, recall that a basic MPI message consists of a number of data items that are adjacent in memory. In C, for example, the elements of a 2d array are stored in row major order. So with the decomposition advocated above, where processors need to send (edge) rows of their displacement matrices, the relevant data are adjacent in memory and the message is easy to construct. (For Fortran the situation is exactly reversed: columns of the array are adjacent in memory, so columns are easy to send in MPI.) Sending a column in C, by contrast, requires sending a group of numbers that are not adjacent in memory (although they are evenly spaced, by a number of steps equal to the number of columns in the array). This can certainly be done, but it is not the easiest way to begin!

5.6 Exercises

1. Write a serial program to simulate the 2d case. Verify that the behavior of your solution is reasonable. You should experiment with a variety of boundary and initial conditions. Make sure you are able to visualize the solution.

   Regarding visualization, `gnuplot` can be used to make surface plots, e.g., plots of the wave displacement plotted over the xy plane. You may find if easiest to use contour plots – these are quick and show quite clearly the general progression of a disturbance (wave). For basic contour plotting try

   ```
   gnuplot> set nosurface
   gnuplot> set contour base
   gnuplot> set view 0,0
   gnuplot> splot 'data.file'
   ```

   The first command turns off the surface that would be normally produced by `splot`, the second sets the contour plot to appear directly on the xy plane, and the third sets the viewpoint to be directly above the plane. The last command plots the data in `data.file`.

To avoid having to specify for `gnuplot` the organization of your data, make the grid square and have your code separate the displacements for each row with a blank line. For a 2×2 grid, for example, the data file should look like this:

```
a
b

c
d
```

where a, b, c, d are some numbers. Unless instructed otherwise, `gnuplot` assumes that data in this form – with N "blocks" separated by blank lines, each of which has N numbers – represents data on a 2d uniform grid, thus:

```
c d
a b
```

2. Write an MPI program to simulate the 2d problem, using the strip decomposition discussed above. Specifying the initial state of the system is up to you, but one possibility is to have a sinusoidal "source" or driving mass located along one edge. You could then watch the wave disturbance spread out from the source, reflecting off the walls, etc. Multiple sources will allow examination of interference phenomena.

   A way of visualizing the results will again be quite helpful here. Watching the disturbance pass over a domain boundary will be the quickest way of discovering whether there is a problem with the message passing. The problem of combining the output from multiple processors to show the entire plane is still a problem, though the concatenation method discussed above will work for C.

3. (Advanced) Write an MPI code for the 2d case using decomposition in the "unnatural" direction for your chosen language (i.e., for C decompose into vertical strips).

4. (Advanced) Write an MPI code for the 2d case using a checkerboard decomposition, that is, partitioning in both directions simultaneously.

If you have completed the previous exercise, no fundamentally new issues arise, though the details are more complicated. Be sure to think carefully about how processors are assigned to domains; this determines which processors are "neighbors."

If you wish to explore more advanced features of MPI, look in the documentation under "virtual topologies." These allow you to define a virtual grid of processors, each of which knows who its neighbors are. This feature, along with the use of a derived datatype to send non-adjacent strips of arrays, makes this advanced decomposition quite tractable.

With a working checkerboard simulation, study whether checkerboarding or strip decomposition is faster for your system. Can you estimate the parameters T_d and T_{lat} from eq. (4) above?

References

[1] The MPI Forum defines and maintains the MPI standard, and is the official repository for the standards documents: `http://www.mpi-forum.org`.

[2] Some other potentially useful resources on MPI include:
Peter Pacheco, *Parallel Programming with MPI* (Morgan Kauffmann, 1996);
Introduction to MPI and *Intermediate MPI*, on-line courses available at NCSA: `http://webct.ncsa.uiuc.edu:8900/webct/public/home.pl`.

[3] `gnuplot` is available at `http://www.gnuplot.info`.

[4] S. E. Koonin, *Computational Physics* (Benjamin/Cummings, 1985).

[5] T. Pang, *An Introduction to Computational Physics* (Cambridge University Press, 2006).

[6] L. D. Fosdick, E. R. Jessup, C. J. C. Schauble, and G. Domik, *Introduction to High-Performance Scientific Computing* (MIT Press, 1996).

[7] See, for example, chapter 9 of R. P. Feynman, R. P. Leighton and M. Sands, *The Feynman Lectures on Physics* (Addison-Wesley, 2005).

[8] For example, S. T. Thornton and J. B. Marion, *Classical Dynamics of Particles and Systems* (Brooks Cole, 2003).