Parallel Programming with MPI: Instructor Notes*

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

January 1, 2008

Abstract

Some notes and suggestions for the exercises are presented. The sample codes are briefly described. Please contact the author if you have any comments or suggestions.

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

1 Goals and Time Needed

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.

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 many of the features that make parallel programming subtle and interesting.

My estimate is that about two weeks of class time, assuming 3-4 hours per week in class and some work outside of it, should be sufficient for this module at least through the 1d parallel project. With very well-prepared students the 2d case with simple decomposition (strips in the "natural" direction) can also be completed in this time. More likely a third week will be needed for the 2d case, especially if any of the advanced projects (checkerboard decomposition) are attempted.

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.) The notes are currently focused on C, in the expectation that this is the more common choice. C++ can also be used with the C MPI library, of course. Students using C will certainly need to be familiar with at least the basics of pointers. They will also ideally have taken (or be taking) an upper-level course in classical mechanics, though students with

only introductory physics should also be fine, especially if they have had some exposure to differential equations.

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. Many such centers offer "classroom" accounts for teaching. Alternatively, a network of workstations connected using ethernet can be used, though in this case performance may be so poor that significant speedups are hard to achieve.

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` 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 VTK, Matlab or Mathematica, could also be used.

3 Notes on the Exercises

Here I provide some hints and suggestions on the various exercises, with the idea that you will choose what is appropriate to share (and when is the appropriate time to share it!) with your students.

3.1 Exercises from Section 4.4

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.*

   This is a starter to make sure students know how to compile, link and run on the system at hand.

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.*

   Basic MPI exercise, with branching on ranks.

3.2 Exercises from Section 4.6

Most of these exercises are straightforward applications of the send and receive functions, and require no comment.

3.3 Exercises from Section 5.2

The goal of these is to sort out all the "physics" issues involving time stepping, boundary conditions, etc., before tackling the parallel version. This part of the problem will be essentially unchanged.

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.*

   You will need arrays `x[]` and `v[]` to hold the displacements and velocities. Since we are using fake masses on the ends to implement the boundary conditions, it is convenient to make these arrays $N + 2$ dimensional, and use the 0 and $N + 1$ elements of `x[]` to hold the fake mass displacements. Thus elements $1, \ldots, N$ hold the displacements of the actual masses. (The fake mass velocities never enter the problem so we don't really need these, but doing the same with `v[]` as with `x[]` means that element i of each array can refer to the same mass in the chain. The wasted memory in the two extra elements is insignificant.)

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.)*

Exploration should reveal that time steps as large as $\epsilon \sim 0.05$ are reasonable for a couple hundred time steps. The naive Euler method will be markedly worse.

3. *In this case there are two normal modes of oscillation, with frequencies $\omega_2 = \sqrt{k/m}$ and $\omega_1 = \sqrt{3k/m}$. Guess, or derive for yourself (or consult a mechanics textbook [7] to find), the displacements for the two normal 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.*

   The two modes have the masses oscillating exactly in phase and exactly out of phase, respectively.

4. *Compare your numerical results with the exact solution for one or both of the normal modes, as well as a general linear combination of them, and use this to further refine your value for ϵ.*

   This is a nice test that will quickly show how far ϵ and the total number of time steps can be pushed. It also gives the student some experience with resolving the general motion into normal modes.

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?*

   The energy fluctuates a surprising amount, but oscillates fairly close to the initial value.

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*

$$x_0(t) = A \sin \tilde{\omega} t \tag{1}$$

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.

Looking at the motion graphically will quickly reveal whether you are close to a resonance, but to get accurate (say, three significant figures) results you will probably want to have the code keep track of the maximum displacement over several periods of the normal mode.

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 resonant 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?*

3.4 Exercises from Section 5.4

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.*

Remember that the standard send `MPI_Send` blocks either until the message is received or it is buffered – its completion criterion is undefined. So if two neighboring processors were to first send, then receive, the code might hang: if `MPI_Send` does block until its message is received then there is no way for the messages to get through, since no

receives ever get called! If `MPI_Send` buffers the message, by contrast, then this actually works okay: the sends return after the buffering and the (blocking) receives then collect the messages. But even if this works on one system it may fail on another, so it is better to structure the code so that it works whatever the local behavior of `MPI_Send`. (If you are feeling ambitious, you can use non-blocking sends and receives here: call one of each, then wait for the receive to complete and you're done.)

Outputting the string displacements will also require some thought. The issue is that each processor has its own values and nothing more; thus if each writes values to a file then you get a bunch of files, each with data for part of the string. (This assumes the processors write to files that have unique names; if they all write to the *same* file then you get all the data interleaved randomly, or in other words gibberish – not good.) Here are some ideas for approaching this. One simple way is to have each processor write its displacements to a unique file, say with a name like `out.0.32` for the output of processor 0 at timestep 32. Then you can just examine each of these individually using whatever visualization tool you used with the serial code. Only slightly more work would be to concatenate the files using a Unix command like

```
cat out*.32 > bigout.32
```

Since `cat` will take the output files in the proper order, the data in `bigout` will be organized correctly to represent the whole string. *Then* use the viz tool to look at it.

One could also do the assembly of the big file for the whole string in the MPI code. The idea would be that whenever output is to be written, each processor sends its displacement data to one "master" processor. (Hence this will require a slightly different organization of the code than that proposed above!) The master processor has an array big enough for the whole string; it receives each processor's contribution, stores it in the appropriate place in the big array, and then writes the big array to a file.[1] This will slow down the simulation somewhat since the worker processors have to periodically send their whole configuration

[1] To avoid unnecessary waste of a possibly large amount of memory, this big array could be allocated dynamically. Thus each processor would have a pointer declared, but only the master would call `calloc` to allocate space for it.

to the master, but it we only write out every 100 timesteps or so this should not be too burdensome. Once the data is received by the master, of course, the workers can go back to computing while the master does the comparatively slow job of writing to disk.

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.

   It is a good idea to time the code on every processor, to see how much variation there is. Note also that `MPI_Wtime` can be used in the serial code too (be sure to `#include <mpi.h>` but don't call anyy other MPI

routines). It is a nice high-resolution timer and so useful in many contexts.

The eventual drop-off of performance as the number of processors increases is because that increase reduces the size of the sub-domains treated by individual processors (for fixed overall problem size, of course). At the same time the communication load is increasing, at least in terms of number of messages. Eventually the computational load decreases to the point where the communication overhead dominates, and the parallel code actually runs more slowly than the serial code.

3.5 Exercises from Section 5.6

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
```

Again, this is mainly intended to work out the kinks in the physics part of the problem and the visualization before tackling any parallelization.

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.

   It will probably be easiest conceptually to imagine the horizontal strips (for C) assigned to increasing rank processors as you go up, i.e. processor 0 has the bottom strip, etc.

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.)*

   This is really just a warm up for the next problem, as there is no reason otherwise to do this – the extra overhead in managing the non-adjacent data can only make the code slower compared to the previous version.

Your main problem will be sending the edge values, since the needed array elements are not adjacent in memory in this case. You can of course send each element individually, though this may not be the most efficient approach. Another possibility would be to have 1d arrays to hold the edge values to be sent/received. You could first copy the needed elements from the 2d array of displacements to the 1d holding array, then send this array with one message. On receipt, the data could be stored in a holding array, then copied to the correct elements of the 2d displacement array.

The best option would be to define a special datatype that corresponds to an array column (for C) or row (for Fortran). This will require use of the MPI "vector" datatype.

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?

   The most involved application by far, especially if virtual topologies are not used. If you try this one, you should at least use derived datatypes for array columns (and perhaps even rows, for uniformity of treatment).

4 Sample Programs

A collection of sample programs in C is distributed with the module. They typically take command line arguments, with some other parameters hard-

wired via `#define` statements. Comments in the programs explain the arguments.

The programs included are:

1. `helloworld.c`

   A basic MPI version of the old classic.

2. `chain2serial.c`

   The two-element chain in serial form. Code for calculating energies is present but commented out.

3. `chainNserial.c`

   The N element chain in serial form, with a sinusoidal driver on the left.

4. `chainNmpi.c`

   MPI version of the N element chain. The number of masses to simulate is an input but it must be evenly divisible by the number of processors. Note carefully the order of sends and receives.

5. `mattressNMserial.c`

   Serial version of the 2d problem with $N \times M$ masses (hard wired). There is a single sinusoidal driver mass on one wall but the rest of the boundary is fixed to zero displacement. Pure Neumann boundary conditions could be implemented here as well. Note that `OUTSKIP` is the number of time steps between each writing to disk of the wave configuration. With $\epsilon \sim 0.1$, say, not much happens in a single time step!

6. `mattressNMmpi.c`

   MPI version of the 2d code, using the natural strip decomposition. Observe here where the message data is taken from and deposited. The format for output file names is `out.timestep.rank`; this will allow concatenation of files from the same time step to produce a file for the entire domain.