Parallel Computing on the CPUG Beowulf with mpiJAVA

Kristopher Wieland, Kevin Kyle, Rubin Landau
Oregon State University, Department of Physics

July 19, 2002

1 Using JavaMPI

The Computational Physics for UnderGraduates (CPUG) program at OSU has set up a Beowulf cluster. This is a do-it-yourself supercomputer made up of some 20 Sun Unix workstations connected to each other via an HP switch on a 100BT network. The Beowulf was built by students for the purpose of teaching parallel computing and for computational physics research.

This multicomputer can be used as separate workstations or with any number of CPU's working on the same problem. We use MPI for the Message-Passing Interface needed for parallel computing. Usually MPI is called from within a C or Fortran program, yet we have also installed some "wrappers" called mpiJava (or sometimes MPJ) so that MPI can also be invoked from within a Java program.

We will give you some examples of Java programs using JavaMPI. Your job is to get the experience of running MPI, to understand some of the MPI commands within the programs, and then to run some timing experiments.

2 Setup

1. First you need an account on all the Beowulf computers. Although these computers run the same operating system as does the berry patch, you need separate accounts for the Beowulf. The system administrator must help you with that. You can contact the system administrator by e-mailing support@physics.orst.edu.

2. Next, make a subdirectory `mpiJava` in your home directory with the command
   ```
   > mkdir mpiJava
   ```

3. Once you have a Beowulf account, you need to configure it so that you can run MPI commands from the Unix command line and from within Java. Command line access requires that the operating system knows where to find the MPI commands that you and your program may issue. The places where the operating system looks for

commands are listed in a configuration file `.cshrc` that resides in your home directory. Since this file begins with a "dot", it is usually hidden from view and is read by the operating system only when you log on the computer. Likewise, the different places where Java looks for class files are listed in the `.cshrc` dotfile.

(a) The places where Unix looks for commands is called your *PATH*, and the places where Java looks for classes is called your *CLASSPATH*. Your *PATH* needs to include the entries for MPI, namely:
/usr/local/mpich-1.2.1/bin and */usr/local/mpiJava/src/scripts*.
Your *CLASSPATH* needs to include the three directories 1) */home/username*, 2) */home/username/mpiJava*, and 3) */usr/local/mpiJava/lib/classes*

(b) Since your `.cshrc` file controls your environment, having an error in this file can lead to a nonfunctional computer for you. And since the format is rather unforgiving, it is not hard to make mistakes. So first make a backup copy with the command:
`> cp .cshrc .cshrc_bk`.
You can use this backup file as reference, or copy it back to `.cshrc` if things get to be too much of a mess. Even then, you may need help from an administrator if your file gets messed up.

(c) Edit your `.cshrc` file so that line which has `setenv PATH` includes */usr/local/mpich-1.2.1/bin* and */usr/local/mpiJava/src/scripts*. Find a line containing `setenv PATH` and add these in after one of the colons, making sure to separate the names with colons.

(d) Edit your `setenv CLASSPATH` to include */home/username*, */home/username/mpiJava*, and */usr/local/mpiJava/lib/classes*. Here too use colons as separators.

(e) As an example, the `.cshrc` file for user `rubin` is

```
 # @(#)cshrc 1.11 89/11/29 SMI umask 022
setenv PATH /usr/local/bin:/opt/SUNWspro/bin:/opt/SUNWrtvc/bin:
/opt/SUNWste/bin:/usr/bin/X11:/usr/openwin/bin:/usr/dt/bin:/usr/ucb/:
/usr/ccs/bin/:/usr/bin:/bin:/usr/sbin/:/sbin:
/usr/local/mpich-1.2.1/bin:/usr/local/mpiJava/src/scripts: setenv
PAGER less setenv CLASSPATH
/home/rubin:/home/rubin/dev/java/chapmanjava/classes/:
/home/rubin/dev/565/javacode/:
/home/rubin/dev/565/currproj/:/home/rubin:/home/rubin/mpiJava:
/usr/local/mpiJava/lib/classes:
set prompt="%~::%m> "

# BEGINNING of automatic update setenv MANPATH
/usr/local/man:/usr/man:/opt/SUNWspro/man:/opt/SUNWspci/man # END
of automatic update
```

(f) Since dotfiles are read by the system when you first log on, you will have to log off and back on for your changes to take effect.

(g) Once you have loged back on, you can check the values of your `PATH` and `CLASSPATH` environment variables with the Unix commands

```
> echo $PATH
> echo $CLASSPATH
```

4. Let's now take a look at what has been done to the computers to have them run as a Beowulf cluster. On Unix systems the directory "/" ("slash") is the root or top directory.

5. Change directory to / (`cd /`) and you will see file names there which are the operating system, kernel, devices, and all. You may not be permitted to examine these files, which is a good thing since modifying them could cause real problems (it's the sort of thing that hackers do).

6. MPI is our local addition to the operating system. Accordingly, it has been placed in the directory `/usr/local`, where the first / indicates the root directory. Change directory to `/usr/local` and notice the directories `mpiJava` and `mpich-1.2.1` (`mpich` is a symbolic link to `mpich-1.2.1`). Feel free to explore these directories, but please do not try to write in them. You will notice that there are examples in `mpiJava`.

7. Copy the `examples` subdirectory into your personal mpiJava directory:

```
> cp -r examples /home/userid/mpiJava
```

   If you wish, try out some of the examples (not all have been tuned to work on our setup, however).

8. Copy the file `/usr/local/mpich/share/machines.solaris` to your home directory and examine it. This file contains a list of all of our workstations that are on the Beowulf cluster and can be utilized with MPI. Although this list may change as machines come and go, at present the list is:

```
> cat machines.solaris
# Change this file to contain the machines that you want to use #
to run MPI jobs on. The format is one host name per line, with
either #   hostname or hostname:n # where n is the number of
processors in an SMP. The hostname should # be the same as the
result from the command "hostname" daphy chris albert david erik
henri kirk paul pom rose rubin tom tomek cortez cpug #lady silas
emma #popcorn 11/13/01 withdrawn from cluster joe manuel
```

2.1 Run HelloWorldMPI.java

Now test the configuration by running
`/usr/local/mpiJava/examples/simple/HelloWorldMPI.java`
from your home directory (there is also a copy on the Web). This file contains the simple code in which each of the processors print `Hello World from processor #` followed by the rank of the processor sending the message. We'll talk more about rank below.

```
/*
  HelloWorldMPI.java: a simple MPI Program to demonstrate basics.
*/

import mpi.*;  //get the mpi classes
public class HelloWorldMPI
{ public static void main(String[] args) throws MPIException
    {
    int myrank = MPI.COMM_WORLD.Rank(); // what is my rank?
    MPI.Init(args); //start mpi for "args" machines
    System.out.println ("\n\t Hello World from processor #" + myrank );
    MPI.Finalize();  //end MPI
    }
}
```

Compile `HelloWorldMPI.java` using the usual Java compiler:

```
> javac HelloWorldMPI.java
```

Run this code on 2 processors using the JavaMPI command

```
> prunjava n HelloWorldMPI
```

where `n` is the number of processors. Processor 0 will be the master and processors 1-(n-1) will be slaves.

Optional: Modify the code so that only the slave processors say hello.
Hint: What do the slave processors have in common?

3 MPI Basics: HelloWorldMPI Explained

```
import mpi.*;
public class HelloWorldMPI
    {
    public static void main(String[] args) throws MPIException}
    {
```

The first thing done in HelloWorldMPI.java is to import the MPI libraries. As you write more complex code, you may also need to include other libraries, such as the Java IO, JAMA, SLATEC, etc. The class is defined next, and a main method, as is routine for java. Notice that main throws and MPIException, and probably should throw an an IOException to allow more in-depth error reports. The variable `args` in main is passed to it as whatever argument the user enters when they run the program from the command line. This is important because this argument is passed to the `MPI.Init` routine later where it defines how many processors you will be using.

`int myrank;`
The next part of `HelloWorldMPI.java` sets up the variables that MPI and the program

need. This program is simple and so the only variable we will use is `myrank`. This variable stands for the rank of the computer as will be explained below.

```
        // MPI Initialization
        MPI.Init(args);
        myrank = MPI.COMM_WORLD.Rank();
        System.out.println ("\n\t Hello World from processor #" +myrank );
```

The next phase of the initialization is to initialize MPI. This is accomplished by the `MPI.Init(args)` command. The args variable is passed to MPI.Init and defines how many processors you are requesting. Remember that each processor is in parallel and thus runs its own code after the MPI.Init argument. The next statement sets the myrank variable for each of the processors. This is critical to how parallel computing works.

The processor that the program is executed on is called the *host* or *master*, and all other machines are called *guests* or *slaves*. The host is always has a myrank value of zero (as an integer). All the other processors, based on who responds first, have their processor numbers assigned to myrank. So, the variable myrank is 1 for the first slave to respond, 2 for the second, an so on. This needs to be concreted in your mind to further understand the `if` statements in more complex programs later. Then, because all the processes are in parallel, they each run the next line, a printout command. Because only the host has a screen, all messages from the other processors are printed on the host's screen. You may notice that processor 5 responds before processor 3, for example. Remember, all processes are independent and so whichever processor completes the code first is printed first. If things need to be done in order, use the `MPI.COMM_WORLD.Barrierz` command. This command acts as a stop sign, and makes all the processors wait until they are all at that point in the program before continuing.

The last basic MPI command is `MPI.Finalize`. This command is at the end of the main routine, and is the counterpart of `MPI.Init`. The `MPI.Finilize` command tells all the slaves that the program is done, and terminates MPI in a nice way.

Now, with an understanding of how to initialize MPI, you are ready to move on to sending and receiving date among processors.

4 Basic Sending and Receiving: Hellow.java

The next example does exactly the same thing as `HelloWorldMPI.java`, but in a different way. In this example, the master sends the message `Hello, there` to machine 1, and then prints the message out when it gets returned from machine 1. The difference from before is that `Hellow.java` uses the basic send and receive commands of MPI.

```
import mpi.* ;
class Hellow {
    static public void main(String[] args)
    throws MPIException
    {
        MPI.Init(args) ;
        int myrank = MPI.COMM_WORLD.Rank() ;  // what is my rank?
        if(myrank == 0)
        {                     // am I the master?
            char [] message = "Hello, there".toCharArray() ;
//Send variable "message",array offset,no items,variable type
//, destination, message tag
            MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR, 1, 99) ;
        }
        else
        {
            char [] message = new char [20] ;
            //mpi receive matches send
                MPI.COMM_WORLD.Recv(message, 0, 20, MPI.CHAR, 0, 99) ;
            //printed out on 1 only
            System.out.println("received:" + new String(message) + ":") ;
        }
    MPI.Finalize();
    }
}
```

When you run this code you should get processor 1 to say hello.
Optional: Modify the code so that all processors say hello.

5 More MPI Basics

Sending and receiving data makes up the heart of parallel computing. The slave processors need to report back the data that they have computed to the host. As we just learned, there are some basic MPI commands, Send and Recv (receive) to do that. More powerful commands include Bcast (broadcast) and the Allreduce, which we will explore next. First, understand that if one processor runs code to send data, another processor must run code to receive data. Otherwise the one sending simply waits indefinitely! Likewise, if a processor has a receive command, that processor will wait indefinitely for data. Always keep this in mind as you send and receive data.

Now look at the `MpiPi.java` program. It is a simple program that computes π in parallel using the "stone throwing" technique. In it you will find the basics that were discussed above, with the inclusion of a few new MPI commands.

The first new MPI command in `MpiPi.java` is `MPI.Wtime`. As might be evident, this command returns the wall time in seconds. This can be useful when computing speedup curves, which will be discussed later. Before we talk about the communication commands, it is important to remember that
MPI will only allow arrays to be passed.

When you write your own code, make sure that the data you are passing is stored in arrays!

The next MPI command is `MPI.COMM_WORLD.Bcast()`. This command broadcasts the data from one processor to all the others. So, in this program, the number of iterations is entered on the host, and then the host broadcasts this to the other computers. They replace their value of `n` with the one received from the host.

The final new command is `MPI.COMM_WORLD.Allreduce()`. This is a glorified broadcast command. It takes a variable (`mypi`) from each of the processors and then performs an operation on them (`MPI.SUM`) and then broadcasts the result in the form of another variable (`pi`).

Compile and experiment with the code.

Optional:

1. Notice how long each run takes and how accurate the answers are. Do round off errors enter in? What could you do to get a more accurate number?

2. There are two different kinds of parallel programs, one in which the host acts just like a slave and one in which the host does nothing at all but control the action. These are called *active* and *lazy* hosts respectively. Is MpiPi.java a lazy host or active host? How could you change it?

3. What does a plot of the time verses number of processors for a calculation of π look like? What does the speedup graph look like? (Plot the time divided by the time for one processor versus the number of processors.)

6 MpiPi.java Listing

```
/*  MpiPi.java: a simple parallel program to compute pi
        converted to mpijava from cpi.c
*/
import java.io.*;
import mpi.*;
//Version 6-3-02
public class MpiPi {
    public static void main(String[] args)
    throws MPIException, IOException
        {
        int done = 0, myrank, worldsize, i;
        int [] n = new int [1];
        double PI25DT = 3.141592653589793238462643;
        double h, sum, x;
        double startwtime = 0.0, endwtime;
        double [] mypi = new double [1];
        double [] pi = new double [1];

        MPI.Init(args);
        myrank = MPI.COMM_WORLD.Rank();
        worldsize  = MPI.COMM_WORLD.Size();

        n[0]=100000000;
        while (done!=1)
                {
                if (myrank == 0)
                        {
                        System.out.println("\nEnter the number of intervals: (0 quits) ");
                        InputStreamReader keyboard = new InputStreamReader(System.in);
                        BufferedReader keyin = new BufferedReader(keyboard);
                        n[0] = Integer.parseInt( keyin.readLine() );
                        startwtime = MPI.Wtime();
                        }
//Bcast variable n,array offset,no items,variable type, destination, message tag
                MPI.COMM_WORLD.Bcast(n, 0, 1, MPI.INT , 0) ;
                if (n[0] == 0) done = 1;
                else
                        {
                        h   = 1.0 / (double) n[0];
                        sum = 0.0;
                        for (i = myrank + 1; i <= n[0]; i += worldsize)
                                {
                                x = h * ((double)i - 0.5);
                                sum += (4.0 / (1.0 + x*x));
                                }
                        mypi[0] = h * sum;
//Allreduce variable mypi,array offset, place to store result pi, array offset,
//count, variable type, operation to be preformed
                        MPI.COMM_WORLD.Allreduce( mypi, 0, pi , 0,1, MPI.DOUBLE, MPI.SUM );
                        if (myrank == 0)
                                {
                                System.out.println("\nPI is approximately " + pi[0]
                                 + " \tError is " + Math.abs(pi[0] - PI25DT) + "\n");
                                endwtime = MPI.Wtime();
                                System.out.println("wall clock time = "
                                + (endwtime-startwtime) +"\n");
                                }
                        }
                }
        MPI.Finalize();
        }
}
```

7 A Bad Example: TuneMPI.java

Recall the `Tune` program that we experimented with previously to determine how memory access affects the running time of programs. You should also recall that as the size of a matrix was made larger, the execution time increased more rapidly than did the number of operations the program had to perform, the extra increase coming from communication time. Since parallel programming often involves a balance between computation and com-

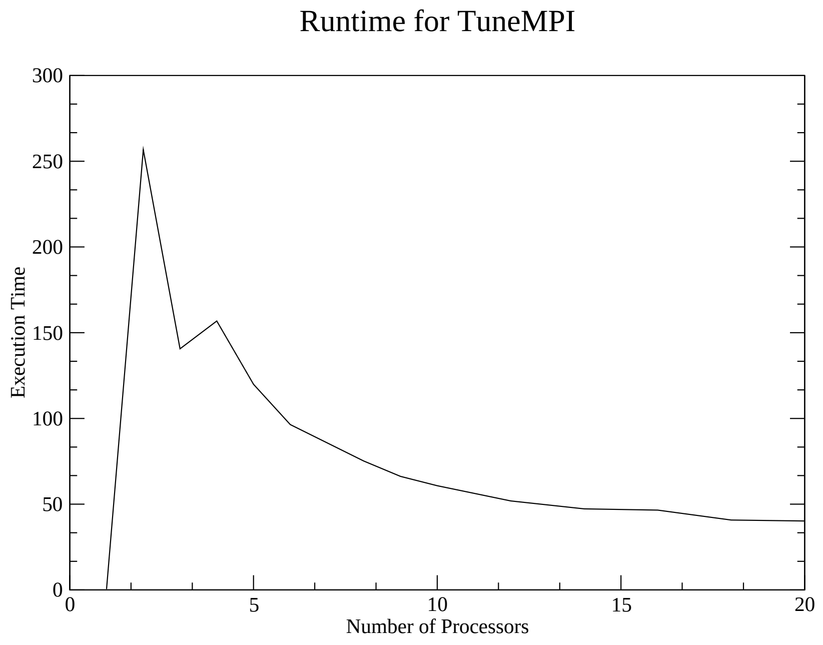

Figure 1: Execution time *versus* the number of processors for the `TuneMP` program. Note that here the single processor result is very small since it is the time *without* running MPI. A single processor running MPI takes approximately the same time as two processors running MPI.

munication (although in this case communicating with another computer rather than with the disk), the `Tune` program is also be a good teaching tool for parallel computations.

Below you will find a listing of the program `TuneMPI.java`. One running of this code yielded the speedup curve shown in Fig. 7 It is the basic `Tune` program with modifications to run under MPI. The basic modification is to have each row of its large matrix multiplication performed on a different processor (called *rank* as explained above):

$$[H]_{N\times N}[\Psi]_{N\times 1} = \begin{bmatrix} \Rightarrow & \text{rank 1} & \Rightarrow \\ \Rightarrow & \text{rank 2} & \Rightarrow \\ \Rightarrow & \text{rank 3} & \Rightarrow \\ \Rightarrow & \text{rank 1} & \Rightarrow \\ \Rightarrow & \text{rank 2} & \Rightarrow \\ \Rightarrow & \text{rank 3} & \Rightarrow \\ \Rightarrow & \text{rank 1} & \Rightarrow \\ & \vdots & \end{bmatrix}_{N\times N} \times \begin{bmatrix} \psi_1 & \Downarrow \\ \psi_2 & \Downarrow \\ \psi_3 & \Downarrow \\ \psi_4 & \Downarrow \\ \psi_5 & \Downarrow \\ \psi_6 & \Downarrow \\ \psi_7 & \Downarrow \\ \vdots & \end{bmatrix}_{N\times 1} . \tag{1}$$

Here the assignment of a row to each processor continues until we run out of processors, and then starts all over again. Since this multiplication gets repeated for a number of iterations, this is the most computationally intensive part of the program, and it makes sense to parallelize this load.

However, life is not that simple. If the matrix size is made small enough so as to avoid memory page conflicts, then the computational demand of the program is too low to work out well running on a parallel computer. Accordingly, to slow down the program we have inserted an inner `for` look over `k` that takes up time but accomplishes little (sound familiar?). This slowdown permits you to accumulate more realistic timings.

7.1 TuneMPI.java Listing

```
/*  TuneMPI.java: MPI version of Tune with slowdown inner loop
converted to MPI by Kristopher Wieland based on tune.f in
"Computational Physics" by Landau and Paez copyrighted Paul J Fink
and Rubin Landau, Oregon State Univ */ import java.io.*; import
mpi.*;   //Get mpi routines
//use for N X N Matrix speed tests with MPI
public class TuneMPI { public static void main(String[] args)
throws MPIException //args is no of machines
    {
    final int N=200, MAX = 15; //matrix size, no of iterations
    final double ERR = 1.0e-6;
    double dummy = 2.;
    int h = 1, myrank, nmach;   //mpi: whoami, total no
    int i, j, iter = 0;
    double [][] ham = new double [N] [N]; double [] ovlp = new double [1];
    double [] coef  = new double [N]; double [] sigma = new double [N];
    double [] ener = new double [1]; double [] err = new double [1];
//  MPI variables. "my": specific to each CPU, can have own values
    double [] mycoef  = new double [1]; double [] timempi = new double [2];
    double [] mysigma = new double [1]; double [] myener = new double [1];
    double [] myerr = new double [1]; double [] myovlp = new double [1];
    double  step=0.,time = 0.0;
    // MPI Initialization
    MPI.Init(args);     //args from main = no machines
    myrank = MPI.COMM_WORLD.Rank(); //assign rank number for this CPU
    nmach  = MPI.COMM_WORLD.Size(); //determine no machines actually working
    MPI.COMM_WORLD.Barrier();       //syncronize: make all processes wait here
        if (myrank == 0)            //0 for host or master
            {timempi[0] = MPI.Wtime();  //mpi timing, done only on master
            // Store initial time
            time = System.currentTimeMillis();
            }
        // set up Hamiltonian and starting vector
        System.out.println ("\n\t Processor " +myrank + " checking in...");
        for ( i = 1; i < N; i++)
            {
            for ( j = 1; j < N; j++)
                {
                if (Math.abs(j-i) >10) {ham[j][i] = 0.0;}
                else  ham[j][i] = Math.pow(0.3, Math.abs(j-i));
                }
            ham[i][i] = i   ;
            coef[i] = 0.0  ;
            }
        coef[1] = 1.0  ;
        //  start iterating towards the solution
        err[0] = 1.0;
        iter = 0 ;
    if (myrank==0)System.out.println ("Iteration #\tEnergy\t\tERR\t\tTotal Time ");
        while (iter <MAX && err[0] > ERR)
            {//--------------------start while loop--------------------
            iter = iter + 1  ;
            mycoef[0]=0.0;      //initialize for each machine, for each loop
            ener[0] = 0.0; myener[0] = 0.0 ;
            ovlp[0] = 0.0; myovlp[0] = 0.0 ;
            err[0] = 0.0  ; myerr[0] = 0.0;
            for ( i= 1; i < N; i++)
                {
                //h =key variable to pick cpu no, ea row on different cpu
                h = (int)(i)%(nmach-1)+1 ;
                if (myrank == h)
                    {
                    myovlp[0] = myovlp[0]+coef[i]*coef[i] ;
                    mysigma[0] = 0.0;
                    for ( j= 1; j < N; j++)

                        {mysigma[0] =  mysigma[0] + coef[j]*ham[j][i];}
                    myener[0] =  myener[0]+coef[i]*mysigma[0] ;
                    // mpi Send + matching Receive
                    //(array offset,no items,type,destination,item sent)
                    MPI.COMM_WORLD.Send(mysigma,0,1,MPI.DOUBLE,0,h);
                    }
                if (myrank == 0)
                    {
                    MPI.COMM_WORLD.Recv(mysigma,0,1,MPI.DOUBLE,h,h);
                    sigma[i]=mysigma[0];
                    }
                }
            //mpi sum all machines' values and then broadcast so same on all machines
            MPI.COMM_WORLD.Allreduce( myener, 0, ener , 0,1, MPI.DOUBLE, MPI.SUM );
            MPI.COMM_WORLD.Allreduce( myovlp, 0, ovlp , 0,1, MPI.DOUBLE, MPI.SUM );
            MPI.COMM_WORLD.Bcast(sigma, 0, N-1, MPI.DOUBLE , 0) ;
            ener[0] = ener[0]/(ovlp[0]);
            for (  i = 1; i< N; i++)
                {
                h = (int)(i)%(nmach-1)+1 ;
                if (myrank == h)
                    {
                    mycoef[0] = coef[i]/Math.sqrt(ovlp[0]) ;
                    mysigma[0] = sigma[i]/Math.sqrt(ovlp[0]) ;
                    MPI.COMM_WORLD.Send(mycoef,0,1,MPI.DOUBLE,0,nmach+h+1);
                    MPI.COMM_WORLD.Send(mysigma,0,1,MPI.DOUBLE,0,2*nmach+h+1);
                    }
                if (myrank == 0)
```

```
                {
                MPI.COMM_WORLD.Recv(mycoef,0,1,MPI.DOUBLE,h,nmach+h+1);
                MPI.COMM_WORLD.Recv(mysigma,0,1,MPI.DOUBLE,h,2*nmach+h+1);
                coef[i]=mycoef[0];
                sigma[i]=mysigma[0];
                }
            }
        MPI.COMM_WORLD.Bcast(sigma, 0, N-1, MPI.DOUBLE , 0) ;
        MPI.COMM_WORLD.Bcast(coef, 0, N-1, MPI.DOUBLE , 0) ;
        for ( i = 2; i < N ; i++)
            {h = (int)(i)%(nmach-1)+1 ;
            if (myrank == h)
                {
                step = (sigma[i] - ener[0]*coef[i])/(ener[0]-ham[i][i]) ;
                mycoef[0] = coef[i] + step ;
                myerr[0] =  myerr[0]+Math.pow(step,2);
                                  for ( int k= 0; k <= N*N; k++)// slowdown loop
                        {dummy = Math.pow(dummy,dummy);
                         dummy = Math.pow(dummy,1./dummy);}
                MPI.COMM_WORLD.Send(mycoef,0,1,MPI.DOUBLE,0,3*nmach+h+1);
                }
            if (myrank == 0)
                {
                MPI.COMM_WORLD.Recv(mycoef,0,1,MPI.DOUBLE,h,3*nmach+h+1);
                coef[i]=mycoef[0];
                }
            }
        MPI.COMM_WORLD.Bcast(coef, 0, N-1, MPI.DOUBLE , 0) ;
        MPI.COMM_WORLD.Allreduce( myerr, 0, err , 0,1, MPI.DOUBLE, MPI.SUM );
        err[0] = Math.sqrt(err[0])   ;
        if (myrank==0)
            {
            System.out.println ("\t#"+iter+"\t"+ener[0]+ "\t" +err[0]
            + "\t"+(System.currentTimeMillis() -time)/1000);
            }
        }//-----end whileloop--------------------------
    // output elapsed time
    if (myrank == 0)
        {
        time = (System.currentTimeMillis() - time)/1000;
        System.out.println("\n\tTotal time = "+ time + " s");
        timempi[1] = MPI.Wtime();
        System.out.println("\n\tMPItime= "+ (timempi[1]-timempi[0]) + " s");
        }
MPI.Finalize();     //stop mpi
}
}
```

7.2 Running TuneMPI.java

1. Issue the command

```
> prunjava 2 TuneMPI
```

This is your base program. It will use one processor as the master and another one to do the work. To determine the speedup with multiple processors, you issue the same command with any number of processors (`prunjava 1...` leads to an error message as there is no one to do the work).

2. Since you are already familiar with the `Tune.java` program, find the old scalar one, or modify the present one so that it runs on only one processor. Run the pure scalar version of `TuneMPI` and record the time it takes. Note that communications costs are so high with message passing systems, that it may require the use of several processors to beat this time!

3. Open another window so that you can watch the processing of your MPI jobs on the master computer.

4. While the code is running, a file beginning with `PI` should appear.

   It lists all the machines running your mpiJava job. For example:

```
> more PI*
rubin 0 /home/rubin/mpiJava/TuneMPI.jig albert 1
/home/rubin/mpiJava/TuneMPI.jig david 1
/home/rubin/mpiJava/TuneMPI.jig erik 1
/home/rubin/mpiJava/TuneMPI.jig henri 1
/home/rubin/mpiJava/TuneMPI.jig kirk 1
/home/rubin/mpiJava/TuneMPI.jig paul 1
/home/rubin/mpiJava/TuneMPI.jig pom 1
/home/rubin/mpiJava/TuneMPI.jig
```

   Once your job is complete, this file is removed automatically. If you do terminate MPI prematurely, you will need to clean up these files along with the processes that could be still running.

5. To look at the processes in more detail while running, try issuing several versions of the Unix processes command:

   `> ps` or `> ps -ef` or `> ps -a` or `> /bin/ps -ef`

   A typical output of `ps` is

```
  PID TT       S  COMMAND
23583 pts/1    S  -tcsh
24203 pts/1    S   /bin/sh /usr/local/mpiJava/src/scripts/prunjava 4 TuneMPI
24206 pts/1    S   /bin/sh /usr/local/mpiJava/src/scripts/prun 4 TuneMPI.jig
24207 pts/1    S   /bin/sh /usr/local/mpich-1.2.1/bin/mpirun -pg -np 4 TuneMPI.jig
```

```
24289 pts/1    S   /usr/java/bin/../jre/bin/../bin/sparc/native_threads/java TuneMPI
24298 pts/1    S   /usr/java/bin/../jre/bin/../bin/sparc/native_threads/java TuneMPI
24299 pts/1    S   rsh albert -l rubin -n /home/rubin/mpiJava/TuneMPI.jig rose
24300 pts/1    S   rsh david -l rubin -n /home/rubin/mpiJava/TuneMPI.jig rose
24301 pts/1    S   rsh erik -l rubin -n /home/rubin/mpiJava/TuneMPI.jig rose
```

(a) This shows that the `prunjava 4` command we issued from the command line to start JavaMPI has issued the `prun` command, which, in turn, has issued the MPI command `mpirun`. That, in turn, has led to some Java native threads (lightweight Java processes that can handle individual chores).

(b) We also see that the master has started remote shells (`rsh`) running on the named computers (`albert, david` and `erik`), and that they were started from the computer `rose`. You too can issue the `rsh` command to run on different Beowulf machines.

6. You now want to collect data for a plot of running time versus number of machines. Make sure your matrix size is reasonable but not too large; we found that `N=200` works well.

7. Run `TuneMPI` on a variable number of machines, starting at 2, until you find no appreciable speedup (or an actual slowdown) with an increasing number of machines (see warning below).

8. *Warning:* While you will do no harm running on the Beowulf when others are also running on it, in order to get meaningful speedup graphs, you really need to have the cluster all to yourself. Otherwise, the time it takes to switch around jobs and to setup and drop communications tends to remove an parallel advantage.

9. You can ask the system to check who is on some other machine (the names are in the `machines.solaris` file) by invoking a remote shell and asking who. For examples, to check on `rose, rubin` and `albert`:
   ```
   > rsh rose who
   > rsh rubin who
   > rsh emma who
   ```
   You can also make a script to check it for you. Here is one such script that checks who is on the machines. You can copy this command to a file and then make the file executable (`chmod +x` *filename*).
   ```
    ksh 'for h in daphy albert chris david erik henri kirk paul pom
   rose rubin tom tomek cortez cpug joe manuel silas emma; do rsh $h
   who;done'
   ```

10. Try large and small values of matrix size and see how it affects the speedups. Remember, once your code is communications bound due to memory access, distributing it over many processors should only make things worse!

8 A Good Example: PhaseMPI.java

PhaseMPI.java

```java
import java.io.*;
import mpi.*;
//30 Apr 2002
//Au-Ni binary phase diagram
//Set Ni A component, Au B component, Solid A phase, Liquid B phase


public class PhaseMPI {
    public static void main(String[] args) throws MPIException
    {
        PrintStream pfout = null;
        PrintStream iout = null;
        int guest_processor=1, host_processor=0, myrank, worldsize, con_res = 0, h=0;
        double temp = 0, T_STEP, T_MAX, T_START;
        MPI.Init(args);
        myrank = MPI.COMM_WORLD.Rank();
        worldsize  = MPI.COMM_WORLD.Size();
        double timer_array[] = new double [2];
        System.out.println("\n"+myrank+": Started");
        MPI.COMM_WORLD.Barrier();
        timer_array[0] = MPI.Wtime();

        if ( myrank == host_processor )
//I am the host processor
        {
            System.out.println("\n\n MPI Started \n");
            try
            {
                FileOutputStream fout = new FileOutputStream("AuNi1.dat");
                pfout = new PrintStream(fout);
                FileOutputStream infoOut = new FileOutputStream("MPIinfo0.dat");
                iout = new PrintStream(infoOut);
            }
                catch(IOException ioe)
            {
                System.err.println("Error in FileOutput...."+ioe);
            }
        }
        //these define the start and stop of the temperature loop in Kelvin
        T_START = 300;
        T_MAX = 1600;
        T_STEP = 1;
        //con_res defines the steps in consintration, the step size across
        //the concentration axis is 1/con_res
        con_res = 1000;
        for (temp = T_START; temp < T_MAX; temp += T_STEP)
        {
            if (myrank == guest_processor) //Other computers
            {
                CALPHAD auni=new CALPHAD();
                TestG test=new TestG();
                test.setOmega(21689,0);
                test.setT(temp);
                double G0Aa=auni.getG0Aa(temp);
                test.setG0Aa(G0Aa);
                double G0Ab=auni.getG0Ab(temp);
                test.setG0Ab(G0Ab);
                double G0Ba=auni.getG0Ba(temp);
                test.setG0Ba(G0Ba);
                double G0Bb=auni.getG0Bb(temp);
                test.setG0Bb(G0Bb);
                double results[]=test.runTest(con_res);
                MPI.COMM_WORLD.Send(results,0,2*con_res,MPI.DOUBLE,host_processor,10);
            }
            if (myrank == host_processor) //Host Computer
            {
                double receive[] = new double [2*con_res];
                if(temp%10==0){System.out.println(temp);}
                double TdegC=temp-273.0;
                MPI.COMM_WORLD.Recv(receive, 0 ,2*con_res, MPI.DOUBLE, guest_processor, 10);
                while(receive[h]!=0)
                {
                    pfout.println(receive[h] + "\t" + TdegC);
                    h++;
                }
                h=0;
            }
            //If we reach the end of the processors,
            //then reset the current processor to 1 otherwise advance
            if (guest_processor == (worldsize-1))
            {
                guest_processor=1;
            }
            else
            {
                guest_processor++;
            }
        }
        timer_array[1] = MPI.Wtime();
        MPI.Finalize();
        if (myrank==0)
        {
            iout.println("\n\n\t Total Processors: "+worldsize+" (including host),\n");
            iout.println("\t Processors computing: "+ (worldsize-1));
            iout.println("\n\t MPI Total time: "+(timer_array[1]-timer_array[0])+"\n");
        }
    }
}

```

8.1 Introduction to PhaseMPI

Above is the listing of a program, `PhaseMPI.java` that calculates the binary phase diagram for metallic alloys, that is, it plots the composition versus temperature. This type of program permits a "nearly-trivial" parallelization in which the same code is run on different machines, each with its own range of temperatures (a truly trivial parallelization would run the same parameter set, as would happen, for example, with a Monte Carlo calculation).

`PhaseMPI.java` is a good example of how some problems scale well as the number of processors increases. It scales well since the computation is essentially a complicated mapping in which you can run different parameter ranges on different machines and thereby keep each machine busy for quite some time without any need to communicate.

Although the physics in `PhaseMPI.java` takes some explaining (see the section entitled *The Physics Problem*), for those readers not taking the "advanced" course, you may just want to run `PhaseMPI.java` to see the kind of speedup plot produced by a mapping. You adjust the computational intensity of the problem simply by changing the step size `T_STEP` or the temperature maximum `T_MAX`. Plots of composition *versus* temperature for Temperature steps of 1 K, 10 K, and 200 K appear to work well.

Alternatively, the non advanced readers can get the same effect by modifying their logistics program to scan over a very fine grid of growth parameters, and run the program on an arbitrary number of processors. In either case, the results should scale better than `TuneMPI`. But again be aware that your plot will change (for the worse) if other users are on the same cluster machines as you are.

9 Advanced Computational Lab: the Physics Problem

The intermixing of several metals to form alloys is a common practice in metallurgy and manufacturing, usually done to improve the structural qualities of the alloy. Rarely are pure materials used for practical applications since are they expensive to manufacture and often have less desirable properties than alloys.

The key to understanding the properties of alloys is the phase diagram, an example of which is shown in Fig. 9. A phase diagram shows the different phases present in a material as a function of temperature and composition. For pure materials, the phase diagram is one dimensional, or *uniary*, showing just melting points. For an alloy containing a mixture of two elements, the phase diagram is two dimensional or *binary* with the temperature as the ordinate and the composition as the abscissa.

Our **problem** is to compute the binary phase diagram from the thermodynamic equations. While drawing a binary phase diagram by hand is tedious, it is a straight-forward task for a computer.

10 Review: the Thermodynamic Equations for Alloys

Because manufacture of most alloys is done at constant pressure, we shall calculate constant pressure phase diagrams. In particular, we shall compute the Gibbs free energy since it determines the amount of and types of phases present as a function of composition and temperature.

Consider a single component solid. Equilibrium thermodynamics tells us that a complete description is possible by considering the Gibbs free energy $G(T, p, N)$ as a function of temperature T, pressure p, and the number of atoms N. Temperature is convenient since it is easier to measure and vary than the than the entropy S. Likewise, pressure is easier to

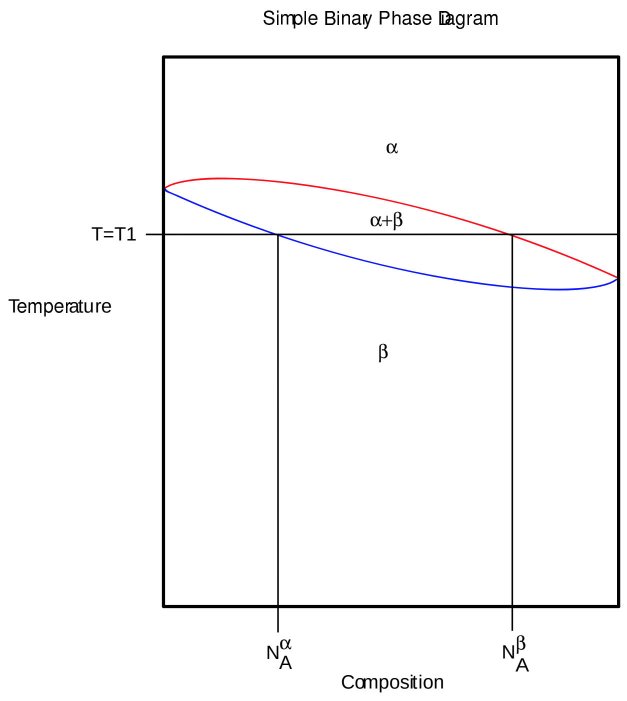

Figure 2: A simple binary phase diagram.

vary than volume V, and most crystalline metallic alloys are formed at constant pressure (usually 1 atm). Finally, the number of atoms present N is easier to vary than the chemical potential μ.

Recall from thermodynamics that for the Gibbs free energy,

$$\begin{aligned} G &= F + pV, & (2) \\ \text{and} \quad dG &= -SdT + Vdp + \mu dN, & (3) \end{aligned}$$

where F is the Helmholtz free energy.
Or in terms of the enthalpy

$$G = H - TS. \tag{4}$$

Recall now from statistical mechanics that the entropy S is related to the probability P by

$$S = k \ln(P), \tag{5}$$

where k is Boltzmanns constant (8.99×10^{23}).

10.1 Entropy of Mixing

Consider a box containing two ideal gasses, A and B. The total volume V of the box is divided into two portions V_A and V_B,

$$V = V_A + V_B. \tag{6}$$

If the box was initially empty and an A atom was placed randomly in the box, the probability of it being in V_A would be V_A/V. If a second A atom were placed in the box, the probability of both atoms being on side A would be $(V_A/V)^2$. And so the probablity of n A atoms being in volume V_A is $(V_A/V)^{n_A}$. Likewise, the probablity of n B atoms being in volume V_B is $(V_B/V)^{n_B}$.

Let us now populate the box with n A atoms and n B atoms, each confined totheir respective sides by a partition. If we now remove the partition, the probability of finding all A atoms in V_A and all B atoms in V_B is:

$$P_1 = \left(\frac{V_A}{V}\right)^{n_A} \times \left(\frac{V_B}{V}\right)^{n_B}, \tag{7}$$

where P_1 is the probability of the box being in the unmixed state. The probability of a homogeneous mixture is $P_2 = 1 - P_1$.

Consider now one mole of gas evenly divided between types A and B and with equal volumes $V_A = V_B = V/2$. The probability of an unmixed state is then

$$\begin{aligned} P_1 &= (1/2)^{10^{23}} \times (1/2)^{10^{23}} \simeq 10^{-10^{23}} \simeq 0, & (8)\\ \Rightarrow \quad P_2 &\simeq 1. & (9) \end{aligned}$$

Even for 125 atoms, the probability of an unmixed state is $(1/2)^{125} \simeq 2 \times 10^{-38}$, the limit of precision for sinfle precision. So to a good approximation,

$$P_1 \simeq 0, \quad P_2 \simeq 1. \tag{10}$$

We now calculate the entropy of mixing gases A and B by calculating the difference in entropy between the mixed and the unmixed states:

$$\begin{aligned} \Delta S &= S_2 - S_1 = k \ln\left(\frac{P_2}{P_1}\right) \simeq k \ln\left(\frac{1}{P_1}\right) = -k \ln P_1 & (11)\\ &= -k \ln \left(\frac{V_A}{V}\right)^{n_A} \left(\frac{V_B}{V}\right)^{n_B} & (12)\\ &= -kn_A \ln\left(\frac{V_A}{V}\right) - kn_B \left(\frac{V_B}{V}\right) & (13) \end{aligned}$$

Because we are dealing with ideal gasses at constant temperature and presssure,

$$\begin{aligned} \frac{V_A}{V} &= \frac{n_A}{n} = N_A \quad \text{and} & (14)\\ \frac{V_B}{V} &= \frac{n_B}{n} = N_B = 1 - N_A, & (15) \end{aligned}$$

where N_A and N_B are the concentrations of A and B atoms. The entropy of mixing is, accordingly,

$$\begin{aligned}\Delta S &= -R[N_A \ln N_A - N_B \ln N_B] && (16)\\ &= -R[N_A \ln N_A - (1-N_A)\ln(1-N_A)], && (17)\end{aligned}$$

where $R = nk$ is the gas constant 8.314.

10.2 Ideal Gases

The entropy of mixing for ideal gases A and B is

$$G = G_A^0 N_A + G_B^0 + T\Delta S_{\text{mix}}, \tag{18}$$

where N_A is the concentration of A atoms, N_B is the concentration of B atoms, T is the absolute temperature, ΔS_{mix} is the entropy of mixing, and G_A^0 and G_B^0 are the Gibbs free energies for atoms A and B. Substituting for the entropy of mixing gives

$$G = G_A^0 N_A + G_B^0 N_B + RT\left(N_A \ln N_A c + N_B \ln N_B\right). \tag{19}$$

If we take a differential of G with respect to N_A or N_B we obtain the partial molar free energies:

$$\begin{aligned}\frac{dG_A}{dN_A} = \overline{G}_A &= G_A^0 + RT\ln N_A, && (20)\\ \frac{dG_B}{dN_B} = \overline{G}_B &= G_B^0 + RT\ln N_B. && (21)\end{aligned}$$

The total Gibbs free energy is now

$$G = N_A\overline{G}_A + N_B\overline{G}_B. \tag{22}$$

Note, in the ideal, non-interacting case, the heat of mixing (also known as the enthalpy of mixing) is zero.

10.3 Non-Ideal Gases

We need now to include the interaction between A and B and atoms. We use the "regular solution" for the enthalpy of mixing,

$$\Delta H_{\text{mix}} = \Omega N_A N_B, \tag{23}$$

where Ω is the weight for any given phase. The total free energy for any given phase now becomes

$$G = G_A^0 N_A + G_B^0 N_B + RT\left(N_A \ln N_A + N_B \ln N_B\right) + \Omega N_A N_B. \tag{24}$$

The partial molar free energies are

$$\overline{G}_A = G_A^0 + RT\ln N_A + \Omega(1-N_A)^2, \tag{25}$$
$$\overline{G}_B = G_B^0 + RT\ln N_B + \Omega(1-N_B)^2. \tag{26}$$

10.4 Determining Phase Equilibrium

We assume that each phase of our alloy has its own Gibbs free energy. In a binary alloy these are functions of temperature and composition. As shown in Fig. 10.4, a plot is next made of the free energies at constant temperature for each phase as functions of composition. At any point along the free energy curve for one of the phases, a tangent line is drawn to a point on the curve and continured to the two vertical axes. The point where the tangent lines intersect the axes of the graph determines the partial molar free energy of the phase and element.

Equilibrium between phases occurs when:

1. the partial molar free energy of component A in phase α equals the partial molar free energy of component A in phase β, and
2. the partial molar free energy of component B in phase α equals the partial molar free energy of component B in phase β.

Graphically, this occurs when two tangent lines lie on top of one another. The composition where this "common" tangent line touches each of the two phases is the critical composition at a constant temperature. These compositions N_A^α and N_A^β are then transferred to the phase diagram. Repeating this process point-by-point eventually determines the entire phase diagram.

11 Computational Solution

The repetitive nature of converting one plot into another process clearly makes it an excellent candidate for computational methods, especially since several thousand points are needed to construct a single phase diagram. Yet the graphical process of drawing common tangent lines, while straight forward for people, does not automate well on a computer. However, we are aided by having set of strict rules for the computer to follow.

The algorithm is straightforward:

1. A temperature T is set.
2. The composition N_A^α of the α phase is set.
3. The composition N_A^β of the β phase and the partial molar free energies are compared until a match is found.

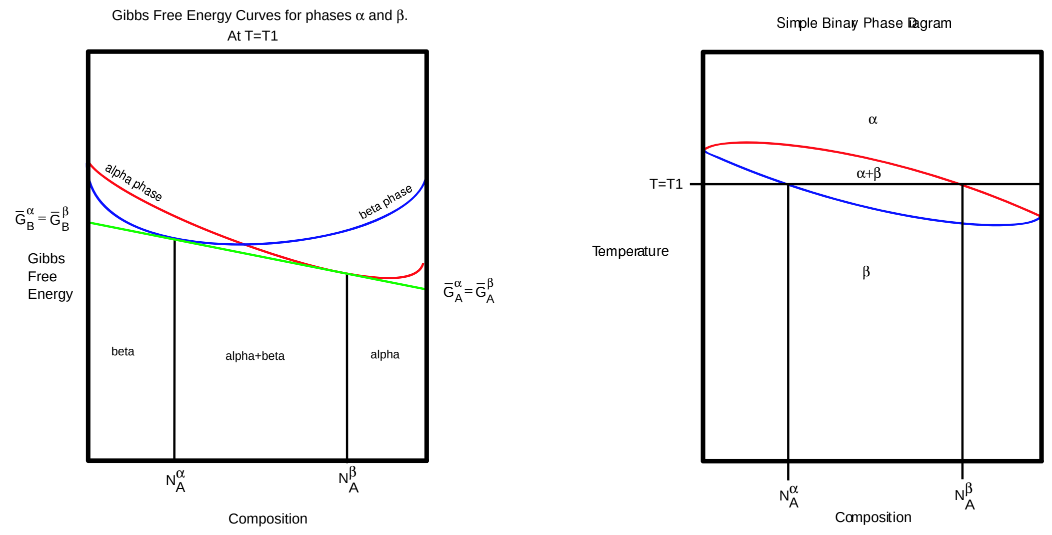

Figure 3: Gibbs free energy curves and binary phase diagram.

4. Once a match is found, the compositions of α and β phases and the temperature are written to a file.

5. The cycle repeats until all temperatures have been checked.

6. The point $N_A^\alpha = N_A^\beta = 0$ is avoided due to mathematical problems. (However, this point should match the single-element phase transformations, for example, a melting point.)

There are various insights that this computation provides regarding the alloying process. We can "tune" the code by adjusting the number of temperature steps, as well as the number of composition steps. Additional control comes from comparing partial molar free energies with each other. This latter tuning is called the *fuzz factor* because it determines how fuzzy the lines in the phase diagram will be. Best results were obtained using 1 K temperature steps, 0.1% composition changes, and ± 20 *fuzz*.

12 Parallel Implementation

Even the rather crude calculations needed to do a homework problem for temperatures ranging from room temperature to 1800 C, required three hours of computing time. More realistic computations, of the type needed in industrial settings, could take orders of magnitude more time. Because of the independent steps in the process, it is straightforward to break the computation up so that it will run in parallel.

PhaseMPI has all the basics features that we have seem in `tuneMPI` in addition to two new features: file IO and looping. The free energy graphs are determined from previously published tables, and these must be read in. The looping over parameters is what we distribute over machines.

There are a number of ways to paralellize this code. One is to send each processor a temperature value, and have it run through the composition calculations. Another is to divide the temperature range by the number of processors, and have each processor work on a specific range of temperatures. Because our solution only exists in some regions, we chose the first approach.

Some of the challenges encountered in parallelizing the code were data sharing and MPI communications. Having each processor work on the solution for individual temperatures was easy enough, but the results for different temperatures had to be collected into one place. We solved this problem by having one processor open a file for writing, and all the other processors send the temperature information to the processor with the open file.

Of course, doing this file IO means dealing with MPI communications, and that is a challenge. In MPI, a processor will wait at a *send call* until the data are sent, and likewise at a *receive call* until it gets the data. For example, if processor 4 sends processor 1 information, and processor 1 does not have a receive call for it, then processor 4 will wait indefinitely for processor 1. This all means that you must have the right processors sending and receiving to avoid such a lock up. We solved this problem by using a simple *for* loop as described below.

The final challenge would be to truly optimize the code. To do that we should set up a temperature queue where processors simply take the next available temperature rather than deal with fixed numbers. We leave this challenge for the reader.

12.1 Algorithm

The basic algorithm is simple: loop over the desired temperature range and increment the processor doing the computations. The main advantage, as you can see from the code, is that all the processors are computing at the same time. Processor 10, for example, skips through the first nine loops and starts right away on its temperature. When it is done, it waits for a *receive* command from the host processor. There is a possible chance for lockup condition here, but in test cases the computations took 95 percent of the time, while the sending and receiving took 5 percent.

The complete program is in the Appendix. It has six steps:

1. initialize the variables and MPI
2. host processor initializes an output file
3. all processors loop over the desired temperature range
4. processors compute the free energy

5. each processor sends its data to the host
6. the host receives and records the data

12.1.1 Step 1. Initialization

On lines 1 and 2 we import the basic java input/output libraries and MPI libraries. We then define the class and main method, with the `args` variable passed to the main routine (line 9). The value for `args` is entered when the user issues the run command and is passed to the `MPI.Init` routine.

```
 1: import java.io.*;
 2: import mpi.*;
 3: //30 Apr 2002
 4: //Au-Ni binary phase diagram
 5: //Set Ni A component, Au B component, Solid A phase, Liquid B phase
 6:
 7:
 8: public class PhaseMPI {
 9:    public static void main(String[] args) throws MPIException
10:    {
```

The next part of the program sets up output file and variables. MPI differentiates between host and guest processors by using a variable called `rank`. The host processor has rank 0, and each guest processor has a unique integer rank greater than 0. Thus we initialize `guest_processor` at 1 and `host_processor` at 0. The variable `myrank` is a local variable that holds the rank of the individual processor. The variable `worldsize` is the total number of processors assigned to the program, which means that 1 - `worldsize` is the total number of guest processors.

The next set of variables are for the computation part of the program. `con_res` is the number of steps between 0 and 1 for the concentration axis, which means that 1/`con_res` is the concentration step size. Thus if you wanted to determine the phase diagram to 100 parts per million, you need to set `con_res` to 10,000. Note that we use `h` as an integer step for receiving and printing files. The doubles `temp`, `T_STEP`, `T_MAX` and `T_START` are used for temperature controls. `temp` is the loop variable, `T_STEP` is the temperature step size, `T_MAX` is the maximum temperature, and`T_START` is the starting temperature.

```
11: PrintStream pfout = null;
12: PrintStream iout = null;
13: int guest_processor=1, host_processor=0, myrank, worldsize, con_res = 0, h=0;
14: double temp = 0, T_STEP, T_MAX, T_START;
```

On line 15 MPI is initialized. At the end of the main method you will also find a call to `MPI.Finilize()`, which terminates MPI in a nice way. Remember that each processor is running in parallel after the `MPI.Init` argument, and so you need to ensure that there are no job left running on these other computers.

Line 16 sets `myrank` for each of the processors. This is critical. Line 17 sets `worldsize` to the total number of processors. Because all the processes are now running in parallel, line 19 causes each one of them to print out their rank on the host terminal. Because all the processes are independent and some may be busy with other jobs as well, the ranks may appear to be out of order. If things need to be done in a particular order, then use the `MPI.COMM_WORLD.Barrier()` command to make the processors wait until all the processors are to that point in the program.

```
15: MPI.Init(args);
16: myrank = MPI.COMM_WORLD.Rank();
17: worldsize  = MPI.COMM_WORLD.Size();
18: double timer_array[] = new double [2];
19: System.out.println("\n"+myrank+": Started");
20: MPI.COMM_WORLD.Barrier();
21: timer_array[0] = MPI.Wtime();
```

12.1.2 Step 2. Host Initializes Output File

Next the host initializes the output file for just itself.

```
23: if ( myrank == host_processor )
24: //I am the host processor
25: {
26:    System.out.println("\n\n MPI Started \n");
27:    try
28:    {
29:        FileOutputStream fout = new FileOutputStream("AuNi1.dat");
30:        pfout = new PrintStream(fout);
31:        FileOutputStream infoOut = new FileOutputStream("MPIinfo0.dat");
32:        iout = new PrintStream(infoOut);
33:    }
34:    catch(IOException ioe)
35:    {
36:        System.err.println("Error in FileOutput...."+ioe);
37:    }
38: }
```

12.1.3 Step 3. Temperature Loop

Next we define the temperature values to be looped over. The variable `T_STEP` is the temperature step size or increment. The `con_res` variable on line 45 is for the scanning of the compositions in the later part of the program.

```
39: //these define the start and stop of the temperature loop in Kelvin
40: T_START = 300;
41: T_MAX = 1600;
42: T_STEP = 1;
43: //con_res defines the steps in concentration, the step size across
```

```
44: //the concentration axis is 1/con_res
45: con_res = 1000;
```

The next piece of code is at the heart of the parallel processing. Line 40 sets up the temperature loop from `T_START` to `T_MAX` in steps of `T_STEP`. Remember that all the processes will run this same code simultaneously, but will use their individual values of `myrank` to decide on which part to work.

```
46: for (temp = T_START; temp < T_MAX; temp += T_STEP)
47: {
```

12.1.4 Step 4. Number Crunching

The `if` statement on line 48 determines which part of the computation will be done by each processor as the variable `guest_processor` loops from 1 to the total number of processors. For example, if `guest_processor` = 3, then when the processor with `myrank = 3` gets to this point in the program, it will enter the `if` loop. Other processors will simply repeat the `if` loop doing no computations until their rank matches `guest_processor`.

```
48: if (myrank == guest_processor) //Other computers
49: {
50:    CALPHAD auni=new CALPHAD();
51:    TestG test=new TestG();
52:    test.setOmega(21689,0);
53:    test.setT(temp);
54:    double G0Aa=auni.getG0Aa(temp);
55:    test.setG0Aa(G0Aa);
56:    double G0Ab=auni.getG0Ab(temp);
57:    test.setG0Ab(G0Ab);
58:    double G0Ba=auni.getG0Ba(temp);
59:    test.setG0Ba(G0Ba);
60:    double G0Bb=auni.getG0Bb(temp);
61:    test.setG0Bb(G0Bb);
62:    double results[]=test.runTest(con_res);
```

12.1.5 Step 5. Processor Sends Data to Host

Line 63 sends data from each `guest_processor` processor to the `host_processor`. Note that the variable `send` is an *array* containing the data, with `2*con_res` as the size of the array, with `MPI.DOUBLE` indicating that the data are doubles, with `host_processor` the rank of the destination, and with 10 a unique identifier for this particular message.

```
63: MPI.COMM_WORLD.Send(results,0,2*con_res,MPI.DOUBLE,host_processor,10);
64: }
```

12.1.6 Step 6. Host Receives and Outputs Data

This `if` statement on line 65 selects the computer with rank = 0, that is, the `host_processor`, to execute the following block of code. Here the `host_processor` sets up an array to receive the data from processor `guest_processor`, and then converts the temperature from Kelvin to degrees Celsius. The host then receives an array from the `guest_processor` and writes it to file. The *if* statement on line 80 makes the `guest_processor` loop again. A simple `mod` operation would also work, but this seems clearer to us. Line 90 terminates MPI and closes the main routine and the `PhaseMPI` class.

```
65: if (myrank == host_processor) //Host Computer
66: {
67:    double receive[] = new double [2*con_res];
68:    if(temp%10==0){System.out.println(temp);}
69:    double TdegC=temp-273.0;
70:    MPI.COMM_WORLD.Recv(receive, 0 ,2*con_res, MPI.DOUBLE, guest_processor, 10);
71:    while(receive[h]!=0)
72:        {
73:            pfout.println(receive[h] + "\t" + TdegC);
74:            h++;
75:        }
76:            h=0;
77:        }
78: //If we reach the end of the processors,
79: //then reset the current processor to 1 otherwise advance
80:        if (guest_processor == (worldsize-1))
81:        {
82:            guest_processor=1;
83:        }
84:        else
85:        {
86:            guest_processor++;
87:        }
88: }
89: timer_array[1] = MPI.Wtime();
90: MPI.Finalize();
```

13 Results

The initial speedup results are quite good. As shown in Fig. 13, for up to 15 processors the speedup increases linearly with the number of processors. This means that there is little time lost to communications.

13.1 Accuracy

Essentially this program transforms data from one form into another. Its accuracy is only as good as accuracy of the input free energy data and the thermodynamic equations. The

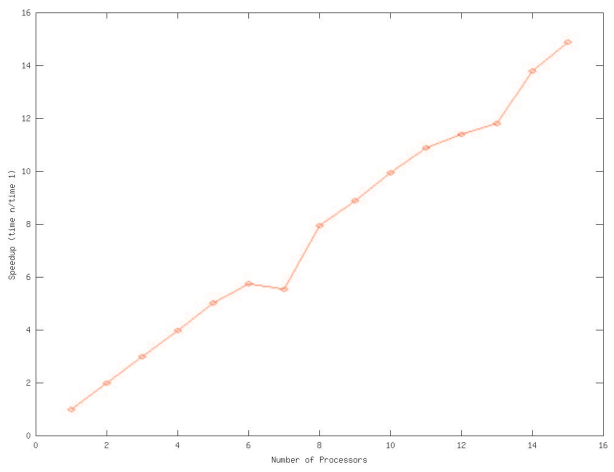

Figure 4: Speedup *versus* the number of processors for `PhaseMPI`.

easiest verification we have made is to check with the large number of published phase diagrams, and especially those that are adopted as industrial standards. Other checks include comparisons with the melting temperature of the pure elements.

For parallel computing, we should also discuss consistency, that is, whether the same results are obtained as the problem is spread over more processors. For this program, we found no change.

13.2 Multi-Phase Extensions

Up until now, the program calculates only single-phase binary alloys. Clearly this can be extended to multi-phase systems. Where for a single-phase alloy we had two phases, liquid and solid, that needed to be checked for equilibrium, in a three-phase system, for example a liquid FCC solid and BCT solid, we would need to check for equilibrium between all the l-FCC, l-BCT, and FCC-BCT phases. This means running the program three additional times. For a four-phase system we would need to run six additional times. For five phases we run the program ten times, and so on.

14 References and Other Reading

Physical Metallurgy Principles, 3rd ed.: Robert E. Reed-Hill, Reza Abbaschian, PWS publishing, Boston 1992.

Phase Transformations in Metals and Alloys, 2nd ed.: D.A. Porter, K.E. Easterling, Chapman & Hall, London 1992.

ASM Handbook v. 3, Alloy Phase Diagrams: ASM International, Materials Park, Ohio 1992.

Binary Alloy Phase Diagrams, 2nd ed.: editor T.B. Massalski, ASM International, Materials Park, Ohio 1990.

"Notes for PH 641/642 Statistical Thermophysics": H.J.F. Jansen, Physics Department, Oregon State University. Corvallis, OR 2002 (Unpublished).

"mpiJava 1.2 API Specification": B. Carpenter,G. Fox, S. Ko, S. Lim, Northeast Parallel Architectures Centre, Syrcause University. Syracuse, NY.

(On manuel on the beowulf cluster: /usr/local/mpiJava/doc)

15 For Further Information

You should be well on your way to understanding and writing parallel code. For more information there are two web sites that are constantly updated:

1. mpiJava home page: `http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html`
2. MPI home page: `http://www-unix.mcs.anl.gov/mpi/index.html`