# Excursions into Parallel Programming with MPI Computer Systems Lab, 2007-2008

Michael Chen

January 23, 2008

## Abstract

With more and more computationally-intense problems appearing through the fields of math, science, and technology, a need for better processing power is needed in computers. The solution can be found through parallel processing, the act of linking together multiple computers and cutting run-time by using all of them efficiently. MPI (Message Passing Interface) is one of the most crucial and effective ways of programming in parallel, despite its difficulty to use at times. However, there has been no effective alternate to this date, and continues as the de facto standard of parallel programming. But technology has been advancing, and new MPI libraries have been created to keep up with the capabilities of supercomputers. Efficiency has become the new keyword in finding the number of computers to use, as well as the latency when passing messages. The purpose of this project is to explore some of these methods of optimization in specific cases like the game of life problem and heat dissipation. **Keywords:** Message passing interface,

# 1   Introduction

Because of the increasing demand for powerful computations, MPI has become a standard for even companies like IBM, which recently began using their BG/L supercomputer along with an MPI library to tackle problems like protein folding. Because of this, efficiency in parallel programming has become a high priority, finding what yields the best latency, and what type

of processors can best suit each problem. For example, IBM tried using co-processor mode (both processors calculate the problem), as well as virtual node (one processor receives and sends data, while the other is calculation intensive).

Parallel programming is a field that will have much use in the future. As a programmer, it is important to reach into bodies of knowledge that will become important in time, and that includes a thorough understanding of MPI.

Many hopes for the future lie in this field. Molecular biology, strong artificial intelligence, and ecosystem simulations are just a few of the multitude of applications which will surely require parallel computing. Though my plans only include optimization of the game of life and heat dissipation problems, it is these basic skills that carry over into real-world applications, except on a much larger scale.

# 2 Background and review of current literature and research

Parallel programming is the concept that multiple processors can be used to split up a task and then combine the separate parts to enhance processing speed. With advancements in many fields requiring computation-intensive calculations, parallel programming has become increasingly popular. This has been explained in many books such as Introduction to Parallel Programming as well as Parallel Programming in MPI. In specific, the type of parallel programming being used in this project is message passing, having computers send and receive data from each other to complete programs. The first message passing interface was released in 1994 for Fortran. 14 years later, though it retains much of its old functions, MPI-2 has become a new standard, focusing more on parallel I/O, dynamic process management and remote memory operations.

Project ideas that have been explored include taking images taken from aircraft and placing them on a map in terms of longitude and latitude, as well as symbolic computations for recognizing speech and facial features. The complexity of algorithms and functions ranges greatly and often depends on the efficiency of the code. For example, though master-slave message passing is probably the easiest to code, it runs much slower than the divide and

conquer strategy. (Figure 1) Whereas the first only has messages passing between the main computer and slave computers, divide and conquer has each processor communicating with any other processor it needs information from. But these are still just the basics. As stated above, companies like IBM are actively searching for new methods to tackle their problems, using collective and torus networks (that allow quick communication in the least number of steps), and different modes with which to use dual processors.

Also, a quick overview of the game of life, one of the projects I am working on. It begins with a board with a certain number of cells, with each one either alive (1), or dead (0). It is a very basic simulation of social interaction, since whether or not each cell lives or dies depends on the status of the cells around it in all eight directions. If 4 cells around it are alive, then it dies from overcrowding. If less than 2 cells are around it, it dies from overcrowding. It lives if 2 or 3 cells are alive, and if the cell is currently dead, it comes back to life (reproduction) when there are 3 cells as neighbors. Running this repeatedly leadings to a series of interaction between cells that can cause eventual patterns to form (squares and crosses survive).

# 3  Development

## 3.1  Requirements and Limitations, Overview, Development plan

The project will be deemed successful under a few conditions. At the beginning of the year and up till now, it has been to tackle problems and programs involving MPI, like the wind velocity lab from first quarter, and Mandelbrot and game of life from second quarter. However, from third quarter on, the focus will shift from implementation to optimization. My goal by the end of the year is to write a program that can test the run results of two specific problems: the game of life and heat dissipation problem, which both use similar cell by cell processing.

The technology demand of MPI will be no problem to meet, since at TJHSST, all the computers in the Systems Research Lab are compatible with MPI, and have enough processing power to suffice for any computational power I will be using.

## 3.2 Research Theory

Although I started with programs like embarrassingly parallel computations, I have moved through more difficult aspects of coding, namely the divide and conquer algorithms. My programs begin with writing a non-MPI program, and converting it to work in parallel. I now have a clear grasp of the basics of MPI (through much trial and error), and it will allow me, in the next quarter, to change the view of my project.

Latency has many definitions, but in terms of MPI, it represents the amount of time a processor has to wait to receive a message from another processor. In MPI, this number is essential in making a program as efficient as possible, and will become a major focus in the next quarter. Run-times for parallel processing is basically a combination of two factors: latency and processing power. Depending on which one the computer excels at, a program will be better off either sending more messages and calculating less per message, or sending large messages at once, and spending more time between messages calculating. Finding the perfect balance is the problem that is being posed.

## 3.3 Developmental Procedures

During first and second quarter, I have been following along with the supercomputing class, which has been diving into parallel programming with MPI. However, nearing the end of the 2nd quarter, I decided to break off, and work more on the game of life program. The process began with writing the game of life without MPI, which was done quickly. However, making it run in parallel was the hard part, hindered by two problems: Java has been my main language for the past three years, so I run into syntax errors in C quite often, and because I encountered problems at first with sending and receiving in MPI (array sizes caused problems in the game of life).

The theory behind the game of life with MPI is that the board will be split by the number of processors used in calculation. But each section also needed a limited amount of information from surrounding areas, and this is where message passing came into play. My time focused on writing code that would allow each processor to send one row or one column to another computer, as well as receive it. This is also where the latency problem comes into play. If a computer sends and receives not one, but two rows and columns at once, then the first time it runs, the data will be accurate up to the first row. Then, after

being run the second time, the data is accurate inside the assigned section. Thus, by doubling the amount passed (or actually more than doubling), the amount of time needed before information needs to be passed again is also doubled. (Figure 2) But a limit has to be drawn eventually. There is little use in, say, passing the entire board to each cell. Also, when the main computer receives a print command, it will wait for all the other computers to send information on its section, and combine them to display output on the screen (this is just parent-child).

## 3.4   Testing and analysis

Because of the nature of MPI, it is not restricted to certain capabilities, though it does excel at some. Thus, a variety of possible displays and processes can be run, and depending on the specifics of each programming, different debugging and error analyses are required. In specific, every program I write needs to have the number of processors specified. For the game of life, I am working on a feature that allows the user to input the exact width and length and number of cells to start alive (in the form of a percentage), before the program runs, and also use mouse clicks to let the simulation run, or to run it step by step. The heat program will have a similar interface.

But what is important is not what the programs look like when they run, since they will be the same if I pass one row or five rows. What is important is the records of their run-times, which I will try to write a general program to record, so I dont have to manually run everything. Fortunately, since the game of life and heat programs use similar boards and characteristics, it should be relatively simple to write one testing program for both of them that will record speeds based on amount of message passing vs. amount of internal processing. The final product will be creating a program that can find the optimal conditions for a certain problem (in this case, game of life, though it can be expanded to fit for other problems)

# 4 Results, Discussion, Conclusion, Recommendations

## 4.1 Expected Results

Once again, I am hoping to learn about parallel programming more through MPI, a field that will become more important in the future. Even now, there are many research teams comparing and contrasting different ways of using MPI, including IBMs supercomputing team. For example, one research I read was exploring the possibility of a graphical interface called MPI-Delphi for workstation networks that allows quick and easy access for programmers. Even at the professional level, I read about testing done on blocking v. non-blocking coordinate checkpointing, another method of MPI. So whether or not the research I do yields a substantial result this year, or if I find a direction to follow, the knowledge and skills I obtain will become indispensable in the future.

The methods and programs to find optimization pale in comparison to the works that are being studied by programmers outside of a classroom setting. But even these simple tests provide an important basis for problems that are sometimes not more complex, but just much larger in magnitude. Though I know I will not be able to try the coprocessor and virtual node modes (which require dual processor nodes on each computer, the one IBM uses has over 65,000 nodes, each dual), by learning from the basics and working through these ideas, I can grasp the concept behind research that is going on now and will continue in the future.

## 4.2 Results

From my results from the wind velocity lab, the efficiency in adding number of processors increased, but steadily increased less until the number hit 8, at which the processing time increased with each added computer. From that point on, adding any more processors only made the time initializing and passing messages wasted. However, I do not have information to use for the game of life, which is a much more difficult problem to analyze. That is what I am going to work on for the majority of third quarter, along with efficiency of heat dispersion.

The problem with measuring these will be that it will measure not on the number of processors used, but also the amount of information passed

between steps (passed every step? Or every three steps?) This will affect the latency and the amount of processing that each computer has to do. Depending on which task the processors are more suitable for, one will yield better results.

## 4.3 Future Testing

There is much to be done in the following two quarters, and more research to be done. Mr. Torbert has suggested using latency algorithm and formulas that can be found online to gauge efficiency and run-times, and I need to look into that while working on the testing programs in the near future. And by doing so, I will to move my testing and programming more from the theoretical and experimental, into something more practical.

# 5   Appendices

Sample Code from Wind Velocity Lab:

```
for(angle=0.0;angle<=maxangle;angle+=0.5) //parent process
for (wvel=-50.0;wvel<=wmax;wvel+=0.1)
{
//if all child processes are busy, wait to receive info
if (flag>=size)
{
MPI_Recv(args,3,MPI_DOUBLE,k,tag,MPI_COMM_WORLD,&status);
printf("receiving! endx:%3.2f\n", args[2]);
flag--;
}
//after receiving, send info to the same process
args[0]=wvel;
args[1]=angle;
args[2]=1337.0;
MPI_Send(args,3,MPI_DOUBLE,k,tag,MPI_COMM_WORLD);
//move to the next process and go around the for loop
k++;
flag++; if (k>=size)
k=1;
//making sure not to exceed number of computers available
```

```
printf("sending! angle:%3.2f,wvel:%3.2f\n",angle,wvel);
}
//this section of code makes sure that all values have been received
int recall=0;
int ori=k-1;
if (ori<1)
ori=size;
while (recall==0)
{
if (size==2)
{
MPI_Recv(args,3,MPI_DOUBLE,k,tag,MPI_COMM_WORLD,&status);
printf("%3.2f\n",args[2]);
break;
}
MPI_Recv(args,3,MPI_DOUBLE,k,tag,MPI_COMM_WORLD,&status);
printf("%3.2f\n",args[2]);
k++;
if (k>=size)
k=1;
if (k==ori)
recall=1;
}
//this portion of code ends all child processes
args[0]=0.0;
args[1]=0.0;
args[2]=666.0;
int n;
for (n=1;n<size;n++)
MPI_Send(args,3,MPI_DOUBLE,n,tag,MPI_COMM_WORLD);


-------------------------------------------------------------------------------


MPI_Recv(args,3,MPI_DOUBLE,0,tag,MPI_COMM_WORLD,&status);
double vx = v0*cos(args[1]*M_PI/180);
double vy = v0*sin(args[1]*M_PI/180);
double vw = -args[0];
//checks to see if parent process ends this process
```

```
if (args[2]==666.0)
break;
while (py>=0) //as long as the projectile has height
{
//recalculates acceleration and accounts for it
ax = (vx-vw)*(vx-vw)*rf;
//acceleration of wind depends on direction of wind
//with respect to the direction of the projectile
if (vw<vx)
ax=-ax;
//checks if wind velocity is working with/against gravity
if (vy>0.0)
ay=-g-rf*vy*vy;
else
ay=-g+rf*vy*vy;
vx+=(ax*dt);
vy+=(ay*dt);
px+=vx;
py+=vy;
}
//sets value as the ending location of projectile and sends it
args[2]=px;
MPI_Send(args,3,MPI_DOUBLE,0,tag,MPI_COMM_WORLD);
```

Sample Code from Game of Life Lab:

```
int xymin = 0;
int xymax = max/sqrt(size);
int xs=((int)rank)%((int)sqrt(size));
int ys=rank/sqrt(size);
int xa=xs*xymax;
int ya=ys*xymax;
int n;
int m;
//number here represents how many turns the game goes through
while(flag<10)
{
//all processors make a move (only in their area)
step(xa,ya,xymax,size);
```

```
//corners are not covered, need to swap around the passstatements
if (xs>0)
MPI_Send(arr,max*max,MPI_INT,rank-1,tag,MPI_COMM_WORLD);
if (xs<sqrt(size)-1)
MPI_Send(arr,max*max,MPI_INT,rank+1,tag,MPI_COMM_WORLD);
if (ys>0)
MPI_Send(arr,max*max,MPI_INT,rank-sqrt(size),tag,MPI_COMM_WORLD);
if (ys<sqrt(size)-1)
MPI_Send(arr,max*max,MPI_INT,rank+sqrt(size),tag,MPI_COMM_WORLD);
//after sending, all computers waiting for other processes, then processes them
if (xs>0)
{
MPI_Recv(rec,max*max,MPI_INT,rank-1,tag,MPI_COMM_WORLD,&status);
for (m=ya;m<ya+xymax;m++)
arr[xa-1][m]=rec[xa-1][m];
}
if (xs<sqrt(size)-1)
{
MPI_Recv(rec,max*max,MPI_INT,rank+1,tag,MPI_COMM_WORLD,&status);
for(m=ya;m<ya+xymax;m++)
arr[xa+xymax][m]=rec[xa+xymax][m];
}
if (ys>0)
{
MPI_Recv(rec,max*max,MPI_INT,rank-sqrt(size),tag,MPI_COMM_WORLD,&status);
for (n=xa;n<xa+xymax;n++)
arr[n][ya-1]=rec[n][ya-1];
}
if (ys<sqrt(size)-1)
{
MPI_Recv(rec,max*max,MPI_INT,rank+sqrt(size),tag,MPI_COMM_WORLD,&status);
for (n=xa;n<xa+xymax;n++)
arr[n][ya+xymax]=rec[n][ya+xymax];
}
//1 if you want to print!
//computers all send print info to main computer
if (rank!=0 && pflag==1)
MPI_Send(arr,max*max,MPI_INT,0,tag,MPI_COMM_WORLD);
```

```
//main computer prints
if (rank==0 && pflag==1)
{
for (k=1;k<size;k++)
{
MPI_Recv(rec,max*max,MPI_INT,k,tag,MPI_COMM_WORLD,&status);
int xtemp=((int)k)%((int)(sqrt(size)))*xymax;
int ytemp=((int)k)/((int)(sqrt(size)))*xymax;
int xmax=xtemp+xymax;
int ymax=ytemp+xymax;
for(n=xtemp;n<xmax;n++)
for(m=ytemp;m<ymax;m++)
arr[n][m]=rec[n][m];
}
display();
}
//one step over
flag++;
}
```

Comparison of Master-Slave and
Divide and Conquer Methods of
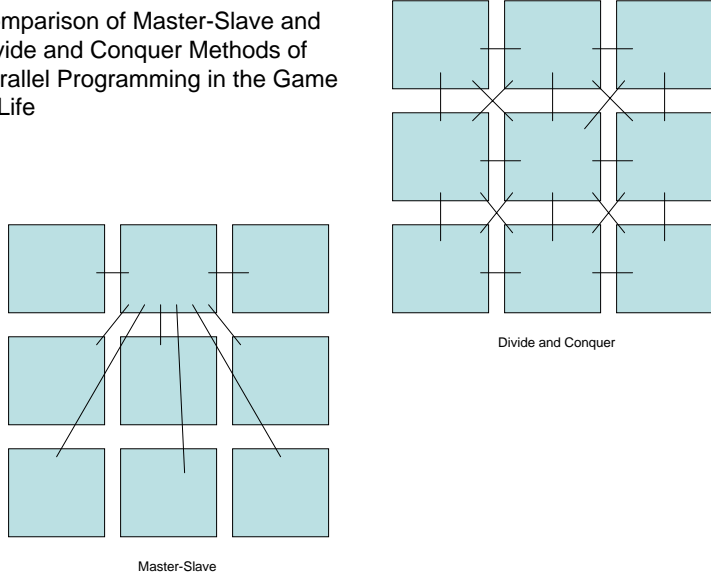Parallel Programming in the Game
of Life

Divide and Conquer

Master-Slave

Figure 1

Example of Latency vs. Processing Power

a processor before any information is passed

After one row/column has
been passed                 Before "step"

after two rows/columns have been
passed

After "step"

Inaccurate data

Accurate data that is
inside the processor's
area

Accurate data that is
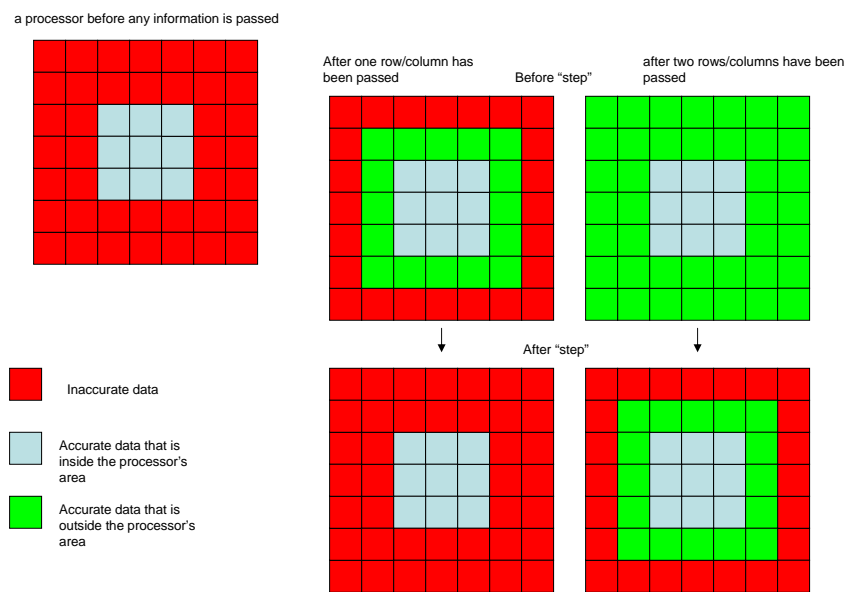outside the processor's
area

Figure 2

# 6  Literature Cited

P. Pacheco, Parallel Programming with MPI, M. Kauffman: San Francisco, California, 1997.

Almsi, G., Archer, C., Castaos, J. G.,  Gunnels, J. A. (2005, March). Design and implementation of message-passing services for the Blue Gene/L supercomputer. IBM Journal of Research and Development, 49(2/3), 393-406. Retrieved from ProQuest database.

Buntinas, D., Coti, C., Herault, T., Lemarinier, P., Pilard, L., Rezmerita, A., et al. (2008, January). Blocking vs. non-blocking coordinated check-pointingnext term for previous term large-scale fault tolerant MPI Protocols. Future Generation Computer Systems, 24(1), 73-84. Retrieved from ScienceDirect database.

Gregoretti, F., Laccetti, G., Murli, A., Oliva, G.,  Scafuri, U. (2008, February). MGF: A grid-enabled MPI library. Future Generation Computer Systems, 24(2), 158-165. Retrieved from ScienceDirect database.

# 7  Acknowledgements