Optimizing Parallel Programming with MPI, Computer Systems Lab, 2007-2008

Michael Chen

June 7, 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.
Keywords: Message passing interface, latency

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 coprocessor 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 problem, it is these basic skills that carry over into real-world applications, except on a much larger scale. The processes of automated testing are becoming increasingly popular, and the results that my project should yield as a result hopefully will point to possible relationships between latency, computation power, as well as the optimum number of processors to use.

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 pro-

gramming 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. But because individual computers each have their own latency, the question of how many computers to use as well as how often and how much to communicate become key issues, ones that I address and attempt to solve in this project.

Also, a quick overview of the game of life, the basic simulation I am using to test effiency. 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 leads 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 shifted from implementation to optimization. I chose the game of life program because it has variables that are easily manipulated through cell by cell computation. By having a simple scheme to test, I can find out exactly how much latency and number or processors plays a role in run time.

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. Since I've developed a clear grasp of the basics of MPI, I've moved on to more difficult tasks, such as automating the testing as well finding real results within my data rather than just writing example programs. This has really shown in the 3rd and 4th quarters, in which I've created a system in which the user simply has to input data into a text file that will be read in and run by a secondary program that transmits data to the main program.

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 followed 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). However, in the 3rd and 4th quarters, the problems instead revolved much more around file i/o and the inability of the C language to use strings effectively, which caused problems when I was attempting to automate the whole process. Though this was fixed eventually, there are still other problems that I have been mostly unable to address, which I will discuss later, like overhead.

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). In theory, a graph will be constructed based on the data I received based on run time correlated with information passed as well as number of com-

puters. Then, by analyzing this chart and converting it to a graph, I hope to be able to find a general formula on how to approach similar problems involving latency.

4 Results, Discussion, Conclusion, Recommendations

4.1 Expected Results

Although my original goal is to learn about parallel programming more through MPI (something that is still a goal actually), 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. However, I hope that the results I find will shed some light on the importance of latency and multiple processors in efficiency.

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. My 3rd and 4th quarter work were focused on the game of life problem and optimizing it with the variables that I've discussed. Though I was hoping to find a clear relationship between latency, run-time, and number of processors, the data that I've produced (Figure 3) suggest otherwise. In fact, some of the results are actually quite counterintuitive.

The largest problem was the fact that as I added processors, the run-time actually increased, almost linearly. From the wind velocity problem, where the optimal processors was around 8, we can see that this is not supposed to happen. However, there are reasons for why this occured. The first is simply that the code I've written in MPI is flawed and inefficient. Even so, that isn't what accounts for most of the inefficiency. After looking into the issue a bit when I discovered the results, I saw that the problem is the sheer amount of overhead from the passing of information at every step. Even though only a little bit of data is transported, it plays a large difference when many processors are in play.

However, not all the results were against my original hypothesis. From Figure 3, we can see that there are some interesting things to note. First is the fact that with less processors, the amount passed plays a smaller role. With 4 processors, the variation in the running time between when 1 row/column was passed vs 5 has a difference of around 25 millseconds. However, with 9 processors, the variation is closer to 100 milliseconds. The second was that there was a general trend for faster run speed as the amount passed neared 4, and then a gradual increase in run time again above 4. For the board size with which I ran for testing was 108x108. This means that for 9 processors, each processor covered an area of 36x36, and with 4 processors, each computer covered an area of 54 x 54. It appears the the optimal number does

not scale with the amount of area that the processors have the cover, but rather, stays at the number 4. Exactly why though, I'm not sure, and if I had more time, I would test different scenarios to discover a better correlation.

One problem with measuring data was that it measured not only 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 and create a situation with more than two variables, therefore adding more chances of confounding factors in the experiments. Depending on which task the processors are more suitable for, one will yield better results.

4.3 Future Testing

Even though my task was somewhat complete, there is always more research to be done as well as different tests I could have run had there been more time. Mr. Torbert has suggested using latency algorithms and formulas that can be found online to gauge efficiency and run-times, and there are many other possibilities to consider as well. Even so, I have moved my testing and programming more from the theoretical and experimental, into something more practical. Although the results I have uncovered may be insignificant compared to the grand-scale projects that major companies are pursuing, they have much of the same basics, and will provide a basis for me in the future to work on, as well as an idea of how research is formally done.

There are four main things that I would have liked to complete for the project. The first is the reduction of the excessively high overhead that is involved as more processors are added to pass data, since this has severely skewed the data in the favor of less processors. Second would be more testing on whether or not four is really the "magic number" for latency, since I highly doubt it would be a constant, but this ties in with the third issue as well, since I would need more computers to test the effect of latency. The third is the use of more processors so that I could test for even higher numbers. There seems to sometimes be a problem with the research lab computers in which some of them refuse to run MPI, therefore limiting the total of computers I had access to in the end to less than 20, even though there should be around 40. Finally, I would have wanted to use the heat map program instead, since that is a more relevant problem and one that could actually be of use in the real world, whereas game of life is really just a simple scheme.

5 Appendices

Sample Code of information passing section from Game of Life Lab:

```
int xymin = 0;
int xymax = max/sqrt(size);
int xs=((int)rank)%((int)sqrt(size));
int ys=((int)rank)/((int)sqrt(size));
int xa=xs*xymax;
int ya=ys*xymax;
step(xa,ya,xymax,size);
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
if (xs>0)
{
MPI_Recv(rec,max*max,MPI_INT,rank-1,tag,MPI_COMM_WORLD,&status);
for (x=amount;x>=1;x--)
for (y=ya;y<ya+xymax;y++)
arr[xa-amount][y]=rec[xa-amount][y];
}
if (xs<sqrt(size)-1)
{
MPI_Recv(rec,max*max,MPI_INT,rank+1,tag,MPI_COMM_WORLD,&status);
for(x=amount;x>=1;x--)
for(y=ya;y<ya+xymax;y++)
arr[xa+xymax+amount-1][y]=rec[xa+xymax+amount-1][y];
}
if (ys>0)
{
MPI_Recv(rec,max*max,MPI_INT,rank-sqrt(size),tag,MPI_COMM_WORLD,&status);
for (y=amount;y>=1;y--)
for (x=xa;x<xa+xymax;x++)
arr[x][ya-amount]=rec[x][ya-amount];
}
if (ys<sqrt(size)-1)
{
MPI_Recv(rec,max*max,MPI_INT,rank+sqrt(size),tag,MPI_COMM_WORLD,&status);
for (y=amount;y>=1;y--)
for (x=xa;x<xa+xymax;x++)
arr[x][ya+xymax+amount-1]=rec[x][ya+xymax+amount-1];
}
```

Sample Code from the game of life automated testing program

```
FILE* outfile;
outfile = fopen("lawl.txt","w");

board=atoi(message);
int q;
char ch[100];

printf("%i\n",amount);
sprintf(ch,"%d",amount);
strcat(ch," ");
fwrite(&ch,sizeof(char),strlen(ch),outfile);

sprintf(ch,"%d",nump);
strcat(ch," ");
fwrite(&ch,sizeof(char),strlen(ch),outfile);

sprintf(ch,"%d",board);
strcat(ch," ");
fwrite(&ch,sizeof(char),strlen(ch),outfile);

char run[100];
strcpy(run, "mpirun -machinefile hosts.txt -np ");
char numpr[10];

sprintf(numpr,"%d",nump);
strcat(run,numpr);
strcat(run," golmpi13");

int trials;
fclose(outfile);
for (trials=3;trials>=0;trials--)
system(run);
m=0;
value=0;
for (q=0;q<3;q++)
message[q]='\0';
```

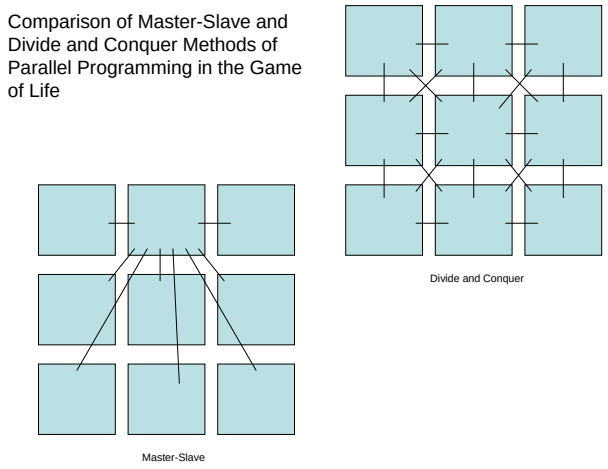


Figure 1

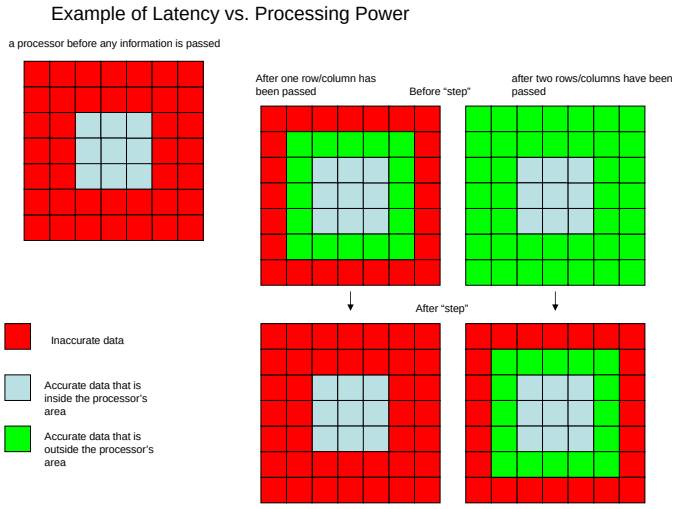
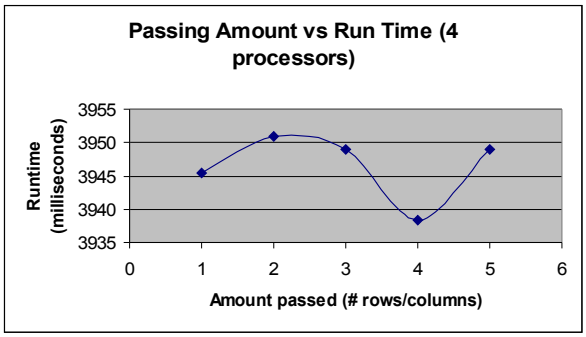


Figure 2

Processors	Passed	Runtime
1		495.2
4	1	3945.4
4	2	3951
4	3	3949
4	4	3938.4
4	5	3948.9
9	1	7109.8
9	2	7131
9	3	7092.4
9	4	7060.9
9	5	7100.6



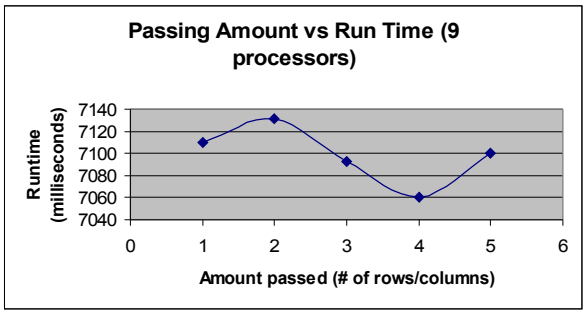


Figure 3: Chart with data and graphs plotting data correlations

Figure 3

6 Literature Cited

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

Acacio, M., Cnovas, O., Garca, J. M., Lpez-de-Teruel, P. E. (2001, October). MPIDelphi: an MPI implementation for visual programming environments and heterogeneous computing. Future Generation Computer Systems, 18(3), 317-333. Retrieved from ProQuest database.

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 checkpointing for 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.

Gupta, R., Vadhiyar, S. S. (2007). An efficient MPI allgather for grids. High Performance Distributed Computing, 169-178. Retrieved from Portal database.

Lisandro, Dalcn, Paz, R., Storti, M., D'Ela, J. (2008). MPI for Python: Performance improvements and MPI-2 extensions. Journal of Parallel and Distributed Computing, 68(5), 655-662. Retrieved from Portal database.

7 Acknowledgements

Special thanks to Mr. Latimer for help throughout the year.

Also thanks to Mr. Torbert for help with MPI with programs like wind velocity and Mandelbrot, as well as much help with debugging during the 3rd and 4th quarter, as well as ideas for future projects