Dillon Cowan

Senior Project Documentation

 For my senior project I made a mock up version of MPI. My imitation of MPI I called Dsc\_mpi. I copied the calls MPI\_Send, MPI\_Init, MPI\_Comm\_size, MPI\_Comm\_rank, MPI\_Finalize, MPI\_Abort, MPI\_Recv, MPI\_Bcast, and MPI\_get\_processor\_name. For each different call I just specified by putting “Dsc\_” in front for my calls and lower cased “MPI” part. For example MPI\_Send() is now Dsc\_mpi\_send() and so on with the rest of the calls. I used the master and slave concept to design the program. The first thing I had to do for my project was find at least 4 different computers and a switch to connect the computers up. I had checked 14 different computers and only found 4 working ones and the switch was set up in Japanese. I wanted to set up an Ssh server on my head machine so I could work on it at home but I couldn’t configure the port forwarding on the switch. Once I had the working computers I had to either install Rocks cluster or Ubuntu. I figured I would go with Ubuntu and start the configuration from scratch (verses the Rocks cluster which I think there was almost no set up in). I named each machine head, node1, node2, node3 respectively. The first thing I set up on each machine was password less Ssh servers on all the nodes. On the head machine I installed cssh (cluster Ssh) and that allowed me to connect to each node (including its self) all at once. So any command I typed in would be sent to all of them at once. The next item on the list to do was change the hosts file to allow each machine to know who the other is. Each node had the same hosts file.

Head 192.168.x.x

Node1 192.x.x ETC.

I then installed nfs-clients on nodes 1, 2 and 3 and the nfs-server on the head node. On the head node I created the master file system called “/mirror” and mounted each node to this directory. Then I installed the necessary compilers GCC and MPICH2. In the mirror folder I created a file called “machinefile” and this held all the names of the nodes. Because I changed the hosts file the program is able to just read the names in and know what the ip’s are. At this point I wanted to make sure that if I ran a normal MPI program that it would run on all the machines. I did this by testing a small program that would output each of the nodes names, how many machines were running and what rank they were. Well, it worked and I was now done with setting up the machine. It took me a week and a half to set everything up.

 The hardest part about this was installing the actual operating systems. Installing the operating systems was the longest process of setting everything up. I did have problems with using the terminal. I never took the 300 level class on administration, so it took longer to understand what arguments to type in. I was glad I choose to actually configure instead of using Rocks cluster because I can now set it up on any machine that can use Ubuntu or another version of it and use it for computations.

 When I was first going into the project the only idea I had was that I had to get certain methods from the mpi calls done and the first one to even start was MPI\_init. I had it started out at 20pts. I knew that I had to have the program that could connect to nodes 1, 2 and 3 before I could even pass messages back and forth. I started with a simple client program and server program and built it up from there. The server file eventually became the daemon on each node, including the head. So at first Dsc\_mpi\_init only read the file called “machinefile” and connected to each server that was running on every machine including the head node and the daemon then quit (or killed itself). Then I soon realized that in order to have the other nodes run the mpi code, they needed to actually look at the same code to run. I needed to use the command exec (execve (prog, newargv, newenviron) ;) and have the daemon servers call exec to start up the “master program” from the mirror folder. Since I had a shared folder, each node only had to look in the mounted folder instead of actually transferring my mpi file to each machine. At first I thought I had to transfer the file to each computer. Each node ran exec on the master program, but the program initially took different parameters, so I set up 3 different parameters. At this point Dsc\_mpi\_init takes in 6 parameters. When you first call the program to start on the master machine your input is ”./main\_machinefile\_4\_1902\_0\_0\_0” (underscore is a space). The first 4 arguments are used when starting up initially; the last 3 are used to start on the other machines. The first one is what nodes your running on, the second one is how many nodes there are, the third is the port the daemon servers are running on, the 4th is the phase of the initiation you’re on (start with 0), the 5th is used for what rank the machine was, and the 6th and last was what their name was (head, node1, etc).

void Dsc\_mpi::Dsc\_mpi\_init (char \*file\_name, char \*Number\_of\_processes, char \*port, char \*phase, char \*Rank, char \*Process\_name)

 When I initially connected to each node I had sent over what there rank was going to be and the name I was giving them. When exec is called from the servers, the example parameters for the program are 0\_0\_0\_2\_1\_head. Again the first 3 are garbage parameters. After the master program tells each daemon to start the slave programs the master program switches from phase 0 to phase 1 and starts up a server and starts polling for connection. The slave programs now connect back to the master. Each slave waits a certain amount of time based on its own rank so the master program knows which rank goes to what file director. . I used pulling in the init method because I wanted the slave programs to connect back as soon as possible. The master program stops pulling for connections as soon as the number of slaves connected back are equal to the number of slaves that was told to start up. If the nodes don’t connect back it sits and waits. If the node closes the master program clears the file director. For Dsc\_rank, Dsc\_size and Dsc\_get\_processor\_name they are getter methods, where each node knows its own rank, size and name from when first initialized. Instead of MPI\_Comm\_size (&world\_size); I called the method this way int wr=mpi->Dsc\_size (); and so on for each of the getter methods. For bcast, send and revc they return 0 if the send or revc was successful to the master, the master node exits if the write or recalls return badly. To get a message to any side the nodes must send and revc everything from the master node. The master node knows which node is which and handles all the redirecting. Because I don’t know how much information is sending each time, I send over two write calls each time. The first one is how much data I’m sending over for the second call (the length of the string) and the second is the string. The second write (the string) I send over is where to go, data type and data, separated by space in each one.

 er1=write (head\_fd, one, 100);

er=write (head\_fd, temp.c\_str (), temp.length ());

 The master side parses the where to go and replaces where to go with where the message just came from and then sends it over to the receiver. So the master side does two reads and two writes per message sent, and the revc side will do two reads every time. For revc I read in the information and then parsed the part of the message that was sent to double check where it was sent from. If everything is good, it will change the variable; otherwise it won’t crash on you. The method will ignore setting the variable and tell the user that it was sent to the wrong place or will tell you if the data type was wrong. I only got integers working to be passed over for sure, I might have got the type double to work but I never tested it. I wanted to spend my time on getting everything correct versus more data types. The way I implemented the abort call is different than the way mpi specified it. The way I have it working is if the Dsc\_mpi\_send returned an error, then you can call abort. Abort then tries to send another message to the receiver saying that there was a send error the receiver then reads and doesn’t do anything but tell you that the abort message was received. The way Dsc\_mpi\_finalize() works is each node calls finalize and closes its connection to the head file director and sends a message to the master to have it close itself on the master side and then the calls exit on the master program. Any time you make a “cout” call, it “couts” to the terminal the node is running from. I couldn’t figure out how to make the “cout” call to be called out on the master node. I don’t believe bcast should have been worth 20 points. The method takes two parameters, the data it’s sending and where the data is being sent from. If the rank of that node equaled where it was coming from you called the method send to each other node, and if you weren’t that specific node you were calling revc. I knew once I had send and revc set up that this method wouldn’t take that long to write out.

 I got really good with setting up the write and read calls; however, I didn’t completely understand what to put in the parameters. I always used sizeof(x) as the last methods parameter. Now I know it’s the number of bytes being sent over. If it was an integer or double etc., sizeof(x) works but if you have an unknown size string to send over then you have known in advance the size of the string being sent. So how I fixed that is I did two writes. The first is to tell you how many bytes are coming over in the second write and of course the second is the data. I think the best thing I learned was how to set up the cluster. That wasn’t something we have learned in any of the classes being taught. There wasn’t anything in my program that was overly difficult to think about. The hardest part is realizing the scope of the project and what it actually entailed. I did have problems with thinking about how each machine looked at the same code. If I needed to, I wouldn’t have known how to transfer the executable over to each machine and how to start the executable up. Once I got init method done everything flowed pretty well. I didn’t understand how void pointers worked and how to change a variable by passing in its reference. Memcpy can really screw up your program if you do not give it the right amount of bytes to copy over. That is why I didn’t bother with doing the rest of the data types, each one would have been different on what their size is.

 I didn’t do a lot of planning ahead. However, I did have to plan on how sending and reading would work and what information had to be sent over. I had to plan that part just because you need to know what you are going to parse out on each side. I needed it to be consistent to do the parsing. I went method by method and tested each method multiple times every time to make sure I got what I just wrote correct. I thought about having the master program send over its file directors of the other nodes over to each node. I soon realized that only the master program knows what the file directors really mean. I didn’t want each node to also be a server to every other node. If you didn’t have to go through a master node that also parses the data being sent to it, then it is probably faster to send and read messages also. The total of sends and reads that my program uses to get one message over to another node is 8. Where if I set up each node to know where the other node was it would cut the message send and reads in half, there is no middle man.

 I use polling on the master node while waiting for messages instead of using signaling. I figured the faster the master node is the faster speed I can get on sending and reading. After init is done in the master program, it switches to a different mode that I called master. This master mode is a while loop that does two reads, parses and then does the two sends to the correct slave. It also polls for the messages being sent to it. I figured that the messages are going to be going back and forth between nodes a lot, which is why I used polling. It might take more of the CPU but that shouldn’t be happening as much if the network is busy enough. In any case you can have a separate machine for just doing the redirecting if the load is too much. In my project I only had four machines so I designated the head machine to also run the master program side.

 If anyone wanted to use my program they would need 3 files, Dsc\_mpi\_server.cc, Dsc\_mpi.cpp, and Dsc\_mpi.h. When you are making your program you have to include my header file and the program has to be called “main”. The daemon server starts up the “main” program only. From there you have to make the calls something like this:

Dsc\_mpi \*mpi= new Dsc\_mpi ();

 mpi->Dsc\_mpi\_init(argv[1],argv[2],argv[3],argv[4],argv[5],argv[6]);

 int r=mpi->Dsc\_rank ();

 int wr=mpi->Dsc\_size ();

 string pn=mpi->Dsc\_processor\_name ();

Any send, revc, abort and bcast calls are:

mpi->Dsc\_mpi\_send (&funsize, 0, 1);

 if(er!=DSC\_mpi\_success) {

 cout<<"failed on sending on rank" <<r <<endl;

 mpi->Dsc\_mpi\_abort (1);//abort the rank your sending to

 }

mpi->Dsc\_mpi\_revc (&rut, 0,-1);

 if(er!=DSC\_mpi\_success)

 cout<<"failed on revc on rank" <<r <<endl;

int bu=0;

 if(r==0) bu=19;

 mpi->Dsc\_mpi\_bcast (&bu, DSC\_INT, 0);

That snippet of code is what you must have minus the variable names that can be your choice. The arguments for send are the variable you are sending over, the type and where sending to. 0 is the type INT. I also have DSC\_INT defined as 0 also if you wanted to have that in the 2nd argument also. For revc its arguments are the variable you wanted changed, the type, and who you’re receiving from. -1 being receives from anyone. All you do for finalize is call mpi->Dsc\_mpi\_finalize (); There is nothing to it. For abort you have to have one argument and that is where you’re sending the failed message to. The only time you can call abort is after a failed send call. All my calls are blocked communication.

 If I had more time to work on this project I would do a couple things differently. I would try to get the number of arguments down from 6 for init, I would put more types that the program can handle, and I would make it so that you are able to name your program anything else instead of having to be the name “main”. The way the slaves connect back to the master program isn’t very efficient, it works, but if you have more than 4 computers to connect it becomes a long wait. It is half a second wait for every connection. There is a way to have each slave send a message over that will tell the master who is who instead of waiting on them. And I would definitely implement more mpi library calls. The only advantage I believe I have over the real mpi is that my calls are easier to understand what arguments are needed.

If I had to do this project over I would want to have around 10 raspberry pi computers and a better switch. Once I unplug the switch from the wall, they will lose their ip addresses. I wasn’t able to configure the switch to have static ip addresses because again, the language was in Japanese. If I were to reconnect the cluster somewhere else I would have to make sure the hosts file would match up with the newly assigned ip’s and possibly remount the network drives to the mirror folder.

 For my demo I’m going to have to compare the time of the real mpi versus my code. I’ll send one message to a node and back about 200 times and average out the messages. Then, for the next one I’ll just do one send and receive from one node to another.

|  |  |  |
| --- | --- | --- |
| Milliseconds | Dsc\_mpi | Mpi |
| Back and forth | ≈114  | ≈0.8  |
| One way | ≈0.5  | ≈10  |
| bcast | ≈4.5 | ≈25 |

When I tested this, mpi varied a lot, but my code was very consistent. I’m not sure what exactly makes my code slow on the back and forth, yet one way is completely faster. Nothing crashes, I think it might be the master node taking a little bit to read and rewrite.