ECE 420

LAB 4: Implementing Page Rank With MPI

SEC: H2

Kelvin Liang 1395077

John Symborski 1387814

Date: Mar 23rd, 2017

***Description of Implementation***

For lab 4 we were given the task of creating a parallel/distributed version of the PageRank algorithm while using MPI based iterative updates. The PageRank algorithm is used by search engines such as Google to evaluate the popularity of webpages and rank the results from the search query. Essentially what we had to do was parallelize the work for which is calculated. Given that the sequential version does work on a single array r we had to find a way to separate the work done on this array by p number of processes and have MPI communicate updates between the processes.

During the lab, we came up with three different implementations with various performances. All three implementations utilized the same method of parallel work however, they differed in how they communicated information between processes. To start off before worrying about how our program would communicate data between threads we focused on defining how each thread would perform the necessary calculations for PageRank. To make the work parallel we had to define three new variables: lowbound, highbound and nodecount\_local. The three variables are local to a single process and help determine the bounds of the main array r in which the process will do work. So essentially lowbound to highbound is the range of elements of r that the current process thread will handle. Nodecount\_local is the number of nodes in which to handle. The local nodecount is found by dividing the total nodecound by the total number of processes. With this we also derived a new local r array to handle the local work for the thread. Now finally to truly parallelize the work in the iterative portion of the algorithm we swapped out the main array r for the local array version in which work within the boundaries defined earlier would be done.

Now with the parallel work done we focused on how our processes would communicate with each other. We came up three different implementations.

**Implementation 1: Send and Receive**

In this implementation, we tried using simple MPI send and receive calls. Once one iteration of work was done the algorithm would check if the current process is the main one (rank being equal to 0). If the rank was zero we would proceed to first update the main array r with the values in the local r. For processes not ranked zero they would send their values via MPI\_send. Once this was done the main thread would iterate between 1 and number of processes to receive the sent values from all other processes. Finally the main thread would send the new updated r to all other threads. This process would be done every iteration.

**Implementation 2: Broadcasting**

When designing implementation 1 we realized that we created a lot of overhead with the send and receive calls. To tackle this, we made use of MPI\_Broadcast which would essentially communicate information from one process to all other processes. In this implementation, we would check again for rank 0 and if rank 0 the thread would update its local values. Other threads would again send their information to the main thread to update the main array. Now instead of the main thread sending to each other thread we use MPI\_Broadcast here to instead send the newly updated array r to every other process which would have a corresponding MPI\_Broadcast call to receive the new array. This method was much more efficient as we had less idle blocking.

**Implementation 3: AllGather**

In our last implementation, we greatly sped up the performance of our communication by using a many to many communication pattern. This was done using the MPI method MPI\_Allgather which given a set of elements distributed across all processes would gather all of the elements to all the processes. Essentially what happens is a MPI\_Gather is performed followed by a MPI\_Bcast. This was very effective as we essentially got rid of any of idle blocking done by MPI send and receive and instead had a streamlined communication method where our main array r would self-update by gathering all local work done by all other processes. Efficiency is than maintained again as a broadcast is used to tell all other processes of the newly updated main array r.

***Testing and Verification***

As we were developing the program we essentially had 4 different periods of verification testing. The first was when we first developed our parallel method of doing the iterative work in the PageRank algorithm. We had to test this first using just a single thread to insure that the funcunality of the algorithm was still maintained despite our changes. By making size equal to one the bound variables mentioned above would essentially be in the range of the main array. The given check.sh script was used to test that our output was valid. Timing was added appropriately to the system. Once we knew the program still solved the data input correctly our focus was redirected to testing the communication of our system.

When testing the communication of our algorithm we ran into several issues. Early on our program would deadlock as processes would be stuck waiting to send or receive information. To Debug this, we made use of print statements to tell us where processes were located within their respective work. Fixing these deadlocks were easy as we simply realized we had placed send or receive calls in incorrect places within our algorithm. After running into these problems testing for MPI\_Bcast and MPI\_Allgather went without any trouble.

Overall we saw steady improvement in the performance of our implementations. As shown below we see that we went from an initial speedup of 3.08 to 3.32 to 3.93 when testing with 4 threads on a sample size of 13000.

Table 1: Implementation #1 MPI\_Send & MPI\_Recv

|  |  |  |  |
| --- | --- | --- | --- |
| Time | Sample Size | Threads | Speedup |
| 0.0173 | 5300 | 1 | 0.982774566 |
| 0.013083 | 5300 | 4 | 1.299549033 |
| 0.347795 | 13000 | 1 | 0.816213574 |
| 0.092042 | 13000 | 4 | 3.084189826 |

Table 2: Implementation #2 MPI\_Bcast

|  |  |  |  |
| --- | --- | --- | --- |
| Time | Sample Size | Threads | Speedup |
| 0.015683 | 5300 | 1 | 1.084103807 |
| 0.013696 | 5300 | 4 | 1.241384346 |
| 0.333294 | 13000 | 1 | 0.851725504 |
| 0.085509 | 13000 | 4 | 3.319825983 |

Table 3: Implementation #3 MPI\_Allgather

|  |  |  |  |
| --- | --- | --- | --- |
| Time | Sample Size | Thread | speedup |
| 0.015102 | 5300 | 1 | 1.126 |
| 0.005341 | 5300 | 4 | 3.183 |
| 0.260394 | 13000 | 1 | 1.09 |
| 0.072214 | 13000 | 4 | 3.93 |

Table 4: Distributed Cluster Testing With All Implementations

|  |  |  |  |
| --- | --- | --- | --- |
| Time | Sample Size | Thread | Implementation |
| 0.117708 | 5300 | 8 | 3 |
| 0.205726 | 5300 | 8 | 2 |
| 0.3004385 | 5300 | 8 | 1 |

***Performance Discussion***

For this lab we were required to to use MPI whilst attempting to speed up a page ranking algorithm. The page ranking algorithm requires a large amount of iterative calculations so we will be using MPI to generate and communicate different iterations to the the main process. We have done 3 different implementations each of them better then the prior one. Implementation one has a speedup of around 3.08 for the 13000 sample size. Implementation two has a speedup of around 3.32 for the 13000 sample size. Finally implementation three has a great speedup of around 3.93 for the 13000 sample size. I will first cover what was done in each of the implementations and why it should cause improvement. After that I will discuss the testing we did on cluster of computers and what this did to our speedup.

**Implementation 1:** This implementation was done using a simple send a receive communications from the main process(process 0) and all the child processes. Basically we split up our rather large ranking array into equal sized local array for each process. Once all the calculations are completed the main process receives all the data from its child processes and assigns it to the large ranking array not the local one. This implementation causes a significant speedup because it has four processes running these long iterative calculations. We have some overhead introduced in the form of communication between processes but the amount of time it saves on the lengthy calculations is much more influential on the overall time. Since each calculation will take approximately equal time splitting up the calculations equally allows for this calculation section to go approximately four times faster. The problem with this implementation is our overhead for communication is much larger then it needs to be.

**Implementation 2:** We noticed that when the main process has the whole page ranking array it has to send it to every single other process. Our prior implementation had us actually calling a send with the same data to every process. We replaced this call with a broadcast. As mentioned in the notes broadcast utilizes all the processes to ensure the data is sent at the fastest rate possible. For example when process two get the data from process one process two will send to process four at the same time as process one sends to process three. This significantly reduces the overhead of our process and communication. Bringing our speedup to 3.32.

**Implementation 3:** In this implementation we took advantage of some optimized function calls in the MPI library specifically the call MPI\_Allgather. This call on a basic form is an MPI gather followed by an MPI broadcast. So our significant increase to a speedup of 3.92 was caused by replacing a series of receives by a single MPI\_Gather. This MPI\_gather is an optimized called that gathers all the processes data into a main buffer on the root process. Using this call automatically splits up the receives by rank rather then us having to do this manually. This reduces idle time waiting for the processes because the messages are simply received in the order they come not in order of 1, 2, 3. This MPI\_Allgather once we set up our environment properly also allowed us to complete all process communication in one line. This made our code very easy to follow while being almost as efficient as possible with four processes running.

**Cluster Computers:** Using multiple machine we would assume that the speedup would increase because we were able to have more CPUs running our code. This was not our observation at all. We saw huge slowdowns when running on a cluster of computers. The reason is the speedup is bottlenecked by network communication. Even though we may have completed the calculation portion of the program much faster the communication overhead shot way up. This is because our messages were now needed to be sent between different computers through a network. This type of communication causes significantly more overhead as compared to just communication locally between processes. The overhead caused by network communication mitigated any speedup we may have seen from having a larger number of CPUs running this code.

***Conclusion and Experience***

In this lab, we were required to make a parallel solution to the problem of solving a linear system of equations. As part of a requirement we had to use Gauss Jordan Elimination. Given to us was a sequential version to the problem. Essentially what we had to do was use OpenMp directives to parallelize the solution and make it more efficient and faster. In this lab, we learned how to implement OpenMp directives to help make for loop implementations faster and more efficient. We used these directives to help make the calculation section of the Gaussian elimination and Jordan elimination more efficient. We also learned how to properly schedule these directives as well as use private variables properly. Each for loop had different characteristics that required us to better understand how scheduling worked so that we could choose the optimal type for the given problem. Allocating the proper shared and private variables helped make the system better as iterations could occur as variables were not shared with other threads when not needed. Overall this lab taught us about proper parallelizing techniques, the effects of scheduling, and how to refactor code to better suit parallelization. Overall we reached a max speedup of 3.70.

Top of Form

Bottom of Form

Questions:

1)

It was significantly faster on the single machine versus the cluster of machines. This is because we are bottlenecked by our network communication. When we ran it on multiple machines it was required to send messages over the network which caused a significant added overhead that diminished the speedup of the parallel implementation. We will also note that people were running the same tests on this network which may have caused the network to be congested and slower than usual. It would be faster if there was less communication required between processes. This would allow the speedup caused by additional CPUs running the problem to be more significant than the overhead introduced by network communications. Hence it would be better to run these types of problems on a clustered network.

2)

We partitioned our data equally among processes. We defined a lower and higher bound for each process. This caused each process to have equal number of nodes. The first chunk is assigned to process 0, second to process 1 ... ect. The best number of processes is four. Finer granularity will mean greater communication and synchronization overhead in our runtime. Fine granularity our task have a smaller amount of work to perform hence they are less likely to have a load imbalance. While this is the case the synchronization and communication may cause too much overhead to have speedup. While a coarser granularity means there will be less communication overhead and more processing time for each process. This will cause a greater likelihood of load imbalance and hence effect the performance of the program. Overall granularity is something to be balanced in order to reach the best performance.

3)

For first implementation we used traditional blocking send/receive. The second and third implementation made use of broadcast and gather commands. Broadcast and gather operations are optimized to perform a very specific communication task which made them have significantly faster runtime that the tradition send/receive. The send/ receive have the advantage that they are able to be used in any communication situation. Gather is only useful when we need to gather all the info from a number of processes. Broadcast is only useful when we need to send out common information to all other processes.

Optional:

4)

Our first implementation contained a blocking send/receive while the second and third implementations contained non blocking broadcast and gather calls. Specifically the gather call does not dictate an order for the way the main process receives all the information from other processes. This allows there to be less waiting and whatever process is done first can properly send out the information. Just in general non blocking is almost always a better practice because it will ensure programs do no deadlock and wait time is reduced.