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 use OpenMP whilst attempting to speed up solving a system of linear equations by Gaussian elimination with partial pivoting. We have broke our step by step improvements into 4 distinct implementations of improvement. By the end of the experiment we we were able to get a speedup up of approximately 3.7 when comparing to the single thread run time. I will first cover the what was done in each step and why it should cause improvement. After that I will discuss the chunk size and scheduling pattern trends in the tables above.

**Implementation 1:** In this implementation we made the calculations after the proper pivoting parallel. This causes a significant increase in the speedup because the entire section that does all the calculation can be ran in parallel. This causes a significant speedup because there is no critical section overhead to cause additional overhead and the size of each iteration is constant. Since the size of these iteration are constant the static scheduling algorithm by default is the best scheduling. Basically each loop will take the same amount of time so simply split up the iteration equally among thread for best performance.

**Implementation 2:** In this implementation we added a parallel component to the inner for loop in the Jordan elimination process. Even by just putting the inner for loop in parallel we were able to see small increase in speed up. We are able to compute this inner loop much faster for each iteration. This causes the overall Jordan elimination process. The scheduling we chose is the static with the default chunk size. This makes sense because each for iteration is doing 3 computations and hence each iteration is going to take approximately equal time to complete.

**Implementation 3:**  In this implementation we simply added a “#pragma for” to the spot where we assign the index array and to where we calculate the solution. These for loops may not take a huge amount of time but we did find an increase in the speedup when we put these for loops in parallel. Since we are dealing with such a small run time in total any increase in time causes a increase in speedup. Using the default scheduling policy and thread number is appropriate because of the simplicity of these loops.

**Implementation 4:**  In this implementation we simply changed the way syntax in our openMP Jordan Elimination. We moved “#pragma omp parallel” above the upper loop and then simply have a “#pragma omp for” before the start of the inner loop. According to the notes there is one fork and join with each parallel for iteration and putting this on the inner loop would cause an overhead due to excess of fork and join calls. By moving it to the outside of the loop we will reduce the number of fork and join calls hence reducing the overhead of making these calculation parallel. We saw a small increase in speedup by making this improvement from ~3.67 to ~3.699.

**Chunk Size and Scheduling policies:** In order to achieve the max speedup we found that when it came to scheduling policies less is more. For both major areas we made parallel we found that the default scheduling policy was the fastest. This is a static scheduling policy that specifies chunk size of iteration/number of threads. For both sections this makes sense because for the most part the iterations would take almost identical time. By dividing this iteration up equally each thread should take the same amount of time to finish and cause the largest increase in run time without taking any overhead for complex scheduling policies. The dynamic and guided scheduling policies saw less increase because it takes additional overhead to actual implement a scheduling policy that changes at run time.Changing the chunk size too large or too small imbalances the workload for each thread causing some thread to do more work then others. This causes an increase in run time because some thread would finish before the other did and at some point would not be doing any work. These factor contributed to us using the default scheduling policy with the default chunk size.

***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