ECE 420

Lab 4: Implementing PageRank with MPI

Lab section H42

Kelly Luc
1498694
Nicolas Serrano
1508361

Implementation methods:

The implementation in this lab will be based on the PageRank Algorithm, making it a distributive version using MPI. This can be achieved by following the details on iterative updates provided. This lab requires us to create an optimal solution to parallelize the PageRank algorithm and test its performance on multiple machines and nodes.

Solution:

Using the serial implementation gave us a guideline on where to start our parallelization. First we added new local variables that each process will use in order to distribute the tasks. This includes a start and end index on a specific chunk of the original array with a local copy of it and as well as the contribution array. When the program starts its processes will get the specific chunk size they will work on by dividing between the nodes over the processes. In case of when the number of processes cannot equally divide the nodes, we created a check that will increase the chunk size by one to account for it. Once the processes initialize the variables it needs, it will proceed to the two equations (Eq 1 and Eq 3) provided by the lab.

During the process through Eq 3, the processes will find its starting and end index in order to parallelize the algorithm and calculate the local pagerank and then store it in the contribution array. To optimize this, we use MPI_allgather at the end of each iteration to gather the elements in each process which sends it to all the other processes to receive. This automatically gathers the local calculation and distributes it to the other processes, reducing idle time like using MPI_send and MPI_recv, which could potentially have a race condition. MPI_Allgather provides an efficient way to complete the communication between processes allowing our program run faster than having a root process waiting for communication between its other processes to receive data.

The program will continue iterating the calculation process until it converges to a certain point where it does not change as described from Eq4. Finally the root process will print the runtime of the program and save the result in an output file.

Performance Discussion:

We observe the performance of our program on bothon a single and multiple machines with a node size of 1112, 5424, and 10000. Additionally, the machines will run with 1,4,8, and 16 processes.

From our observation we can see that the single machine performance decreases significantly as the number of processes increases regardless of the amount of nodes present. This is likely because when the program is serially running, having multiple processes requires a significant amount of context switching.When more machines are added we can see that adding more processes actually increases our performance (up to a point), and will allow us to achieve the best possible run time results. Having more machines distributes the workload, and allows computations to be done in parallel. From this, we can conclude that our program overall works best with multiple machines in the cloud.

We also found that generally having more machines running the program with large scale problem sizes is faster than when compared to a single machine. However if the problem size was small we noticed that a single machine is still a very plausible solution. Our data shows that even with 4 processes, the single machine still performs better than 3 machines. The decrease in performance for the multiple machines under smaller problem sizes is possibly from the time it takes to communicate with each machine. Overall, based on our data, we were able to observe that 5 machines were able to produce the best results over a variety of different problem sizes.

The best number of processes for a single machine is 1 as its efficiency decreases significantly when it has more processes. This should be used for small scale problem sizes so that it can focus more on calculating the program instead of communication between machines or switching context. In the case of multiple machines, the best number of processes varies between the problem size and number of machines running.

In terms of granularity, the overall runtime will increase as we increase the amount of processes. However, there is a point as we increase the number of processes that the runtime decreases significantly. This makes our granularity between fine and coarse. It is because that there should be an optimal amount of processes that should be used and if there is an excess amount, it will decrease it performance instead because of idle time or context switching.

Appendix A: Runtime Data