Overview

The purpose of this assignment as outlined in Canvas is implement a real-world computing problem using the MPI libraries to distribute work across multiple nodes in Beocat. We are given 2 files generated by Dr. Andresen: wiki\_dump.txt and keywords.txt. Available is also a file which does some of the underlying work (reading in the files, keeping track of time, etc.) called find\_keys.c. My task was to read through variable lines of wiki\_dump.txt and use the keys in keywords.txt to find matches and record the line at which they were found, as well as the index. I copied find\_keys.c to my home directory and began trying to implement MPI.

I separated some of the code in find\_keys.c to functions, mainly so I could keep the main method clean. I knew from the onset I would need to implement some type of data structure to hold my matches, but I couldn’t be sure what to use until I understood MPI. I found that by calling MPI\_Comm\_rank() & MPI\_Comm\_size() I could assign a variable to see what rank or PID was running at any given time & tell the entire system the number of tasks to work with. I then initialized two double pointers to chars to hold the keys of key\_words.txt and the lines of wiki\_dump.txt. To store my global results of all matched keys, I built a data structure. I created a struct of two integers and a pointer to an integer so I could store the size, count, and occurrences of each key word in the double pointer, lines. This was stored as a double pointer to a List in ‘results’. Now I calculated the starting rank/PID by multiplying the current rank and by the max\_words divided by the number of threads. The end was calculated by adding the start to the division of max\_words and number of threads. Each process would have a set of local\_results, not directly stored in a global data set until I received the work they had completed. If the current process was not the host (rank of zero), I used MPI\_Send() for all occurrences, relative to the start and end of the rank. Otherwise, I allocated memory for a global list of structs called ‘results’. Then, I looped for all existing processes and allocated memory for each struct, as well as each structs int pointer. I then received all occurrences and matches for all existing processes using MPI\_Recv(). Finally, I blocked the rest of the nodes and printed all results found using the host and ended my clock and printed my data.

The environment of this project had the following traits:

* Linux
* Bash – V. 4.2.26
* Slurm – V. 18.08.0
* Open MPI – V. 3.1.1
* ICC – V. 18.0.3

Jobs

I then wrote some shell files to compile and load the correct environments for my code. I wrote two shell files containing bash scripts. I submitted some test jobs using mpiexec after compiling using mpicc. These two commands I put into one of my scripts, as well as a module load OpenMPI. My main shell was launch\_mpi.sh. In this file, I tried to automate what I could for job submission but found that (after submitting jobs which required 192GBs and 64 cores) I needed to change the number of nodes, tasks per node and memory very specifically. Instead, I had 4 main loops to run multiple trials. For 1000 lines on 1 node with 1 task, I used 1G of memory with a time constraint of 3 minutes. As I increased the number lines, I increased the memory by a factor 10, and the memory by a factor 2. As for the jobs with one million lines, I allotted 18 hrs for the case with 1 node and 1 task per node.

Conclusion

In general, by increasing the number of nodes, I could have an increase in time by a factor of 2. For instance, on the case where I ran 1000 lines with 1 node, and 1 task, I had a time of 39.84s. When I used 2 nodes and 2 tasks, I had a time of 20.29s. However, when the number of lines was increased by a factor 10, I had vastly different results. On the first trial when running one million lines on 2 nodes and 2 tasks, I had a time of 22313.67s, or approximately 6 hours and 20 seconds. When the number of nodes in use changed to 4 and number of tasks changed to 16, the time was 2903.4s, or approximately 48 minutes and 40 seconds. This speed up was close to a factor of 8, with a literal speed up by a factor of 7.68.