The Experiment

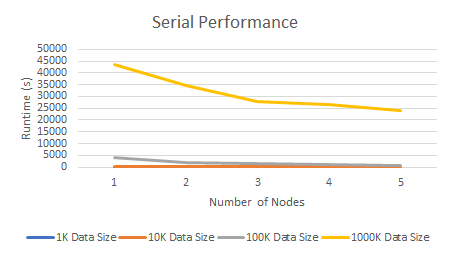
In order to test the effectiveness and scalability of OpenMPI, the file find\_keys.c was designed and built to analyze and read varying amounts of data across multiple nodes and processes. The program reads in a file of keywords (keywords.txt) to search for and a file to search in (wiki\_dump.txt). The program separates the data into “chunks”, searches for matches, and stores the keywords and their corresponding line numbers in a dynamically generated output file (wiki.out). In addition, the file linked\_list.c was created to dynamically allocate memory storage for the matching results. The program was ran using varying numbers of cores, nodes, and memory amounts. The final results are comparisons of the total runtimes for each test. Due to time constraints, each test was given as little RAM as physically possible. Lower memory requirements allowed the data to get through the queue faster allowing the data to arrive on time. Tests of with the 1000 and 10000 lines were given 1GB of RAM per core while the 100000 and 1 million-line tests were given 2GB of RAM.

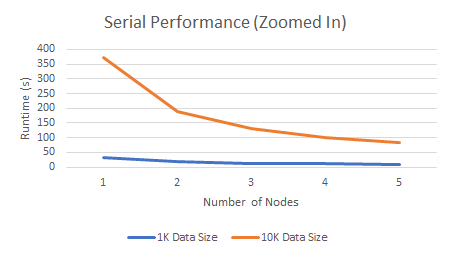
The Environment:

All of the code in the following tests was ran on the Elf class of nodes. More specifically, the tests were running on the first two tiers, elves 01-72. These nodes have two 8-Core Xeon E5-2690 processors with 64GB of RAM. At runtime, the program was compiled and ran using OpenMPI v3.1.1. The client side of the tests was a Dell Latitude E7470 running Windows 10 Pro with a 2.45GHz Intel Core i5 with 16gb of RAM. The SSH client was MobaXterm v10.7 build 3650.

The Results: Single Threaded

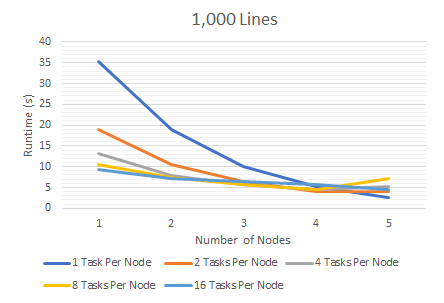
The single threaded version of the program performed more or less as expected. In this case, performance shouldn’t be affected much by giving it extra nodes. It should only run on a single core, on a single node, and it should produce the same runtimes across the board. Curiously, the test of 1 million lines reported an almost 40% increase in runtime from the single node test to the five-node test. This was most likely caused by the increased number of nodes, doubling of RAM, and increased number of cores. Even though those resources are utilized, it would cause the program to run on a higher tier within the same node class which would result in better runtimes.



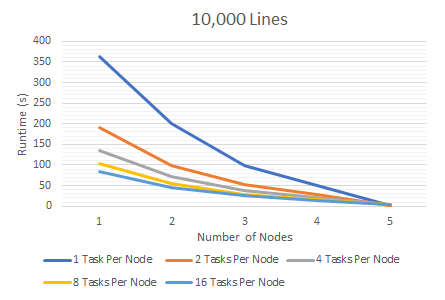


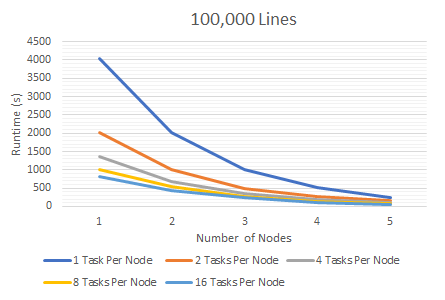
The Results: OpenMPI

In terms of scalability, OpenMPI offers a massive improvement over the standard performance. This is of course when the program is implemented correctly, the test runs without race conditions, and the data size is to a point that parallelization makes sense. For example, given the smallest dataset, (50,000 words, 1,000 lines) the code reached and almost 4x speed up when compared to the serial version on four nodes. Interestingly, as the number of tasks per node increases, the starting runtime is reduced significantly but there’s less return as the number of nodes increases. This is because setting up processes is expensive. Creating multiple processes is only effective given that the time saved by dividing up work is greater than the time required to set up the tasks.

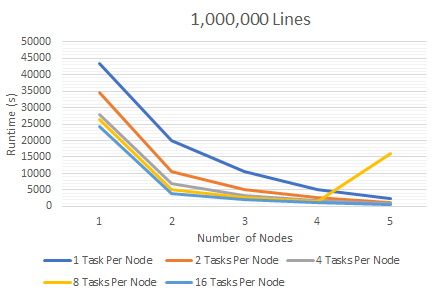


The tests for the data sets of 10,000 and 100,000 performed in a similar fashion to the first test. As expected, at one node, the runtimes increased tenfold in conjunction with the increase in data size. This is because the code here is effectively running like it’s single threaded. Meanwhile, at five nodes with 16 tasks per node, the code ran in only 4.5 and 65 seconds respectively. For comparison, the serial code ran at 84 and 814 seconds respectively. This is an 18.7 and 12.5-times speedup over the serial version. Also like the above, tests ran on 16 tasks per node featured the best speedup on a single node but had the least amount of return as the number of nodes increases. In the 10,000 lines with 16 tasks per node on a single node, had a roughly 4.4 times speed up over the serial code and the 100,000-line test with the same number of nodes and tasks per node, had a roughly 5 times speedup over the serial code.





As expected, the 1 million line tests followed the same trend as the above. Much like the serial code, the runtime for the test with one node and one task per node increased tenfold and as the number of nodes and tasks increases, the runtime decreases exponentially. However, between the tests of one node and two nodes, the increase in runtime is linear and consistent across all levels of tasks per node. The test given 16 tasks per node started significantly higher than in previous tests. This was likely caused by the CPUs running at full capacity. There’s only so much data a processor can handle at once before runtimes start to go up. From here, the only way to increase output further is to spread the data handling out to another node. Furthermore, among these tests is an anomaly at five nodes with 16 tasks per node. The runtime jumps significantly to over 16,000 seconds, almost 16 times longer than the previous test. This was most likely caused by a race condition. Load on the system and differences between nodes could have had an effect on the runtime, however, it’s highly unlikely that it could cause a discrepancy of this scale.



Conclusion

Throughout the tests, the only test that really seemed out of the ordinary was the potential race condition in the 1-million-line test. Everything else makes sense. Given that a data size is large enough to warrant parallelization, runtime can be significantly reduced by increasing the number of tasks per node and the number of nodes. In term of the optimal configuration, it really depends on what is needed to determine the most effective split of nodes and tasks. For example, if only a single node is available, maximizing the number of tasks per node with offer significant improvements over the standard code but with marginal returns as the number of nodes increases. In general, increasing the number of nodes and tasks per node will result in an exponential decay in terms of runtime. The biggest challenge in completing this experiment is time. Polishing OpenMPI code and running tests, especially serial code and tests with fewer nodes, takes an exceptionally long time. If I had to run the experiment again, I would’ve run all of the above tests with 2GB of RAM per node for consistency and I would’ve tested even greater data sizes and number of nodes to determine at what points is OpenMPI no longer effective.