Distributed Computing Analysis

Assignment

This assignment we worked with distributed computing. The function of the program was supposed to use MPI. This API is used to communicate with processes either on the same machine or different machine. At the point of splitting/forking all the processes run the same code, each assigned a different rank from 0 to the number of processes – 1.

Hardware

We used the 'Elf' nodes. They have 8x 10-Core Xeon E5-2690 v2 Processor with 64GB of RAM, 2x 10-Core Xeon E5-2690 Processor with 96GB, 2x 10-Core Xeon E5-2690 v2 Processor with 384GB of RAM, and 2x 10-Core Xeon E5-2690 v2 Processors with 64 GB of RAM.

Software

The OS Beocat runs is 'CentOS Linux 7(Core)'. The kernel version is '3.10.0-862.11.6.el7.x86_64'. We used the command 'module load OpenMPI' to load OpenMPI/3.1.1-iccifort-2018.3.222-GC. This API is used to cross communicate among processes using MPI and the MPI commands.

OpenMPI

We read a keyword file and a wiki_dump file. The keyword file contains words which we use as substrings through wiki_dump file. We then record the line number on which we found that substring. We output the keywords with the line numbers on which they were found in the wiki_dump file.

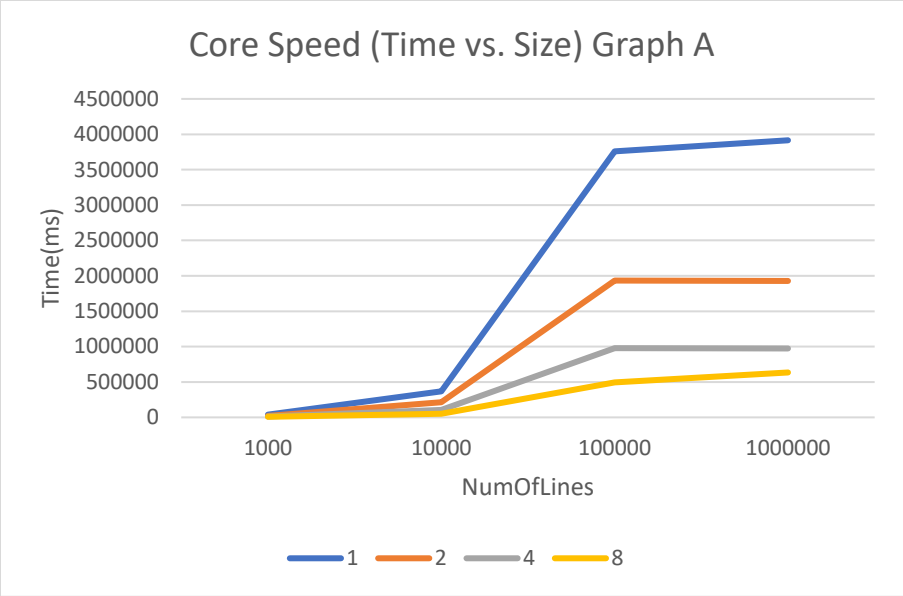
Where the data varies is how many lines we read, how many nodes and cores per node we used, and also the chunk size (number of lines sent to each process). We increase from 1000, 10000, 100000, to 1000000 lines. We increase the number of nodes from 1, 2, 4, to 8 nodes, and for each node we increase the processes by 1, 2, 4, to 8 tasks-per-node.

The table of data collected for all the runs. Each row is a numOfCores and nodes vs data size.

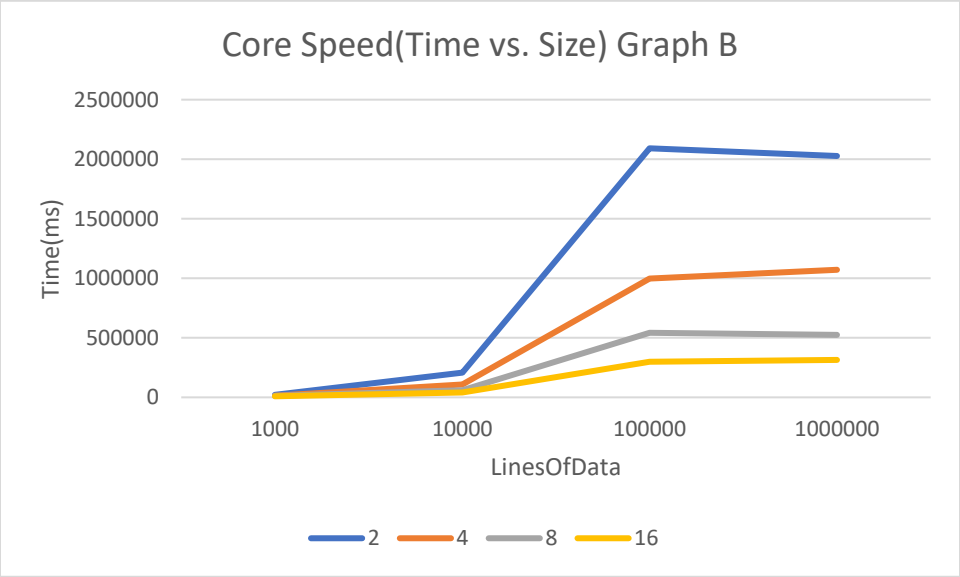
Cores Per Node/Data Lines	1000(lines)	10000(lines)	100000(lines)	1000000(lines)
1 node, 1core	38965(ms)	366732(ms)	3761079(ms)	3915313(ms)
1 node, 2core	20687(2x)	215432(2x)	1933507(2x)	1926477(2x)
1 node, 4core	13025(3x)	104150(4x)	978339(4x)	974732(4x)
1 node, 8core	8234(5x)	50547(7x)	494021(8x)	634520(6x)
2 node, 1core	21252(2x)	206507(2x)	2091476(2x)	2026722(2x)
2 node, 2core	13567(3x)	107057(3x)	997788(4x)	1071340(4x)

2 node, 4core	12772(3x)	59429(6x)	542214(7x)	523535(7x)
2 node, 8core	8868(4x)	42129(9x)	298344(13x)	314415(13x)
4 node, 1core	17448(2x)	106924(3x)	1054964(4x)	1083469(4x)
4 node, 2core	13361(3x)	61995(6x)	542821(7x)	564565(7x)
4 node, 4core	10596(4x)	43641(8x)	323380(12x)	416389(9x)
4 node, 8core	12020(3x)	33298(11x)	172580(22x)	182728(21x)
8 node, 1core	14464(3x)	63863(6x)	555972(7x)	515327(8x)
8 node, 2core	14758(3x)	46362(8x)	299335(13x)	338253(12x)
8 node, 4core	19905(2x)	179013(2x)	176824(21x)	175731(22x)
8 node, 8core	168217(0.23x)	124265(3x)	131210(29x)	121995(32x)

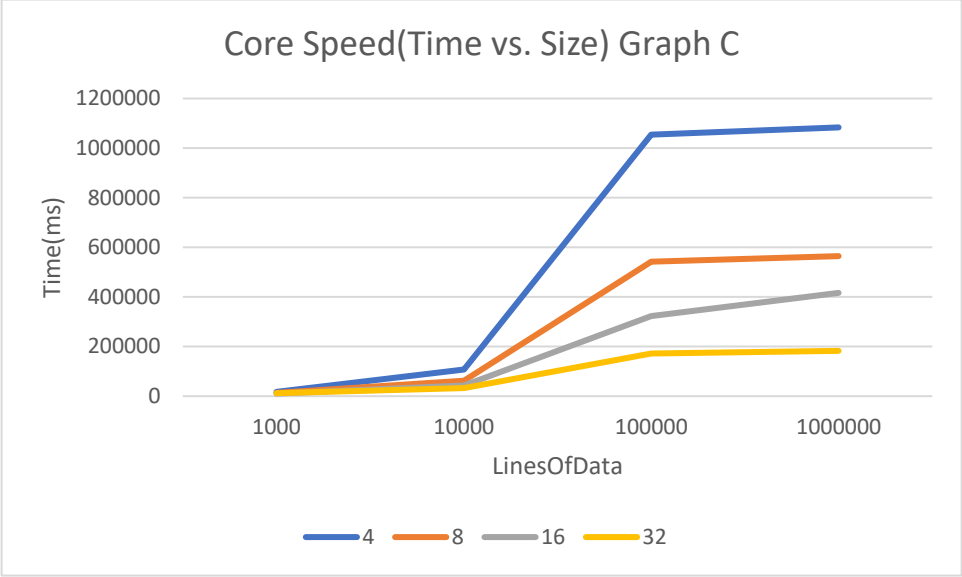
Some things that affect our run times: the number of machines(nodes) because of latency between the machines, the number of cores because it affects how many chunks we can process at a time. For the 1000 lines the fastest run was the 1 node with 8 cores. This is likely because the amount of data is small and there is no latency because we are only communicating on one machine. We are splitting the chunks into pieces of 100 and since we have 8 cores each core and work on a chunk at a time. Any core run that has a larger number of cores than 10 cores would have cores made that would be waiting for a chunk that never comes. For the other runs that have smaller number of cores on multiple machines, we still see a speedup in runtime. However, since they have latency related to communication their overall runtime is slower than the single machine with 8 cores. This can then be related to other runs with more lines of data and a different number of cores and machines. Already for the next number of lines, 10000, we see that the single 8 core 1 machine set up is no longer optimal. This is because with multiple machines we can have more cores than a single machine. This further increases the number of chunks that can be processed at a time because we have a larger quantity of cores available.



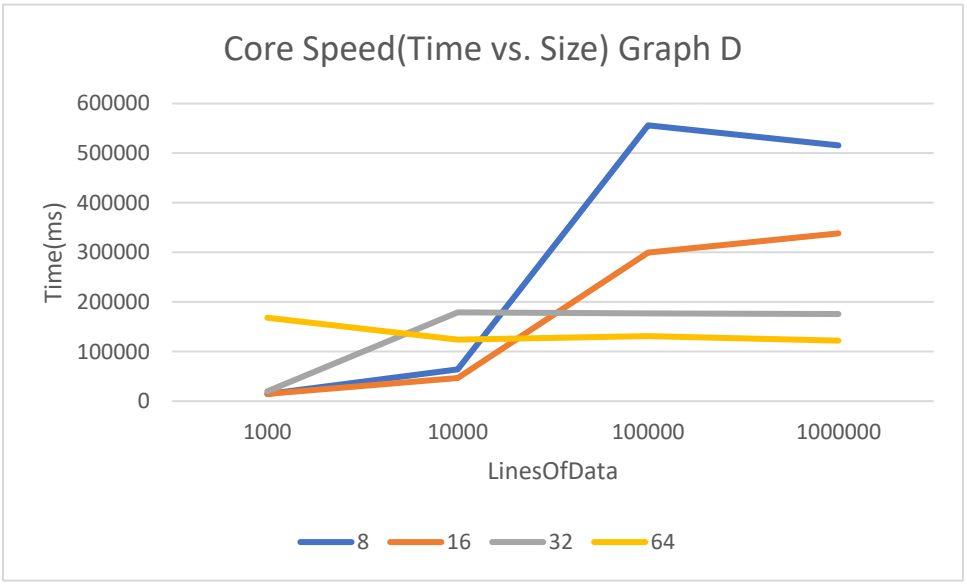
On Graph A shows multiple cores on one machine. This graph shows a relation that as the number of cores grow, data is processed faster. This is especially true with larger numbers of data. For the one core run on 1,000,000 millions of lines of data the time is tremendously slow in comparison to the 8 core run. This is because 8 cores can theoretically process 8 times faster than 1 core, because each core would be able to perform a process at a time.



Graph B relates data on 2 different machines(nodes). Here already the fastest speed up over all the data sizes is with 8 cores on 2 nodes, 16 cores total. Even though with 8 cores on 2 machines, and only 10 chunks. It is still faster than the 2 core 2 node run. This can be explained by the amount of latency and data required to process. The amount of data was processed quicker with more cores, even if some cores were waiting for a chunk that never arrived.



Graph C shows the same kind of relation as do Graphs A and B. However, it describes data with 4 nodes. Even though it may not look it on the graph, the fastest runs were the 32 core runs (4 nodes, 8 cores). And as the data size increased, the speed only decreased slightly in comparison to other core and node variations.



Graph D looks most different compared to the other 3 graphs. This is because the 64 core run took a tremendous amount of time on the 1000 lines. Why is this? Because we split the lines into chunks of 100, which means that of the 64 cores we only utilized an entire 9 other processed and the server process. We observed a massive amount of communication latency due to the lack of data and the number of stagnant cores requested. However, as the data size increased to 100,000 and 1,000,000 we a factor of 29x and 32x time speedup. These were the fastest runs on the entire table. This is due to the fact that we had enough data to pass around and a massive number of cores to process a lot of data in parallel. However, the difference in run time from 100,000 lines to 1,000,000 million lines was not as significant of a change as from the 32 core rune. Possibly because we have reached a sort of spot where even though we can observe that the data is being processed faster. The communication amongst all these cores kept the run time from increasing much more. Had all the cores been on a single machine, we would have seen a faster run time than the 64-cores on different machines.

I did not include a full table with the physical and virtual memory used for the runs, because each process created added additional data to the table and created a massive file. A table with

partial sums is included in the submission. The virtual memory remained within a similar range for all the runs in with 1000 lines of data. As the data increased so did the amount of virtual memory which was used by the server process. The physical memory bounced around frequently among each different number of processes. This could have been due to the fact of usage of each of the processes during the run. If one process processed more chunks than another then its physical memory usage would have had a higher amount. Therefore, with runs that had idle cores, the physical memory was smaller than the ones without idle cores.

This relation is described through the rest of the data size runs. The virtual memory increased with the more data we brought in and the physical memory remained similar on runs where each core was given an equal workload. While on runs where some processes were used less than others (idle ones as well), the physical memory fluctuated greatly there.

Conclusion

With MPI there are a few factors that need to be considered when using the API. One item that should be considered is the amount of data we need to process. As we saw, that not always was the largest number of cores available the most optimal option, latency among the processes communicating across machines has an impact on overall run time. If the data set is not massive enough and can be processed with only the cores on one machine, then the processes would save the time required to pass messages across the network to another machine. From the table we can see that in the smaller data sets, the faster runs were on a single machine which meant we excluded cross node communication. Another thing that needs to be considered is how much overall latency will we be expecting when communication, if latency can be an issue that could result in the loss of data or data taking too long to communicate. Processes could remain idle for longer amounts of time than they should be, wasting time sitting and doing nothing. Also, as the number of processes increases, so does the memory usage. Considered how much is available to each process to work properly is necessary to decide if the MPI API can be used.