Performance Analysis of Pthreads, OpenMP, and MPI.

Lev Kavs levnikolaj@ksu.edu Kansas State University

Sam Moylan smoylan22@ksu.edu Kansas State University

Mitchell Slavens mslavens@ksu.edu Kansas State University

Hardware Specifications

The programs were ran on the same hardware to keep the testing environment constant between the different programs. The machines were constrained to ‘elves’ which contain two 8-Core Xeon E5-2690 processors or a two 10-Core Xeon E5-2690 V2 processors.

System Specifications

The OS Beocat is using is the CentOS Linux. The Linux kernel is the 3.10.0 – 957.1.3.el7.x86_64. The code is compiled using the GCC version 4.8.5 (Red Hat 4.8.5-36). Slurm version is 18.08.6-2

OpenMP Software Architecture

The number of threads is passed in as a command line argument. The file is read into memory, each line read from the file is placed into an index of our wiki_dump array. That array is then iterated over using the ‘#pragma omp parallel for’, this distributes the iterations among the threads. Inside the for loop the call to our ‘algorithm’ is made where it compares the line index passed with the line after. Algorithm is also passed a reference to the wiki_dump array. Each separate call to ‘algorithm’ processes another line from wiki_dump. When a substring is found the corresponding index into the ‘longestCommonSubstring’ is allocated and the substring is copied into the place. Once ‘#pragma omp parallel for’ loop is done the time and memory use is collected and printed out.

Pthread Software Architecture

The number of threads is passed in as a command line argument and the file is read into memory with the same process as the OpenMP version. A loop iterates through the number of threads. Inside each iteration a algorithmArgs_t structure is allocated which contains a reference to the longestCommonSubstring array, a reference to the populated wiki_dump array, and a start and stop index to process of the wiki_dump array. This struct is passed in as the final argument into the pthread_create function call. The algorithm is passed as the function to run to the pthread_create function. Because the struct must be passed as a (void *) in ‘algorithm’ we must cast the parameter to its appropriate type to extract the fields out of the object. A for loop inside ‘algorithm’ causes the threads to perform the substring identification process across its section of wiki_dump array between the start and stop bounds passed inside the struct. When a substring is found its corresponding place in longestCommonSubstring is allocated and the substring is copied into its place. After each thread finishes its section it exits.

The pthreads are then combined with pthread_join command and the output is printed by the main thread.

OpenMPI Software Architecture

The implementation of MPI was completed differently than Pthreads and OpenMPI. We wanted to eliminate as much sending of data as possible to increase processing time and we did that by eliminating all data sending. Every process has access to the 'wiki_dump.txt' file in the 625 directory, every process knows the number of overall tasks and its own rank. The beginning of our MPI implementation starts with finding a chunk size by dividing the 'linesToProcess' by the number of tasks. From there, each task can find its own section of the 'wiki_dump.txt' by using its rank and the chunk size. Each task allocates its own 'wiki_dump' array equal to the chunk size and reads the 'wiki_dump.txt' file and places its section of the file into its own 'wiki_dump' array. Each process also allocates its own 'longestCommonSubstring' which is of equal size to chunk size and the 'wiki_dump' array. When the process finds a substring it places it into the appropriate part in its 'longestCommonSubstring' array. We have an MPI_Barrier that causes processes to wait for all the processes to complete after 'algorithm' is done working. The output is printed in order by using MPI_Send and corresponding MPI_Recv calls to synchronize the data being printed in order. Then the processes print out their memory usage and the rank 0 process prints out the time it took to complete 'algorithm'.

Performance Analysis

In the performance analysis we talk about the memory usage of physical and virtual memory and have a discussion regarding the relationship between the run time and the cores. Another note, the OpenMP and Pthreads data did not collect a single core run because we underestimated the time it would take for that run to work. The 'sbatch' time constraint was requested to have 3 hours which we assumed would be plenty of time, however it did not complete. Therefore, our base case for OpenMP and Pthreads are our 2 core runs.

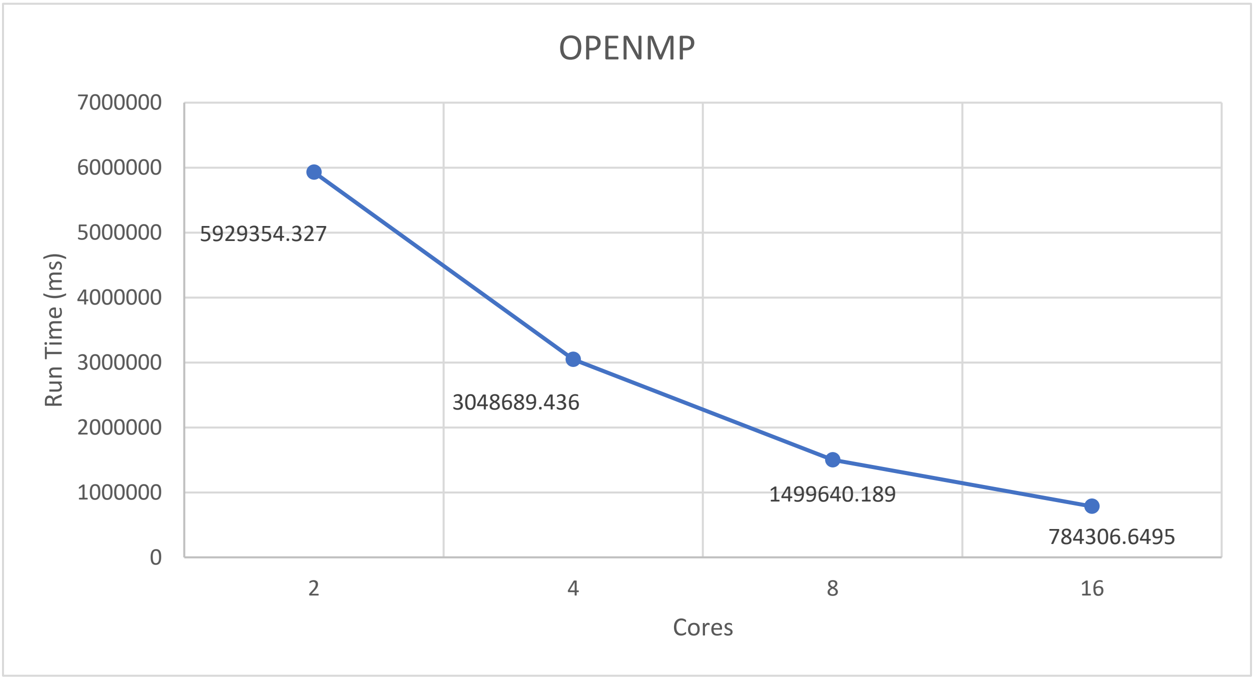
OpenMP

Our OpenMP implementation is the simplest using only a few commands to parallelize the code. These are compiler directives which remove almost all the synchronization and parallelization work from us. The fastest run time was our 16 core run which had almost and 8 time speed up compared to our 2 core run. The runs were ran 5 times each to get a more accurate average run time.

CORES	AVG TIME
2	5929354.327
4	3048689.436(2x)
8	1499640.189(4x)
16	784306.6495(7.5x)

The virtual memory usage by the 2 core run was 8733.124 MB while the 4 core run was close to 8804.152 MB and the 16 core run was 10161.300 MB. This is likely related to inefficient memory usage and incorrect freeing of objects that were allocated throughout the program. The physical memory used by the 2 core run was 6291.641 MB while the 4 core run came near 8747.052 MB and our 16 core run had close to 9179.347 MB of physical memory use. Between all the runs the 2 core run stood out because the physical memory used was much lower than the virtual memory assigned. A difference of almost 2000 MB between the values while among the higher core jobs the two memories remained relatively close.

The graph below shows the run time in milliseconds vs the number of cores used. Here we can see that the slowest run time is the 2 core job and the fastest run time is the 16 core job. The speed up between the job run times is about 7.5 times between the 2 to 16 core jobs. Each time the core amount double (2 to 4 to 8 to 16), the run time almost cuts itself in half.



Pthreads

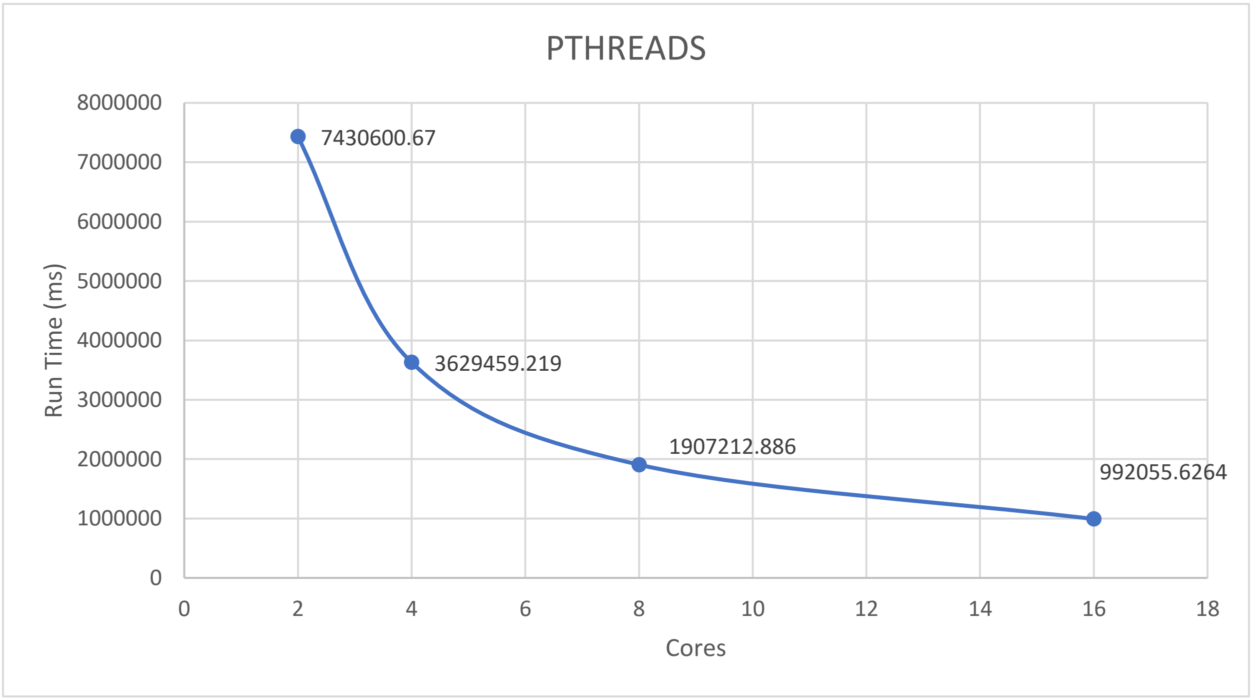
Pthreads are less of an abstraction than OpenMP giving us more control and power over the code implementation. This also gives more room for errors because of the control. Each run was also ran 5 times and the times were averaged among them to get a more accurate result.

CORES	AVG TIME
2	7430600.67
4	3629459.219(2x)

8	1907212.886(3.8x)
16	992055.6264(7.5x)

The virtual memory used by the 2 core job was 1774.23MB, the 4 core job had 2013.45MB and the 16 core job was given 2872.016MB. While the physical memory used by the 2 core job was 1691.63 MB, the 4 core job used 1691.59MB, the 16 core job used 1694.14MB. The physical memory remained very constant throughout the runs while the virtual memory fluctuated and increased as the number of cores increased. The biggest difference between the physical and virtual memory comes at the 16 core job which has almost 8000 MB more virtual memory than physical memory.

The graph below shows the run time vs the cores used of pthreads for processing the wiki_dump.txt file. The graph shows that the slowest run time was the 2 core run having about 2 hours of run time while the fastest run time was our 16 core job which ran in about 16 minutes. This is close to a 7.5 times speed up from the 2 core job. From this trend it seems to show that as the core number doubles the run time is almost cut in half.



MPI

MPI allows us to run distributed programs meaning that they are placed on different machines. While in OpenMP and Pthreads latency was not an issue because there was no need for communication, with MPI latency can cause the overall run time to be slower. However, our implementation removed the need for much of the communication making latency almost have no effect on the run time.

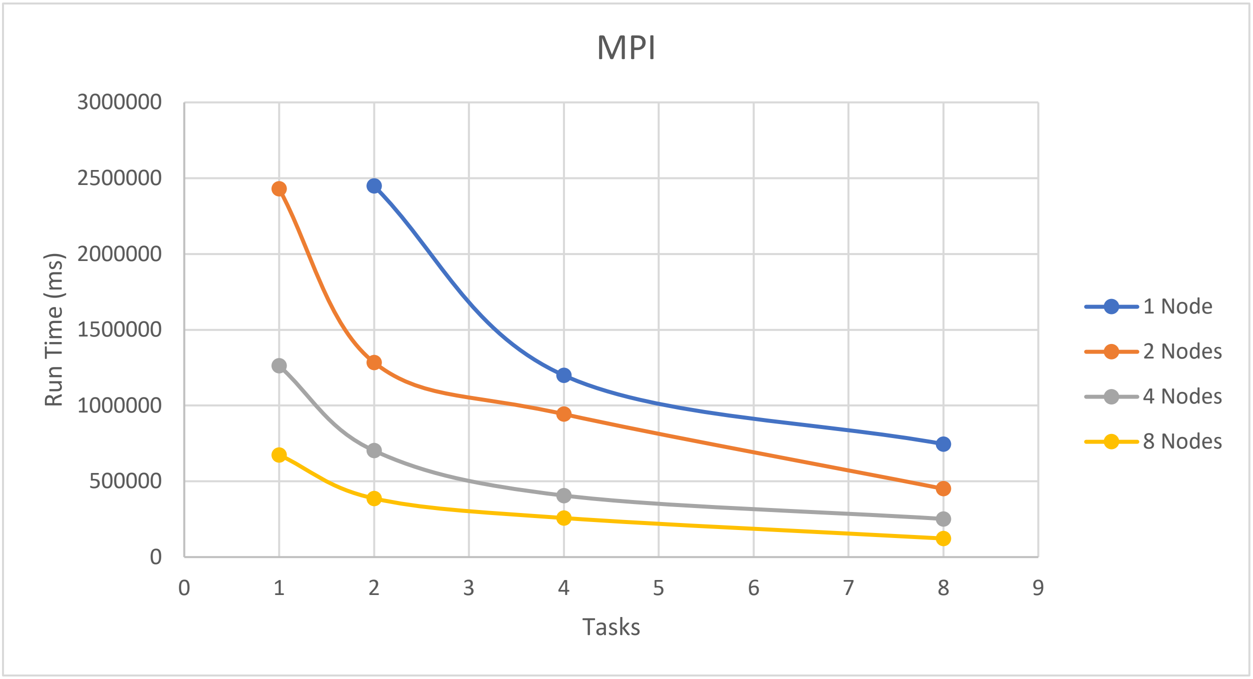
The table below shows the run time with the left column showing how many nodes and how many tasks were requested and the right column showing the overall run time in milliseconds.

We are using the 1 NODE 2 TASKs(Cores) as the base case from which all comparisons are done.

CORE/PROCESS	AVG TIME
1 NODE 2 TASK	2448511.855
1 NODE 4 TASK	1198931.825(2x)
1 NODE 8 TASK	745807.4452(3.2x)
2 NODE 1 TASK	2430192.893(0x)
2 NODE 2 TASK	1284412.489(2x)
2 NODE 4 TASK	943385.153(2.6x)
2 NODE 8 TASK	450769.6562(5.4x)
4 NODE 1 TASK	1261922.666(2x)
4 NODE 2 TASK	702071.1328(3.5x)
4 NODE 4 TASK	405994.088(6x)
4 NODE 8 TASK	251912.2472(9.7x)
8 NODE 1 TASK	673953.748(3.6x)
8 NODE 2 TASK	387103.1594(6.3x)
8 NODE 4 TASK	257629.9768(9.5x)
8 NODE 8 TASK	122586.4926(20x)

For the 1Node 2Task job the virtual memory usage for the main task was 4756.26 MB and the physical memory was 3109.76 MB. For the 2Node 8Task job the main thread had 1046.71 MB of virtual memory usage and 639.784 MB for the physical memory usage. Because of our implementation we only read the necessary amount of data into memory, each task only allocates memory for the part of the file that is necessary to read. The 4Node 8Task run had a virtual memory use for the main thread of 7758.44 MB and only 368.98 MB for the physical memory. The 8Node 8Task run used 636.67 MB with the virtual memory only being 229.84 MB. Throughout all the runs the virtual and physical memory continuously decreased as the number of Nodes and Tasks increased. The 64 Task run had the least amount of memory for a single process, however we must take into account that this was divided among the processes.

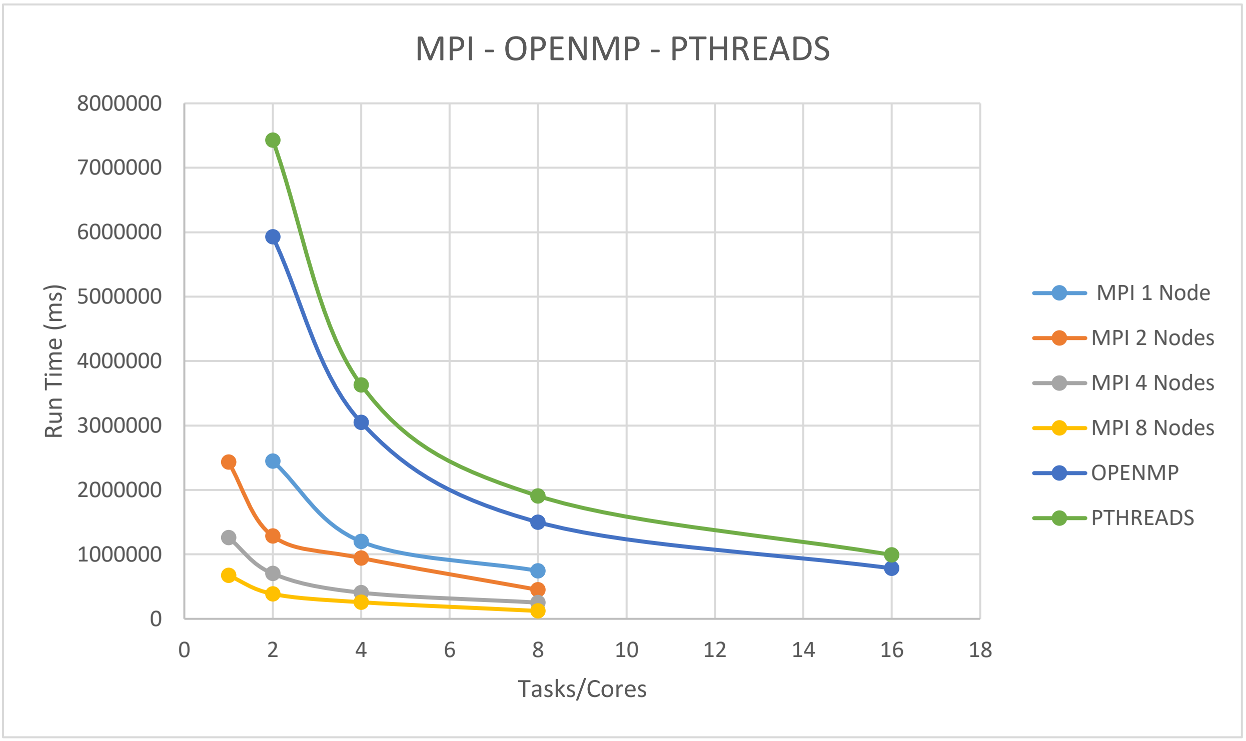
The graph below shows the run time with the number of nodes and tasks on each node. From the graph it is clear that splitting up the jobs was much faster then adding more cores. Even the 1 node 2 task was slower than the 8 node 1 tasks. This is probably because we eliminated a large portion of communication necessary for the algorithm to work. Therefore, eliminating almost all the latency between process communication.



The graph above more easily portrays the differences between splitting the tasks up. Even with 8 Tasks on 1 Node, the 8 Node and 1 Task per node was slightly faster. This could be attributed to Tasks having their own machine, not having to share RAM, and very little communication.

Conclusion

The final graph in this analysis is a composition of the previous three graphs to show in a clearer way which versions of the program had the fastest run time.



From the graph above we can clearly see that the fastest run time overall was from the 64-process job using MPI. This is because the tasks the find the longest common substring takes a generous amount of compute time and memory. Being able to split that task up efficiently with little communication means that the comp-to-comm ration is very good. A lot of computation is happening and little latency affecting the overall run time. Pthreads and MPI are both slower in almost all aspects. The only time that the two become faster then MPI is when they have more tasks than MPI and even then as we can see with the 8-task job of OpenMP it is still slower than the 1 Node 4 tasks job. This could be attributed to the code quality and that the processes don’t need to allocate as much memory in the MPI version because they are separate processes and only allocate the space for the work they are doing. While in the Pthreads and OpenMP versions they must allocate space for the whole file. On top of that Pthreads and OpenMP both have critical sections that make execution halt and cause threads to wait. Where in MPI there is no waiting for other processes besides the MPI_Barrier to wait for all processes to finish. Overall, MPI had the fastest run time followed by OpenMP and finally Pthreads.