Performance Analysis of Parallelism (OpenMP vs. PThreads)

Lev Kavs, levnikolaj@ksu.edu, Kansas State University

Hardware and Software Specifications

To receive more accurate results, we constrained the machines we used to only 'elves'. Elves can have machines can have either a two-processor machine working with 8-Core Xeon E5-2690 processors or a two-processor machine working with 10-Core Xeon e5-2690 v2 processors.

The OS Beocat is using is the 'CentOS Linux 7 (Core)'. The Linux kernel version is '3.10.0-862.11.6.el7.x86_64'. The code is compiled with the GCC version 4.8.5(Red Hat 4.8.5 - 28). Slurm 18.08.0.

Commands used to find these: cat etc/os-release, uname -r, gcc -v, scontrol -V.

OpenMP Code Version

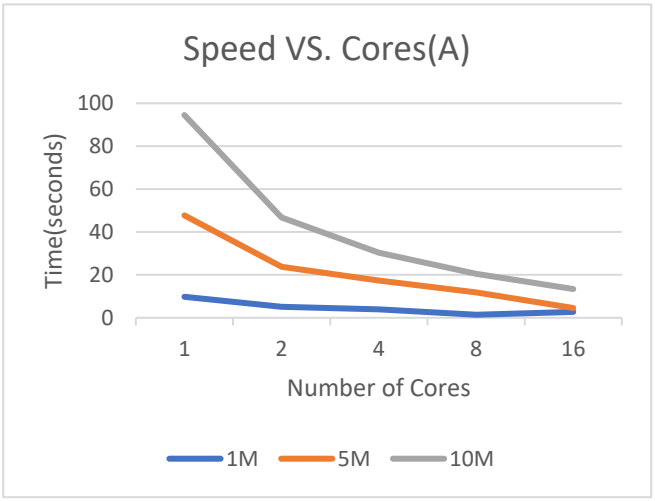
The first version of the code being discussed was parallelized using the OpenMP library (specifically OpenMP version 3.1). OpenMP uses preprocessor directives that modify is used by the C compiler (GCC) that transforms the program before actual compilation. We are using the '#pragma' directive which gives the machine or operating-system additional features. Information taken from https://developerinsider.co/preprocessor-directives-c-programming/. The goals of OpenMP is to be 'Lean and Mean' and provide significant parallelism using only three or four directives. It is designed for multi-processors/core, shared memory machines. A key difference between OpenMP and PThreads is that OpenMP is more abstracted (hides complexity) than PThreads. Taken from https://computing.llnl.gov/tutorials/openMP/.

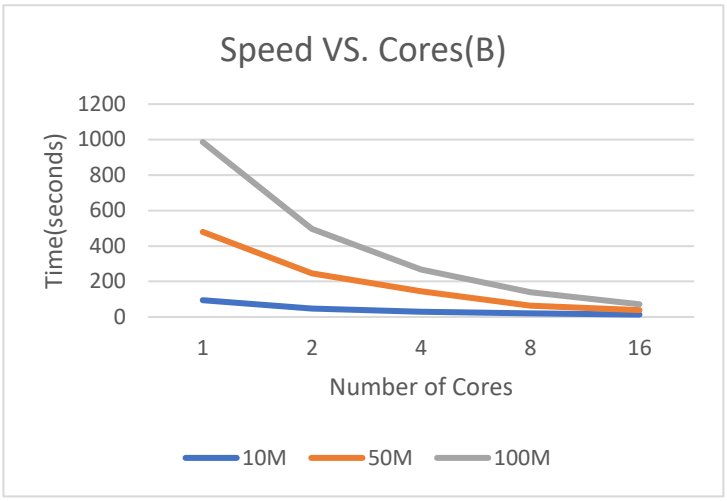
The variances between trial runs are the number of iterations done by the loop and the number of threads (cores from the machine) used to complete each program run. Each column entry is averaged from 5 runs with the top row being the number of iterations and the farthest column being the number of cores used in that run.

Cores	1M	5M	10M	50M	100M	500M	1B
1(base)	9.8	47.7	94	480	986	4788	9865
2	5.1(2x)	23.8(2x)	46(2x)	247(2x)	497(2x)	2500(2x)	5014(2x)
4	3.9(3x)	17.3(3x)	30(3x)	145(3x)	267(3.7x)	1498(3x)	2529(4x)
8	1.4(7x)	11.9(4x)	21(5x)	63(8x)	140(7x)	643(7x)	1277(8x)
16	2.7(4x)	4.6(10x)	13(7x)	39(12x)	72(14x)	322(15x)	632(16x)

The values in parentheses following the runtime (milliseconds) represents the factor by which the runtime has decreased from the base (1 core) run. As the core quantity increases so does the virtual memory, however it remains constant through various iteration amounts. The physical memory remains quite constant floating around 604 KB.

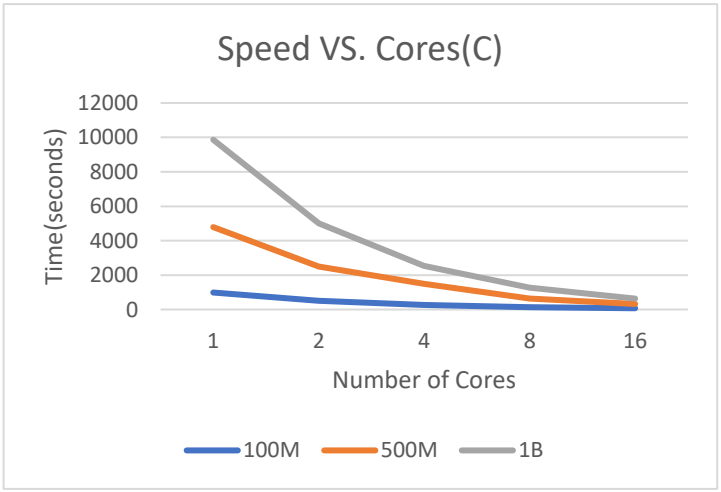
Virtual Mem	8172 KB	16908 KB	33300 KB	66084 KB	131652 KB





Graph A shows the runs for 1, 5, and 10 million loop iterations. The time taken to complete the loop decreases as the number of cores increases. This is because each core is assigned a different section of the loop iteration. We see that for the 1 million iterations increasing the number of cores does not drastically affect the speed of the program. This is because the data set is small. However, from 8 to 16 cores the average time slows. This is because the runtimes for 16 cores and 1 million iterations are: 6.064, 0.986, 0.912, 0.982, and 4.569. This is possibly due to the increased amount of overhead involving creating and allocating 16 threads and the corresponding small task size.

For the 10 million iterations the runtime with 1 core is 94 milliseconds, with 16 cores the time is decreased to 13 milliseconds. This is a speed up of a factor of 7 times. The virtual memory used with 1 core was 8172 KB which continues to increase to 131652 KB which is the amount used by 16 cores. 131652/8172 =~ 16, we can then conclude that each additional thread requires an additional 8172 KB. For the 50 million iterations run the overall speed up going from 1 to 16 cores is a factor of 12. For the 100 million iterations run the overall speed up from 1 to 16 cores is 14 times faster.



The trend of increase continues with Graph C. where we see for 500M iterations having a speed up of a factor of 15. And for 1 billion iterations a final factor of 16.

With this we can conclude that working with small data size adding more cores does not always increase runtime speed. Adding too many threads/cores can cause the program and processor to have too much overhead, in turn decreasing overall runtime.

PThread Code Version

The second version of the parallelized code was done using PThreads and their accompanying library. PThreads are the less abstract version of OpenMP, where the programmer has more control over the creation and use of threads. PThreads are suited for parallel programming, whatever applies to parallel programs also applies to parallel PThread programs. A few functionalities of threads: they all have access to the same global/shared memory, they also have their own private data, with less abstraction means that the programmer is responsible for synchronizing access and protecting the shared data (remove race conditions). Taken from https://computing.llnl.gov/tutorials/pthreads/.

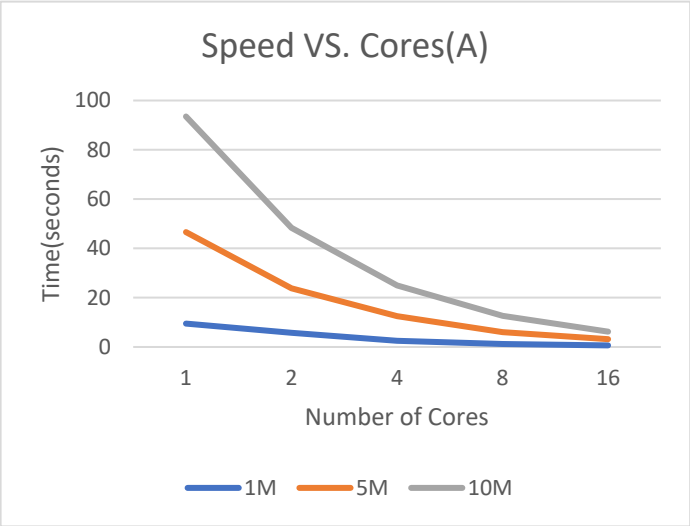
Each trial run was keep the same from OpenMP to PThreads for the ability to compare their performances. Each different number of cores and iterations was each ran 5 times and averaged for a more accurate run time. The units of each entry in the table is milliseconds and the entry in parentheses is the speed up factor from the 1 core (base) run.

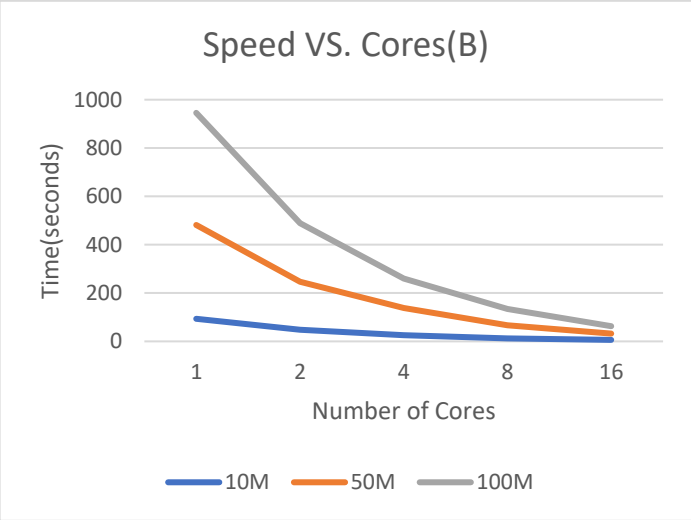
Cores	1M	5M	10M	50M	100M	500M	1B
1(base)	9.5	46.6	94	482	946	4768	9669
2	5.7(2x)	23.8(2x)	48(2x)	247(2x)	489(2x)	2480(2x)	4973(2x)
4	2.5(4x)	12.6(4x)	25(4x)	138(4x)	261(4x)	1254(4x)	2523(4x)
8	1.3(7x)	6.1(8x)	13(7x)	67(7x)	134(7x)	637(8x)	1264(8x)
16	0.6(16x)	3.2(15x)	6(16x)	32(15x)	63(15x)	314(15x)	638(15x)

The virtual memory table hold the values of the amount of memory used for each type of core run. Going from left to right it holds the memory value for 1 core, 2 cores, 4 cores, 8 cores, and 16 cores

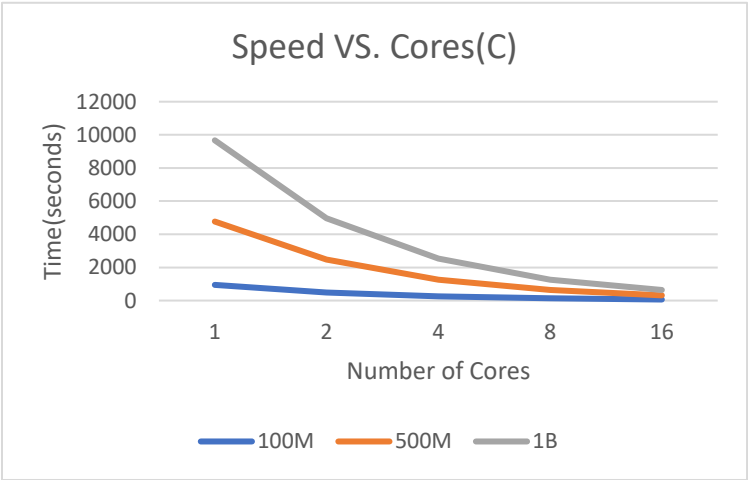
vMem	14704 KB	22900 KB	39292 KB	72076 KB	137644 KB

The virtual memory used for 1 core is 14704 KB, for 16 cores is 137644 KB. 137644/14704 =~ 9. Our data shows that PThreads and OpenMP both at 16 cores have approximately similar virtual memory use. However, at 1 core PThreads uses more virtual memory. This could be because the code working with PThreads is more complex and longer. But as the core number increases, PThreads does not use as much memory per thread compared to OpenMP as the thread count increases. If we continued adding cores we might see OpenMP using more virtual memory than PThreads. The PThreads also seem to hover at a lower physical memory amount, around 380KB.





Graph A, B, and C show the run times for a different number of iterations as the number of cores changes. Unlike in the OpenMP 1 million iterations run where the average runtime from 8 to 16 cores slows down. The PThread 1 million iterations run sees a speed up from 8 to 16 cores. An explanation for seeing an increase instead of a decrease could be that with the lesser abstraction of PThreads decreased the overall overhead of allocating and creating of threads.



Analyzing the data, we see that for 1 million iterations using PThreads, increasing the cores from 1 to 16, the factor of speed up is 16x, while the more abstract OpenMP version only saw an 4x factor speed up. The 10 million iterations run also saw the same 16x factor speed up. Every single other run saw a factor of 15x run time speed increase.

Final Analysis

Through the various iterations and core numbers we can conclude a few things. First, because PThreads give us more control over the flow of execution and data access they appear to be more optimized and useful when working with smaller data sizes. The speed up from PThreads and 16 cores has a similar change across all different iteration runs, staying between a factor of 15 to 16 times runtime speed up. In comparison, OpenMP runs do not have a constant 16 core speed up on all task sizes. Instead as the number of iterations increase so did the speed up from using 16 cores. Even though our sample size of different iteration runs is limited, we can hypothesize that the increasing speed up of OpenMP would continue as the number of iterations increased. This could eventually show that for very large data sets OpenMP is faster

and more optimized in comparison to PThreads. However, we must also consider that this result could come from programmer inefficiency.

When discussing the memory efficiency of PThreads to OpenMP, OpenMP start with using less code with one core. We did discuss how this could relate to the smaller amount of code required to use OpenMP. We then see as we continue to add cores OpenMP uses an additional constant 8172 KB per additional core, while PThreads use a smaller amount per additional core.