David Rohweder
Parallelization Analysis
Penn State University
State College, PA, USA
Djr6005@psu.edu
March 37, 2022

# 1 Introduction

For this research we will attempt to reduce the execution time of complex code using parallelization. We will have two benchmarks to parallelize and study. To begin our analysis, we will first need to give the specifics of the equipment and tools used in the experiment. First, the study was executed on the Penn State Roar supercomputer using the ACI-b Standard Core resource profile and queued batch jobs to fully utilize open allocation on the cluster. The allocation for this setting is as follows: 20 available cores, InfiniBand interconnect, and 256 GB total RAM (for open allocation users). More specifically, the VM is using an Intel Xeon CPU E5-2680 v3 @ 2.5GHZ with 32KB L1 cache, 256KB L2 cache, and 25600KB L3 cache. It is important to note that this experiment utilizes multiple cores for program execution because we aim to study the effects parallelization can have on code performance. Next, we will further discuss the specific scenarios we aim to explore and compare to further our understanding of benchmarking different computations. The first benchmark our experiment sought to analyze was a mean and standard deviation experiment that also computed the minimum and maximum and values over a predetermined threshold value. The analysis of this benchmark will consist of a comparison of how parallelization will improve the execution time. Secondly, we will be testing the matrix matrix multiplication benchmark. In this benchmark we will be looking for how we can improve the performance of this benchmark using the loop blocking technique, parallelization, and combination of both on the execution time of the experiment. Finally, we will analyze the effects cache has on parallelization and the scalability of parallelizing programs and code.

# 2 Analysis

## Mean and Standard Deviation

In the first benchmark test we sought to speed up the original mean and standard deviation calculations and min/ max finding using OpenMP. Then, after calculating the mean and standard deviation we wanted to find the indices of all the elements that exceeded a predetermined threshold and store those values in an array.

In the original code, the mean was found by iterating over the array of elements. Then, after the loop exited the mean was calculated by dividing the sum over the number of elements. When introducing OpenMP to increase the speed of the code execution, it is necessary to break up the code into individual steps. The first step was to isolate the job. The job here for each thread to do was to accumulate the array given. Second, we need to partition the job so that each thread is solving an equal portion of the array or so that each thread has a problem size of N / P (where p is thread number and n is problem size). The work will be evenly divided into each of the threads in the thread pool because of the

OpenMP for directive which is a work-construct that automatically gives each thread a portion of the problem, the amount of which is specified by the scheduling method. Since the computational domain is relatively simple, load balancing and partitioning can be done with the static scheduling method of partitioning work in OpenMP without worry of a huge work imbalance. Next, we need to execute the threads by firing off each thread for the total amount of threads specified which is done automatically in OpenMP when the parallel region is entered in the code. Then, the Fork-Join model will be used to fire off the threads. After all the threads are executed, they must be joined together and wait until all threads have finished execution before the master execution continues. In OpenMP this process is also automatically done with implicit and mandatory barriers (synchronization points) at the end of certain pragma directives. Now that all the threads have computed the local sum of the array from the threads start to end point, we will use the reduction method and apply the addition binary associative operator to each of the partial results of each of the threads. After the partial results are combined using the addition operator then the result is pushed back into the main mean value in the function. After the total is found and placed into the mean variable we can compute the mean by dividing the mean variable which as of now is just the sum by N. When just only implementing the change, with 20 threads, we get a speedup of 3.7s serial / 2.88s parallel = 1.28X. However, we also want to speed up the standard deviation calculation as well. Applying the same process as before we can isolate the for-loop calculation where the operations are accumulated to the sd_temp variable. This part can be isolated and broken into its OpenMP parallel region to be executed in parallel. Next, as before we will use the OpenMP for directive to specify that we will be using the for work-sharing construct as well as the static scheduling method. Now that the standard deviation process has been isolated into each function and each thread contains information on which part of the array to iterate over, we can fire off the threads and execute. Like before, each thread will be automatically fired off from the thread pool when it enters the parallel region and will contain the information that the scheduling method sends to it. Also, we will have to wait for the threads to reach all the barriers before finishing. Then, like before we will use the reduction method and the binary associative addition operator over all the partial results in each of the threads. Now, we can calculate the standard deviation by taking the square root of the sum / N. When implementing just this change, with 20 threads, we get a speedup of 3.7s / 2.79 =1.32X. With both threaded calculations together, with 20 threads, we get a speedup of 3.7s / 1.7s = 2.17X. Another statistical calculation that is collected in the calculations is the min and max element of the array. In the original program, after the standard deviation calculation occurs, another loop goes through the whole array again looking for the min and max elements. However, this operation is completely unnecessary and imposes redundant computations. In the standard deviation calculation, we can easily integrate the search for the min and max elements and store the local min and max of each thread into privatized versions in the OpenMP parallel block. Along with the addition reduction on sd_temp we can also do min and max reductions on the local min and max partial results of each of the threads. When integrating this computation into the standard deviation calculation, with 20 threads, we get a speedup of 3.7s / 1.56s = 2.37X. This means that the overall speedup of threading the original function is 3.74 s serial/ .46s parallel = 8.13X speedup.

In the analysis, we looked at the speedup when threading the code with 20 threads at N =

1,000,000,000 where it resides in RAM. In theory, when adding a new thread, the speedup should be 2X. For instance, if you have 1000 tasks and add a thread then you have half the amount of work to do. However, in practice that ideal result is not the case because of communication overhead, load imbalances and OS jitter delaying tasks, false sharing, shared hardware, synchronization loss, startup overhead, and many more factors.

When forcing the array into lower cache levels and then threading the task, we can use bigger array sizes before spilling over to the next cache level since each processor has their own cache (sometimes they can share L2 or L3). To pick the size of N that will force the program to occupy these cache levels is easy. For L1 cache we follow the following formula: (Thread# * 32KB / 8 double size) is the max size of an array to fit into L1 cache. Next, for L2 cache we follow the following formula: (Thread# * 256KB /8 double size) is the max size of an array to fit into L2 cache. Finally, for L3 cache we use the following formula: (Thread# * 25600KB / 8 double size) is the max array size of an array to fit into L3 cache. These formulas are calculated by factoring in the number of threads which each have their own cache, along with the size of the cache, and finally that result divided by the byte size of the array type double which is 8 on this architecture.

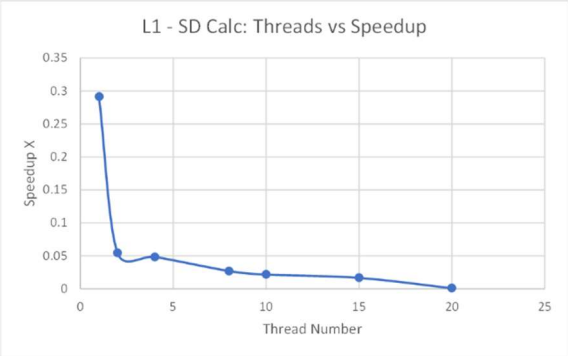


*Figure 2.1: L1 Cache Threads vs Speedup*

As mentioned before threads have a lot of overhead that come from system calls to the OS, setting up address spaces, and many more. Therefore, when comparing threads compared to serial execution in L1 cache, or the fastest memory, we can see that the parallel version is several times slower than serial. Moreover, as more threads are added to the already slow problem, we can see the rapid decline in performance as we go to 20 threads. In this data set, using the L1 cache formula above we got a size of N = 2000 (32KB * .5 / 8) for our array size. In short, the reason the performance declines rapidly with more threads compared to serial version is all the compounded setup and overhead for such a small data set. This effect can be visualized in *Figure 2.1*.

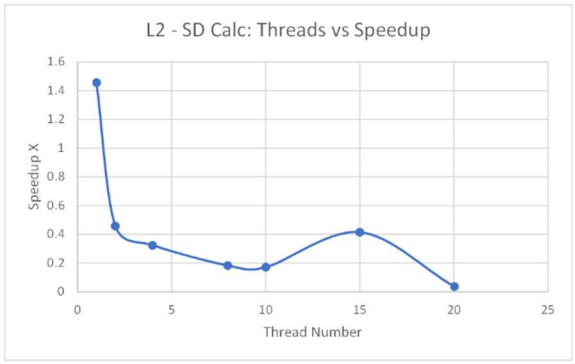


*Figure 2.2: L2 Cache Threads vs Speedup*

The same effect can be seen with L2 cache compared to L1 cache. The calculated size for the L2 cache array size was N = 16,000 (256KB * .5 /8). However, since there is a bigger data size, there is still a slowdown, but the penalty and overhead are somewhat made up for due to the larger data to be computed from the threads. This explains why the curve in *Figure 2.2* is not as steep of a performance degrade as *Figure 2.1* when more threads are added.

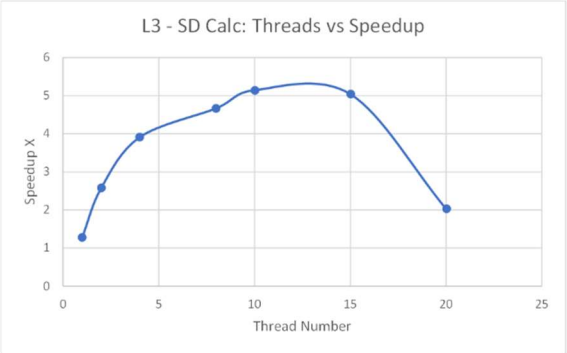


*Figure 2.3: L3 Cache Threads vs Speedup*

Different from L2 and L3 cache we get a speedup in L3 cache. This can be attributed to the array size N = 1,600,000 (25,600,000 * .5 / 8). Since the array size is so much larger than the array sizes of the L1 and L2 cache, the performance hits from overhead are diminished greatly since each thread has a lot more work to do. However, as more threads are added it is apparent that the threads share L3 cache on this system because from 10 threads until 20 threads we see the speedup factor decline slightly as shown in *Figure 2.3*.

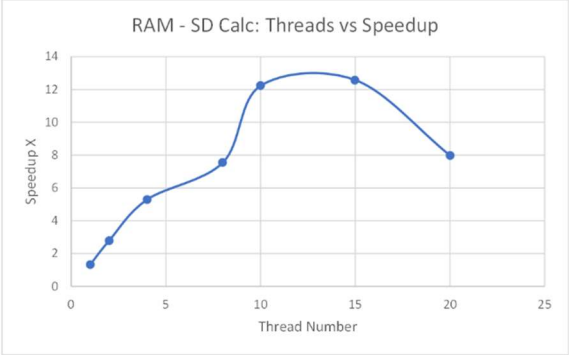


*Figure 2.4: RAM Threads vs Speedup*

The final and slowest memory level has the greatest speedup when parallelizing the code compared to when the task size can fit into the faster, but smaller cache memory. This is because of the factors mentioned before and overhead becoming so irrelevant due to the massive task size to be in RAM N = 1,000,000,000. As we can see from *Figure 2.4* there is a perfect correlation between the predicted speedup vs the experimental speedup for the first couple threads. However, for the additional threads we do see a slight impact from the additional threads as the correlation is not perfect; however, unlike L3 cache the task size is so large there is still a massive speedup factor.

The next computation that was compared was the threshold function, which found, counted, and placed elements whose values was over a predetermined limit into a new array storing the elements index. First, the original function iterated over each element of the array and compared the current elements value to the threshold and counted all the elements that were over. After, the array was iterated over again and each element whose value was over the threshold had its index placed in another array. To speed up the original code, we parallelized it using OpenMP. First, when counting the elements that were over the threshold, the loop iterating over the array was identified as the job that needs to be isolated. Next, we needed to use the for work-sharing construct to give each thread an equal amount of work to do according to the static scheduling method. After, we had to fire off the thread pool and wait for each thread to finish or reach the mandatory synchronization point at the end of the parallel block. Then, after the execution of the threads the result was computed using the reduction method and the addition operator on all of the threads partial results. The speedup, with 20 threads, for this code was 2.16 / 1.8s = 1.2X. Next, we needed to parallelize putting elements over the threshold value into a new array containing the arrays index. First, we needed to identify the job which was to check all elements of the array and see if the value of the element was over the threshold value, and if so the index of the array should be placed in a new array containing the index. Next, we needed to isolate the task we just identified into a parallel block and partition the task into jobs for each thread. The partition steps follow the

steps taken in previous tasks with the for work sharing construct and static scheduling method. Next, we need to fire off each thread to work on their task and then wait for them to finish and join them all together. Once all the threads are done, we can finish since the elements over the threshold will be put in the array via reference and therefore no data is sent back. When only parallelizing this function the speedup, with 20 threads, is 2.32s/ 1.7s = 1.36X. When combining both parallelization's for counting the elements over the threshold and then putting those elements into a threshold array the speedup, with 20 threads, was 2.1s/ 1.454s = 1.45X.

Like the previous mean and standard deviation threaded code, we saw similar results when comparing threads to speed up in the cache levels and RAM. Below in *Figures 2.5 – 2.8* we can see the computed results.

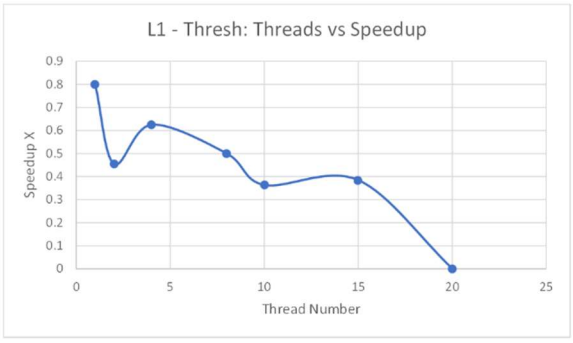


*Figure 2.5: L1 Cache Threads vs Speedup*

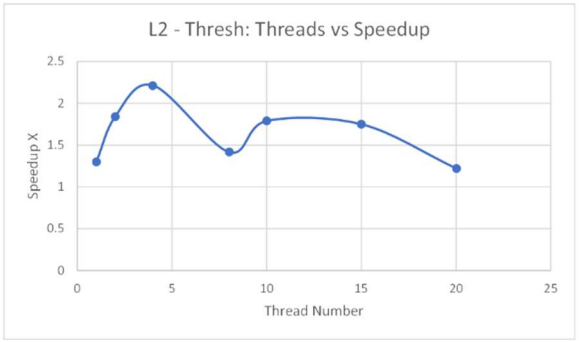


*Figure 2.6: L2 Cache Threads vs Speedup*

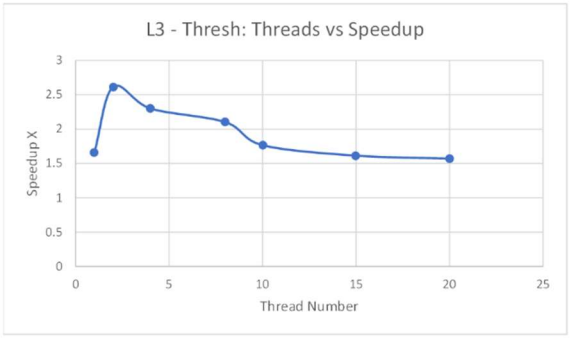


*Figure 2.7: L3 Cache Threads vs Speedup*

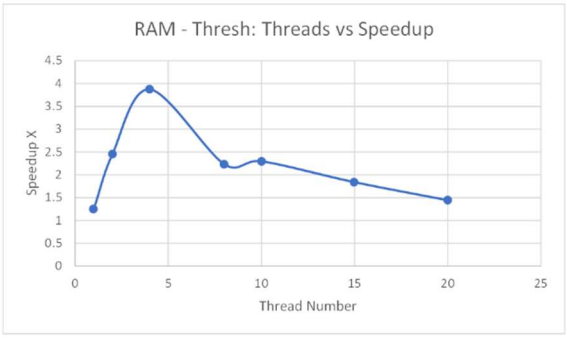


*Figure 2.8: RAM Cache Threads vs Speedup*

## Matrix Matrix Multiplication

To begin, for the matrix matrix multiplication we sought to analyze how this algorithm can become more efficient using several strategies. The original function for the matrix matrix code has a runtime of $O(N^3)$ as for each entry it multiplies by every cell down the row and column. The execution time of this function without any improvements is around 17s. However, multiple improvements can be made to drastically reduce the execution time.

The first method of improving performance is to use parallelization and in this test, we used 20 threads. The first step to parallelizing the code is to isolate the job which in this case is the triple for loop structure computing the calculation. Next, we need to partition the work so that each thread has an equal amount

of work to compute on the outer loop. The work will be partitioned via the parallel for work-sharing construct in OpenMP and the threads will be fired off once the code enters the parallel region. Next, now that the threads have been fired off, we must wait for all of them to finish executing which will happen once all of the threads reach the mandatory barrier at the end of the parallel block. Once the threads finish execution, the results are already calculated to the storage array so there is no other calculations to be done. Note: since each thread will be working with different indices in the storage array there is no concern regarding race conditions and therefore no need for a critical block. The execution time of parallelizing the original function was (4s-4.5s) and the speedup was 17.44s / 3.27s = 5.33X speedup.

The next method to improve the code performance is by blocking the calculations so that they fit into cache. This does not happen when the calculations go down every row and down every column of the matrices and therefore there is little to no utilization of the values being put into cache, effectively cache thrashing, and not getting any benefit as all calculations are missing in cache. This is where the notion of loop blocking comes in because calculations are broken up into manageable chunks (in this case block sizes are 10) that can fit into cache and not become so huge that they spill over into lower-level memory without being utilized by cache first. By using loop blocking the execution time was (4s-5s) and the speedup of the code was 17s / 4.5s = 3.7X speedup.

The final performance improvement to decrease the execution was the combination of parallelization (still 20 threads) and loop blocking. This was done by partitioning the rows to be worked on in OpenMP. By breaking up the optimized version into smaller chunks such that each chunk has a range of rows to work on the work can be broken up more and reduce execution time. The combination of each thread having only certain rows to work on reduces the complexity of the calculations and along with loop blocking makes it so that caching works much more effectively. With the speedup of both loop blocking and parallelization on the code was 17.1s / .38s = 45X speedup.

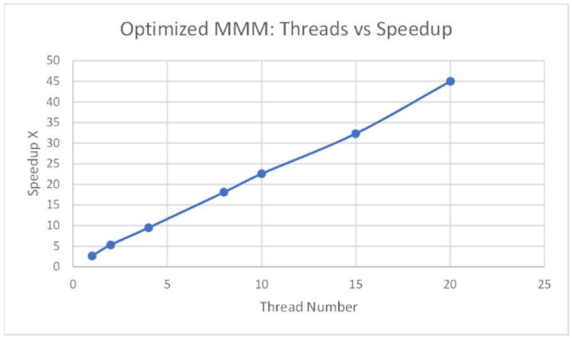


*Figure 3.1: Optimized Matrix Matrix Multiplication Threads vs Speed up*

In the optimized matrix matrix multiplication, we see that as more threads are added there is a huge performance increase with each new additional thread. Furthermore, since we are cache blocking, each cache in each thread is effectively used for maximum performance of each thread essentially zeroing out any performance hit from thread setup and overhead. Essentially, with the optimized MMM we see an effective usage of multiple threads being maximized for performance with cache usage per each thread.

# 3 Conclusion

In conclusion, we tested two different benchmarks to better understand the impact that parallelization has on the performance of different code scenarios. The results gathered clearly demonstrate that although more workers exist, the resulting speedup isn't a perfect divide by the number of processors. Moreover, while the program might be parallelized, it does not necessarily guarantee massive speed improvements. Although, there

were still clear and noticeable impacts on execution time from the parallelization. Furthermore, threading tends to favor larger data set applications rather than ones with smaller data sets as the serial has less overhead and set up time to complete minimal work in fast cache-based environments.

Appendix

While parallelizing code can often lead to some degree of performance under the correct conditions, it also leads to different computations of code compared to serial. This change causes small, but still relevant differences down to the smallest decimal value between serial and parallel implementations. For example, in the standard deviation calculation the mean had visibly the same values in the serial implementation as the parallel but was flagged as wrong for having different values. If the full value of the serial mean and parallel mean was shown the smallest decimal differences would be apparent. The reason for this difference of accuracy between the algorithms is due to the floating-point addition which is non-associative due to round-off errors. When a series is being computed, the master thread computes $s_1+\ldots+s_n$, where N is the size of elements in the array. However, when parallel computations are computed they result in a series of partial sums or $s_1^p+\ldots+s_n^p$, where N is the size of the thread pool, to be further combined and calculated. Since floating point addition is non-associative the order of operations matters when computing the mean result. However, there are many things that alter the order of the partial results being added in parallel environments causing non-deterministic execution. Furthermore, as operations and the total number of threads in the thread pool grow, the amount of thread interleave and permutations grow and consequently alter output results (Villa).

## Bibliography

Oreste Villa, et al. *Effects of Floating-Point Non-Associativity on Numerical Computations on Massively Multithreaded Systems*. Penn State University, https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.458.2473&rep=rep1&type=pdf.