Algorithm Design

MMScanSEQ

This algorithm computes the scan of a list. This scan uses matrix multiplication as its associative operator. This is done by looping through each element in the list, and then the matrix multiplication is applied to each element. This loop results in a time complexity of O(n).

MMScanDNC

This algorithm computes the scan of a list where the associative operation is matrix multiplication. This scan is computed using a divide and conquer algorithm. This algorithm divides the list in halves recursively until it is one element long. Then, as the recursive calls return, the matrix multiplication operation is applied to the right side. At one level deep, the last element on the left side is applied to all elements on the right side to complete the scan. This algorithm has a time complexity of O(nlog(n)). It has the advantage of exposing multiple opportunities for parallelism.

MMScanDNC1

This algorithm is based on MMScanDNC. It exploits the opportunity to create tasks for one recursive call. The second recursive call is left alone so the main thread can work on it. This algorithm has the time complexity of O((n*lg(n))/p) where p is the number of tasks spawned. We control number of tasks spawned as a command line parameter.

MMScanDNC2

This algorithm is based on MMScanDNC. It exploits the opportunity to parallelize the for loop which applies the scan operation to the subarrays. We optimize this parallelism by throttling the number of tasks spawned via a command line parameter and the size of the subarray. This has a time complexity of O(n*lg(n)).

MMScanDNC3

This algorithm exploits both opportunities previously used. We use the data from the previous two experiments to optimize the algorithm's parameters for limiting number of tasks spawned for the for loop and the recursive calls. This has a time complexity of O((n*lg(n)/p).

Experimental Design

Experimental Conditions

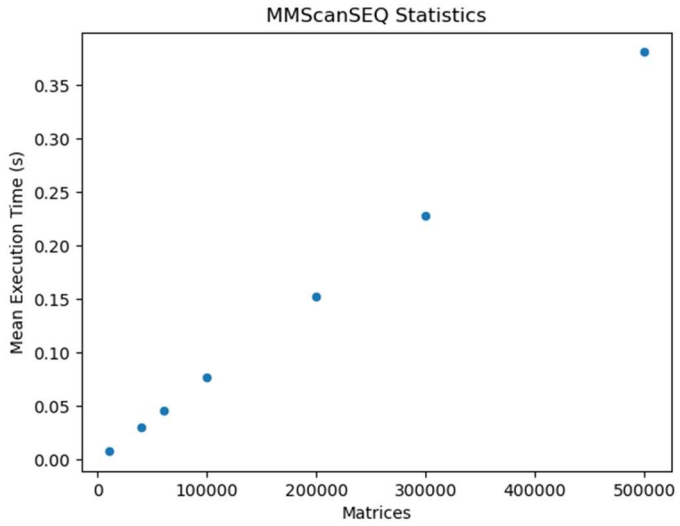
These experiments are conducted on the "Eldora" machine. This lab machine has 16 CPUs, the L1d cache is 256 KiB, the L1i cache is 256 KiB, the L2 cache is 2 MiB, and the L3 cache is 20 MiB. The experiments are conducted sequentially or with parallelism utilizing all 16 CPUs.

Experimental Results

The following experiments use a list of 500,000 matrices of size 6x6.

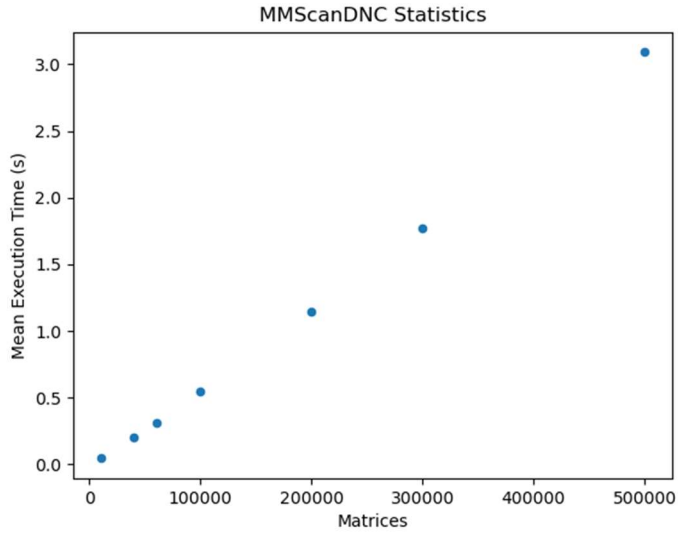
MMScanSEQ

Here we observe the expected linear complexity.



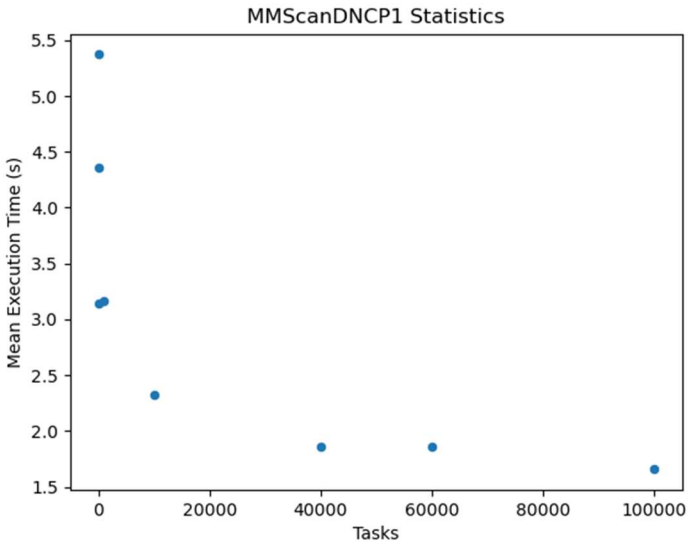
MMScanDNC

Surprisingly, this experiment resulted in a faster mean execution time for a large list of matrices. This is likely due to variances caused by other processes on the machine.



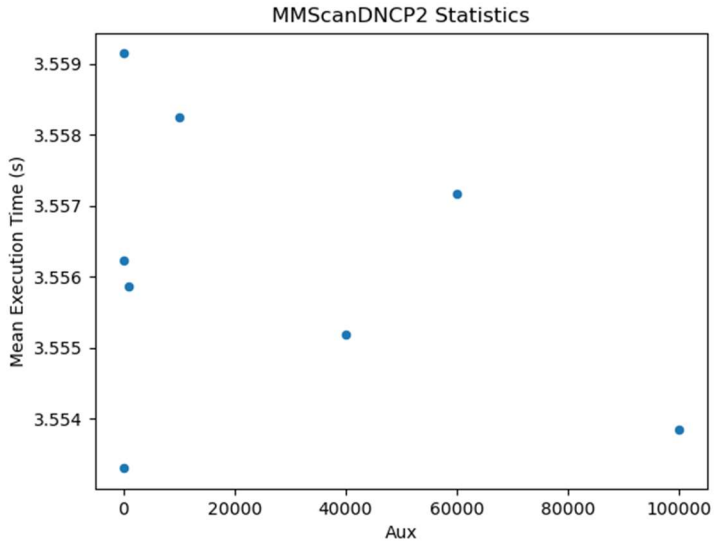
MMScanDNCP1

This experiment confirms our complexity of O((n*lg(n))/p), as we increase the number of tasks spawned for the recursive call, we observe the execution time asymptotically approaching 1.5 seconds. This motivated me to use values larger than 100,000 for P when conducting experiments using MMScanDNCP3.



MMScanDNCP2

This experiment was surprising. We do not see much variation when limiting the spawning of new tasks to large sub-arrays when scanning. The variations are within one thousandth of a second. We may be able to find a larger difference if we conducted experiments with larger amount of matrices. Because Aux = 1 had the fastest execution time, this experiment shows evidence that it is best not to limit the spawning of new tasks. However, the overhead associated with task creation should result in slowdown.



MMScanDNCP3

Here we see results confirming the results we found in the experiment with MMScanDNCP1, as well as better evidence of limiting task creation to large subarrays for the scanning operation. The fastest average execution time was found using a P value larger than 100,000, which limits the number of tasks created when recursively dividing the list of matrices. Additionally, limiting task creation to sub-arrays larger than 100,000 matrices during scanning resulted in the fastest time.

Aux	P	Mean Execution Time (s)
100000	300000	1.6781635
60000	40000	1.884244667
40000	300000	1.887779667
10000	40000	2.342398667
1000	40000	3.2078835
1	60000	3.597465667
100	40000	4.582067333
10	100000	5.946677333

Conclusion

In conclusion, we see that we obtain the best performance when utilizing all available methods of parallelization when it is tuned correctly. Tuning depends on the individual machine, and it is best to determine using automation as it requires many tests to tune. We can obtain better execution time than sequential execution although the time complexity of the algorithm being parallized is worse than the sequential version..