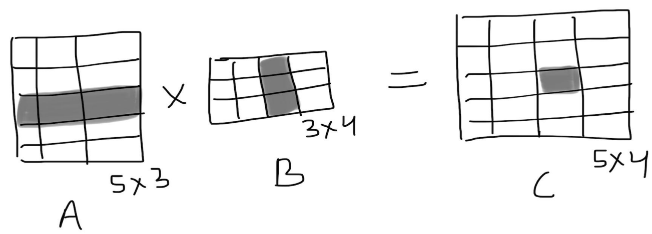
COEN 319 - ASSIGNMENT 1 REPORT
BHANU PRAKASH NAREDLA
00001630571

I. Approach

I have parallelized matrix-multiplication program using two different approaches

Approach-1:

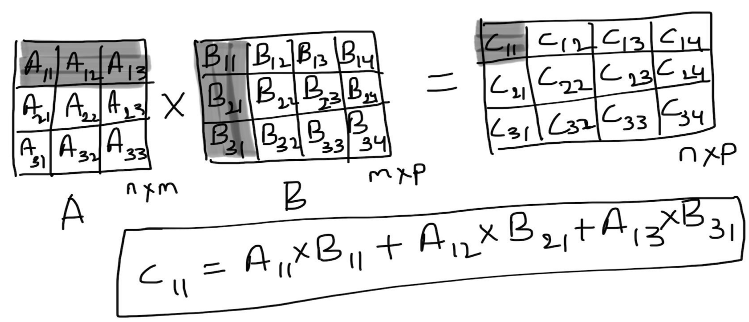
- In the serial matrix-multiplication program we are computing each element of the resultant matrix serially, as the computation of each element in the resulting matrix is not dependent on other elements, serially computing each element is not efficient. So, I decided to parallelize the execution with multiple threads where each thread is responsible to compute each element in the resulting matrix.



- As the number of elements in the resulting matrix can be large, creating and managing threads becomes overhead when compared to work done by each thread. So, I decided to run the program using a different number of threads and on each run we are sharing the computation load equally among the threads.

Approach-2:

- In the above approach we are not utilizing the cache locality i.e., As the matrix sizes can be large, once we multiply a row of matrix A with a column of matrix B the row elements of matrix A will be evicted from the cache and those elements need to be fetched again to compute other elements of matrix C, which is a expensive operation and can slow down our program.



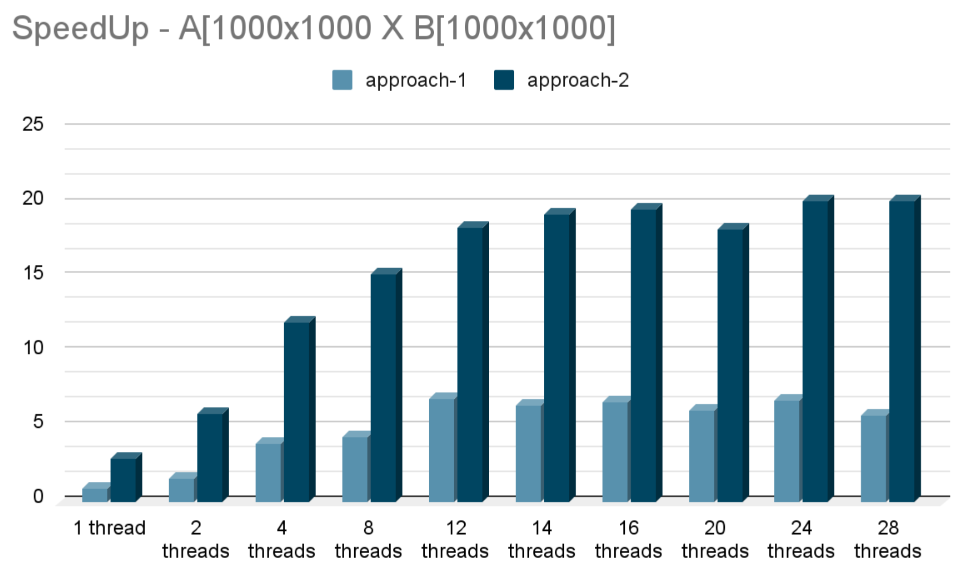
- So I divided the matrices into submatrices(block) of size bxb, where the block of matrix A, block of matrix B and block of matrix C will fit in the cache. Each block of matrix C can be computed serially similar to the figure shown above, By this we are utilizing the advantage of cache locality.
- Now, I decided to parallelize the execution with multiple threads where each thread is responsible to compute each block in the resulting matrix C. As the number of blocks in the resulting matrix can be large, creating and managing threads becomes overhead when compared to work done by each thread. So, I decided to run the program using a different number of threads and on each run we are sharing the computation load equally among the threads.
- I have also experimented with multiple block size b and block size 25 gave best results. This approach runs faster than the previous approach as we are utilizing cache locality.

II. Results

(i) A[1000x1000] X B[1000x1000]

- Time taken with sequential is : 3.224443890 sec

Num threads	1	2	4	8	12	14	16	20	24	28
approach-1	3.298640	1.957777	0.808804	0.734392	0.459575	0.496615	0.478679	0.517866	0.471856	0.550413
approach-2	1.098944	0.536600	0.267823	0.210237	0.174610	0.166228	0.164180	0.175991	0.158990	0.159304

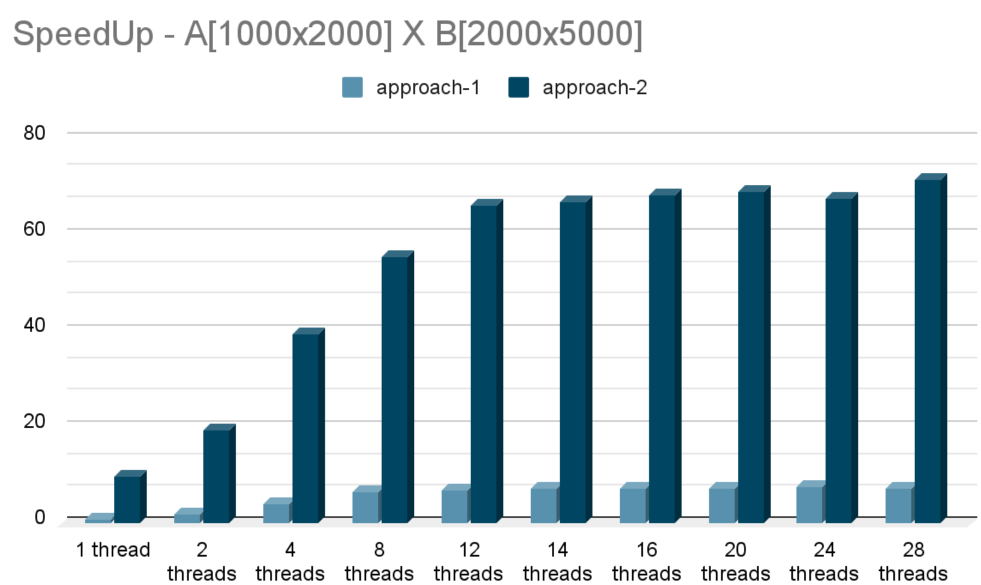


- Here we achieved a maximum speedup of ~20X when run using 24 threads with approach-2.

(ii) A[1000x2000] X B[2000x5000]

- Time taken with sequential is : 104.722312 sec

Num threads	1	2	4	8	12	14	16	20	24	28
approach-1	105.841789	52.899422	26.473632	15.784515	14.814592	14.395988	14.288681	14.209248	13.941713	14.038875
approach-2	10.754005	5.376819	2.656544	1.893215	1.584790	1.566524	1.535516	1.514109	1.549423	1.465638

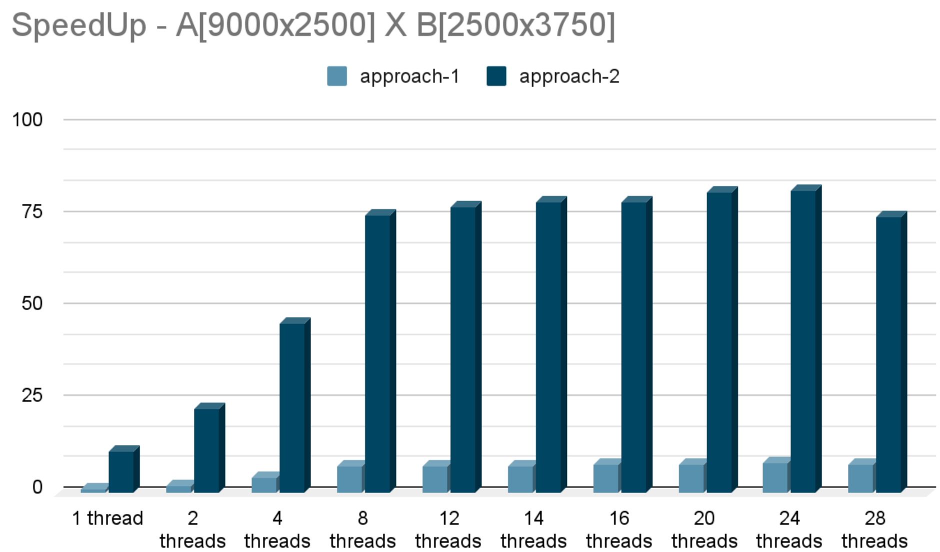


- Here we achieved a maximum speedup of ~71X when run using 28 threads with approach-2.

(iii) A[9000x2500] X B[2500x3750]

- Time taken with sequential is : 989.408794 sec

Num threads	1	2	4	8	12	14	16	20	24	28
approach-1	997.842457	495.180552	246.424479	139.243533	138.703192	134.188185	131.297627	131.250934	123.055023	130.332679
approach-2	91.302418	43.607768	21.835003	13.381103	13.002018	12.755662	12.770895	12.326260	12.303143	13.450321



- Here we achieved a maximum speedup of ~82X when run using 24 threads with approach-2.

III. Analysis of results

- With larger sizes of matrix A, matrix B we took more time to compute the resulting matrix as the number of computations increased. The serial algorithm iterates row x col x col2 times, with larger matrices we need to do more computations to get the resulting matrix. Even with the parallel approach, we are dividing the work equally among all the threads hence, each thread takes more time to complete their own task.
- With an increasing number of threads we cannot guarantee that our program will run faster because it all depends on how many cores does the machine have and those many threads can run in parallel to improve performance.
- If we keep increasing the number of threads our program runs on, we may end up slowing it down, as threads are not free i.e., OS takes up some memory and time for creating and maintaining threads. If the amount of work each thread does isn't significant, time taken to create these threads will be an overhead for the program execution time.
- Also, if we observe speedup results are consistent with approach-1 but not with approach-2.
  - With approach-1 we achieved a speedup of ~7X and couldn't get more by increasing number of threads, this is because the amount of work is divided equally among the threads and ran parallelly but we do not have as many cores as the number of threads thus, we cannot run all the threads in parallel. Interestingly, running a program on more threads may end up slowing it down as the time taken to create and maintain threads by OS becomes overhead.
  - But with approach-2 we observed maximum speedup of ~20X, ~71X, ~82X. We observed more speedup with larger matrices because we utilized cache locality. By utilizing cache locality we get more cache hits thus resulting in less memory access time. Utilizing cache locality with larger matrices, we have more elements to access thus resulting in saving more access time. Interestingly, for larger matrices we can observe that approach-2 with a single thread gave better results than approach-1 with multiple threads.