CS 301 : High Performance Computing

Lab3 - Matrix Multiplication Approaches

Purvil Mehta (201701073)
Bhargey Mehta (201701074)

Dhirubhai Ambani Institute of Information and Communication Technology
Gandhinagar

February 24, 2020

Contents

1 Hardware Details

- CPU model - Intel® Core™ i5-4590 CPU @ 3.30GHz
- Socket(s) - 1
- Core(s) per socket - 4
- Thread(s) per core - 1
- Cache sizes - L1: 32K, L2: 256K and L3: 6144K

2 Matrix Multiplication using Outermost Loop Parallelisation

2.1 Implementation Details

2.1.1 Brief and clear description about the Serial implementation

Since there is no data dependency on the inner statement, we can just initialise matrix C to 0 and then increment each of its elements by the proper term.

```
for i = 1:N
    for j = 1:N
        for  = 1:N
            c(i, j) += a(i, k) * b(k, j)
        end for
    end for
end for
```

2.1.2 Brief and clear description about the implementation of the approach (Parallelization Strategy, Mapping of computation to threads)

There is no dependency on the i loop. Since the result of $i = a_i$ is not dependent on some $i = b_i$, we can parallelise the outermost directly with a parallel for pragma.

2.2 Complexity and Analysis Related

2.2.1 Complexity of serial code

There are 3 loops each running over N. These are nested in each other so the total complexity is $O(N^3)$.

2.2.2 Complexity of parallel code (split as needed into work, step, etc.)

The work related to the outerloop is divided equally among all the threads which can run concurrently so the time taken would be $O(\frac{N}{p}) * O(N^2) = O(\frac{1}{p}N^3)$ since the inner loops take $O(N^2)$.

2.2.3 Cost of Parallel Algorithm

Since the entire code is parallelisable, the total parallel cost would be $p \times O(\frac{N^3}{p}) = O(N^3)$.

2.2.4 Theoretical Speedup (using asymptotic analysis, etc.)

Since the entire code is parallelisable, the theoretical speedup would be $\frac{O(N^3)}{O(\frac{N^3}{p})} = p$.

2.3 Curve Based Analysis

For smaller problem sizes the cost of parallelisation is quite significant as compared to the actual problem size. This cost which is constant takes up majority of the execution time. As we increase the problem size, this constant becomes smaller and smaller. For very large problem sizes, this becomes negligible and hence for very high problem sizes, we achieve a theoretical speedup of $p = 4$.

2.3.1 Time Curve related analysis (as problem size increases, also for serial)

2.3.2 Speedup Curve related analysis (as problem size and no. of processors increase)

2.3.3 Efficiency Curve related analysis

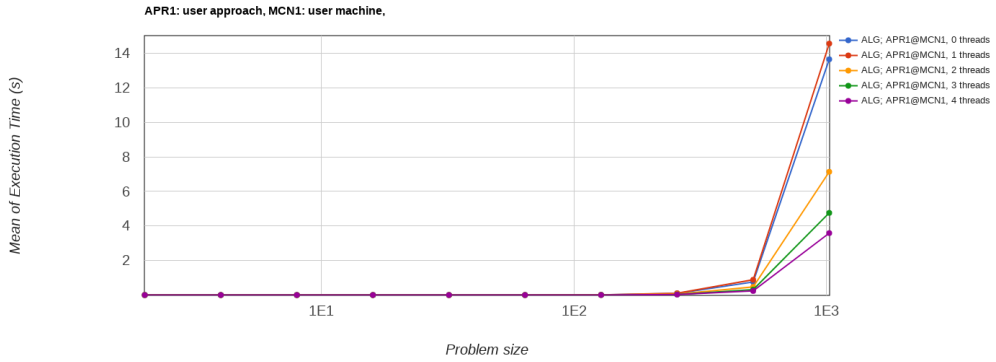

Figure 1: Mean Execution Time

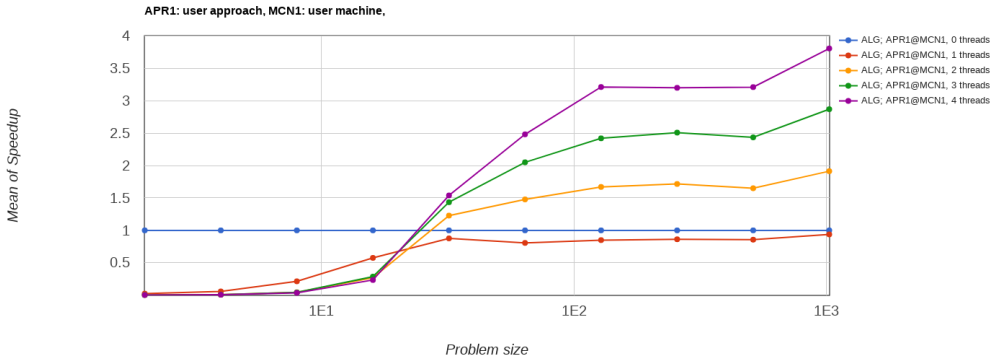

Figure 2: Mean Speedup

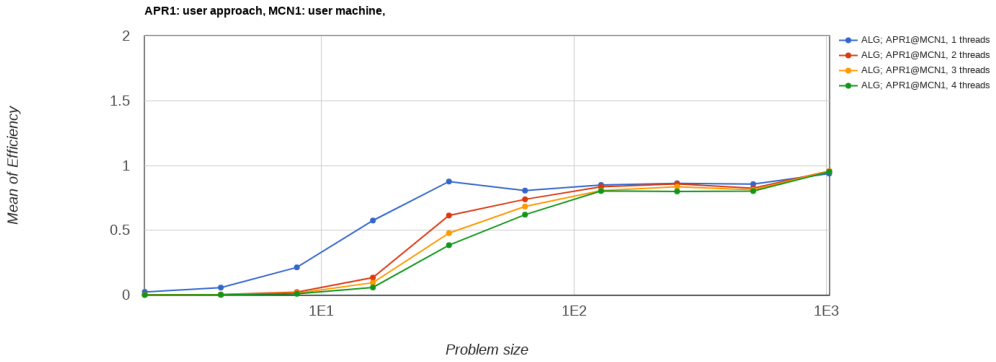

Figure 3: Mean Efficiency

3 Matrix Multiplication using Middle Loop Parallelisation

3.1 Implementation Details

3.1.1 Brief and clear description about the Serial implementation

Since there is no data dependency on the inner statement, we can just initialise matrix C to 0 and then increment each of its elements by the proper term.

```
for i = 1:N
    for j = 1:N
        for k = 1:N
            c(i, j) += a(i, k) * b(k, j)
        end for
    end for
end for
```

3.1.2 Brief and clear description about the implementation of the approach (Parallelization Strategy, Mapping of computation to threads)

There is no dependency of the result matrix elements on the inner loop. That means each column of the matrix can be calculated by a thread independently. So we can directly parallelise the middle loop without any data race or loss of correctness.

3.2 Complexity and Analysis Related

3.2.1 Complexity of serial code

There are 3 loops each running over N. These are nested in each other so the total complexity is $O(N^3)$.

3.2.2 Complexity of parallel code (split as needed into work, step, etc.)

There are N runs of the outerloop. The middle loop runs in $O(N)$ time and since there are p threads, the total time is $\frac{N}{p} \times O(N)$. This combined with the fact that there are N such runs, the total complexity is $O(\frac{N^3}{p})$.

3.2.3 Cost of Parallel Algorithm

Since the entire code is parallelisable, the total parallel cost would be $p \times O(\frac{N^3}{p}) = O(N^3)$.

3.2.4 Theoretical Speedup (using asymptotic analysis, etc.)

Since the entire code is parallelisable, the theoretical speedup would be $\frac{O(N^3)}{O(\frac{N^3}{p})} = p$.

3.3 Curve Based Analysis and Comparison to Outerloop parallelisation

For smaller problem sizes the cost of parallelisation is quite significant as compared to the actual problem size. This cost which is constant takes up majority of the execution time. As we increase the problem size, this constant becomes smaller and smaller. For very large problem sizes, this becomes negligible and hence for very high problem sizes, we achieve a theoretical speedup of $p = 4$.

Since the granularity is finer in the case of middle loop parallelisation, more threads can work concurrently. But in this case we are limited to 4 threads only. Thus the full benefit of parallelism is not exploited and also due to finer granularity we have a parallel overhead which greatly reduces the performance of the code.

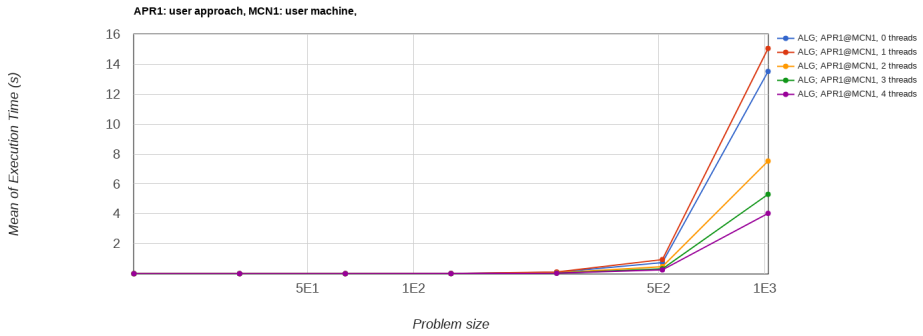

Figure 4: Mean Execution Time

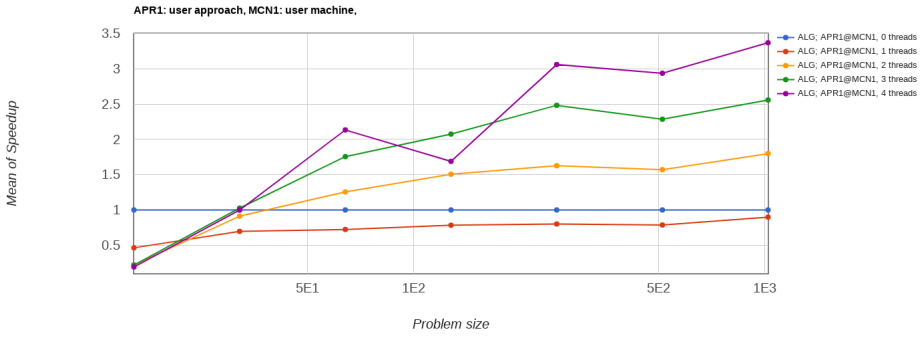

Figure 5: Mean Speedup

3.3.1 Time Curve related analysis (as problem size increases, also for serial)

3.3.2 Speedup Curve related analysis (as problem size and no. of processors increase)

3.3.3 Efficiency Curve related analysis

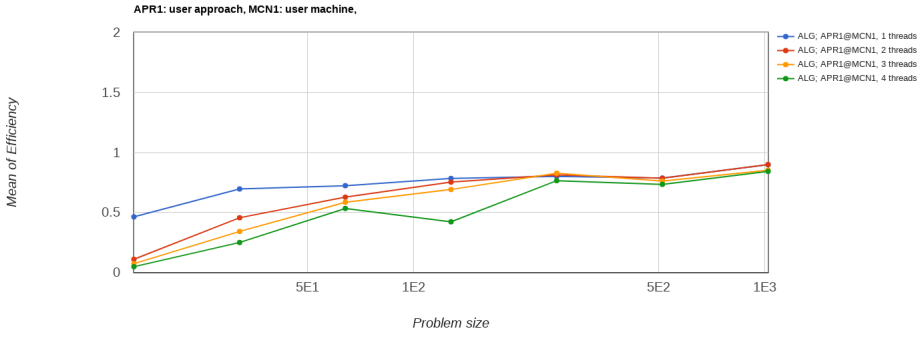

Figure 6: Mean Efficiency

4 Matrix Multiplication using Transposition

4.1 Implementation Details

4.1.1 Brief and clear description about the Serial implementation

Since there is no data dependency on the inner statement, we can just initialise matrix C to 0 and then increment each of its elements by the proper term. In this case we will be transposing one of the multiplicands to try to achieve a better cache hit-miss ratio.

```
for i = 1:N
    for j = 1:N
        for  = 1:N
            c(i, j) += a(i, k) * b(j, k)
        end for
    end for
end for
```

4.1.2 Brief and clear description about the implementation of the approach (Parallelization Strategy, Mapping of computation to threads)

There is no dependency on the i loop. Since the result of $i = a_i$ is not dependent on some $i = b_i$, we can parallelise the outermost directly with a parallel for pragma.

4.2 Complexity and Analysis Related

4.2.1 Complexity of serial code

There are 3 loops each running over N. These are nested in each other so the total complexity is $O(N^3)$.

4.2.2 Complexity of parallel code (split as needed into work, step, etc.)

The work related to the outerloop is divided equally among all the threads which can run concurrently so the time taken would be $O(\frac{N}{p}) * O(N^2) = O(\frac{1}{p}N^3)$ since the inner loops take $O(N^2)$.

4.2.3 Cost of Parallel Algorithm

Since the entire code is parallelisable, the total parallel cost would be $p \times O(\frac{N^3}{p}) = O(N^3)$.

4.2.4 Theoretical Speedup (using asymptotic analysis, etc.)

Since the entire code is parallelisable, the theoretical speedup would be $\frac{O(N^3)}{O(\frac{N^3}{p})} = p$.

4.3 Curve Based Analysis and Comparison with Parallel non-Transposed Approach

Comparing this approach with first one where we parallelised the outermost loop, we see that the overall execution time decreases significantly. This is because earlier, in one of the matrix, only columns were being accessed so we used to have N^2 misses but in this case, there is only 1 miss every C elements now that we have the matrix transposed. The cache miss thus decrease to $\frac{N^2}{C}$. This results in a better performance as compared to parallel non-transposed approach.

4.3.1 Time Curve related analysis (as problem size increases, also for serial)

4.3.2 Speedup Curve related analysis (as problem size and no. of processors increase)

4.3.3 Efficiency Curve related analysis

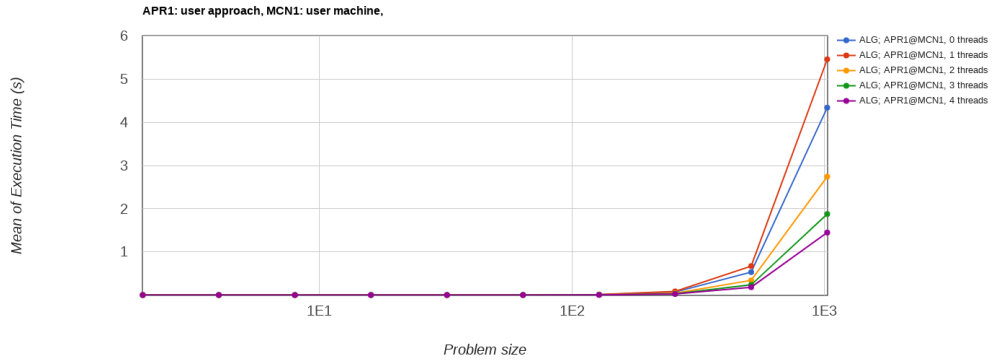

Figure 7: Mean Execution Time

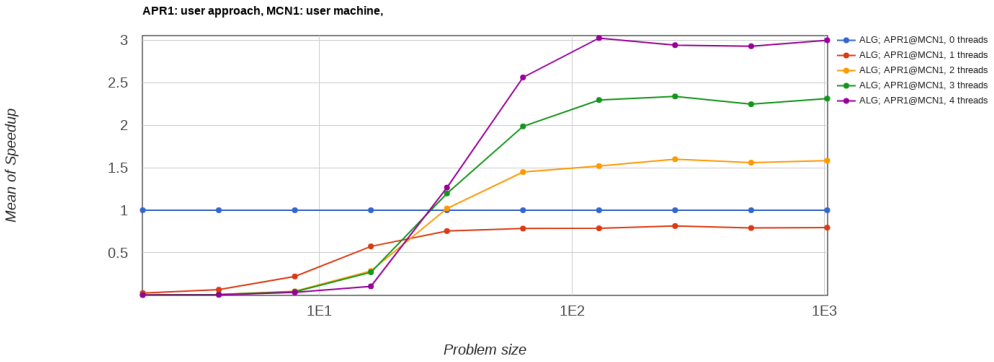

Figure 8: Mean Speedup

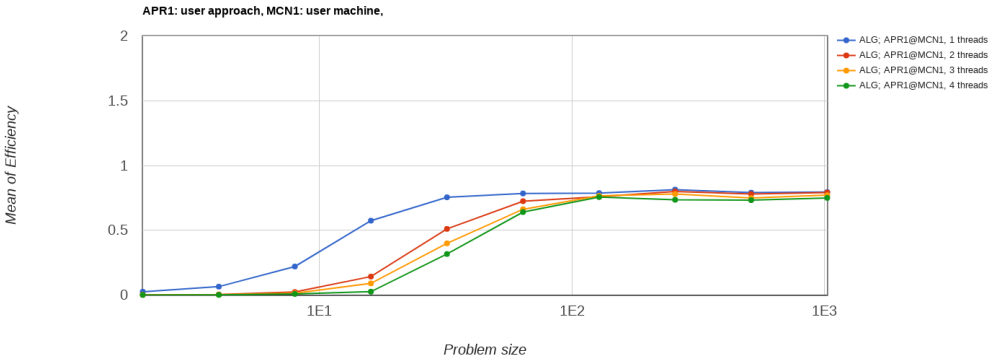

Figure 9: Mean Efficiency

5 Matrix Multiplication using Block Decomposition

5.1 Implementation Details

5.1.1 Brief and clear description about the Serial implementation

In this approach we fill the resultant matrix using blocks of size say B. Each block is calculated and then put into it's correct location in the resultant matrix. This results in a better cache utilisation since we can select the block size such that it can fit in the cache.

5.1.2 Brief and clear description about the implementation of the approach (Parallelization Strategy, Mapping of computation to threads)

Each block is calculated independent of each other, so can decide to parallelise at block level. Each block is calculated independently by some thread. In this particular case, we decide to parallelise the block rows instead of the blocks themselves since we're limited by 4 processors.

5.2 Complexity and Analysis Related

5.2.1 Complexity of serial code

There are 6 nested loops. The first 3 take care which block is to be calculated and hence each runs $\frac{N}{B}$ times whereas the inner 3 loops each run for B times. So in total the complexity is $O((\frac{N}{B})^3) \times O(B^3) = O(N^3)$

5.2.2 Complexity of parallel code (split as needed into work, step, etc.)

The work is divided equally among all the threads each of which runs for $O(B^2)$ time. For each combination of outer 3 loops, there will be $\frac{B}{p}$ such runs. So in total, we have $O((\frac{N}{B})^3) \times \frac{B}{p} \times B^2) = O(\frac{N^3}{p})$

5.2.3 Theoretical Speedup (using asymptotic analysis, etc.)

Since the entire code is parallelisable, the theoretical speedup would be $\frac{O(N^3)}{O(\frac{N^3}{p})} = p$.

5.3 Curve Based Analysis

We can see that there is an optimal value of the block size for which the performance is best. According to the hardware specification that we have it is $n = 256$. Upto this point, the peformance keeps on increasing as we increase the block size since the cache was being underutilised but as soon as we cross this point, the performace drops since the block can no longer fit into the cache and results in lower cache hit-miss ratio.

5.3.1 Time Curve related analysis (as problem size increases, also for serial)

5.3.2 Speedup Curve related analysis (as problem size and no. of processors increase)

5.3.3 Efficiency Curve related analysis

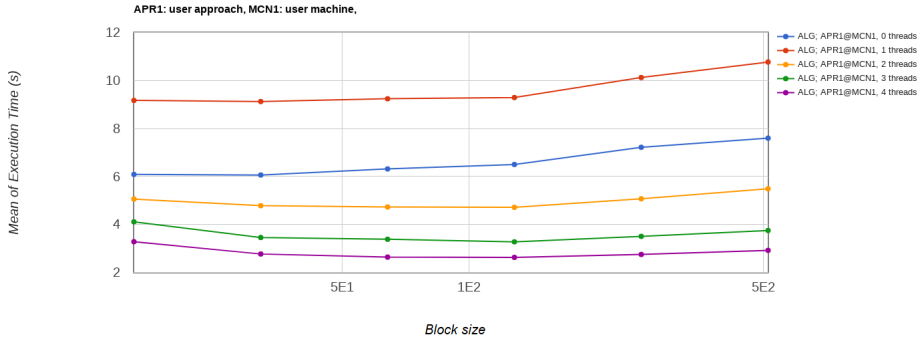

Figure 10: Mean Execution Time wrt. Block Size B

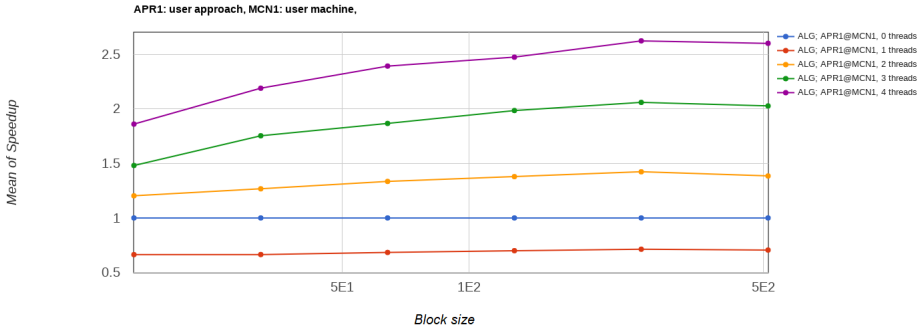

Figure 11: Mean Speedup wrt. Block Size B

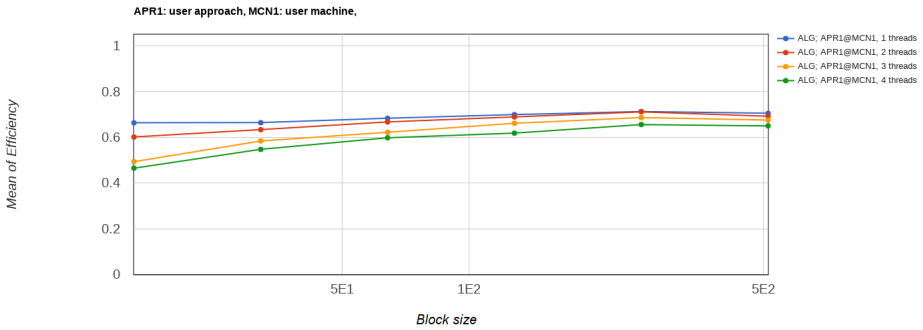

Figure 12: Mean Efficiency wrt. Block Size B