CSCI 503 Homework 2

-Prasanth S Iyer
(piyer@usc.edu)
1958-5902-92

I performed the following experiments using the parallel/optimized code that I wrote and the results are as follows:-

1. Parallel Matrix Multiplication using pThread and OpenMP

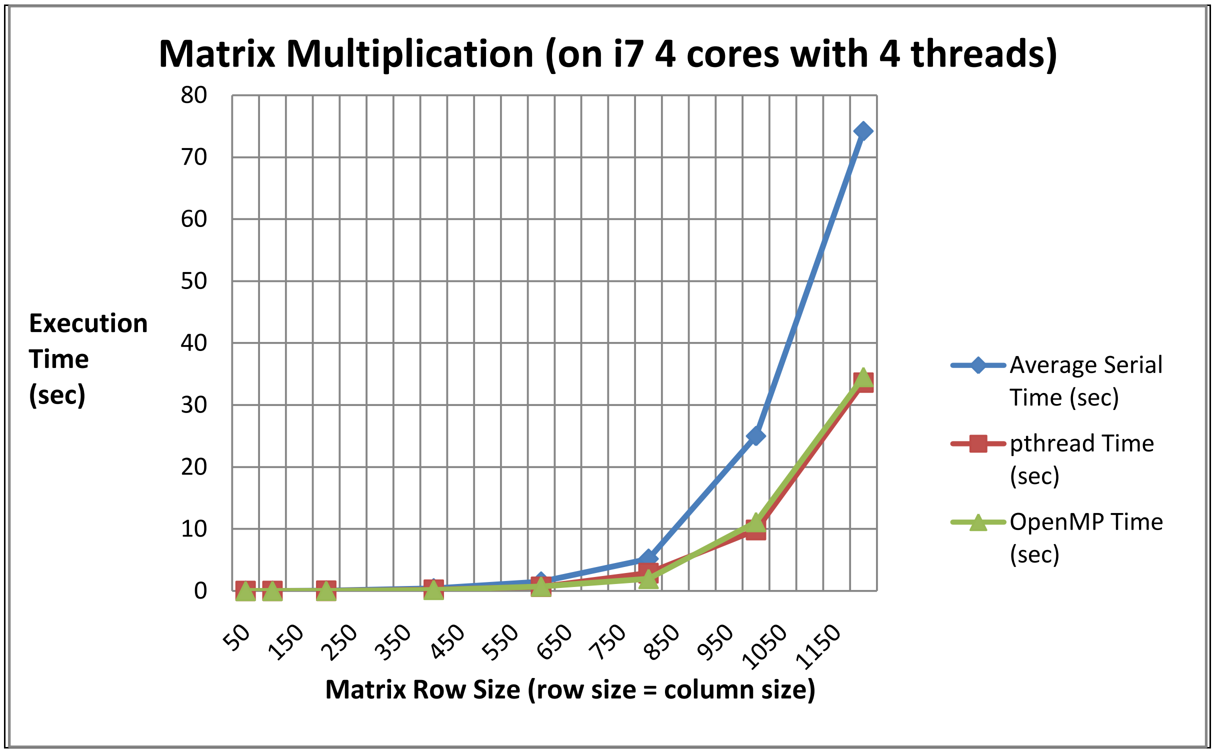
I measured the performance gains of both these algorithms with respect to the matrix size as well as the thread count on two machines - my own i7 machine with 4 cores as well as nunki.usc.edu with 128 virtual CPUs. The configurations of these two machines are as follows:-

My personal machine system configuration	
System Parameter	**Value**
Number of cores	4
Processor type	Intel i7-4700MQ
Clock Speed	2.4 GHz
Memory	8 GB
L1 Data Cache Size	4 x 32 KB
L1 Instruction Cache Size	4 x 32 KB
L2 Cache Size	4 x 256 KB
L3 Cache Size	6 MB
Operating System	Ubuntu 12.04 on VirtualBox 4.3 on a host OS of Windows 8.1
Other features	Supports Intel Hyper threading technology for upto 8 parallel threads

nunki.usc.edu system configuration	
System Parameter	**Value**
Number of cores	16 physical cores supporting upto 128 virtual CPUs
Processor type	UltraSPARC T2 Plus
Clock Speed	2.4 GHz
L1 Cache	24KB per core (16KB instruction cache, 8KB data cache)
L2 Cache	4MB per CPU chip
Memory	64 GB
Operating System	SunOS
Other features	Time-shared over multiple users

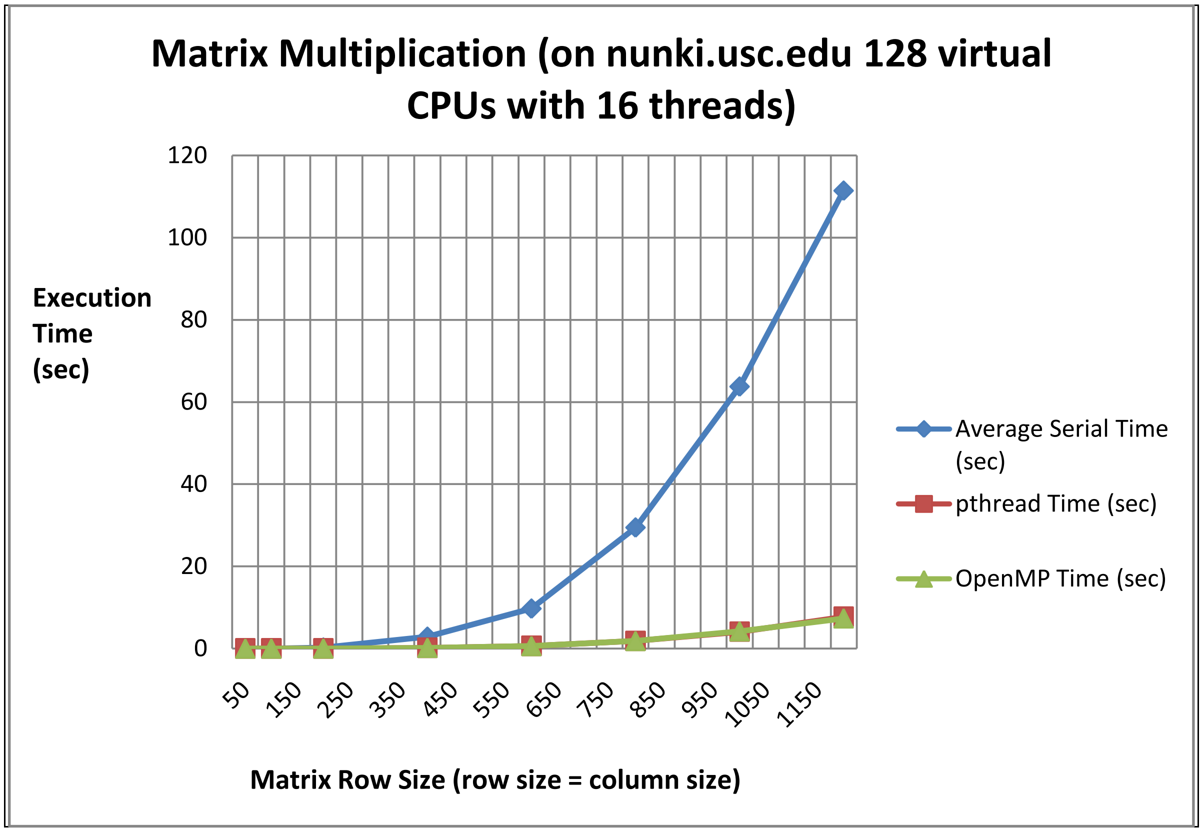
a) ***Plot 1: Matrix Multiplication - Execution Time v/s Matrix size***

On my personal machine with 4 threads.



Size	Average Serial Time (sec)	pthread Time (sec)	OpenMP Time (sec)
50x50	0.000592	0.001741	0.000365
100x100	0.0044695	0.003335	0.003749
200x200	0.0471805	0.021628	0.026044
400x400	0.4019765	0.198968	0.178403
600x600	1.498958	0.677983	0.706936
800x800	5.181228	2.867195	1.947048
1000x1000	24.9774415	9.824392	11.119211
1200x1200	74.1793415	33.60478	34.529848

On nunki.usc.edu with 16 threads.



Size	Average Serial Time (sec)	pthread Time (sec)	OpenMP Time (sec)
50x50	0.0034795	0.001255	0.00148
100x100	0.0276305	0.004226	0.004074
200x200	0.2194135	0.022777	0.023284
400x400	2.8620815	0.188094	0.188641
600x600	9.7085825	0.625906	0.618861
800x800	29.443357	1.845129	1.845639
1000x1000	63.7539705	4.084501	4.184501
1200x1200	111.4285475	7.721517	7.331676

Analysis:

- Based on the timings and the graph, I found consistent performance gain for both my parallel algorithms with increased matrix size from an initial speedup of around 1.2 with a matrix size of 50x50 to a speedup of nearly 2.3 with a matrix size of 400x400 on my personal machine. After that, the gain remained almost constant with increase in matrix size.

- The performance gain on nunki.usc.edu was even more spectacular because of the use of 16 threads with an initial speedup of around 2.3 with a matrix size of 50x50 increasing to a speedup of nearly 15 at a matrix size of 400x400. After that, the gain remained almost constant with increase in matrix size.

- The comparative performance of both the pthread and the openMP algorithms is similar on both machines with almost nothing to choose in terms of the time taken. This was definitely expected since both techniques simply provide different levels of abstraction to thread creation.

- One important observation was that the performance gain on both machines due to increased matrix size tended to max out at a certain value: 2.3 (for a 400x400 matrix) on my personal machine and 15 (for a 400x400 matrix) in case of nunki.usc.edu. Further increases in matrix size showed little to no performance gain.

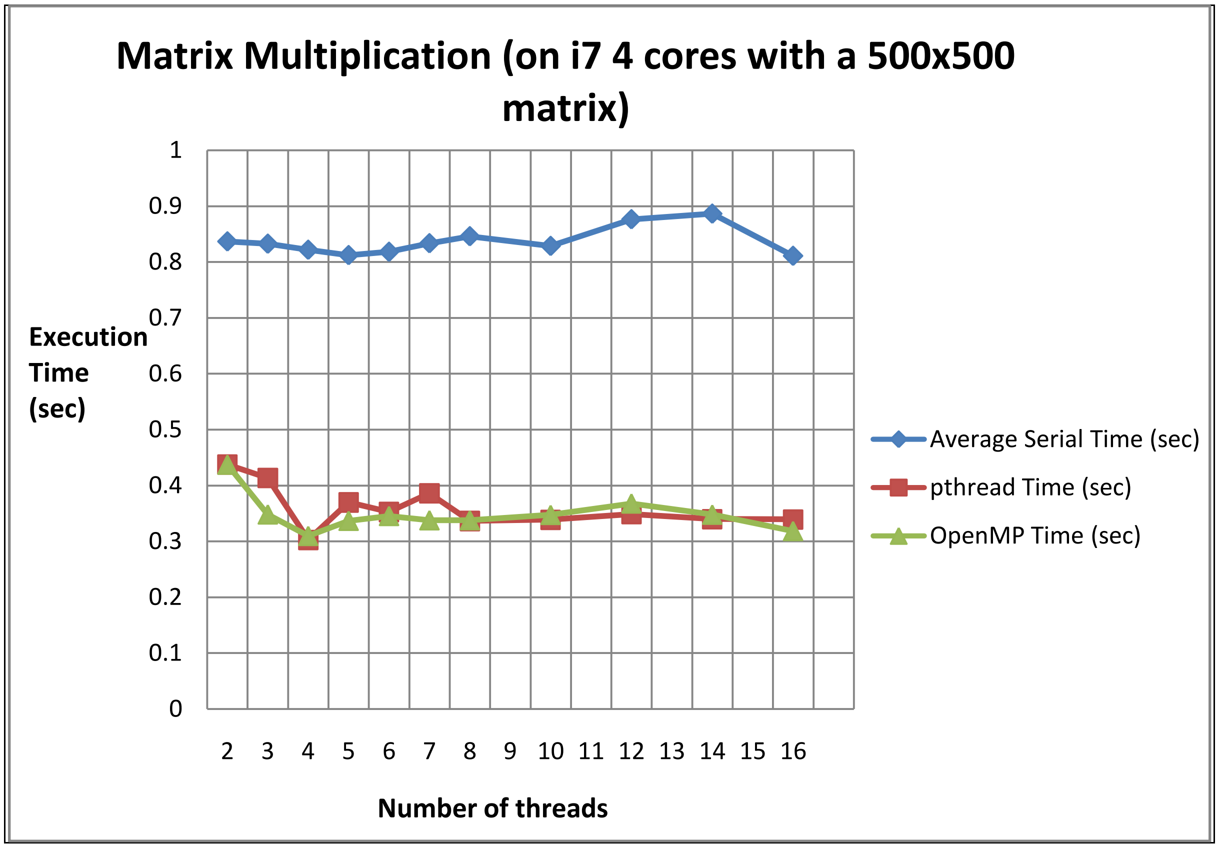
  Now, the potential maximum speedup possible can be 4x with 4 threads on my personal machine and 16x with 16 threads on nunki,usc,edu. However, there is some part of the algorithm like the thread creation and the initial setup which has to be done in serial for all threads. It is because of this serial overhead that my algorithms could not reach the potential maximum of 4x and 16x speedups that one would have expected irrespective of how much I increased the matrix size. This behavior conforms to Amdahl's law.

- One intriguing observation was that for lower matrix sizes (lower than 50x50) the pthread and openMP algorithms actually performed worse than the serial algorithm. The reason for this could be the overhead of creating threads and setting them up. In case of higher matrix sizes, this overhead is extremely small compared to the time required to compute the multiplication. However, for lower matrix sizes, this overhead is comparable to the compute time and consequently slows down the computation.

In conclusion, I did observe that the performance increased proportionally with increased matrix size on both machines through both algorithms upto a size of 400x400. After that, performance gain remained more or less constant. Further, both algorithms performed almost equally well over the set of matrix sizes.

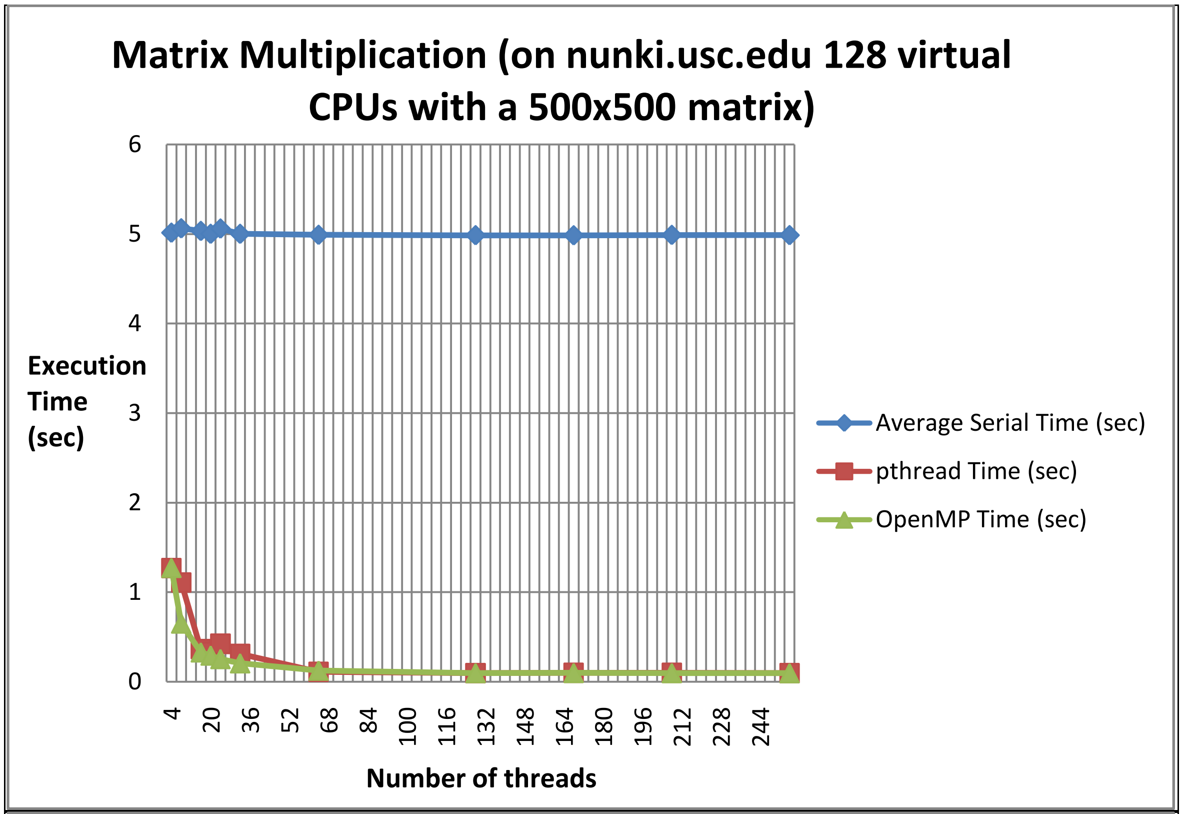
b) Plot 2: Matrix Multiplication - Time v/s Number of threads

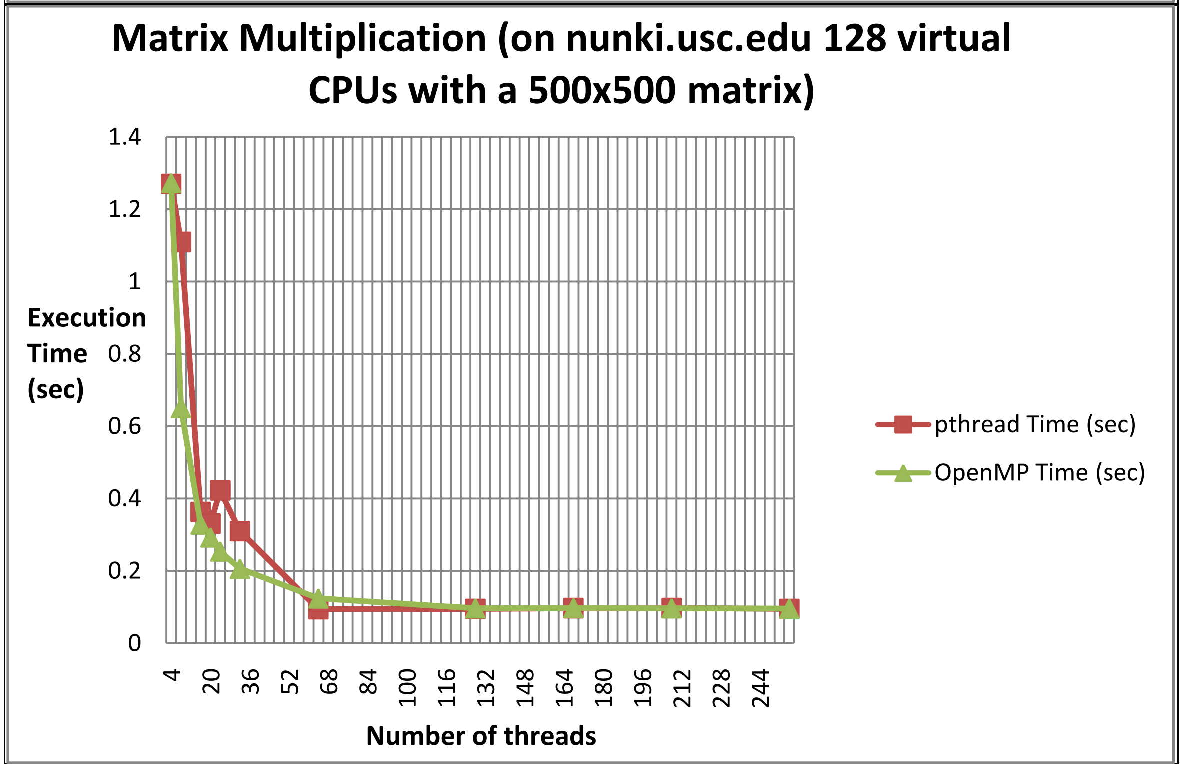
On my personal machine with a 500x500 matrix.



Thread Count	Average Serial Time (sec)	pthread Time (sec)	OpenMP Time (sec)
2	0.836775	0.437413	0.437479
3	0.832664	0.413481	0.3483
4	0.821794	0.302639	0.309796
5	0.81217	0.369715	0.336864
6	0.818122	0.352639	0.345508
7	0.833514	0.386063	0.337856
8	0.846037	0.336594	0.337877
10	0.828623	0.338665	0.34745
12	0.87653	0.34876	0.367895
14	0.886576	0.33978	0.347895
16	0.810879	0.339197	0.318564

On nunki.usc.edu with a 500x500 matrix.





Thread Count	Average Serial Time (sec)	pthread Time (sec)	OpenMP Time (sec)
4	5.014476	1.270502	1.2727
8	5.063296	1.109902	0.650639
16	5.033991	0.363232	0.328388
20	5.001775	0.331102	0.292646
24	5.060999	0.422883	0.253721
32	5.001453	0.310514	0.206138
64	4.990306	0.10453	0.123886
128	4.986185	0.095323	0.096775
168	4.984816	0.097364	0.097878
208	4.989322	0.097822	0.096699
256	4.986706	0.0955	0.095892

Analysis:

- Based on the timings and the graph, I found consistent performance gain for both my parallel algorithms with increased number of threads from an initial speedup of around 1.8 with 2 threads to a speedup of nearly 2.5 with 4 threads on my personal machine. After that, the gain remained almost constant.

- The performance gain on nunki.usc.edu was more easily evident because of the 128 virtual CPUs that the server supports. The gain increased from 1.9 with 2 threads to nearly 51 with 64 threads. After that, the gain remained almost constant.

- The comparative performance of both the pthread and the openMP algorithms is similar on both machines with almost nothing to choose in terms of the time taken. This was definitely expected since both techniques simply provide different levels of abstraction to thread creation.

- However, with certain number of threads like 3 and 7 on my personal machine and 24 on nunki.usc.edu, the openMP algorithm performed marginally better than the pthread algorithm. I would attribute this to the fact that knowing my pthread algorithm, it has some extra logic for work distribution in the cases where the matrix size was not a multiple of the number of threads. I believe this extra work was better handled by the openMP implementation which caused the difference in performance.

- Another observation was that the performance gain on both machines due to increased thread count also tended to max out at a certain value: 2.5 (with 4 threads) on my personal machine and 51 (with 64 threads) in case of nunki.usc.edu. Further increases in thread count showed little to no performance gain.

  Now, one would expect the speedup to increase with the number of threads given that it is possible to execute all threads in parallel i.e. the speedup should increase as long as the number of threads equals the number of cores. However, while my personal machine conforms to this logic, the performance gain should have increased with increase in the number of threads right upto 128 threads in case of nunki.usc.edu. *Here I assume that when nunki.usc.edu supports 128 parallel threads, all 128 threads run in parallel and are not multiplexed on a single core.*

  But, this was not the case in my program because a certain part of the program (thread creation and initial setup) had to be executed serially. So, the maximum speedup will be limited by the time taken to execute this serial portion. This is true for all thread counts from 64 to 128 and this behavior conforms to Amdahl's law.

After that, since nunki.usc.edu has only 128 virtual CPUs, increasing the thread count adds no further value as expected.

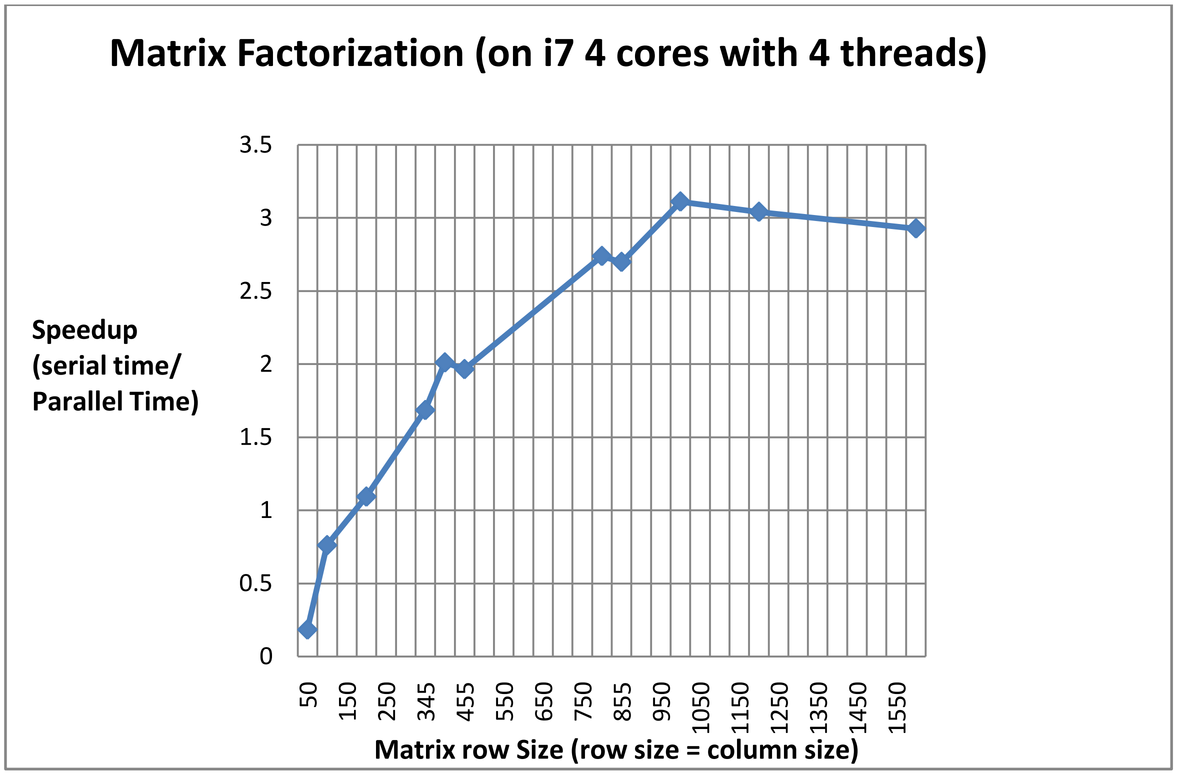
In conclusion, I did observe that the performance increased proportionally with increased number of threads on both machines through both algorithms upto a thread count of 4 on my personal machine and 64 on nunki.usc.edu. After that, performance gain remained more or less constant. Further, both algorithms performed almost equally well over the set of matrix sizes except for certain thread counts which were not multiples of the matrix size in which case the openMP algorithm performed marginally better.

Does scaling the problem size commensurate with performance gain as a result of increasing the number of threads?

Based on my analysis, I observed an average performance gain of almost 1.8 on doubling the matrix size for lower matrix sizes while I also observed an average performance gain of about 2 on doubling the number of threads for lower number of threads. For higher number of threads (>64 on nunki.usc.edu with a 500x500 matrix) and higher matrix sizes (>400x400 on nunki.usc.edu with 16 threads), the performance gain becomes almost constant with further increases. Thus, scaling the problem size does commensurate approximately with performance gain as a result of increasing the number of threads although increasing the number of threads shows marginally better performance gains.

c) ***Plot 3: Matrix Factorization - Speedup v/s Matrix Size***

On my personal machine with 4 threads.



Size	Serial Time (sec)	OpenMP Matrix Factorization Time (sec)	Speedup (serial time/parallel time)
50x50	0.000146	0.000791	0.184576485
100x100	0.001114	0.001463	0.761449077
200x200	0.0084	0.007679	1.093892434
345x345	0.047344	0.028088	1.68555967
400x400	0.064927	0.032277	2.011556217
455x455	0.091967	0.046786	1.965694866
800x800	0.508603	0.185648	2.739609368
855x855	0.631032	0.233792	2.699117164
1000x1000	0.791914	0.254453	3.112221117
1200x1200	1.839736	0.604804	3.041871416
1600x1600	3.978976	1.358848	2.928198003

Analysis:

- Based on the timings and the graph, I found consistent performance gain for parallel matrix factorization with increased matrix size from an initial speedup of around 0.2 with a matrix size of 50x50 to a speedup of nearly 3 on my personal machine with a matrix size of 1000x1000. After that, the gain remained almost constant with increase in matrix size.

- As before, the maximum potential speedup with 4 threads is 4. However just like the matrix multiplication algorithm, there is some part of the algorithm like the thread creation and the initial setup which has to be done in serial for all threads. It is because of this serial overhead that the performance gain smoothes out after increasing with increased matrix size upto a size of 1000x1000. Again, this behavior conforms to Amdahl's law.

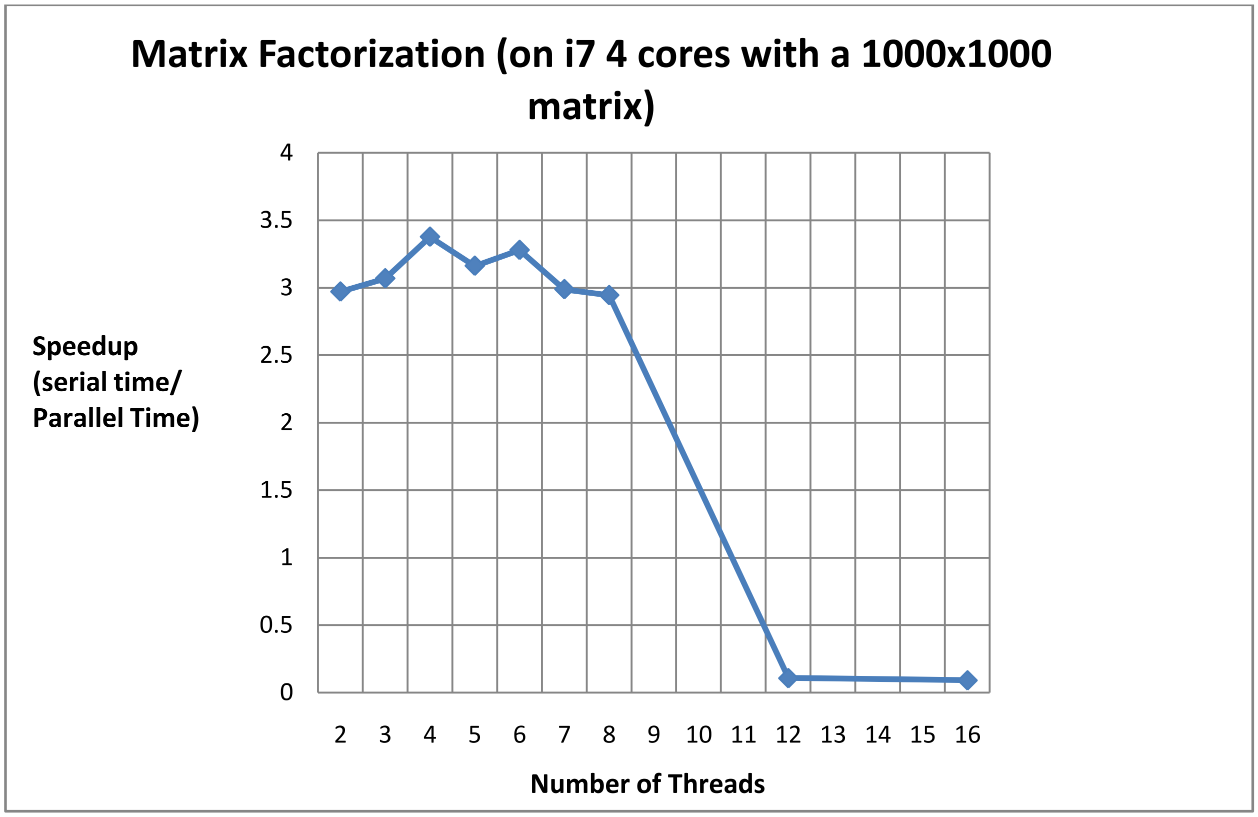
- Also, for lower matrix sizes (lower than 100x100) the algorithm actually performed worse than the serial algorithm. The reason for this would be the overhead of creating threads and setting them up. In case of higher matrix sizes, this overhead is extremely small compared to the time required to compute the factorization. However, for lower matrix sizes, this overhead is comparable to the compute time and consequently slows down the computation.

- Another intriguing observation is the marginal decrease in performance gain in cases where the matrix size is not a multiple of 4 (the number of threads used) - for instance, the matrix sizes of 345, 455 and 855 which I intentionally tested. I believe this decrease is due to the unbalanced load distribution among the 4 threads in these cases because of which some threads end up performing more work than others.

In conclusion, I did observe that the performance increased proportionally with increased matrix size upto a size of 1000x1000. After that, performance gain remained more or less constant.

d) Plot 4: Matrix Factorization - Speedup v/s Number of threads

On my personal machine with a 1000x1000 matrix.



Thread Count	Serial Time (sec)	OpenMP Matrix Factorization Time (sec)	Speedup (serial time/parallel time)
2	1.091943	0.367494	2.971322
3	1.156083	0.376618	3.069644
4	0.791914	0.234453	3.377709
5	0.855199	0.270408	3.162625
6	0.803193	0.244884	3.279892
7	0.871483	0.291549	2.989153
8	0.980012	0.332748	2.945208
12	1.038657	9.689811	0.107191
16	0.997917	10.94412	0.091183

Analysis:

- Based on the timings and the graph, I found consistent performance gain for parallel matrix factorization with increased number of threads from an initial speedup of around 2.9 with 2 threads to a speedup of nearly 3.3 with 4 threads. After that, the gain remained almost constant with increase in number of threads till 8 threads. However, on increasing the number of threads further, I observed a peculiar slowdown to an extent that it is eventually slower than the serial implementation itself. But, the fact that the speedup maxed out at 3.3 < 4 is consistent with Amdahl's law since the maximum speedup that could be achieved was constrained by the serial portion of the program

- The number of physical cores on my personal machine is 4. So, the maximum speedup that I would expect is with 4 threads which was the case here. On increasing the number of threads further, the speedup remains more or less constant till 8 threads since Intel's hyper-threading technology supports upto 8 threads on my machine..

- However, on increasing the number of threads further after 8, I see a huge decrease in performance which I had not expected. However, on analyzing further, I would attribute this decrease to the following:-
    - I used 2 OpenMP "parallel for" constructs for work sharing nested within the loop over the matrix. OpenMP implicitly adds a cache flush and barrier at the end of a parallel for work-sharing construct. Note that within a loop, there would be N such barriers and cache flushes for each thread where N is the matrix size.
    - Till 8 threads, due to Intel's hyper-threading technology, my program was run on 8 logical cores multiplexed on 4 physical cores. Each thread was run on a different logical core and so, all the wait time at the barrier for each thread was comparatively less. Consequently, it provided a consistent speedup of 3 to 3.3.
    - However, once the number of threads increased further, since Intel only provides hyper-threading support for at most 8 threads, at most 8 threads can be run in parallel. Consequently, the time spent by the threads at the barrier increased. The time spent at a barrier is proportional to the number of threads. Since the barrier is hit N times where N is the matrix size, this time significantly reduced the speedup.

In conclusion, I did observe that the performance increased proportionally with increased number of threads upto a thread count of 4. After that, performance gain remained more or less constant until a thread count of 8. Then, performance dropped considerably on further increase in thread counts.

Discussion

1. *Using Amdahl's Law, formulate or draw some conclusions with regards to your findings for the analysis section of the programming assignment above. For example, you might show numbers that either support or reject Amdahl's law -- since this is a well celebrated and accepted law, a statement of rejection should be accompanied by a well thought out discussion [5 points].*

   As I observed in all my experiments, the speedup increased with matrix size/number of threads upto a particular value which was less than the number of processors available to the program. The values for the experiments mentioned above are as follows:-

   1. **Plot 1: Matrix Multiplication - Execution Time v/s Matrix size**
      On i7 4 cores with 4 threads
      The speedup maxed out at 2.3 with a matrix size of 400x400.

      On nunki.usc.edu 128 cores with 16 threads
      The speedup maxed out at 15 with a matrix size of 400x400.

   2. **Plot 2: Matrix Multiplication - Execution Time v/s Number of threads**
      On i7 4 cores with a 500x500 matrix
      The speedup maxed out at 2.5 with 4 threads.

      On nunki.usc.edu 128 cores with a 500x500 matrix.
      The speedup maxed out at 51 with 64 threads.

   3. **Plot 3: Matrix Factorization - Execution Time v/s Matrix size**
      On i7 4 cores with 4 threads
      The speedup maxed out at 3 with a matrix size of 1000x1000.

   4. **Plot 4: Matrix Factorization - Execution Time v/s Number of threads**
      On i7 4 cores with a 1000x1000 matrix
      The speedup maxed out at 3.3 with 4 threads.

   Further increase in either parameter had no effect on the speedup. Now, Amdahl's law states that:-

   The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.

   In my case, the speedup in all my experiments was certainly limited to a certain value. I would attribute this limit to the fact that there was a part of my program that had to be executed serially. This part consists of the initial thread creation and setting up the parameters and variables for these threads. Since no matter what the parameters, I could never get a speedup equal to or more than the number of processors and attributing this to the serial fraction of my program, I would say that my program followed Amdahl's law.

2. *When shopping for a new server, IT procurement managers are generally concerned about the number of cores and the CPU's cache size, possibly because they feel that there is a direct correlation between performance and the number cores/cache size. Let's assume that you've been consulted on the next server purchase. Please discuss the trade-offs between number of cores and the cache size and how they related to your application. In other words, what aspects of code's data and algorithm must be considered when making your recommendation [2 points].*

- Increasing the number of processor cores and increasing the cache size both benefit different types of instructions:-

  1) Increasing the number of processor cores increases the number of compute-based instructions that can be performed in parallel.
  2) Increasing the cache size can decrease the time required for memory accesses.

- So, simply increasing the number of processor cores without increasing the cache size would not be useful if the application requires a large number of memory accesses (reads/writes) and very few computations since more memory accesses would lead to a large number of cache misses which in turn would lead to more memory references that would slow down the application.

- On the other hand, an application that performs a number of computing operations in parallel would not hugely benefit from simply increasing the cache size since the number of possible cache hits may already be maximized. In this case, I would advise to increase the number of cores.

- This can be easily seen using the simple model of machine balance described in class:-

  Machine balance: $B_M = M_M/F_M$, where M_M is the rate at which data can be fetched from memory (max words per cycle) and F_M is the rate at which floating point operations can be performed (max flops per second).

  We also defined the balance for a loop in the application as $B_L = M/F$, where M is the number of memory references and F is the number of flops.

- So, if the application contains code with more loops that have $B_L > B_M$, then the application needs faster access to the memory and in this case, I would advise increasing the cache size. On the other hand, if the application contains code with more loops that have $B_L < B_M$, then the processor cannot process the loop as fast as memory can provide the data and in this case, I would advise increasing the number of processors.

- Finally, if the application contains code with $B_L = B_M$, it means that it has many balanced loops and in this case, I would advise either increasing both the number of processors and the number of cores such that B_M remains the same or not increasing either.

- Here, the question also does not specify whether the cache whose size we plan to increase is an on-core cache, an on-chip cache or a shared cache since increasing the size of each would lead to a different performance characteristic.
    - Increasing the size of the on-core caches for all the processors would increase the rate of memory access for each processor by a large extent. However, it would also increase the amount of data synchronization that needs to be maintained among these caches. In other words, if a value has changed in the cache of one processor, more number of cache locations need to be checked for that value in the other processors.
    - Increasing the shared caches would not increase the rate of memory access for all processors by as much as increasing the on-core cache size would. However, it would not necessitate the amount of data synchronization needed if the size of an on-core cache is increased.

3. *Please discuss the advantages the OpenMP programming model has over the pThread programming model; and disadvantages (if any) [2 points].*

   Advantages of the OpenMP programming model over the pThread programming model:

   1. One major advantage of the openMP model is the ease of programming a multi-threaded application using it. The user does not need to worry about details like how the work is shared among the threads, how private and shared variables are handled and how access to shared variables is synchronized since it provides a very high-level abstraction to multi-threaded programming.

   2. Another advantage is that because of its #pragma lines, it is very easy to port a serial application to an OpenMP program and vice versa.

   Disadvantages of the OpenMP programming model over the pThread programming model:

   1. Since OpenMP provides a much higher abstraction for multi-threaded programming, it lacks some of the low-level thread attributes and synchronization options that pthread supports.

   2. OpenMP provides much less control over multi-threaded code generation and optimization than pThreads since most of the code is generated by the library during compilation. For instance, openMP adds implicit barriers and cache flushes at the end of a worksharing construct. In some cases, this may be unnecessary and may result in performance degradation. In such cases, pThreads provides much more control over the parallelism.

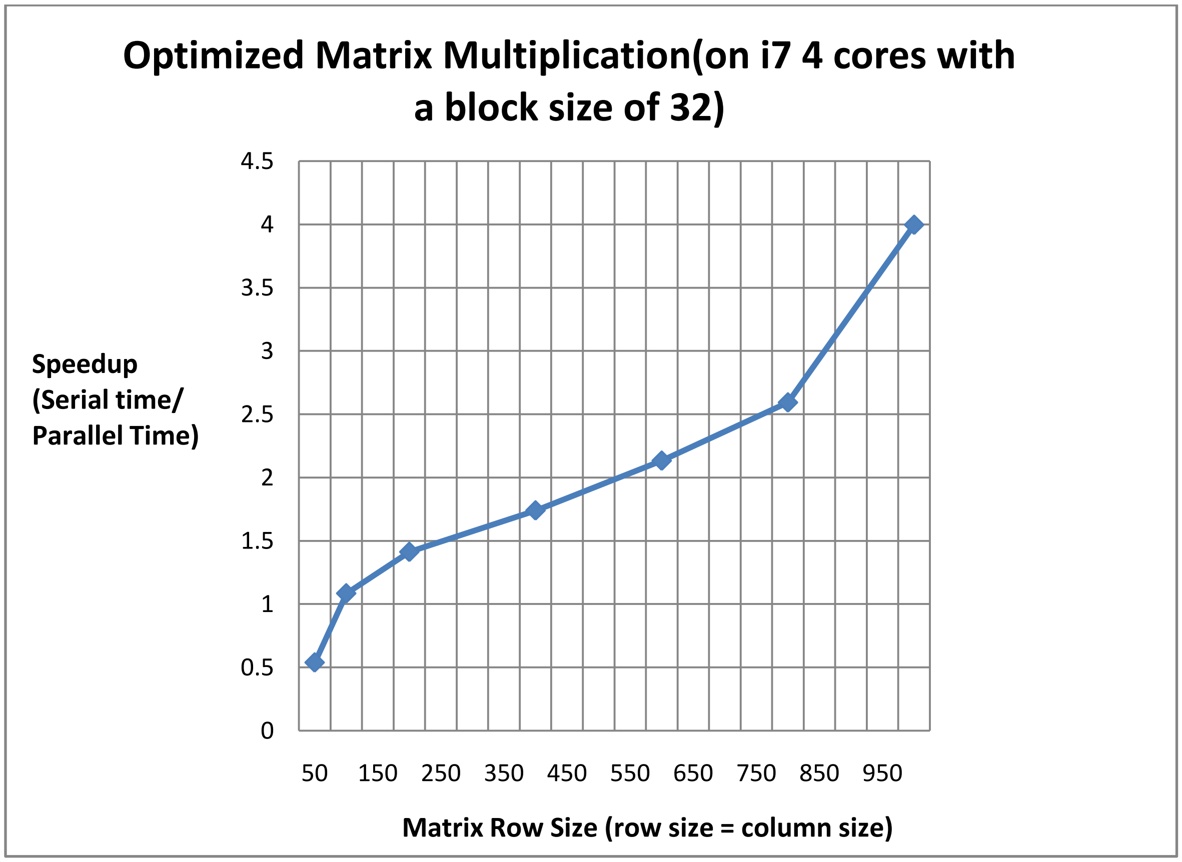
   In conclusion, for fine-grained code-intensive control over parallelism and its implementation details, pthreads are a better option whereas for a high-level, developer-friendly implementation of parallelism, openMP proves to be a better option.

Plots for my optimized implementation of Matrix Multiplication:

I optimized the matrix multiplication algorithm using a combination of block tiling and loop unrolling on all 3 loops used in matrix multiplication. I experimented with various block sizes while using block tiling. I also unrolled each of the 3 inner loops by 2. A combination of the 2 approaches gave me significant increase in performance as shown:-

Plot 5: Optimized Matrix Multiplication - Time v/s Matrix Size

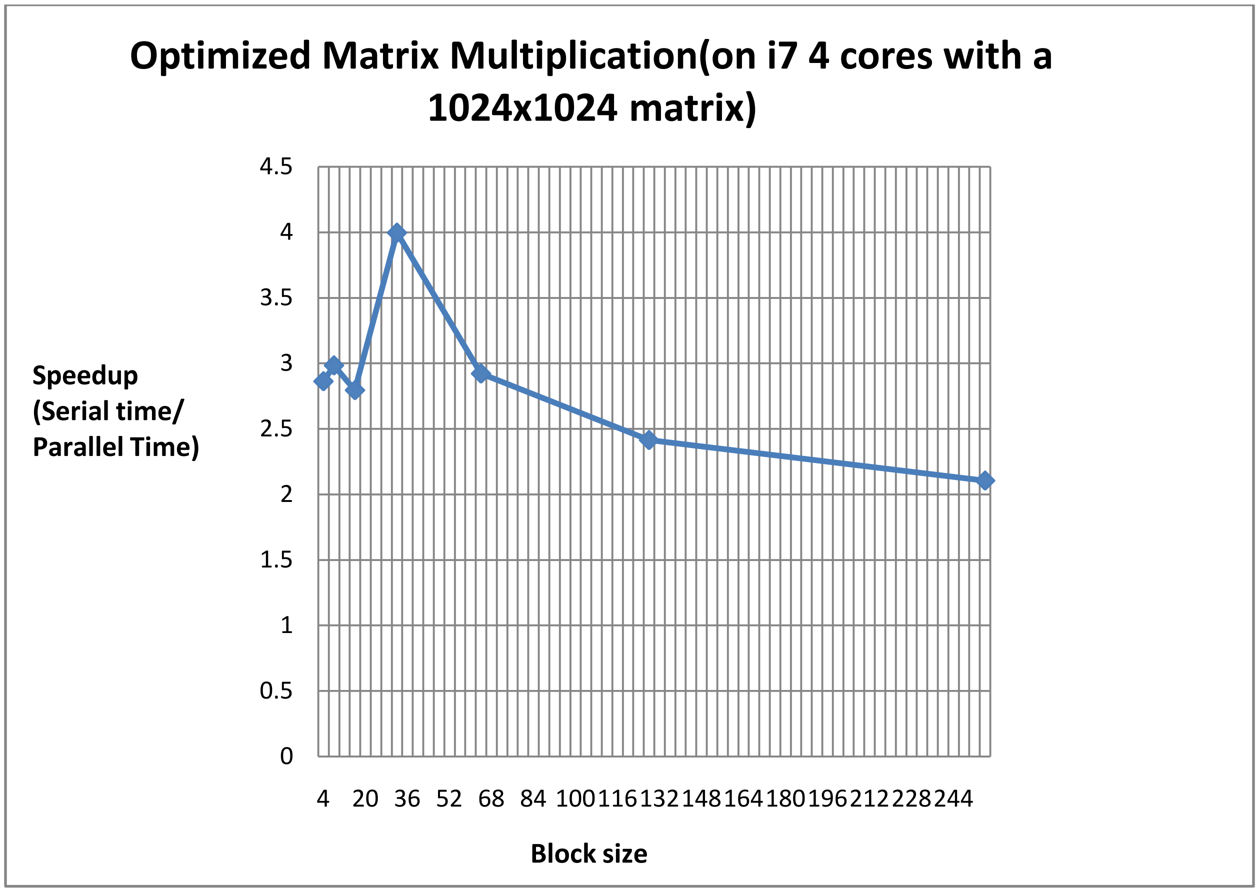
On my personal machine with a loop tile size of 32.



Size	Serial Time (sec)	Optimized Matrix Multiplication Time (sec)	Speedup (Serial time/Parallel time)
50	0.000452	0.000837	0.540024
100	0.006936	0.006387	1.085956
200	0.04897	0.034646	1.413439
400	0.344982	0.198146	1.74105
600	1.345425	0.63048	2.133969
800	3.908752	1.506871	2.593953
1024	10.375708	2.595953	3.996878

Plot 6: Optimized Matrix Multiplication - Time v/s Loop Tile Size

On my personal machine with a matrix size of 1024x1024.



Block Size	Serial Time (sec)	Optimized Matrix Multiplication Time (sec)	Speedup (Serial time/Parallel time)
4	10.65754	3.723631	2.862137
8	10.45989	3.506201	2.983254
16	10.69968	3.829542	2.793983
32	10.37571	2.595953	3.996878
64	10.37966	3.553561	2.920919
128	10.09513	4.183728	2.412951
256	11.89671	5.655365	2.103615

Analysis:

- Based on the timings and the graph, I found consistent linear performance gain for my optimized algorithm for all matrix sizes from 50x50 to 1024x1024. This was expected since the algorithm was not a parallel algorithm constrained by Amdahl's law and so the performance gain was indeed expected to increase as long as the matrix size was increased. The performance with the matrix size of 1024 was especially spectacular and I would attribute this to the fact that in this case, the size of the matrix was a multiple of my cache size.

- On varying the block sizes used while block tiling, I observed that the algorithm performed the best with a block size of 32 on a matrix of 1024x1024. I would attribute this to the fact that a block size of 32 divided the 1024x1024 matrix evenly with 32x32 iterations each working on block sizes of 32x32. Increasing the block size further would have the same effect as using a lower block size since it would have fewer iterations. For instance, a block size of 1024x1024 is exactly similar to having no block tiling since we have exactly 1 iteration working on a single 1024x1024 block. So, it was no surprise to find the optimal block size for this 1024x1024 matrix to be 32.

In conclusion, I did observe that the performance increased proportionally with increased matrix size and did not cap out at a particular value as was the case in the parallel algorithms. Besides this, I also observed that the block size that gave the best performance with a 1024x1024 matrix was a size of 32x32.