# Efficient Large Matrix Multiplication in OpenMP

Maureen Maguire

19213997

CA670 OpenMP Assignment 2

**Disclaimer**

An essay and code submitted to Dublin City University, School of Computing for module CA670 Concurrent Programming, 2019/2020. I understand that the University regards breaches of academic integrity and plagiarism as grave and serious. I have read and understood the DCU Academic Integrity and Plagiarism Policy. I accept the penalties that may be imposed should I engage in practice or practices that breach this policy. I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations, paraphrasing, discussion of ideas from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged, and the sources cited are identified in the assignment references. I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work. By signing this form or by submitting this material online I confirm that this assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. By signing this form or by submitting material for assessment online I confirm that I have read and understood DCU Academic Integrity and Plagiarism Policy (available at: http://www.dcu.ie/registry/examinations/index.shtml)

Name: Maureen Maguire

Date: 19/04/2020

## 

## 

## 

## *Program Design:*

In order to test the most efficient approach for large matrix multiplication several methods were implemented. OpenMP was used in all methods. I have used the C programming language. This study was carried out on an Intel(R) Pentium(R) CPU 4415U @ 2.30GHz 2.30 GHz with 8.00 GB RAM. The CPU has 2 cores with L1 cache of 128 KB. The matrices used in this study are square shaped and are 100x100 in size. Both matrices, ‘m1’ and ‘m2’, have the same number of columns and rows, therefore there is no extra formatting or reshaping required in order to allow the multiplication to take place. The matrices are generated using a for loop and a random number generator (Jayawardana, 2018). The elements in the array are stored in contiguous memory locations of row-major order. In this study, the most efficient method found to multiply large matrices was blocked matrix multiplication which can be found in the file ‘blockedParallel.c’. This involved splitting the calculations of the matrices into separate fixed sized blocks which were carried out by individual threads. The block size was the int variable ‘block’ and was set to 10, so the blocks were 10x10. The block size must be a multiple of the matrix size, which in this case it is. A third matrix, ‘results’, was created to store the results of the multiplication. This solution required 5 nested for loops: ‘i’, ‘j’, ‘x’, ‘y’ and ‘k’. The two outer loops, ‘i’ and ‘j’, were used to iterate through the rows and columns of the matrices respectively. These outer loops determined our blocks and jumped by block size. The following two inner loops, ‘x’ and ‘y’ iterate through the elements of the square blocks that were determined by ‘i’ and ‘j’. ‘x’ loops through the rows of each block of m1. ‘y’ loops through the columns of each block of m2. I choose to parallelize the ‘i’ and ‘j’ loops because it has been found that parallelizing the for loop that iterates through the block elements can increase data sharing and reduce synchronization cost, thus increasing performance (Roktaek Lim, 2018). In order to ensure that each block is calculated by a thread in parallel, the line ‘#pragma omp parallel for collapse(2)’ was added just before the inner ‘i’ loop. Normally, when the parallel construct is immediately applied to a for loop, threads are created for each iteration of that loop. In this case, I have used 'collapse (2)' to collapse the loops, ‘i’ and ‘j’, into a single loop and parallelise the resulting loop, therefore, each thread will get a pair of ‘x’ and ‘y’ to work on. They will represent the row and columns of each block respectively. A block will contain 10 rows from m1 and 10 columns from m2. I used private( y, x, k) to ensure that ‘y’, ‘x’ and ‘k’ could only be accessed by the individual threads. I defined the number of threads to be created using the num\_threads clause with the pragma. If the loop values are private to each thread, then there is no issue with multiple threads accessing that at same time. This means that there is no risk of threads working on the same block as another thread. We set the matrices as shared because we do not need to control synchronised access to them (Calkins, n.d.). The innermost loop, ‘k’, will be used for the block calculation. In order to parallelise a program, you must identify parts of the program that are independent of each and can be carried out at the same time. In our matrix multiplication study that refers to the code responsible for the actual calculation ‘results[i + x][j + y] += m1[i + x][k] \* m2[k][j + y];’ (Andrews, 2000). For each iteration, one row of m1 and one column of m2 per block will be ran through by ‘x’ and ‘y’ and the calculation will be carried out for each iteration of ‘k’. The section in results[i + x][j + y] is each block and will be updated with multiplication result.

## *Evidence as to the efficiency of your implementation:*

To ensure consistency, the same measure of efficiency was applied across each method. The omp\_get\_wtime routine was used to track runtime (OpenMP, 2018). As a baseline for comparison of all methods, the naive approach to matrix multiplication was carried out and timed (Jayawardana, 2018). This can be seen in the file ‘naive.c’. A range of thread numbers and block sizes were experimented on to pin down the optimal combination. The most efficient combination was selected based on runtime. A study carried out found that a multi-threaded program performs better than a single threaded program for matrix multiplication (Hemeida, 2020). This was found to be true in this study too as the solution with 4 threads outperformed the solution with a single thread. A block size of 10x10 and 4 threads was chosen. Figure 1 details the runtime results of these tests. All tests carried out from this point onward were applied to matrices of size 100x100 with block size 10x10 and 4 threads. In the naïve approach, found in file ‘naïve.c’, of large matrix multiplication, the potential of the cache is not utilised. By the time the multiplication of the first row of m1 and the first column of m2 has completed, elements will have been thrown out the cache. As ‘j’ iterates through m2 for each computation, each row in m2 is being stored in cache lines, but we are only using the value from a single column for each iteration. The rest of the values in each row, which we do not require, are in cache, read and unused. This slows down the program and is called cache pollution. *“To work with the cache, the way our program accesses the array elements has to take advantage of spatial locality — once a cache line is in the cache, we need to take full advantage of it.“* (Chen, 2010). In order to create a more efficient approach, our design had to make better use of the cache. Blocking can address this problem. I choose blocking because it is widely used for optimization as it improves and optimises memory hierarchy’s efficacy (Hemeida, 2020). Blocking is used especially with large matrices. Due to their size they cannot be placed in the cache memory, resulting in a lot of cache misses and performance reduction (Ristov, 2014). Blocking merely involves getting full benefit from the cache and thus improving memory utilisation (Intel, 2018) “*The main purpose of loop blocking is to eliminate as many cache misses as possible.”* (Intel, 2018). By splitting up the matrices into blocks, it allows a smaller amount of data, that we are working on, to cache and maximises how much data in cache is used. Once again, we are trying to use the cache to our advantage to improve performance. In the naive approach the matrices would be calculated as a whole, whereas with the blocked approach, threads are calculating each matrix block in parallel. The times of the naive approach and the blocked approach can be seen in Figure 2, with the naive approach taking 0.006544 and the blocked non-parallelised approach taking 0.007228. Parallelising increases performance further, leaving us with a runtime of 0.004989. We can see based on this runtime improvement the blocking design is more efficient than the naive approach. To further improve efficiency, there were several changes made to the blocked design and tested before the final design had been chosen. The results are detailed below. The final design was chosen based on runtime. The first element tested was the parallel critical section. I found that adding the critical construct for the calculation increased runtime. Without the critical section the program took 0.004989 to run, whereas with the critical section included it took 0.127974 to run. I choose not to keep the critical section. The second element tested was loop interchange. This ensures that data locality is benefited from by ensuring m2 is accessed contiguously. Within our blocked solution, we applied loop interchange to loops ‘x’ and ‘y’ which improved our runtime. Without loop interchange the runtime was 0.006007, instead of 0.004989 mentioned above. Therefore, it was kept in the design. The third element tested was the parallelisation of ‘i’ and ‘j’ loops. As mentioned in the design description, parallelising the loops iterating to make the blocks has been found to increase performance (Roktaek Lim, 2018). In this study, this was found to be true. I compared parallelising the outer two loops ‘i’ and ‘j’, with the inner loop’s ‘x’ and ‘y’. The runtime was faster for the outer two loops. Runtime for ij parallelisation was 0.004989 and for xy was 0.011838. There are many alternative algorithms for large matrix multiplication. One of which is, transposing, which was attempted as an alternative to blocking. This involved converting m2 to column major form, allowing for better use of cache, considering the matrices are stored in row major order it is more efficient and optimal to access elements of the matrices horizontally rather than vertically. A for loop was used to convert m2 (Thoma, 2013). The runtime for the parallelised transpose approach, which is file ‘parallelTranspose.c’ is 0.008253 compared to the naive transpose approach, 0.005537. This is faster than the naïve multiplication, however still slower than blocking. Another approach to increase runtime is scalar expansion. In the study, (Mallón, 2009), it was found that using a temporary buffer array instead of a temporary variable in the shared memory improved performance by reducing execution time. This was applied to the naive matrix multiplication. This involves changing a scalar to an array. The runtime was 0.005448. I then parallelised this and got 0.009205 runtime. Following on from this I applied loop interchange and runtime increased to 0.015638. The blocking approach outperforms this approach in terms of runtime. In this study, it was found that blocking outperforms other methods. Blocking does increase the number of operations, as we have 2 extra for loops, however, it reduces the average memory access time and this reduces runtime and increases performance (Ristov, 2014).

*Figure 1: Optimum block size and number of threads test results*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Dimensions** | **Block Size** | **Number of threads** | **Runtime** |
| Blocked | 100x100 | 2x2 | 1 | 0.006978 |
| Blocked | 100x100 | 2x2 | 4 | 0.005600 |
| Blocked | 100x100 | 3x3 | 1 | 0.010017 |
| Blocked | 100x100 | 3x3 | 4 | 0.004926 |
| Blocked | 100x100 | 8x8 | 1 | 0.008332 |
| Blocked | 100x100 | 8x8 | 4 | 0.004410 |
| Blocked | 100x100 | 10x10 | 1 | 0.006975 |
| **Blocked** | **100x100** | **10x10** | **4** | **0.004234** |
| Blocked | 100x100 | 12x12 | 1 | 0.013245 |
| Blocked | 100x100 | 12x12 | 4 | 0.007684 |

*Figure 2: Runtime comparisons of methods and changes to methods*

|  |  |  |
| --- | --- | --- |
| **Method** | **Filename** | **Runtime** |
| Naive Matrix Multiplication | naive.c | 0.006544 |
| Naïve Blocked Matrix Multiplication | naiveBlockedMatrixMult.c | 0.007228 |
| Naive Parallelised Matrix Multiplication (without blocks) | naiveParallel.c | 0.006205 |
| **Final Parallelised Blocked Matrix Multiplication (without critical section, with collapse and with loop interchange)** | **blockedParallel.c** | **0.004989** |
| Parallelised Blocked Matrix Multiplication (with critical section) | blockedParallel.c | 0.127974 |
| Parallelised Blocked Matrix Multiplication (without loop interchange) | blockedParallel.c | 0.006007 |
| Parallelised Blocked Matrix Multiplication (xy parallelised) | blockedParallel.c | 0.011838 |
| Parallelised Blocked Matrix Multiplication (without collapse) | blockedParallel.c | 0.116447 |
| Alternative Blocked Matrix Multiplication | alternativeBlockedParallel.c | 1.320930 |
| Naive Scalar Expansion | naiveScalarExpansion.c | 0.005448 |
| Parallelised Scalar Expansion | scalarExpansionParallel.c | 0.009205 |
| Parallelised Scalar Expansion (with loop interchange) | scalarExpansionParallel.c | 0.015638 |
| Naïve Transpose | naiveTranspose.c | 0.005537 |
| Parallelised Transpose | parallelTranspose.c | 0.008253 |

# Bibliography

Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley.

Calkins, C. (n.d.). *OPENMP*. Retrieved from Object Computing: https://objectcomputing.com/resources/publications/mnb/openmp

Chen, N. a. (2010). Patterns for cache optimizations on multi-processor machines. . *Proceedings of the 2010 Workshop on Parallel Programming Patterns* , pp. 1-10.

Hemeida, A. H. (2020). Optimizing matrix-matrix multiplication on intel’s advanced vector extensions multicore processor. *Ain Shams Engineering Journal.*

Intel. (2018, December 27). *How to Use Loop Blocking to Optimize Memory Use on 32-Bit Intel® Architecture*. Retrieved from Intel Software: https://software.intel.com/en-us/articles/how-to-use-loop-blocking-to-optimize-memory-use-on-32-bit-intel-architecture

Jayawardana, Y. (2018, Feburary 23). *matrix-multiplication*. Retrieved from GitHub: https://github.com/yasithmilinda/matrix-multiplication/blob/master/matrix.c

Mallón, D. T. (2009). Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting* .

OpenMP. (2018, November ). *OPENMP API Specification*. Retrieved from OpenMP: https://www.openmp.org/spec-html/5.0/openmpsu160.html

Ristov, S. G. (2014). Optimal block size for matrix multiplication using blocking. *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (pp. (pp. 295-300)). IEEE.

Roktaek Lim, Y. L. (2018). *OpenMP-based parallel implementation of matrix-matrix multiplication on the intel knights landing. In Proceedings of Workshops of HPC Asia (HPC Asia ’18).* New York, NY, USA: Association for Computing Machinery.

Thoma, M. (2013). *matrix-multiplication*. Retrieved from Github: https://github.com/MartinThoma/matrix-multiplication/blob/master/C%2B%2B/ikj-transposed.cpp