Parallelism of Sparse Matrix Operations

Written by Clayton Herbst (22245091)

Abstract

Matrices are the foundational structure used to represent the data around us. The computer's ability to naturally store and manipulate this information in contiguous memory locations provides exceptional computation benefits to otherwise complicated calculations. Matrix-like data structures closely represent the relationship between various datasets. The term 'vector' and matrix have slightly different interpretations however are interchangeable for the most part. Both of these structures help physicians & mathematicians represent the complex systems in our world through systems of linear equations. Matrix computation on computer systems are often used to give good approximations of computationally intensive calculations. In this way matrices can be used to store and represent nearly all sets of data. Other examples of applications include complex data structures such as binary trees, which can use matrix representation effectively to represent the relationship between various nodes within a tree. Graphics rely heavily on matrices to represent pixels within a file and their respective RGB colours. Graphical effects are the result of manipulating these pixel values using operations such as scalar and vector multiplication to create distortion. An understanding of the many implementations of matrices allows one to gain a greater appreciation for their importance in the modern data driven world. Any improvements gained on the computational efficiency of matrix operations are thus highly valuable to society and the focus of this report.

Background

Computer architectures traditionally consisted of single core hardware and a kernel managing the memory of various process states. Technology improvements led the size of electronic components within computers to decrease. These technological improvements caused drastic advancements in computing power. This phenomenon is described by Moore's law. In the early 21st century, electrical components had reached a bottle neck in their ability to decrease in size slowing the growth in computing power. The continued search for performance encouraged the development of computer systems containing multiple cores. This new architecture for computers rely on the allocation of processes to certain cores by the kernel and the effective management of memory between these processes. My macbook computer for instance has 4 physical cores. A computers apparent computing power can be increased through the effective use of shared memory between all available cores. The memory sharing between the SSD, cache and EPPROM is managed by the kernel by distributing threads for processors to execute. A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler (Wikipedia 2019). A single threaded program is a process that is executed as a single thread. Processes can consist of many threads, each thread executing its own sequence of instructions while accessing the shared memory of the process (Datta 2014). The kernel manages the sharing of memory such as to avoid memory overwrites across processes. Modern computer architectures maintain the independent execution environment of processes, however, introduce the concept of threads within processes that are able to be executed concurrently and share memory in the process's stack.

I have endeavoured to exploit modern computer architecture through the use of the OpenMP library to gain efficiencies in the calculations of common matrix operations. Parallel computing involves the simultaneous execution of processes (Wikipedia 2019) often used to break-down computationally intensive tasks further (Datta 2014). When implemented correctly parallelism can be used to effectively reduce the execution times of a process. In this report the practical processing times of the parallelised matrix operations will be compared to the practical computation times of the respective single threaded program (synchronous program). The algorithms derived will be tested using the macOS Mojave (v10.14.6) operating system, operating on a single Intel Core i5 processor with 2.3GHz processing speed and four physical cores. The hardware installed provides an L2 cache size of 256KB, L3 cache size of 6MB as well as 8GB of random-access memory (RAM). All results gathered will be highly specific to the architecture used due to the nature of parallel computing and multithreading phenomena.

Sparse Matrix Data Structures

Sparse matrices are matrices where most of the elements are zeros. A matrix is generally considered sparse if less than 20 percent of its elements are non-zero. Sparse matrices introduce many complexities and inefficiencies into computer algorithms due to inflated time and space complexities.

Space complexity is the analysis of the relative amount of space required by a process relative to the size of its inputs. Storing large sparse matrices in their entirety is highly inefficient and often not feasible due to restrictions in hardware capabilities particularly in local systems. Time complexity is the analysis of the relative time taken for a program to execute relative to its input size. The time and space complexity required by the algorithm can be reduced by increasing the efficiency of sub-problems within the algorithm. For sparse matrices, this can simply be achieved by compressing the sparse matrix into data structures that preserve the location of all elements however solely store the value of non-zero elements. Thus, sub-operations can be performed on a reduced number of elements, improving the programs time complexity. Additionally, less memory is occupied to store the matrix data.

Typical data structures used to preserve the contents of sparse matrices are the Coordinate (COO) format, Compressed Row Storage (CRS) format and the Compressed Column Storage (CCS) format. CRS and CCS are very similar data structures, containing subtle differences in the order in which the

non-zero elements are stored. It is thus assumed the characteristics of CRS and CCS with respect to storage and time complexities are identical.

The Coordinate (COO) format data structure is the simplest compressed representation of sparse matrices. This data structure consists of three arrays the size of the number of non-zero elements. The non-zero element is stored in one array and the other two store the coordinates of the non-zero element. The space complexity of the COO data structure is $O(3\mathrm{n})$, where n is the number of non-zero elements in the matrix. The coordinate format is advantageous when constructing sparse matrices for performing item-wise operations and allows for fast conversion into other sparse matrix data structures. However, improvements can be made on the space complexity of this data structure.

The Compressed Row Storage (CRS) format and Compressed Column Storage (CCS) format provide increased non-zero element storage density. The space complexity required by the CRS data structure is $O(2n + m + 1)$, where n is the number of non-zero elements and m are the number of rows in the sparse matrix. This is achieved by storing all non-zero elements in contiguous memory locations ($O(n)$), define an array referencing the number of non-zero elements in each row ($O(m + 1)$ and an array referencing the column each non-zero element belongs to ($O(n)$). The trade-off of the reduced space-complexity is increased algorithm complexity due to the extra addressing step required for every scalar operation. This is due to the coordinates of the non-zero elements not being explicitly stored (Dongarra 1995). The time taken to construct the CCS data structure is slower in comparison to COO and CRS when reading from a dense matrix file format, since elements cannot be read in a contiguous manner. The file reading position needs to jump ahead to fetch column elements and store elements in an erratic fashion. This data access pattern requires an interim COO data structure storage step and then a conversion to the CCS data structure.

All three storage structures were implemented in my command line tool in order to maximise the efficient and simple element-wise access provided by COO and make use of the unique data access patterns of CRS and CCS data structure to perform row-wise and/or column wise operations.

Parallel Sparse Matrix Scalar Multiplication

Scalar multiplication involves the traversal of each non-zero element of a matrix. The time complexity of this algorithm is thus $O(n)$, where n is the number of non-zero elements. Linear order traversal complexity is optimal for such an algorithm. Further efficiencies can be gained by taking advantage of modern computer architectures and shared memory systems by incorporating multithreading techniques.

I have implemented the COO data structure for two major reasons. Firstly, the COO data structure is more time efficient when converting from a dense sparse matrix format to the COO sparse matrix format and vice versa. This is due to the linear traversal of all entries and the storage of these elements in their order of discovery. The simplistic storage structure of COO results in fast conversions to and from dense sparse matrices as well as fast element wise access. Secondly the COO sparse matrix data structure efficiently preserves the location of each non-zero element and hence any zero elements as well. Data structures such as CRS and CCS rely on an additional lookup step to determine the location of an element.

The lack of computational intensity results in very fast and efficient execution of the single threaded scalar multiplication program. *Figure 1* demonstrates the execution times achieved with various sparse matrix input files sizes.

The possibility of gaining further speedup through multithreading was investigated. The OpenMP library *pragma omp parallel for* directive was used to effectively distribute the iterative workload amongst the available threads. The *pragma omp for* directive in particular provides useful functionality

in managing the even distribution of tasks to outstanding threads; thus, limiting uneven work distribution and any implicit barriers that may be experienced as a result. The following diagram illustrates the results captured in comparison to the single threaded program.

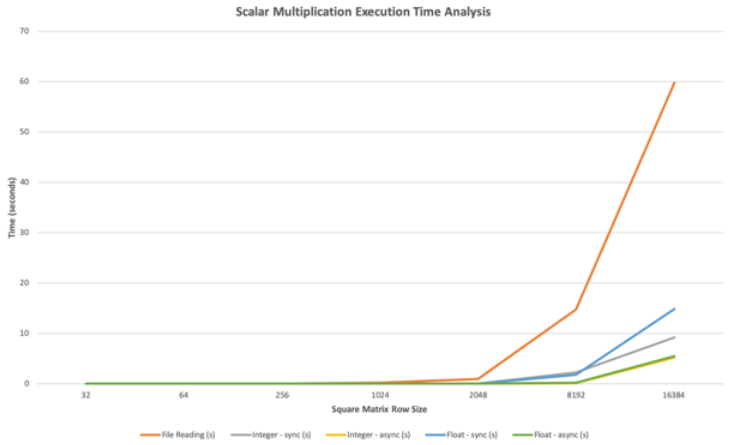

Figure 1: Scalar multiplication execution time graphical analysis

As *figure 1* visualises, for large matrices, the multithreaded program significantly outperforms the sequential program. Using the OpenMP library's support for parallelism, the time taken for execution grows approximately half the rate of the single threaded application. The *pragma omp parallel for* directive effectively manages the task scheduling to allow for efficient parallelisation of the for-loop and avoiding any race conditions. Notably, the result set provided in appendix A, reveals the slightly slower execution times for small matrix sizes ($< 256^2$ non-zero elements). This slowdown is most likely attributable to the hyperthreading feature of intel chips in modern computer architectures. Hyperthreading attempts to maximise processor use by overlapping instructions. The result is one physical core appearing to act as two logical cores (PC Mag 2019). This feature is highly specialised and in the general case results in performance loss.

The input and output (I/O) operations remain the slowest operation. The dependence of the program counter on the open files reading position is a major limiting factor in attempting to parallelise the I/O operations. This characteristic of I/O functionality will remain a bottleneck in the performance of my program's execution times over large matrices.

Improvements on my implementation include using the CRS or CCS data structures as to improve the space complexity of the algorithm. Further testing can be conducted using varied number of threads for certain matrix file sizes. This will allow further inferences to be made as to the ideal thread count to file size ratio specific to my CPU's architecture.

Parallel Sparse Matrix Trace Calculation

The sum of all elements across the diagonal of a matrix is defined to be the trace of that matrix (Wikipedia 2019). The computation of a matrices trace is useful in many applications involving linear algebra, the result of which is only strongly defined for square matrices. The algorithm I have implemented to compute the trace of a given matrix traverses through all non-zero elements of the given matrix summing all elements with equivalent column and row index values. The resulting time complexity is $O(n)$, where n is the number of non-zero elements. The following pseudo code illustrates the algorithm.

```
Given the matrix A represented by the COO struct.

program TRACE(COO A)
  for each non-zero element i in A
    if the row A[i].row equals the column A[i].col of for a non-zero
element
      trace_result <-- trace_result + A[i].element
    end if
  end for
end program
```

Figure 2: Trace calculation algorithm pseudo code

As the pseudo code above suggests, the COO sparse matrix data structure was implemented due to similar reasonings given in scalar multiplication. Other sparse matrix data structures can be implemented however the added complexity in computation is unjustified. The data access pattern for elements within the COO data structure is the best fit to satisfy the algorithms sole purpose of traversing the given matrices non-zero elements and quickly determining their coordinates.

The *pragma omp parallel for* directive was used to efficiently distribute tasks to outstanding threads. In order to ensure no race conditions between threads sharing the trace sum variable, the OpenMP clause, *reductions(+:var)* was implemented. Due to the recurring nature of data sharing patterns such as summing values between parallel threads, the OpenMP library provides the functionality of managing any potential race conditions between these threads through the *reductions(+:var)* clause. This functionality reduces the need to declare a critical section using the OpenMP directive *omp critical* or the need to perform thread specific operations within the containing for loop. The following graph represents the results achieved.

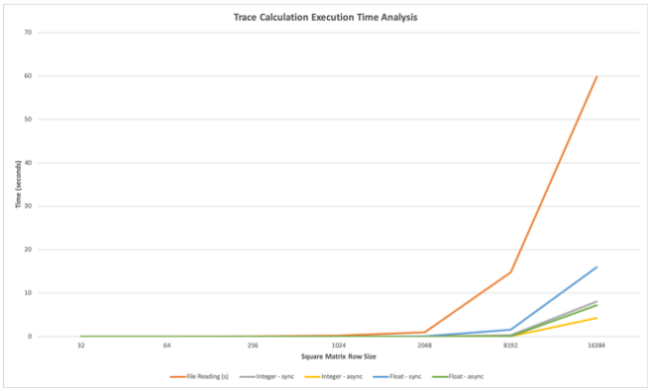

Figure 3: Trace calculation execution time graphical analysis

As *figure 3* suggest, my program was able to achieve speed-up for large input matrices. Race conditions within the parallelised for-loop were avoided and the efficient task load sharing was managed by the *pragma omp for* directive. As previously discussed for scalar multiplication, file reading remains the bottleneck of the application.

The COO data structure remains the most efficient for this algorithm as fast boolean comparisons can be made on the non-zero elements matrix coordinates. Further investigation can be conducted in order to find the most efficient thread count to matrix size ratio, for my computer's architecture.

Parallel Sparse Matrix Addition

Matrix addition involves the element wise addition of two or more matrices. Due to the nature of element wise operations, both matrices specified need to have equivalent dimensions. The resulting time complexity will be relative to the number of rows and columns the program has to traverse through. Thus, the time complexity expressed using big-O notation is $O(n)$, where n is the product of the matrix's rows and columns (i.e the total number of matrix elements). The time complexity clearly remains linear to the size of the input matrix; the optimal time complexity of a sequential algorithm needing to traverse all elements.

The data structures used within my implementation of the matrix addition algorithm were two CSR structures. Elements with identical coordinates are added and the result stored in a COO data structure. The reasoning behind the use of CRS was to benefit from the reduced space complexity as well as benefitting from the row-wise data access pattern. Sequential execution of the program results in fast execution times as seen in *figure 5*. The referential locality of values in matrix addition allows for the parallelisation of the algorithm due to the independence of each calculation from one another. The aim of my multithreaded algorithm was to parallelise the row-wise operation of the outer for-loop control structure in order to achieve speed-up.

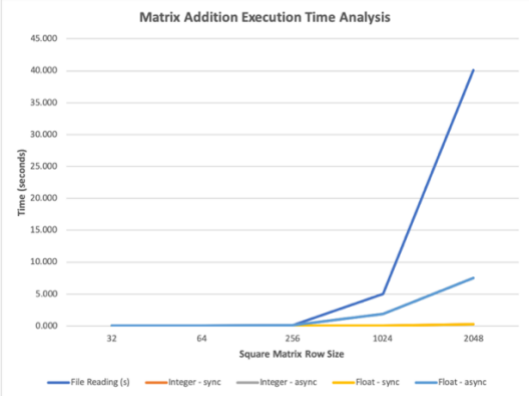

Figure 5: Matrix addition execution time graphical analysis

The results displayed in *figure 5* reflect the disappointing outcomes of the parallelised program. In order to maintain the correctness of the parallelised algorithm, very minimal tasking in the team of threads was implemented. Initial testing conducted on parallelising the outer for-loop using the OpenMP's *pragma omp for* directive yielded incorrect and erratic results. The multiple dependencies that existed between threads executing in parallel exposed many race conditions that could not be explicitly managed through the use of more selective OpenMP memory sharing clauses such as *shared*, *private*, *firstprivate* and *reduction*. The race conditions remained in all further tests conducted. As a result, my final solution highly underutilises modern computer architecture to improve computational performance. The synchronous execution time performance remained strong for large matrix sizes with no apparent bottlenecks.

Improved utilisation of multi-core hardware and OpenMP's api could result in greater program execution performance. In order to achieve this, the dependencies that exist within the nested for-loops need to be removed. This requirement most likely demands a change in choice of matrix data structure used to represent the input file. Further research and testing will need to be covered however the use of the COO data structure seems the most likely alternative due to each non-zero elements independence. See the improvements suggested in the sparse matrix multiplication section of this document for further discussion on the blocking nature of CRS and CCS data structures and potential algorithm improvements.

Sparse Matrix Transposition

The transpose of a matrix is a matrix that has been flipped over its diagonal (Wikipedia 2019). This operation results in the rows of the transposed matrix reflecting the columns of the original matrix. The swapping of matrix coordinates requires the element wise traversal of the original matrix. Considering sparse matrices and how they are represented in data structures, only the coordinates of all non-zero elements need to be traversed and have the swapping operation performed. The time complexity of finding the transposed matrix is thus $O(n)$, where n is the number of non-zero elements stored in the sparse matrix data structure.

The COO sparse matrix data structure seems initially to be the most natural choice given explicit storage of each non-zero elements coordinates. This allows for the explicit swapping of row and column values. The algorithm would consist of a single loop traversing through all non-zero elements, swapping the stored column and row index values for each element. The pseudo code is given in *figure 6.*

```
Given the matrix A represented by COO data
structure

procedure TRANSPOSE(COO A)
  for each element A[i] in A
    A[i].row <=> A[i].col
  end for
end procedure
```

Figure 6: Transpose operation pseudo code

Parallelising the algorithm presented in *figure 6* would involve similar techniques to those implemented in the scalar multiplication algorithm and trace calculation. The for loop containing the swapping operation can be optimised by distributing the workload across outstanding threads. Since the operation of every inner block of code is independent from the other the algorithm is suited to incorporate parallelism. The *pragma omp parallel for* directive can hence be applied. Special consideration must be taken for the cache usage as to avoid any cache misses as this would be the most critical bottleneck of the operation. In order to avoid this, pre-processing may need to be performed in order to prepare the ordering of coordinate index values storage in contiguous memory addresses.

In my implementation I have however used the CRS data structure to store the non-zero elements of the given sparse matrix. This decision has been made in order to benefit from the improved storage complexity of the CRS data structure as well as taking advantage of the element access pattern of the data structure. The contiguous row-wise storage of the matrices non-zero elements allows for a niche conversion to be made between the compressed row storage data structure and the compressed column storage data structure through a single system call. The operation assumes sufficient addressable memory is available for the array of non-zero elements to be copied to the CCS data structure as well as the metadata surrounding the non-zero values. The time complexity of the algorithm remains $O(n)$, where n is the number of non-zero values. The system call used to copy the values of the contiguous memory locations within the CRS struct is *memcpy*. System calls provide optimised execution times as the source code's reliance on compiler optimisation is reduced, thus the actual time performance of the algorithm is expected to be significantly improved relative to the number of non-zero elements in the sparse matrix. Due to the system call's optimisation, multithreading may only be a benefit for very large matrix files.

This techniques conversion to the CCS data structure is also beneficial to support any matrix multiplication operations that commonly follow matrix transposition.

Since the high-performance capabilities of modern architecture remains underutilised the *memcpy* system call can be split in such a manner as to evenly distribute the workload across the number of threads available in the team. All dynamic memory allocations will need to be made before parallelising the *memcpy* system call. In order to efficiently distribute the workload, the number of memory address's copied need to be a function of the total number of non-zero elements and the number of threads

available. This load balance between threads is handled by the OpenMP's *pragma omp parallel for* directive. The implementation hence solely involves wrapping *memcpy* function calls in for-loops the size of the addressable memory needing to be copied. The performance comparison seen in *figure 7* is noticeably similar. From the results gathered, the single threaded program remains the most efficient.

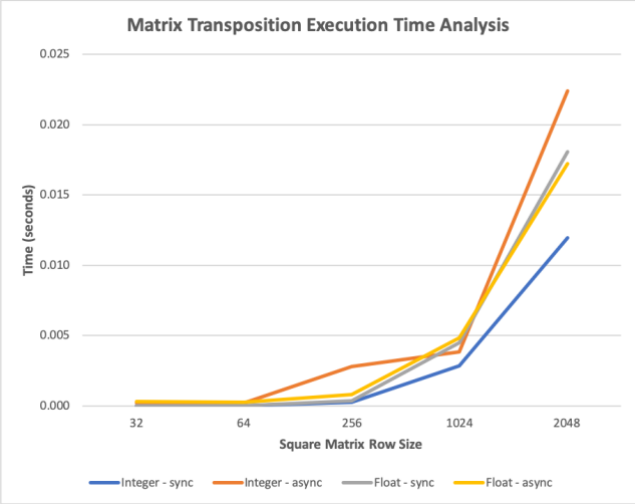

Figure 7: Matrix transposition execution time graphical analysis

Parallel Sparse Matrix Vector Multiplication

Matrix vector multiplication or the '*dot-product'* is a critical calculation in fields such as physics used to determine the magnitude of mechanical work by finding the dot product of the force vector and the displacement vector. Matrix multiplication is also commonly used in graphics to apply distortions to graphical elements. The operation involves taking two equal-length arrays of numbers, finding the product of each element with identical indexing and summing all values in the resultant array. The result of a single operation as described above is a single number (Wikipedia 2019). Mathematically this can be described using two vectors *a* and *b* as:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

Let a_r represent the row vector at row index *r*. Let b_c represent the column vector at column index *c* in the second input matrix. For each row in the first matrix, calculate the dot-product of every column in the second matrix. The scalar result is stored in the resultant matrix at the coordinates (r, c). Considering the computation of such an operation on sparse matrices, the memory access patterns of both the CRS and CCS data structures are closely aligned with the row to column comparisons made between input matrix one and two. The resultant matrix can be stored in a CRS sparse matrix data structure to allow for synchronous storage of computed vector dot-products. The space complexity of such an algorithm relies on three data structures consisting of $O(2n + r + 1)$. Time complexity analysis results in a derived time complexity of $O(r_1 r_2 c_2)$, where:

- r_1 represents the number of rows in the first matrix
- r_2 represents the number of rows in the second matrix
- c_2 represents the number of columns in the second matrix

Evidently this is the most computationally intensive operation considered within this report. The fundamental structure of the algorithm involves three nested for-loops with only the inner most block theoretically containing execution dependencies; hence requiring sequential execution. Within my implementation, the use of the CRS sparse matrix data structure results in a sequential execution dependency forcing the use of blocking operations. This is due to the CRS structures use of element

ordering to implicitly determine the coordinates of the non-zero element. The resultant matrix can instead be represented by the COO data structure, where the coordinates of non-zero elements can be explicitly stated. The revised non-blocking functionality of such an algorithm introduces the concepts of partially and fully distributed memory sharing during sparse matrix vector multiplication.

The partially distributed memory sharing technique for matrix vector multiplication involves the partitioning of the first matrix row wise allowing allocated threads to yield submatrices rather than scalar quantities (Reinders & Jeffers 2014). This technique would involve creating tasks to be executed by the team of threads the size of the external for-loop block. Each submatrix created is thus stored in unique chunks of memory. A major factor to consider other than data races is the load balance between threads. In order to ensure efficient parallelisation, each thread in the team should perform similar amounts of *'work'*. The OpenMP library directive *pragma omp for* manages the efficient load distribution between threads in the team.

The fully distributed memory sharing construct for matrix vector introduces added complexity as the referential locality of the rows and columns is exploited (Reinders & Jeffers 2014). Modern supercomputer systems implement fully distributed algorithms that are additionally parallelised over multiple nodes. These algorithms however are designed to be implemented using high performance distributed systems in addition to local memory sharing, rather than solely using local multithreading techniques.

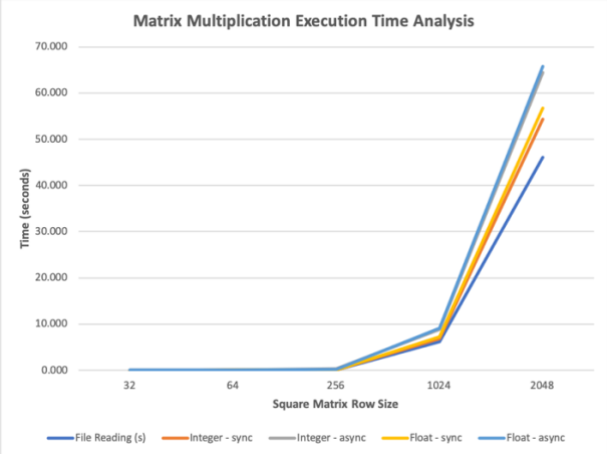

Figure 8: Matrix multiplication execution time graphical analysis

The results displayed in *figure 8* are the outcome of underutilisation of available computer processing power. Many complications were encountered while attempting to parallelise the sequential algorithm described above. Even though the mathematical algorithm for calculating the dot-product of two matrices consists of many independent calculations that can be parallelised, the direct use of the CRS and CCS data structures forced the blocking/sequential execution of the program. This blocking effect is most likely attributable to the sequential storage of non-zero values by the structure. The sequential data access pattern results in implicit dependencies between loop program flow control structures resulting in many race conditions upon parallelising the execution block.

In order to improve the multithreaded matrix multiplication algorithm, necessary pre-processing needs to be implemented. Perhaps the CRS row-wise values can be in a COO data structure. Thus, the outer control loop is able to independently assign contiguous memory addresses to rows for further execution on a per-column basis of the second matrix. For matrices of few non-zero elements, the initial performance may experience a slow-down due to the added pre-processing however for larger matrix sizes, the benefits of parallelised execution may outweigh the pre-processing costs. Further complications that may be encountered in this improved solution include balancing the computation load of each thread in the team as well as managing the cache accesses effectively.

A parallelised local program stores certain chunks of addressable memory in the L2 and L3 cache. Effective use of cached memory results in tremendous program performance due to the fast cache memory access times. However, when threads need to constantly rewrite the stored cache memory due to limitations in cache size, a phenomenon known as cache thrashing is experienced. Cache thrashing occurs when a computers virtual memory resources are overused, leading to consistent data re-writes between comparatively slow levels of memory hierarchy and fast levels of memory hierarchy (Wikipedia 2019). This can be avoided by storing all non-zero values and related metadata in contiguous memory, allowing particular threads to access only particular indexed memory addresses. The additional restructuring will add to the pre-processing cost of the algorithm, however if implemented correctly will result in great program execution speed-up.

Conclusion

Significant execution speed-up was achieved for scalar multiplication and trace calculation programs however the inefficient use of modern computer architectures processing power for the matrix addition and matrix multiplication operations resulted in sub-optimal performance. Further experimentation and testing need to be conducted in order to work towards completely utilising available computer processing power and thus achieve optimal performance on computationally intensive tasks.

Appendix A

Scalar Multiplication Table of Results:

Size	File Reading (s)	Integer - sync (s)	Integer - async (s)	Float - sync (s)	Float - async (s)
32	0.000	0.000	0.000	0.000	0.000
64	0.001	0.000	0.001	0.000	0.000
256	0.018	0.000	0.003	0.000	0.003
1024	0.238	0.006	0.002	0.003	0.001
2048	0.934	0.032	0.010	0.026	0.005
8192	14.740	2.249	0.142	1.762	0.167
16384	59.782	9.226	5.162	14.817	5.437

Note:

- *Input files used during testing were dense in order to ensure consistent computational difficulty across all result sets.*
- *The 'size' integer represents the row and column sizes of the tested square matrices.*
- *Sync represents the single threaded program execution environment.*
- *Async represents the multithreading execution environment.*

Appendix B

Trace Calculation Table of Results:

Size	File Reading (s)	Integer - sync	Integer - async	Float - sync	Float - async
32	0.001	0.000	0.000	0.000	0.000
64	0.001	0.000	0.000	0.000	0.000
256	0.018	0.000	0.000	0.000	0.000
1024	0.238	0.003	0.002	0.003	0.001
2048	0.934	0.027	0.008	0.024	0.005
8192	14.740	0.317	0.080	1.523	0.086
16384	59.782	8.060	4.260	15.925	7.238

Note:

- *Input files used during testing were dense in order to ensure consistent computational difficulty across all result sets.*
- *The 'size' integer represents the row and column sizes of the tested square matrices.*
- *Sync represents the single threaded program execution environment.*
- *Async represents the multithreading execution environment.*

Appendix C

Matrix Addition Table of Results:

Size	File Reading (s)	Integer - sync	Integer - async	Float - sync	Float - async
32	0.001	0.000	0.002	0.000	0.002
64	0.004	0.000	0.004	0.001	0.007
256	0.113	0.004	0.114	0.004	0.119
1024	5.059	0.066	1.876	0.064	1.900
2048	40.080	0.260	7.479	0.308	7.543

Note:

- *Input files used during testing were dense in order to ensure consistent computational difficulty across all result sets.*
- *The 'size' integer represents the row and column sizes of the tested square matrices.*
- *Sync represents the single threaded program execution environment.*
- *Async represents the multithreading execution environment.*

Appendix D

Matrix Transposition Table of Results:

Size	File Reading (s)	Integer - sync	Integer - async	Float - sync	Float - async
32	0.000	0.000	0.000	0.000	0.000
64	0.002	0.000	0.000	0.000	0.000
256	0.064	0.000	0.003	0.000	0.001
1024	2.574	0.003	0.004	0.004	0.005
2048	18.995	0.012	0.022	0.018	0.017

Note:

- *Input files used during testing were dense in order to ensure consistent computational difficulty across all result sets.*
- *The 'size' integer represents the row and column sizes of the tested square matrices.*
- *Sync represents the single threaded program execution environment.*
- *Async represents the multithreading execution environment.*

Appendix E

Matrix Multiplication Table of Results:

Size	File Reading (s)	Integer - sync	Integer - async	Float - sync	Float - async
32	0.000	0.000	0.002	0.000	0.002
64	0.004	0.002	0.011	0.002	0.011
256	0.131	0.114	0.292	0.111	0.260
1024	6.188	6.900	8.812	7.245	9.093
2048	46.149	54.324	64.459	56.689	65.791

Note:

- *Input files used during testing were dense in order to ensure consistent computational difficulty across all result sets.*
- *The 'size' integer represents the row and column sizes of the tested square matrices.*
- *Sync represents the single threaded program execution environment.*
- *Async represents the multithreading execution environment.*

References

Hardesty L, 2013. *Explained: Matrices,* MIT News. Available from: http://news.mit.edu/2013/explained-matrices-1206. [17 Sep 2019].

Dongarra J, 1995. *Compressed Row Storage*. Available from: http://netlib.org/linalg/html_templates/node91.html. [22 Sep 2019].

Reinders, J, & Jeffers, J, 2014. *Higher Performance Parallelism Pearls : Multicore and Many-Core Programming Approaches*, Elsevier Science & Technology, Saint Louis. Available from: ProQuest Ebook Central [29 Aug 2019].

Wikipedia 2019, *Sparse Matrix, 23 July 2019.* Wikipedia Encyclopaedia. Available from: https://en.wikipedia.org/wiki/Sparse_matrix. [16 Sep 2019].

Wikipedia 2019, *Thread (computing)*, 23 September 2019. Wikipedia Encyclopaedia. Available from: https://en.wikipedia.org/wiki/Thread_(computing). [23 Sep 2019].

Wikipedia 2019, *Trace (linear algebra)*, 23 September 2019. Wikipedia Encyclopaedia. Available from: https://en.wikipedia.org/wiki/Trace_(linear_algebra). [23 Sep 2019].

Wikipedia 2019, Transpose, 22 August 2019. Wikipedia Encyclopaedia. Available from: https://en.wikipedia.org/wiki/Transpose. [24 Sep 2019].

Wikipedia 2019, Dot Product, 23 Sep 2019. Wikipedia Encyclopaedia. Available from: https://en.wikipedia.org/wiki/Dot_product. [24 Sep 2019].

PC Mag 2019, *Hyperthreading.* PC Mag Encyclopaedia. Available from: https://www.pcmag.com/encyclopedia/term/44567/hyperthreading. [24 Sep 2019].

Wikipedia 2019, Parallel Computing, 23 September 2019. Wikipedia Encyclopaedia. Available from: https://en.wikipedia.org/wiki/Parallel_computing. [19 Sep 2019].

Datta A, 2014. High Performance Computing. Available from: http://teaching.csse.uwa.edu.au/units/CITS3402/lectures/index.html. [17 Sep 2019].