School of Engineering and ICT
ICT Discipline

UNIVERSITY *of* TASMANIA

Electronic Submission Assignment Cover Shee

If you are required to submit your assignment work via MyLO or if you are required to do an electronic submission only (no paper submission), then you must submit a copy of this cover sheet using the same method as you are required to submit the rest of the assignment.

Student ID:	081136	Login Name:	Cormaccollins (or cormacc)
Family Name:	Collins	Given Name:	Cormac
Unit Code:	**KIT318/ 418**	Unit Name:	Big Data and Cloud Computing
Tutorial Day/Time:	Mon 1:00pm		
Assignment Title/Number:	**Assignment 1 Multithreaded Matrix Calculator**		

UTAS - KIT318/418: Assignment 1

Multithreaded Matrix Calculator

Cormac Collins

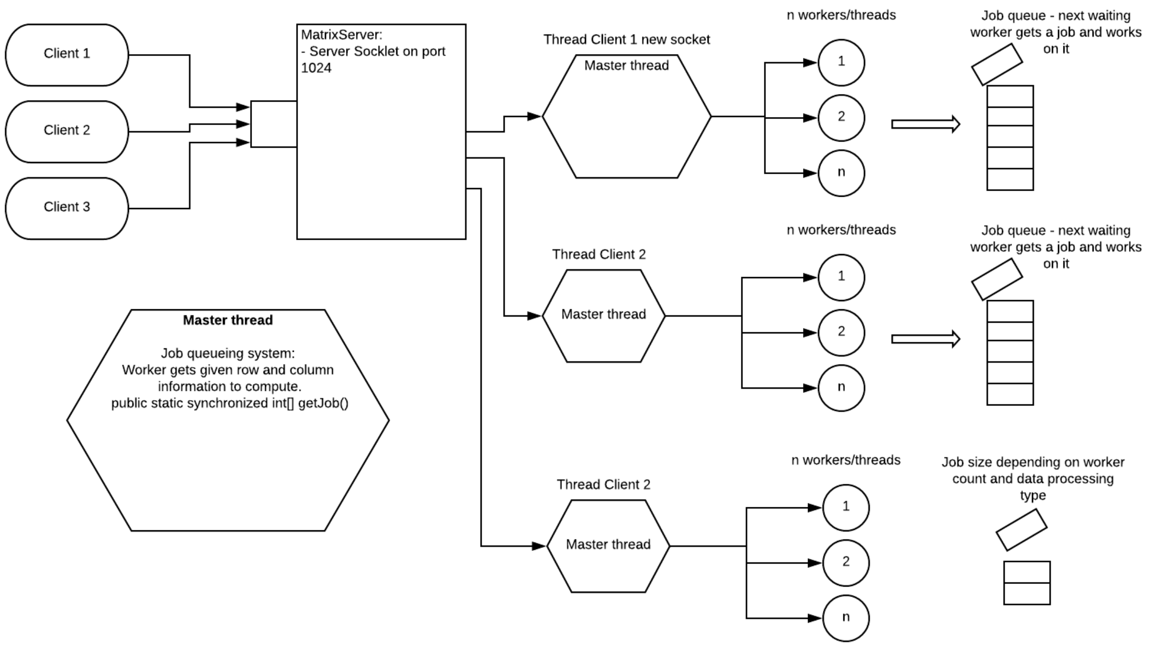
Student ID: 081136

Program Structure

The basic server-client protocol operates on TCP sockets, with the Client making a request via a String, while the server responds with a *serializable object* implemented as the class 'MatrixResult'.

The basic structure of the program includes a client making requests to the 'MatrixServer' class which for each new client, creates a new socket and runs a 'ThreadManager' on it (Thus freeing up the port it is running on for further client requests). The ThreadManager class itself is a thread that then distributes the matrix calculation work between several child threads of the CalculationThread class (the thread count is decided when the server is started).

The work to be performed between threads is divided up by the ThreadManager and placed in a shared data queue, then one of the CalculationThreads can request work from this queue and begin its calculations while the other threads begin also requesting work. The threads finish when the queue has emptied.



Client 1
Client 2
Client 3
MatrixServer:
- Server Socklet on port 1024
Thread Client 1 new socket
Master thread
n workers/threads
1
2
n
Job queue - next waiting worker gets a job and works on it
Thread Client 2
Master thread
n workers/threads
Job queue - next waiting worker gets a job and works on it
Thread Client 2
Master thread
n workers/threads
Job size depending on worker count and data processing type
Master thread
Job queueing system:
Worker gets given row and column information to compute.
public static synchronized int[] getJob()

Figure 1: Client-Server interaction and thread hierarchy.

Synchronization:

Because of the queueing structure this program would be considered SIMD – single instruction multiple data processing. This uses shared data as the 2 matrices are accessible by all workers anytime, however they will never overlap in workflow because of the synchronized provision of jobs from the queue via the thread manager.

As per the division of these tasks, the single row by column operations (Cyclicv1) are easily divided between the workers equally, as well as the full row operations (Cyclicv2), with the rows divided between the workers.

The data splitting division (blockv1), is handling by calculating a 'splitsize' followed by a 'remainder' for uneven division for row-columns and worker counts. We can see in figure 2, given a 5x5 matrix with the request for 2 workers, trying to evenly split the work isn't quite possible here. Therefor the remainder rows and columns are required to complete the workload. These same remainders would occur in an uneven matrix of size 25x25.

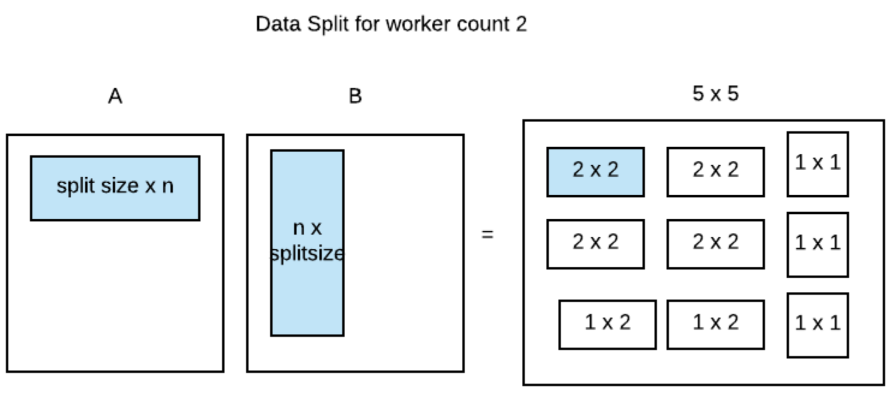
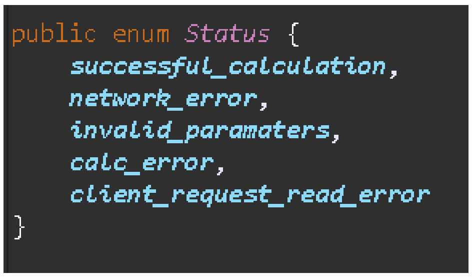


Figure 2: Data splitting calculation model (blockv1), using a 'splitsize' parameter relative to the worker number to divide the work.

Error Handling:

One of the below error's can be sent back to the Client in the 'MatrixResult' object that contains and error code and the result matrix:



```
public enum Status {
    successful_calculation,
    network_error,
    invalid_paramaters,
    calc_error,
    client_request_read_error
}
```

These errors are allocated as per their given names, primary errors coming from things such as an incorrect string parameter. For example, if it does not come in the format of: 'calculationType'-'matrixSize', then the 'invalid_parameters' error will be returned. Or things such as errors in the server reading from the Client socket input will result in a network 'client_request_read_error'.

Calculation errors:

Threading errors are caught within try catch and will return as an unsuccessful calculation (calc_error). During testing, threading errors were picked up within he data partitioning 'blockv1' calculating method, however these did not affect the correctness of the answers and may have been a thread manager queueing issue.

Critical analysis

As might be expected for matrix calculation, as the size of an nxn matrix increases, the multiplication computation efficiency will rise close to the level of exponentially at O(n^2) when calculating it serially. This is very close to what Figure 3 shows through with a roughly parabolic curve shown for the non-parallel calculation (1 thread), however as increasing threads are introduced, much larger efficiencies are gained at large size matrices and we see the slope of this curve decreasing.

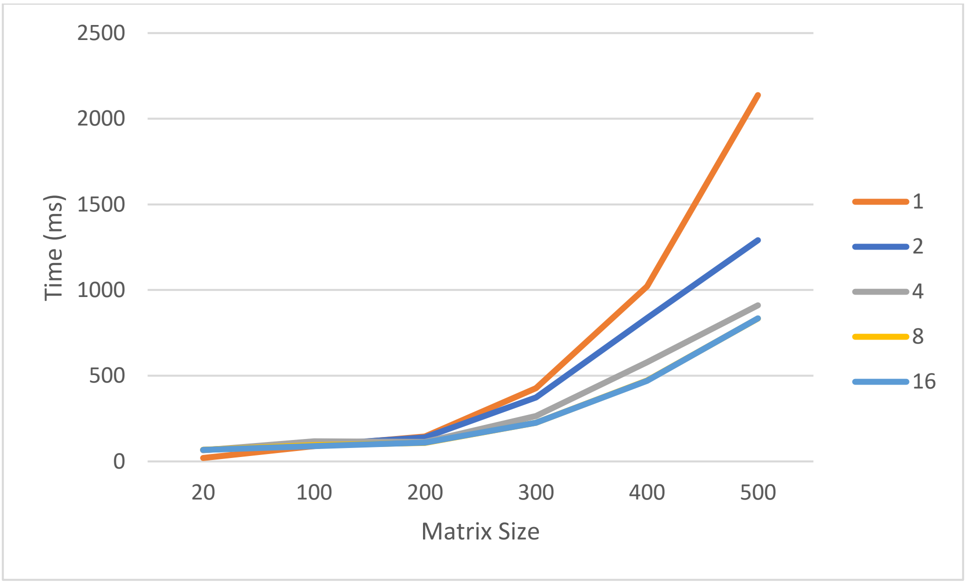


Figure 3: Row by columns calculation method response times for increasing matrix size calculations with increasing threads count (in colour).

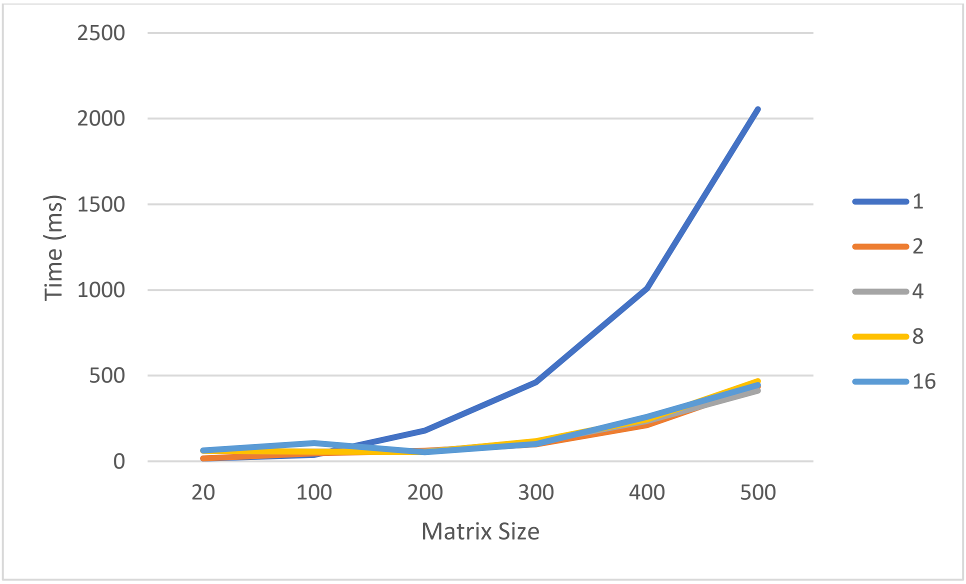


Figure 4: Full row calculation method response times for increasing matrix size calculations with increasing threads count (in colour).

As might appear obvious to the row by column calculation, calculating each individual element for the result matrix may be inefficient with excess overheads for function calls. So by accessing bigger chunks of data for each job we can do more within 1 processing job. This is apparent in Figure 4 as shown by the even greater efficiency in this method at the high matrix size and thread counts. With this form of calculation, we may also be taking more advantage of hardware caching because of the easy linear access to contiguous rows of data in when accessing matrix rows.

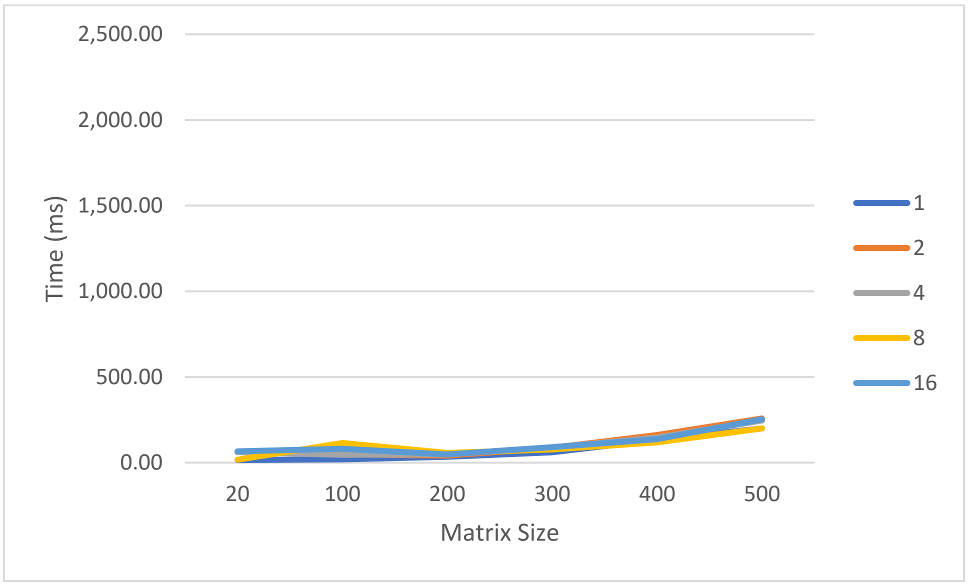


Figure 5: Data Split calculation method response times for increasing matrix size calculations with increasing threads count (in colour).

Finally, the 'Data Split' model appears to have by far shown the greatest improvements in efficiency with rising matrix size and worker counts. This likely takes advantage of a better distributing of work across parallel threads, and the previously menetioned caching.

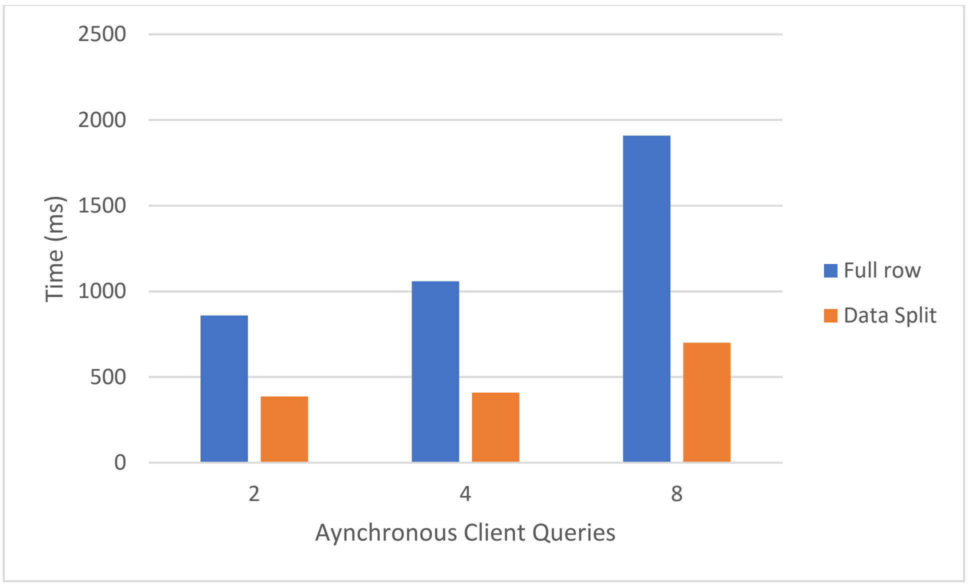


Figure 6: Average response times while querying 500 size matrices using 16 threads during different numbers of client connections.

Lastly, we observe the effects of added client activity on response times. Above we can see the average response times for clients requesting 500 size matrices asynchronously, this was achieved through scripted asynchronous calls for each new client, therefore this would be some minor lag time between the initiation of clients. Despite this we can clearly see at an almost tripling in response time for the 'Data Split' method while querying with 16 threads for 500 size matrices. This displays the obvious drop off when the same machine is performing these calculations, once the advantage of multiple processors is lost, the calculations are simply creating smaller and smaller time slices allocated via the operating system on the same amount of processing memory. This of course would lead the reader to research the area of clustered computing, which utilizes multi-tasking across multiple computers not just threads.