Multi-threaded Matrix Multiplication Client-server Application

Documentation and Performance Analysis

Joshua Stevenson (Student ID: 386572)

August 13, 2018

1 The Application

1.1 Architecture

The application utilises the master/worker concurrency model, in which a master process (here, the *ClientHandlerThread*) creates multiple worker processes (*WorkerThreads*) which share memory in order to complete a task—in this case, the task is to compute the result of a matrix multiplication with two randomly generated matrices of a given size.

The application also implements a client-server model, so that multiple matrix calculations can occur at the same time for different clients. Here, the client executable sends a computation request to the server over the network, and the server performs the necessary calculations, returning the result to the client, again over the network. See Figure 1 for details.

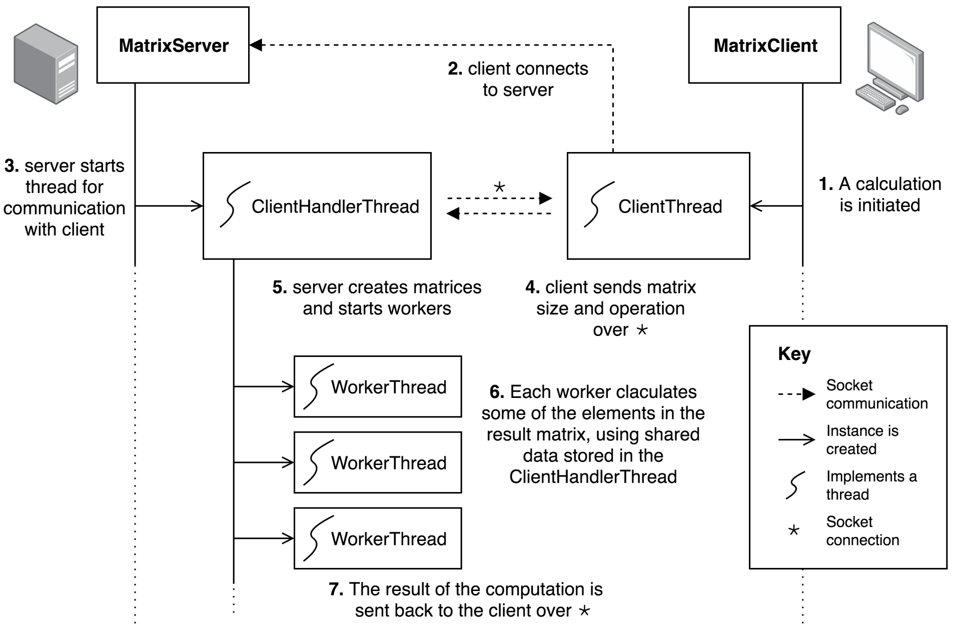
MatrixServer
MatrixClient
2. client connects to server
3. server starts thread for communication with client
ClientHandlerThread
⋆
ClientThread
1. A calculation is initiated
5. server creates matrices and starts workers
4. client sends matrix size and operation over ⋆
WorkerThread
WorkerThread
WorkerThread
6. Each worker claculates some of the elements in the result matrix, using shared data stored in the ClientHandlerThread
7. The result of the computation is sent back to the client over ⋆
Key
Socket communication
Instance is created
Implements a thread
Socket connection

Figure 1: Threads and sockets are used to parallelise matrix computations.

1.2 Client-server Protocols

The software uses Java *sockets* to implement TCP, enabling communication between the client and server programs. This connection-based communication ensures reliability in data exchange between the client and server.

1.3 Synchronisation

The design of the application, including the architecture and data structures chosen, meant that the Java keyword *Synchronize* was not needed. That is, threads writing to shared memory write separate data to distinct locations, and no worker thread ever writes to a memory location that another worker needs to read from.

1.4 Workload Partitioning

The application implemented three different methods for partitioning the task of calculating the result of a matrix multiplication. That is, each of the below methods describe a different way of determining which of the elements of the resulting matrix **C** is calculated by each worker, and hence which of the elements of the matrices **A** and **B** each worker needs to read.

$$(i)\begin{bmatrix} w_1 & w_2 & w_3 & w_4 & w_1 \\ w_2 & w_3 & w_4 & w_1 & w_2 \\ w_3 & w_4 & w_1 & w_2 & w_3 \\ w_4 & w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 & w_1 \end{bmatrix} \quad (ii)\begin{bmatrix} w_1 & w_1 & w_1 & w_1 & w_1 \\ w_2 & w_2 & w_2 & w_2 & w_2 \\ w_3 & w_3 & w_3 & w_3 & w_3 \\ w_4 & w_4 & w_4 & w_4 & w_4 \\ w_1 & w_1 & w_1 & w_1 & w_1 \end{bmatrix} \quad (iii)\begin{bmatrix} w_1 & w_1 & w_2 & w_2 & w_2 \\ w_1 & w_1 & w_2 & w_2 & w_2 \\ w_3 & w_3 & w_4 & w_4 & w_4 \\ w_3 & w_3 & w_4 & w_4 & w_4 \\ w_3 & w_3 & w_4 & w_4 & w_4 \end{bmatrix}$$

Figure 2: Examples of the three partitioning methods (i) cyclicv1, (ii) cyclicv2, (iii) blockv1, being used on a 5×5 matrix with 4 workers. Here, elements marked with w_i would be calculated by worker i.

Using the cyclicv1 method, the *individual elements* of **C** are distributed to each worker in a cyclic manner. That is, left-to-right and then top-to-bottom.
Using the cyclicv2 method, the *rows* of **C** are distributed to each worker in a cyclic manner.
Using the blockv1 method, the elements of **C** are broken up into a grid pattern as evenly as possible (using the most balanced pair of factors of the number of workers). Each block in the grid is then assigned to one worker.

1.5 Error Handling

All exceptions which arise from sockets communication, thread management or other code are handled appropriately. This means that an appropriate error message is displayed, and the application will try to recover where possible, and otherwise close gracefully. For example, if a client attempts to perform a calculation when the server is unavailable, the user will see:

```
Could not connect and send calculation info to server
Call to 'calculate' completed with error code -1
```

2 Class Design

2.1 Server Classes (Server.java file)

MatrixServer

This is an executable class which starts the server to which clients will connect in order to perform calculations. The server is started with two arguments: the *port number* on which to open a connection, and the *number of worker threads* to use when performing calculations.

Upon starting, the *MatrixServer* class will open a socket on the specified port and wait for connections. When a connection request is received over the socket, the server will start a *ClientHandlerThread* to manage the request.

ClientHandlerThread

The *ClientHandlerThread* waits to read data from the client. This data includes the *size* of the matrices to be multiplied, the *operation*, and a boolean to inducate whether the server should return the result matrix. Since only multiplication is supported, the *operation* parameter refers to the method of partitioning the work-load across the multiple worker threads—*cyclicv1*, *cyclicv2* and *blockv1* (see subsection 1.3).

Upon receiving the required parameters over the socket connection, the *ClientHandlerThread* generates three matrices of the given size: two with random entries which will be multiplied together, and one empty array to hold the result, which will be filled-in by the worker threads.

The *ClientHandlerThread* will create and start worker threads—as many as are allocated to the server via its second parameter. Each *WorkerThread* will receive two parameters: a *worker ID* and a reference to the *ClientHandlerThread*, so that each worker can access its data.

Finally, the *ClientHandlerThread* will wait until all workers have completed their work allocation, and send the *Result*—including a completion code and calculated matrix—to the client, over the network. Note that the calculated matrix is only returned if requested in the call to 'calculate' on the client side.

WorkerThread

When started, each *WorkerThread* will use its reference to the parent *ClientHandlerThread* to obtain the specified type of workload partitioning to implement. Each worker thread will then determine the elements of the result matrix which it needs to fill-in, and compute each element using its *calculateElement* method, and the elements from matrices A and B which are obtained via the reference to the parent *ClientHandlerThread*. The worker will then store the results in the result matrix, again via its reference to its parent *ClientHandlerThread*.

2.2 Client Classes (Client.java file)

MatrixClient

MatrixClient is an executable class with a main method, and a method called *calculate*, which takes an *Operation* and a *matrix size.* When *calculate* is called, it will create a ClientThread with the given parameters, and return the *Result* of the matrix multiplication that the *ClientThread* recieves from the server's *ClientHandlerThread* over the network.

ClientThread

When a *ClientThread* is created, it will establish a connection to the server, and write the matrix size and operation type over the socket. It will wait for the server to perform the calculation, and then read the *Result* that the server delivers over the socket (if requested).

2.3 Other Classes

For the purposes of analysing performance, an executable *Driver* class was created to simulate multiple client machines. This class has a main method, and two other methods which are called from the main method. These methods were used for performance measurements. They each create *MatrixClient* objects and call their *calculate* method on separate threads.

A HelperMethods class was created, with one method for printing matrices.

The *Result* object is the object which is sent by the *ClientHandlerThread* to the *ClientThread* over the socket. It stores a matrix *answer* and an *errorCode.*

Finally, an emum called *Operation* is used to refer to the three different methods for partitioning the workload.

3 Performance Analysis

3.1 Initial Performance Tests

To begin analysing the performance of the application, the three different methods for partitioning work (cyclicv1, cyclicv2, blockv1) were compared across different numbers of worker threads (1, 2, 4, 8, 16) and different matrix sizes (100, 200, 300, 400, 500) through the use of the *basicTest* method in the *Driver* class. Results are shown in Figures 3, 4 and 5.

Note that in these tests, the completion time is an average of 10 iterations of the same calculation performed on the server. One client was used, and both the client and server programs were run on a 1.7GHz Intel Core i7 CPU with 4 logical cores.

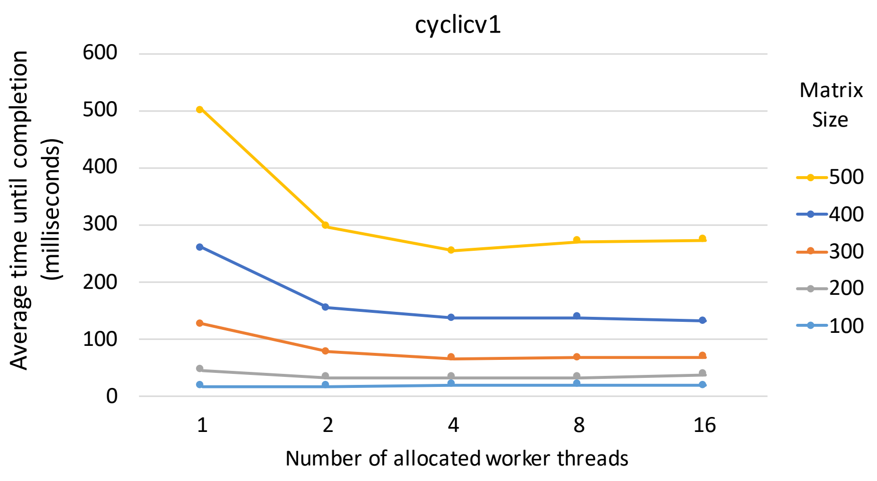

Figure 3: Testing the performance of the cyclicv1 partitioning method with one client.

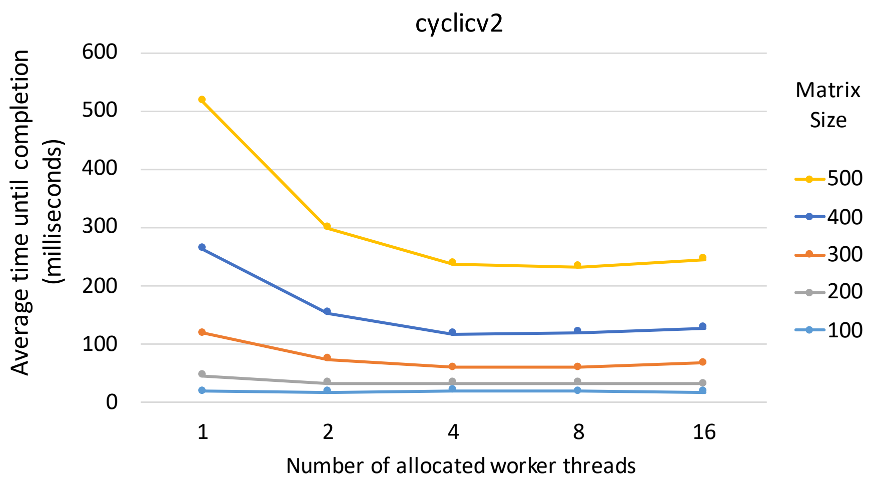

Figure 4: Testing the performance of the cyclicv2 partitioning method with one client.

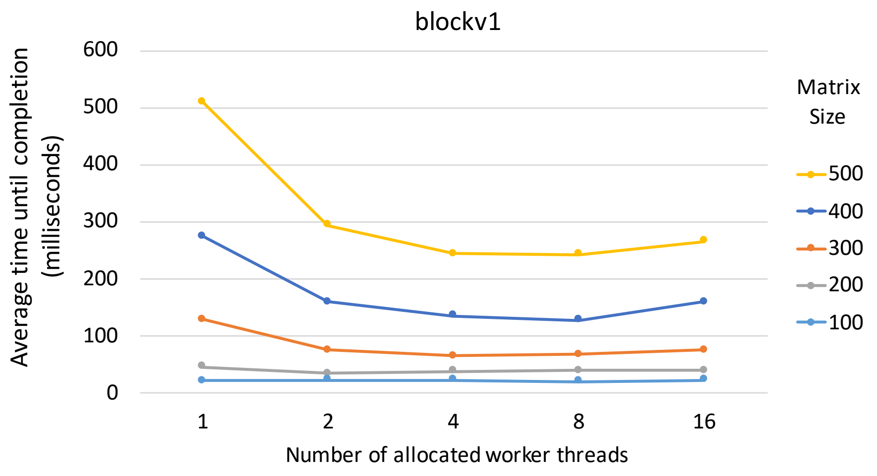

Figure 5: Testing the performance of the blockv1 partitioning method with one client.

Since each partitioning method aims to distribute work evenly amongst the workers, it is unsurprising that in general, the performance when using each of these methods is similar.

These initial plots also show that for the matrix sizes used for testing, there is a significant speed increase when moving from 1 to 2 worker threads. There is a smaller increase when moving from 2 to 4 worker threads, and no significant increase in performance with additional workers. One might hypothesise from these figures that the performance gained by using additional threads will increase further as the matrix size increases.

3.2 Multiple Clients

The next step was to test performance in the case where there are multiple clients connecting to the server at once. Blocking on matrix size, partitioning method and available worker threads per calculation, the *Driver* class was used to test multiple connected clients on one machine. Each ‘client’ performed one calculation. The average client wait time was plotted against the number of connected clients, across a differing number of worker threads per calculation, in Figure 6.

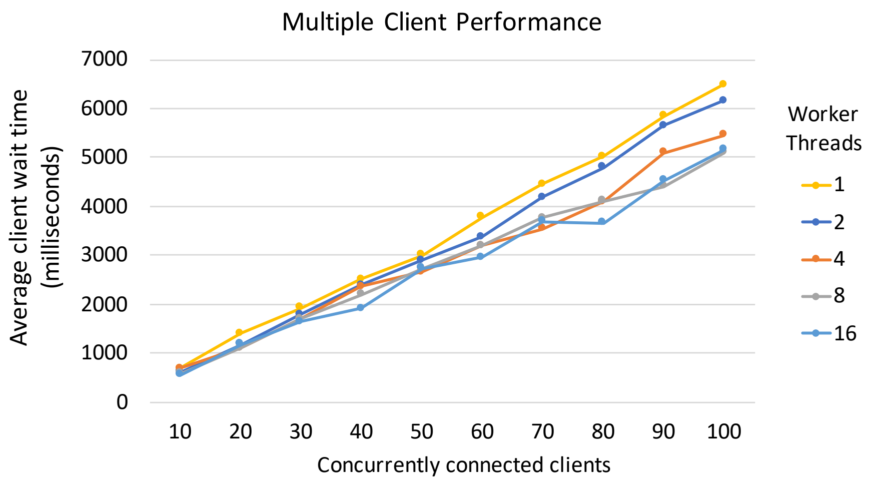

Figure 6: Testing the performance of multiple clients connected to the server simultaneously. Each client connected to perform one calculation with size 300 matrices, using the *cyclicv1* partitioning method. The test was repeated with a differing number of worker threads allocated to each computation.

From the plot, one can see that for size-300 matrix computations completed using cyclicv1 partitioning, the average wait time for clients increases *linearly* with the number of clients connected simultaneously.

It is worth noting that 100 appeared to be approximately the maximum number of concurrently connected clients that the server could handle. Since the server treats each computation as a ‘client’, this means that the server can process around 100 computations at a time.