MCOMD3HPC – High Performance Computing Assignment 2

Nathan Shuttleworth

Due 10/1/22

Contents

[Task 1: Simple Implementation – Gold Standard 3](#_Toc92701463)

[Task 2: Three Models 5](#_Toc92701464)

[Master/Slave 5](#_Toc92701465)

[Symmetric 7](#_Toc92701466)

[Loose Coupling 10](#_Toc92701467)

[Task 3: Testing & Logging 12](#_Toc92701468)

[Gold Standard 12](#_Toc92701469)

[Fixed Threads/Change matrix Size 12](#_Toc92701470)

[Master/Slave 13](#_Toc92701471)

[Fixed Threads/Change matrix Size 13](#_Toc92701472)

[Fixed matrix size/Change number of threads 14](#_Toc92701473)

[Symmetric 14](#_Toc92701474)

[Fixed Threads/Change matrix Size 14](#_Toc92701475)

[Fixed matrix size/Change number of threads 15](#_Toc92701476)

[Loose Coupling 15](#_Toc92701477)

[Fixed Threads/Change matrix Size 15](#_Toc92701478)

[Fixed matrix size/Change number of threads 16](#_Toc92701479)

[All Fixed thread solutions 17](#_Toc92701480)

[All fixed matrix size solutions 17](#_Toc92701481)

[Task 4: Evaluation & Conclusion 18](#_Toc92701482)

[General Evaluation 18](#_Toc92701483)

[Three models 18](#_Toc92701484)

[Master/Slave 18](#_Toc92701485)

[Symmetric 18](#_Toc92701486)

[Loose Coupling 18](#_Toc92701487)

[Recommended Model 18](#_Toc92701488)

[References 20](#_Toc92701489)

# Task 1: Simple Implementation – Gold Standard

This implementation works using 3 methods, to streamline the running and development. The size of the matrices are determined with this variable:



Figure 1- Matrix Size variable

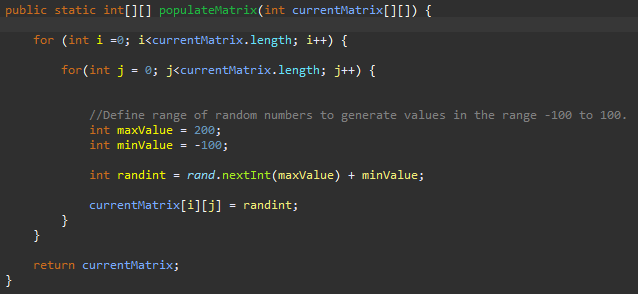
The Main method initializes 3 2-D arrays, two initial ones and the other which will hold the result of the multiplication. In turn, the initial arrays are passed to the populateMatrix method: 

Figure 2- Matrix population method

The values that are generated are limited in the range -100 <= n <= 100 to ensure that overflow errors do not occur when the matrices are multiplied.

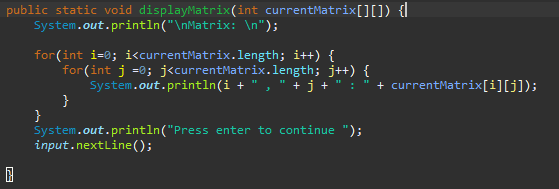
The populated arrays are then passed to the displayMatrix method: 

Figure 3- Matrix display method

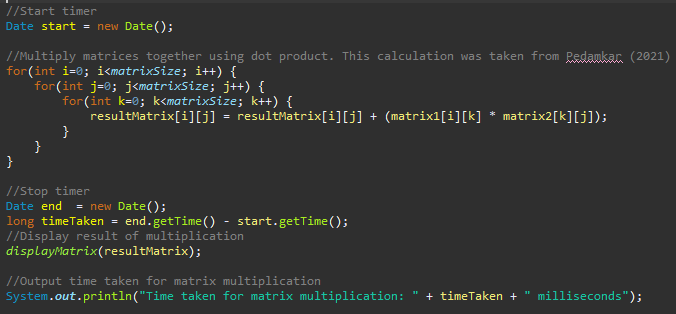
This outputs each value to the console in turn on one line each. A scanner is used to wait for user input after outputting the array to help the user keep track. Next, in the main method, the dot product operation is performed on the arrays, and the results stored in resultMatrix: 

Figure 4- Timing and dot product code

This is achieved using 3 nested for loops. This solution was adapted from Pedamkar (2021). The resultMatrix array is then passed to the displayMatrix array to output all its values in the same way as the other two. The time is tracked by getting the date and time before and after the operation, and finding the difference.

# Task 2: Three Models

All three of these models feature elements from the Gold Standard, the generateMatrix and displayMatrix methods are carried over in the main classes.

## Master/Slave

The three classes for this implementation are Main, Master Thread and Slave. The main class starts the master thread, which is passed the three matrices as parameters.

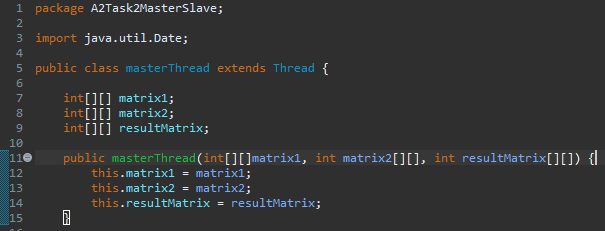


Figure 5- Values passed to master class

Text

Description automatically generated

Figure 6- Master thread main content - timer, thread creation and thread starting

Text

Description automatically generated

Figure 7- Handling of values passed to slave, and calculation

With the slave threads, there is only one line of code for the calculation, the rest is just handling how data is passed to it. This keeps the threads lightweight, so that they run faster and more can be run at a time.

When I started running the tests for this model, I noticed that all the results were very similar. I realized that the program was not actually multi-threaded, since I had been calling the thread.run() rather than thread.start(). .start() creates a new thread and then runs the class, as opposed to .run() simply running the thread like it is another method, there was no concurrency. I fixed this using the article by Samarpit (2021) for reference.

## Symmetric

The symmetric implementation works similarly to the master/slave but uses a ‘job’ class to pass data to the threads, and the master thread is implemented into the main class. As well as helping with passing data, the job class also helps to simulate the process queue that is found in symmetric HPC solutions. Text

Description automatically generated

Figure - Symmetric Job class

The job class features a Boolean variable that determines whether it has been completed. This is used mainly in the threads:

Text

Description automatically generated

Figure - Symmetric thread class

The main method features the displayMatrix and populateMatrix methods, but is more complex since it has to create jobs and threads. A loop break is used to ensure the correct number of jobs is made, if the three nested loops run all the way to the end they try to make more jobs than there are needed/able to be made, causing an error.

Text

Description automatically generated

Figure - Main class Job creation

Text

Description automatically generated

Figure - Main class thread creation and starting, and timing

The main method features a while loop that ensures threads will keep being created, assigned job and started until all jobs are marked as complete. Its running of threads works in much the same way as the master/slave, the job and thread is incremented each time until the number of threads is reached, and then it starts re-using finished, dead threads. Since the threads themselves run so quickly, there is no anti-lock system in place.

This implementation does have an error, it would give out of memory crashes when trying to run at a matrix size of 9000x9000.

## Loose Coupling

The Loose coupling components were adapted from work by Mishra (2020). This primarily includes the interface, threadInterface:

Text

Description automatically generated

Figure 12- threadInterface

Aside from implementing the interface, the thread class is fairly similar to the other implementations: Text

Description automatically generated

Figure 13- Loose coupling thread class

Text

Description automatically generated

Figure 14- Loose coupling main thread creation and running

The main method features the familiar three nested for loops and thread creation, starting and implementation. I did not have much time to make this implementation as well as I would have liked, I would have preferred to make the interface a more key part of this and use it to have more intercommunication between threads.

# Task 3: Testing & Logging

When testing, I will use same computer, without any other programs running to allow the program to have as much computing power as possible. Before running the proper tests, I did pre-testing to determine the matrix sizes and number of threads I should use. When testing with different numbers of threads, I used 200x200. Although this is quite small, some of my implementations were surprisingly inefficient, and it would not have been feasible to use larger matrices.

Some graphs are logarithmic to better display the data, and I was unable to run all tests to the highest matrix sizes due to the high time taken or crashes. All of the test data is viewable in the excel document.

## Gold Standard

### Fixed Threads/Change matrix Size

Only matrix size can be changed, non-threaded solution.

Figure 15- Graph showing gold standard average test times against matrix size

I only ran to 7000x7000 as this one took 2 hours, higher would have taken at least 5.

## Master/Slave

### Fixed Threads/Change matrix Size

Figure 16- Graph showing Master/slave average test time against matrix size

These tests were only run up to matrix size 1000x1000, as this one took over 9 hours, I ran it overnight.

### Fixed matrix size/Change number of threads

Figure 17- Graph showing average test time against varying number of threads

## Symmetric

### Fixed Threads/Change matrix Size

Figure 18- Graph showing Symmetric average test time against varying matrix size

The program crashed when I tried to run with matrix size 9000x9000 due to lack of memory.

### Fixed matrix size/Change number of threads

Figure - Graph showing average time against number of threads for Symmetric implementation

## Loose Coupling

### Fixed Threads/Change matrix Size

Figure 20- Graph showing average test time against varying matrix size

The 500x500 size took 81 minutes, I did not have enough time to run any more tests as they would have taken even longer.

### Fixed matrix size/Change number of threads

Figure 21- Graph showing average test time against varying thread count

## All Fixed thread solutions

Figure 22- Graph showing all average test times against varying matrix size

## All fixed matrix size solutions

Figure 23- Graph showing all average test times against varying number of threads

# Task 4: Evaluation & Conclusion

## General Evaluation

Something that could have been done better is with testing, although I used what should be the same computer (University library PC), I was not able to use the exact same machine and so this did result in different results coming in different tests.

## Three models

### Master/Slave

Whilst my master/slave implementation is functionally complete, as shown in the testing, it is very slow and inefficient. I believe a key reason for this is how the entire resultMatrix is passed to each thread, as opposed to simply returning a variable that is then assigned to the matrix outside of the thread.

On the positive side, I did not experience any crashes which is surprising given that there is a single point of failure, the master thread.

### Symmetric

Out of all of my threaded solutions, symmetric was my most efficient. However, it did begin to become exponentially slower once I started using larger matrix sizes, and once I started running it at 9000x9000, it kept crashing. I would consider it to be the best solution for running small -mid matrix sizes, up to 7000x7000.

A way that my symmetric implementation could be improved is more control for if the threads were to lock, since resources are shared more directly with this model.

### Loose Coupling

The loose coupling implementation was not the most inefficient, however I believe that the coupling and interface could have been implemented better. Coding in a loosely coupled manner is a new concept to me, and so with more practice I should be able to use it better.

In terms of the solution itself, it was fairly efficient with small matrix sizes, however it quickly became very slow. I also noticed that it began to become less efficient with more threads, this means that with this investigation threads were taking longer to create than to run, so it was a waste of resources to have more.

## Recommended Model

In conclusion, the best model for an HPC solution provider to use is Symmetric, as it was by far the fastest compared to the other models. However, it must be ensured that anti dead locking measures are taking, and that the data is passed efficiently so it is not being accessed by the same thing at the same time. This is also beneficial compared to using an asymmetrical cluster, as there is easier scalability. A new node/ processing element can be added easily. As described by Castro (2018) : ‘There is no need to take the entire cluster down to add a new node.’

Master/Slave is not suitable due to having a single point of failure, if the head node/ master is to fial, the whole system will be unusable. Loose coupling is also less suitable as ‘dedicated slices of RAM, (are) less efficient’ (Sahota, 2021). It is also harder to detect CPU failure because each process has its own section of RAM.

# References

Pedamkar, Priya (2021) *Matrix Multiplication in Java.* Available at:<https://www.educba.com/matrix-multiplication-in-java/> (Accessed on 1/12/21)

Samarpit (2021) *Java Thread Tutorial: Creating Threads and Multithreading in Java*. Available at: <https://www.edureka.co/blog/java-thread/> (Accessed on 23/12/21)

Mishra, Neeraj (2020) *Loose Coupling and Tight Coupling in Java.* Available at: <https://www.thejavaprogrammer.com/coupling-in-java/> (Accessed on 5/1/21)

Castro, Kristi (2018) *Asymmetric and Symmetric Clustering System.* Available at: <https://www.tutorialspoint.com/Asymmetric-and-Symmetric-Clustering-System> (Accessed 8/1/21)

Sahota, Vijay (2021) *MPI.* Available at: <https://learn.canterbury.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_17358_1&content_id=_3074736_1&mode=reset> (Accessed 9/1/21)