Threaded Matrix Report

Audrey Smith

Assignment 2

High Performance Computing

Canterbury Christ Church University

Table of Contents

[Task 1: The Gold Standard 3](#_Toc93577650)

[Task 2: The Three Models 5](#_Toc93577651)

[Task 3: Testing 8](#_Toc93577652)

[Task 4: Conclusion 10](#_Toc93577653)

[References 11](#_Toc93577654)

# Task 1: The Gold Standard

The multiplication of two matrices is a relatively straightforward process that is much simpler to write an implementation for than to do the work by hand. Two matrices are initialised, and then using a random number generator and nested for loops, the program iterates through each position in the matrix and gives it a randomly fabricated value between zero and a specified maximum integer. If the maximum range of the random number generator is left open-ended, the multiplication process will cause the values of the resulting matrix to go out of integer bounds and appear incorrect. The size of each matrix can be changed in the initialisation code (see Figure 1).

In the third section, nested for loops use temporary integers ‘i’ and ‘j’ to loop through the newly initialised ‘resultMatrix’. The values placed in each position are the output of the matrix multiplication process, which uses both randomly generated matrices from the previous sections (see Figure 2). Finally, there are two options for the formatting of the console output of each matrix. By default, the matrices are separated by row and printed over a single line; however, when replaced with the commented code the matrices are then printed with each row on its own line (see Figure 3).

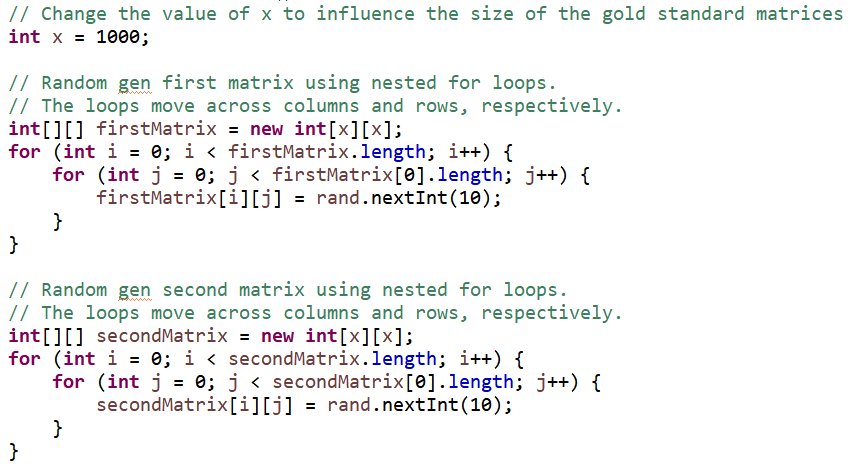


Figure 1

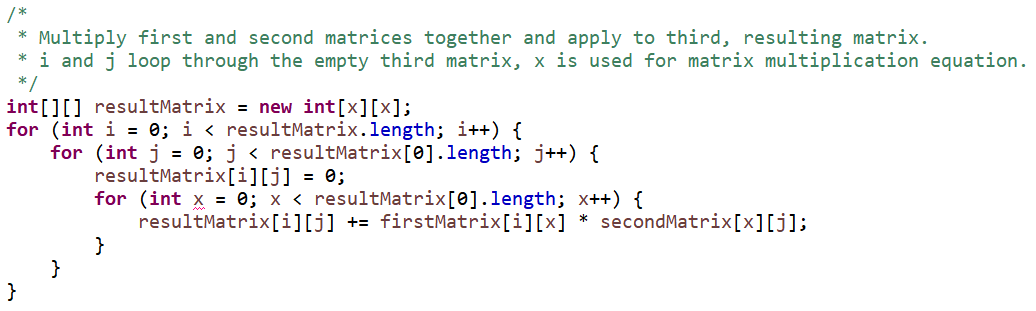


Figure 2

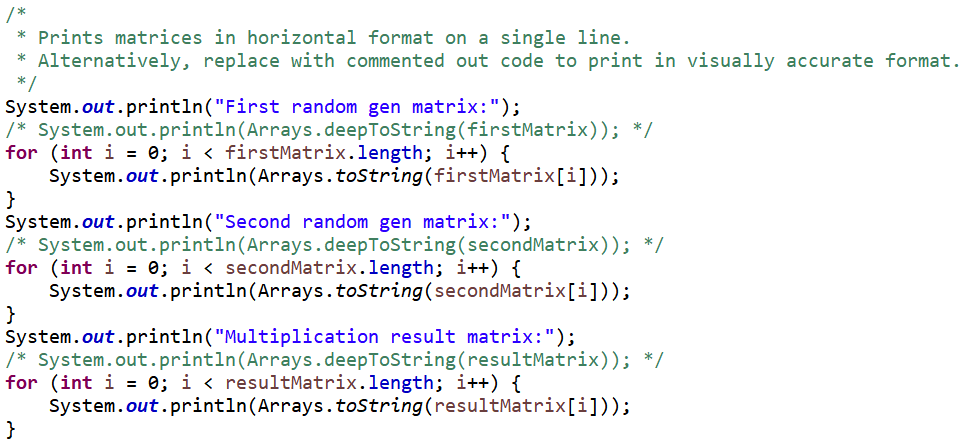


Figure 3

# Task 2: The Three Models

Three lines at the end of the Main class initiate the three models. In the Master-Slave model, the Master is responsible for assigning tasks to the Worker threads, which each handle their task and pass the result back to the Master to put together the results (Goclimb, 2019). Following the process of matrix multiplication, each thread calculates one cell of the resulting matrix using a row of the first matrix and a column of the second matrix, these meeting at the point of the cell which the thread was assigned. The Master thread class is labelled ‘MSMaster’. This immediately creates an integer variable ‘threadCount’, which throughout the project will retain the value of the length of the matrix multiplied by the height of the matrix. For this example, the length, or the value of x, is 3. This would give threadCount a value of 9. Workers are initialised using this value, and then nested for loops paired with a counter variable decide which portions of the arrays go to each Worker thread.

In the Master-Slave model, the Worker threads are given a row from firstMatrix and a column from secondMatrix, as well as the value of x. The Worker iterates through these two arrays and conducts the matrix multiplication process for a single cell. A method ‘getCell()’ allows MSMaster to retrieve the result from each Worker thread, finally creating a resulting matrix and subtracting it from the gold standard result to verify that the process worked successfully. The other two models are quite similar to the first, though also different in a number of ways.

Both the Loose Coupling and Symmetric models work with a Job class alongside the Master and Worker classes. In the Loose Coupling case, the Master class, ‘LCMaster’, differs from the first in that it must initialise Workers *and* Jobs, passing the array sections (hArray and vArray) to the Jobs because the Loose Coupling model requires each thread to have its own set of limited information. The Worker thread takes the information and instructions from the Job it was assigned and uses it to complete the task to its extent, including the subtraction verification with the gold standard. The Master class takes the result from each thread and creates an array to show in the console.

The Job class exists in the Symmetric model as well, though it acts as more of a job pool and less as a processor of extra information. The Symmetric model is somewhat a mix of Master-Slave and Loose Coupling. It takes from Master-Slave the idea that the threads all take from the same main memory, and it takes from Loose Coupling the idea that threads are not dependent on a Master thread to tell it what to do (GeeksForGeeks, 2019). Worker threads are assigned jobs based on the order they were initialised and conduct whatever is contained within that job.

Throughout the code in every class, there are commented lines that print to console whenever an event occurs, such as a job being created, an array being filled, or a Worker thread finishing its process. These can be used to check that the code runs smoothly and that no errors exist. If errors do arise, these lines will fail or show the error in the console, which will make the fixing process easier. Alongside this, there is a try-catch statement in each class that waits until each Worker thread finishes and throws an error if they fail to do so. I found these safety measures to be the most important because they can typically show exactly what the issue is and lead the way to resolving it. In the below Figures 4-6, each set of classes is shown. Figure 4 is the Master-Slave set, Figure 5 is the Loose Coupling set, and Figure 6 is the Symmetric set. By observing these figures closely, one might notice that these models are not entirely different from each other; they are different approaches to the same issue. There are many elements that two or all of them share, such as job pools, initialisation and error catching of Worker threads, the process used to split the initial arrays up into pieces for the Workers, etc. The main difference between the models is that these pieces are tweaked slightly and conducted by different classes at varying times. It can be seen in Figure 4 that the Master-Slave Worker thread has “Thread.sleep(1000);” at line 20. This was added out of precaution, but upon seeing that the model works, this was removed to save time in testing.

Text

Description automatically generated

Figure 4

Text

Description automatically generated with medium confidence

Figure 5

Text

Description automatically generated

Figure 6

# Task 3: Testing

To test the three models and find the best solution for each, console messages were placed throughout the code to check that each section functioned properly. These have been commented out upon finishing the models to keep the console from getting cluttered, but some examples can be seen in the figures below. The most useful of these console print lines were those which displayed data that would not be used in the result. For example, Figure 8 shows the line, “System.out.println(Arrays.toString(hArray));” and another below it that was similar. These types of lines revealed to me issues in the assigning and interpretation of data somewhere in the program, which greatly shortened time spent fixing bugs.

Text

Description automatically generated

Figure 7



Figure 8

While debugging and testing each model, I have manually recorded successes and failures in an Excel spreadsheet that automatically shows me the ratio for each. Code could be added to automate this step; however, I began doing this early in the project so it felt appropriate to keep it that way. If the program runs and outputs a difference of zero, that is considered a success and is marked ‘1’ in the spreadsheet. All other cases are marked ‘0’ in the spreadsheet. This can be seen in Figure 9 below. Note the bottom of the tables – the percentage is calculated automatically by a function input into Excel that takes the average of the values in the respective column. This makes it helpful by reducing time spent testing and taking recordings.

Graphical user interface, application, table

Description automatically generated

Figure 9

Each model was tested using multiple values of x, ranging from 3 to 1000. Once the debug process was concluded, the models passed 100% of the time. The models run incredibly quickly, even at higher values of x. As mentioned previously, the Master-Slave Worker thread had a sleep function that was added out of precaution. This has since been removed due to it causing incredibly long waiting times for the program to run, especially when the value of x is in the hundreds or thousands. Figure 10 below shows multiple timed runs of each model. There are three runs each with a 100x100 matrix, and three runs each with a 1000x1000 matrix. There is an obvious winner in the set for the larger matrix, and another in the set for the smaller matrix. In the 100x100 runs, the Symmetric model finished the quickest, and the Master-Slave was the slowest. This is the opposite for the 1000x100 runs, in that the Master-Slave had the shortest time, and the Symmetric had the longest.

Table

Description automatically generated

Figure 10

# Task 4: Conclusion

Based on the findings of this study, a conclusion can be made about the three models presented, as well as which model is most suitable for an HPC environment. Starting off with time taken to complete the given tasks, it is apparent that the Symmetric model works the best in smaller scale tasks and takes a great amount of time in a larger scale. The model is intended to resemble multiple processors pulling data from one large, common memory and working independently from each other. Symmetric uses only the resources it needs but given that it takes quite a long time to finish the more complex tasks, it may not be the most reliable when it comes to a HPC environment, where loads can have a wide range of complexity.

The middle-ground of the models in terms of time taken to complete the tasks is Loose Coupling. This model, like the Symmetric, has processors that work independently. These, however, pull from their own exclusive memory instead of one common one. A processor is assigned a job, completes the job fully, and then moves on. The main issue with this model is that a failing processor is not easy to detect. Bringing in the timed runs, the Loose Coupling model performed fairly well in the 100x100 matrix set, but it took almost as long as the Symmetric model in the 1000x1000 set. It is faster than the Symmetric model and not very prone to major failure, making it a more viable option for an HPC environment.

Finally, is the Master-Slave model, which consists of one main thread managing the others, along with the memory. It passes jobs and data to the Worker threads and is a lot simpler in idea than the other models. One risk of using this model for threads is that if the Master thread goes down, all the rest do as well. Though this model performed slightly worse than the others in the 100x100 timed sets, it did surprisingly well at the larger scale, 1000x1000 matrix sets, completing the tasks in less than half a minute where the others took multiple minutes. Because this model performed at such a high level, it would be the best option for an HPC environment because it is faster and errors are easier to detect than in the Loose Coupling model.

# References

Goclimb (2019). *Detailed Explanation of Master-Worker Mode in Java Multithread Programming*. Available at: <https://programmer.group/detailed-explanation-of-master-worker-mode-in-java-multithread-programming.html> (Accessed 8 January 2022).

GeeksForGeeks (2019). *Typical Multiprocessing Configuration*. Available at: <https://www.geeksforgeeks.org/typical-multiprocessing-configuration/> (Accessed 14 January 2022).