**Name:** Ruben Vasconcelos

**Student Number:** 16212630

**Module Code:** CA670

**Assignment Title:** Java Threads

**Submission Date:** 10am on Monday 20th March 2017.

**Plagiarism Declaration:**

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I/We have read and understood the Assignment Regulations. I/We have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

Ruben Vasconcelos

19-March-2017

**Simple Approach:**

**Mechanism used:** Executor Interface and Thread Pool.

**Algorithm:** For my simple approach, I decided to use a very basic multithreaded version of the algorithm often used to multiply matrices in a non-concurrent environment, where the result at . Where the number of columns of the matrix “mA” must be equal to the number of rows in matrix “mB”.

|  |  |
| --- | --- |
| **Single-threaded** | **Multi-threaded** |
|  |  |

The multithreaded version creates a new task for each row in “mA”, the number of tasks executed at the same time is equal to 4 times the number of cores available.



**Advanced Approach:**

**Mechanism used:** Fork/Join Framework.

**Algorithm:** I decided to implement a version of the Strassen algorithm very similar to the one shown on the wiki <https://en.wikipedia.org/wiki/Strassen_algorithm>.

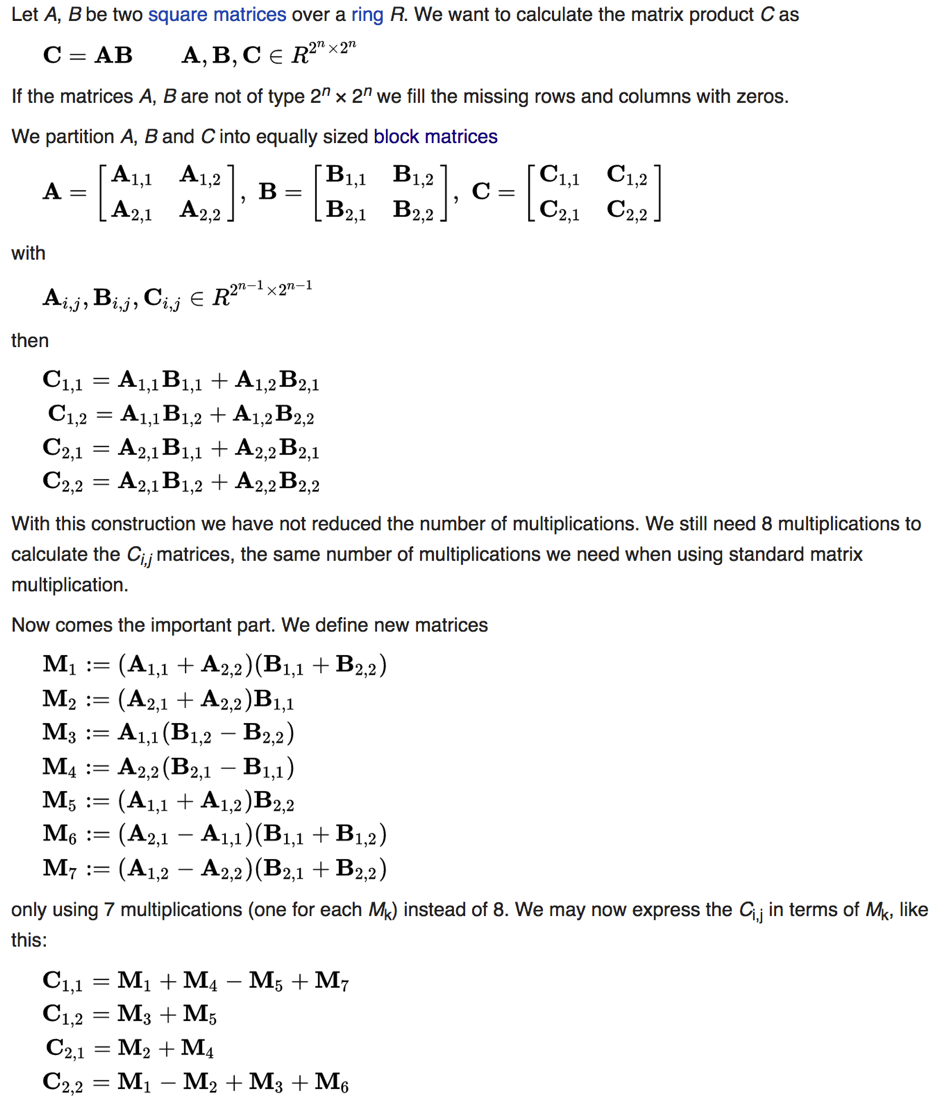


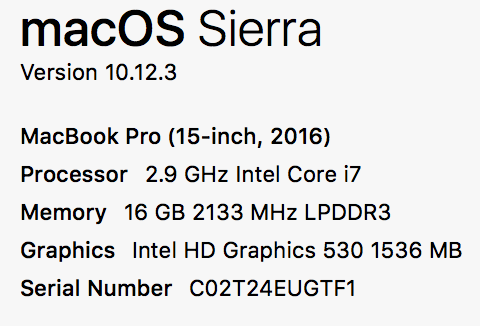
Figure 1 src: https://en.wikipedia.org/wiki/Strassen\_algorithm

The algorithm only works with square matrices, meaning that extra operations are required in order to resize and copy data for all other cases. For smaller matrices, you won’t notice much of a speed, actually for really small matrices Strassen will perform worse than the normal algorithm.

But with matrices of 1000 and over I started noticing an improvement in the test results.

**Results:**

**System:** 8 – cores (physical and virtual). 16 GB ram. I also decided to read in the matrix input from 2 files. This means that some time at the start of each test run is spent reading in data.



**Docker:**

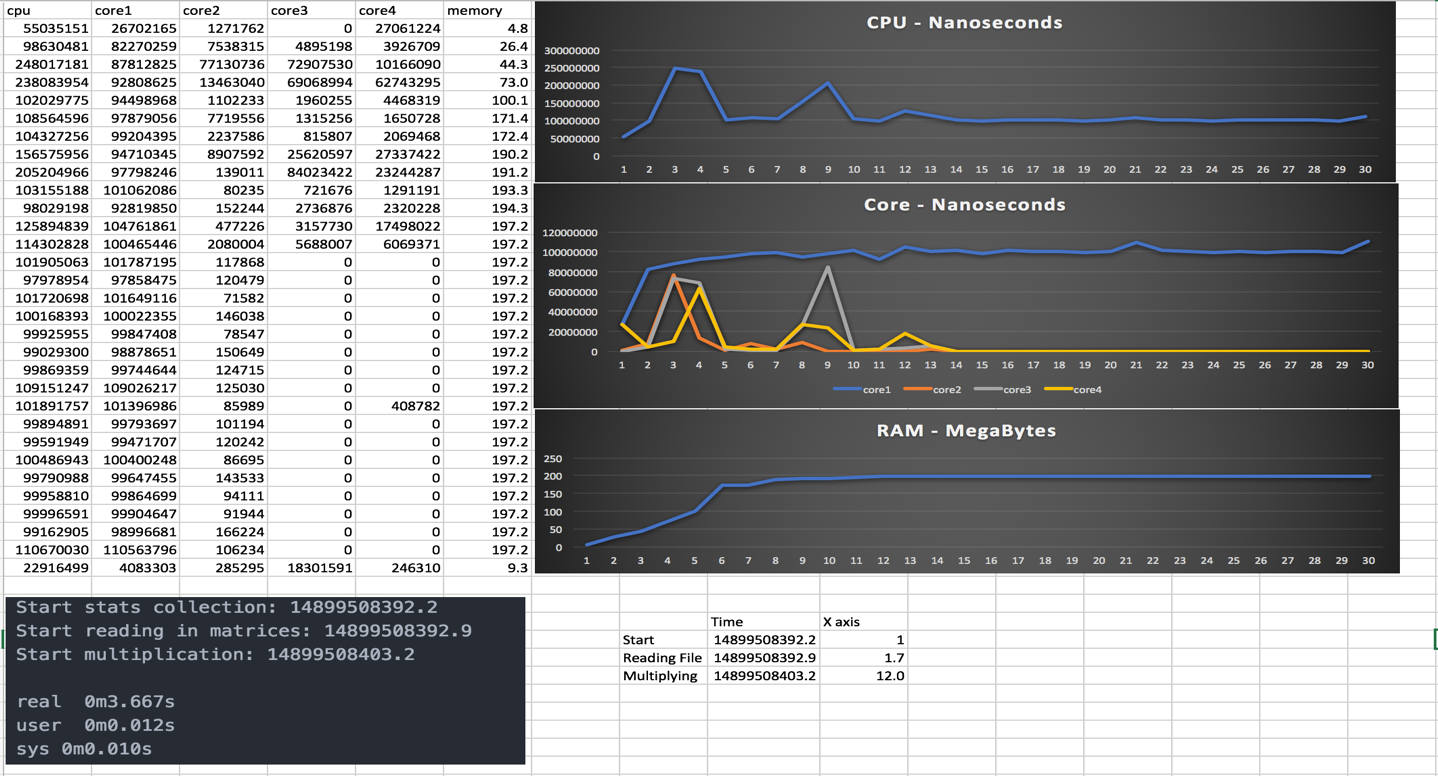
I decided to use Docker for running the different algorithms, because it allows me to:

1. Specify the number of cores available and memory available during each run.
2. Collect RAM, CPU and Individual core usage stats. Docker keeps real-time usage info in the file system, I used a modified version of the following script to collect the stats <https://github.com/vasconr2/py_docker_stats_collector>. The script was created by myself a while back as a proof of concept for a different project.
3. The fact that there are no other applications running inside the containers means that the metrics represent almost only the application’s usage, there is some minimal overhead, mainly during the container start up time which I will be able to show in the charts bellow.

**Test Run 1:** Single threaded algorithm

Matrices size = 1024\*1024

Running on 4 cores with a limit of 2GB ram



I decided to Run the single threaded algorithm first as a baseline comparison. In the two first graphs, above, along the Y axis is the nanoseconds spend by the CPU working on the application. Along the X axis is the time gone by (x axis value / 10 = seconds).

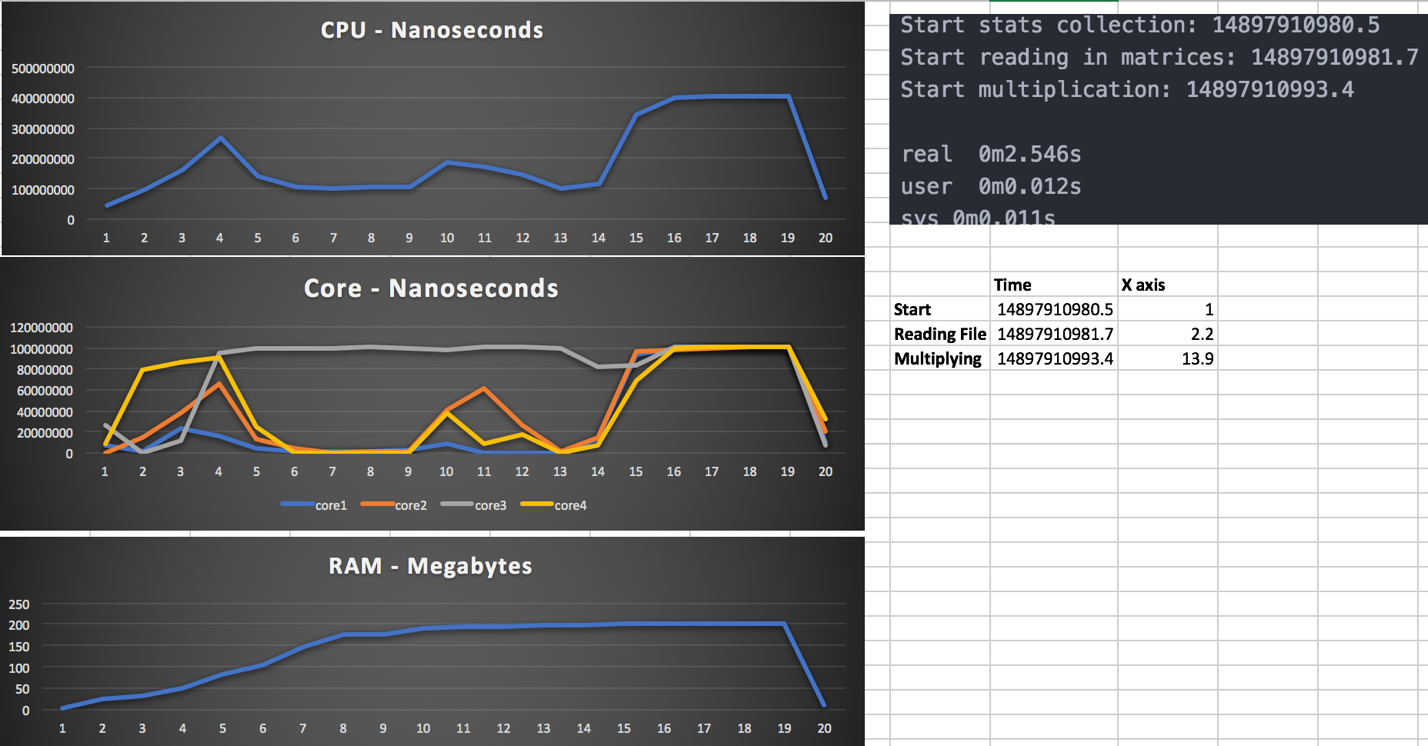
**Important:** You’re probably looking at the Core chart and wondering why did all 4 cores seem active if it was a single threaded application, well this is because of the overhead I mentioned before, Docker itself does some work when initializing/shutting down the container.

The first 1.2 seconds are spend reading in data from the disk and the matrix multiplication only starts around value number 12 in the X axis, if you notice the blue thread is active from start to finish, this is the thread executing the application.

**Test Run 2:** Simple multithreaded algorithm

Matrices size = 1024\*1024

Running on 4 cores - 2GB ram

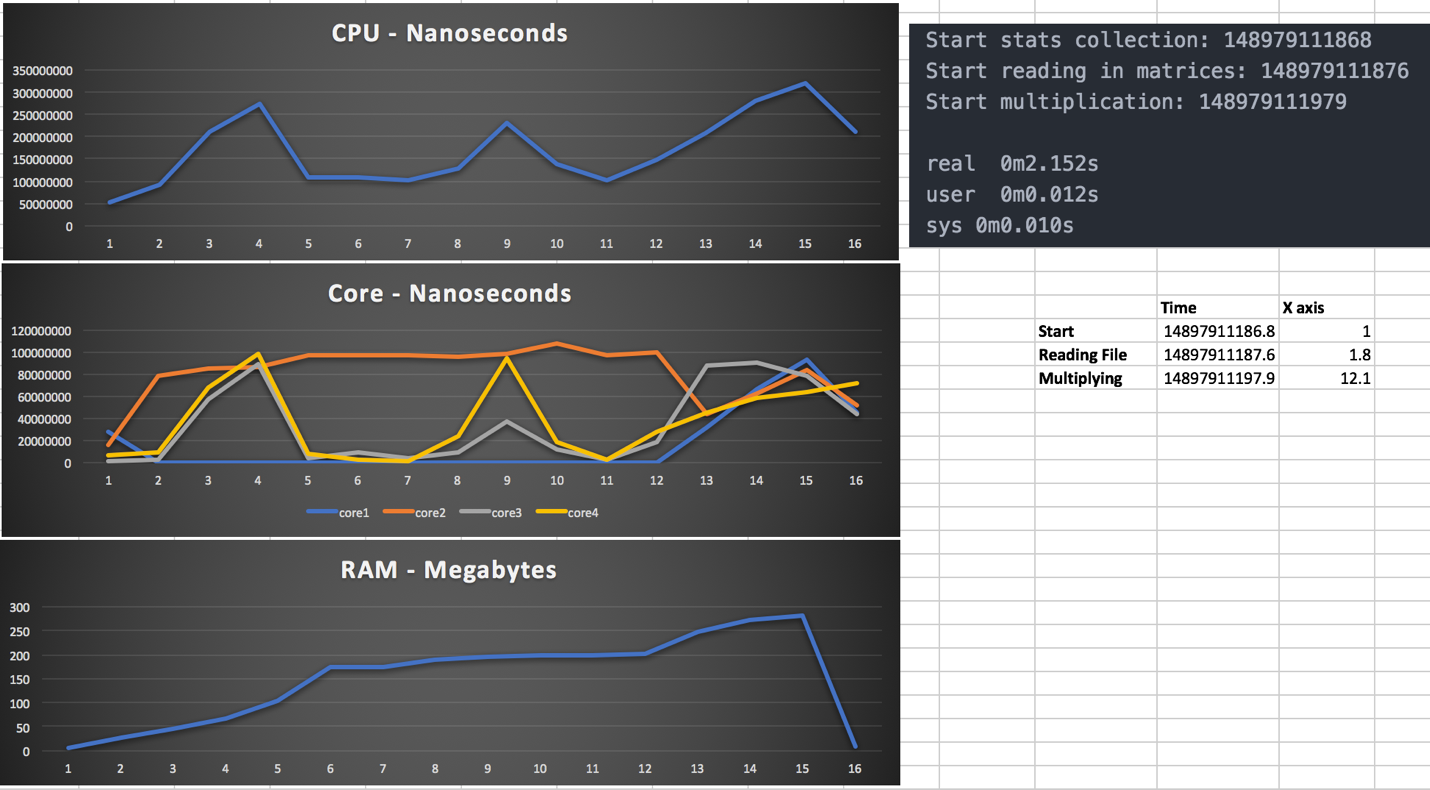


Just like before initially all cores seem to be active with one doing the most work, which I believe to be the one reading in the matrix values which is still single threaded. Unlike before though the Multiplication is executed across all 4 cores, starting around second 13.9.

**Test Run 3:** Advanced multithreaded algorithm (Strassen)

Matrices size = 1024\*1024

Running on 4 cores - 2GB ram



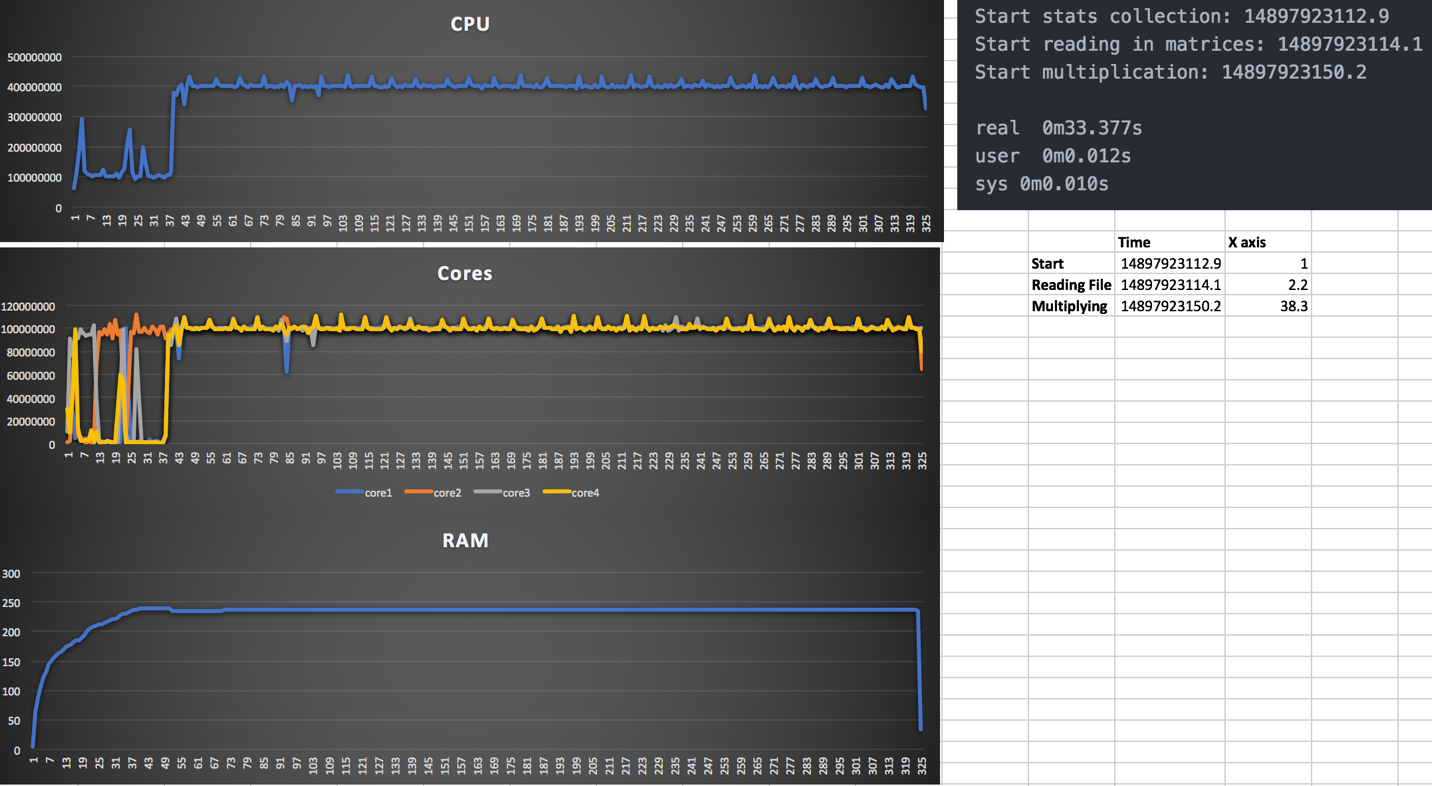
Until now there were no noticeable spikes in the memory usage, but this time nearly an extra 100 megabytes where used, and the spike starts at around 12 (1.2 seconds) which matches up with the start of the multiplication process at 12.1

I guess this is related to the fact that a lot more threads are created during the fork/join approach, and each thread gets their own stack frame in addition to all of the extra temporary arrays used for partitioning data.

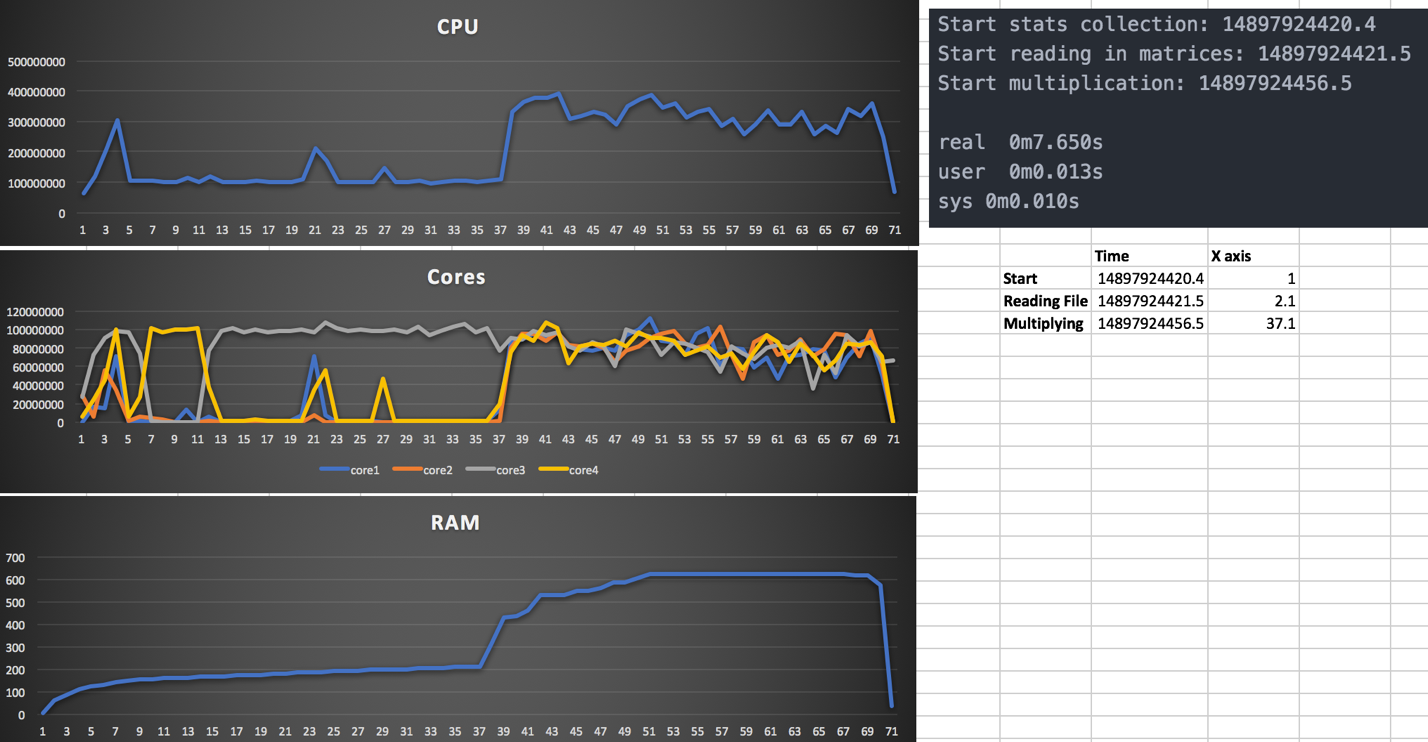
Time wise both the simple and more efficient approach took about the same time to multiply two 1024\*1024 matrices. The efficient algorithm performed slightly better but not enough for the user to notice any difference.

**Test Run 4:** Matrices size = 2048\*2048 **-** Running on 4 cores - 2GB ram

**Simple Approach:**



**Efficient Approach:**

****

This time the results are totally different, the simple approach took a lot longer, while in contrast the more efficient program was much quick but used a lot more memory during its execution. The pattern of the graphs also changed quite a lot, while the simple approach executes the exact same task many time, it’s quite obvious that in the efficient approach the work being done by each core varies a lot more.

**Test Run 5:** Matrices size = 512\*512 Running on 1 core - 2GB ram

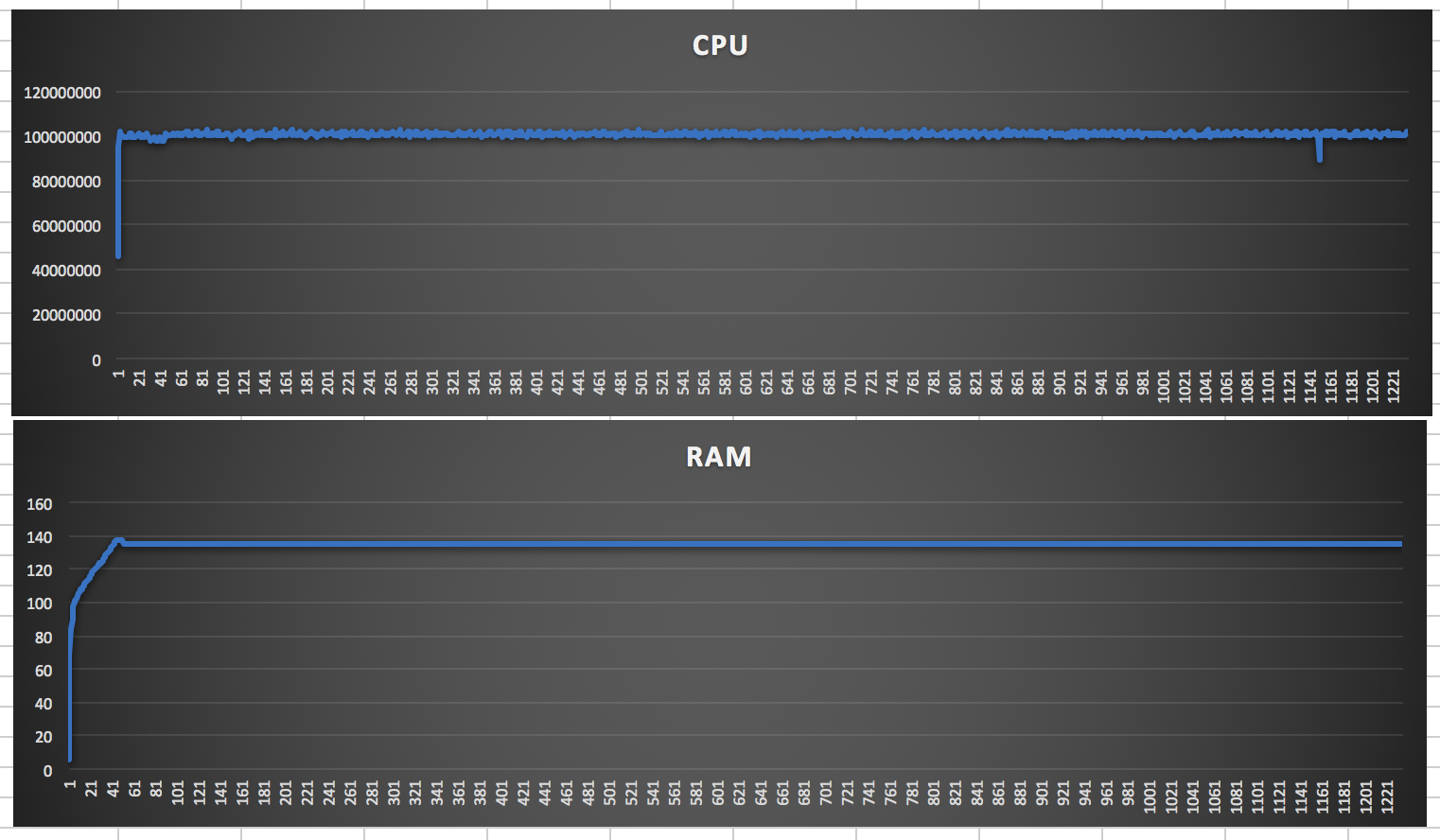
|  |  |
| --- | --- |
| **Simple Approach:** | **Efficient Approach:** |
| **Ram: 30mb** | **Ram: 35mb** |
|  |  |

This time around the results were very similar for both approaches, the quickest algorithm also varied quite a bit.

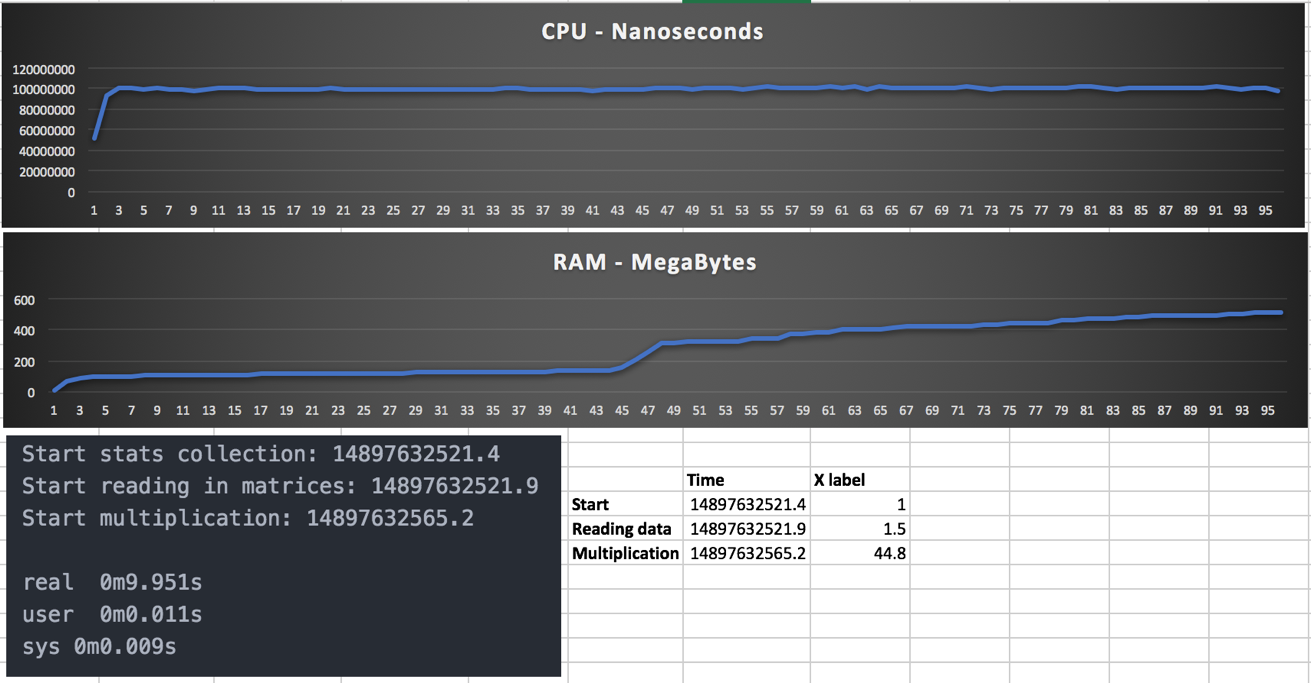
**Test Run 6:** Matrices size = 2048\*2048 Running on 1 core - 2GB ram

**Simple Approach:**

Took over 2 minutes to execute, the charts aren’t that interesting, the chart for the CPU and Core usage look the exact same obviously, with the core being used at 100% all the time (100000000 = 0.1 seconds).

****

**Efficient Approach:**

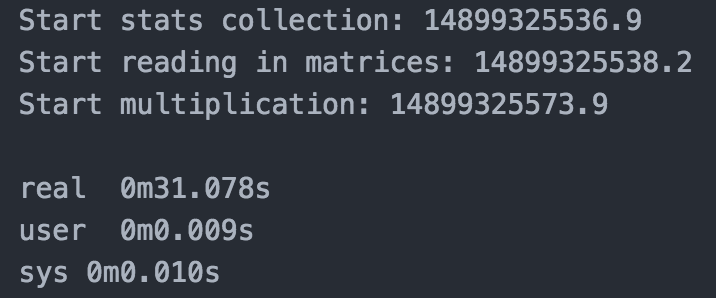
****

The charts once again are very predictable. What I found unexpected is how much quicker the efficient algorithm was on a single core. After running tests on a different number of cores from 1 all the way to 8, I noticed that the simple implementation’s performance was very bad initially and noticeably improved as I increased the number of cores up until 4, with 5+ cores there was a slight improvement not very much though. While the efficient approach always performed quite well and increasing the number of cores only marginally affected performance.

**Test Run 7:** Matrices size =2048\*2048, Running on 8 cores - 2GB ram and 16GB ram

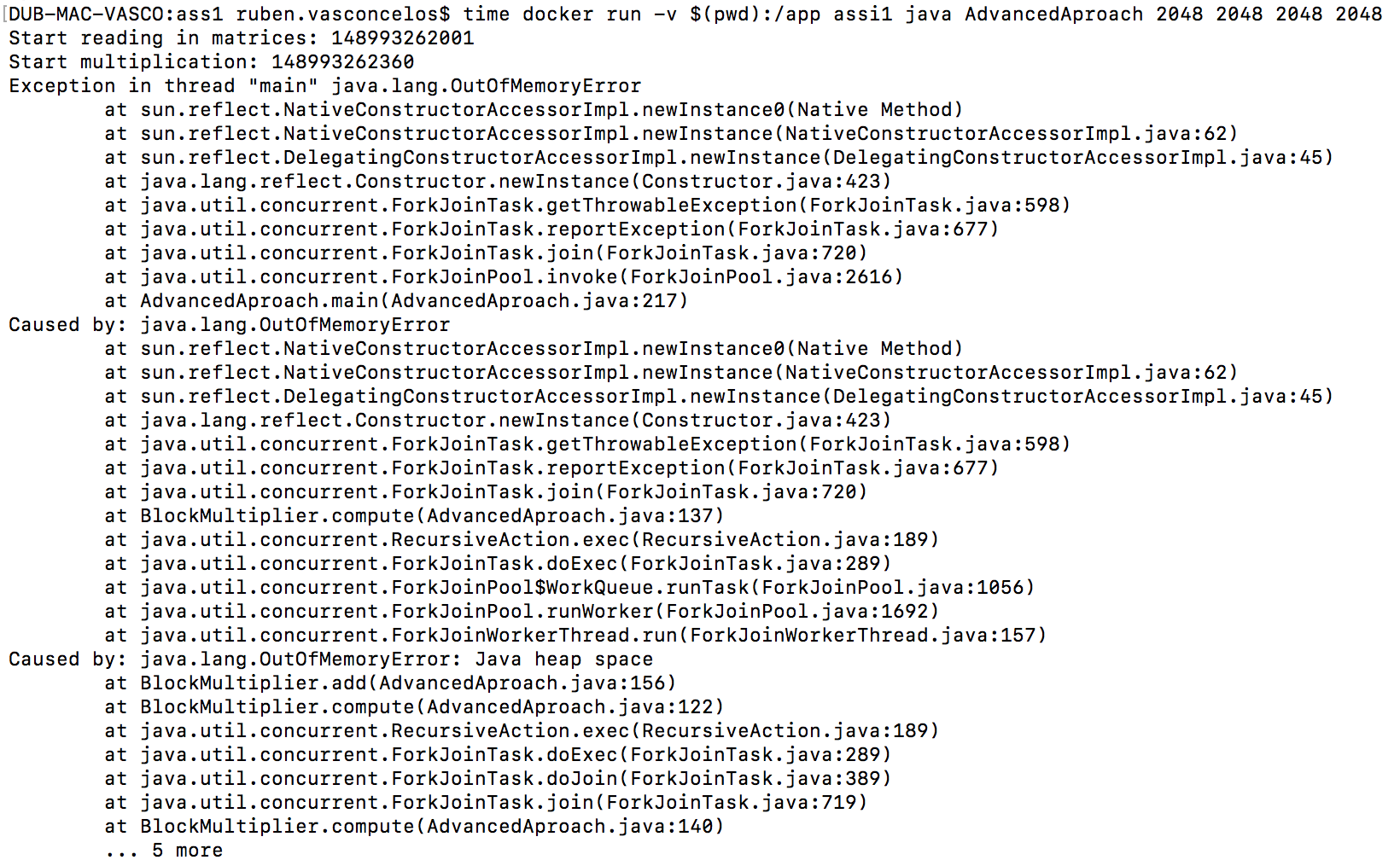
**Simple Approach:**

Very similar the results I got with 4 cores for the same data. The time slightly improved in the run bellow but was very similar in general.

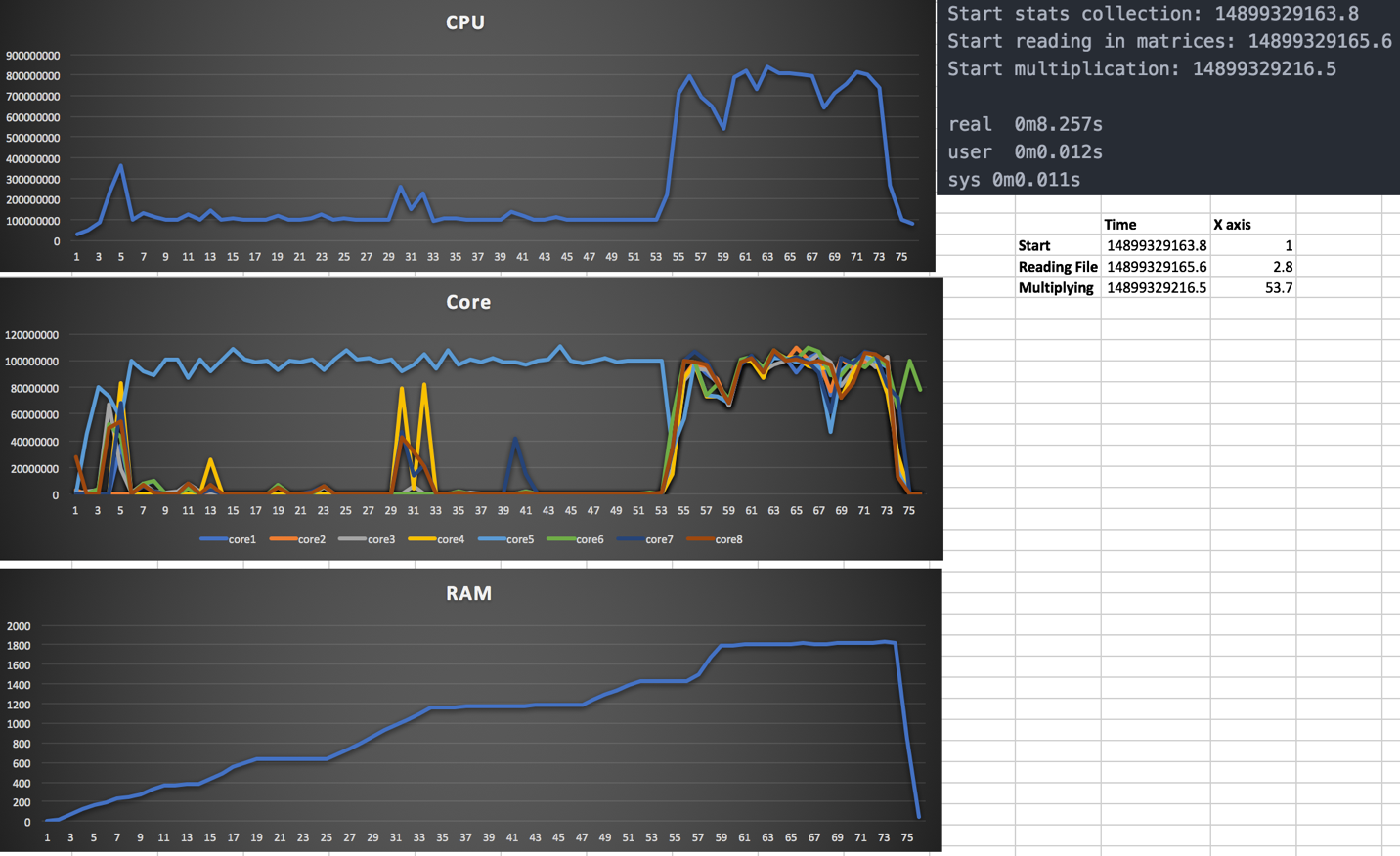


**Efficient Approach:** 2GB ram

The efficient approach actually started failing with an out of memory error.

****

**Efficient Approach:** 16GB ram



Speed wise it took almost twice the time it took to compute the same data with only 4 cores. It used around 1.8 GB of memory. The OutOfMemoryError occurred because I had another container running collecting the stats in simultaneous that was using up about 200 MB.

**Extra testing functions:**

I used the following functions to read in the data and compare the results afterwards in order to ensure both algorithms produced the same result.

