ASSIGNMENT REPORT

Gustavo Alves Bezerra

This is a simple report regarding the matrix multiplication assignment. The assignment required two different implementations of the algorithm: a sequential and a concurrent. Futhermore, a benchmarking was requested. I implemented both algorithms in python and C++ in order to benchmark the languages as well.

Input

A minor change was made regarding the specifications of the assignment. Instead of requiring only the matrices to be multiplied and the number of threads, the algorithm also requires the output file. Examples will be given in later sections.

Output

The result matrix will be stored in the provided file. The file follows the same standard provided by the assignment; i.e. the first line only has two elements: the number of rows and colums, while the remaining data are the elements (each row is separated by a return character, i.e. “\n” and each element inside the row is separated by a space character).

The results of the benchmarking will be printed on screen (standard output).

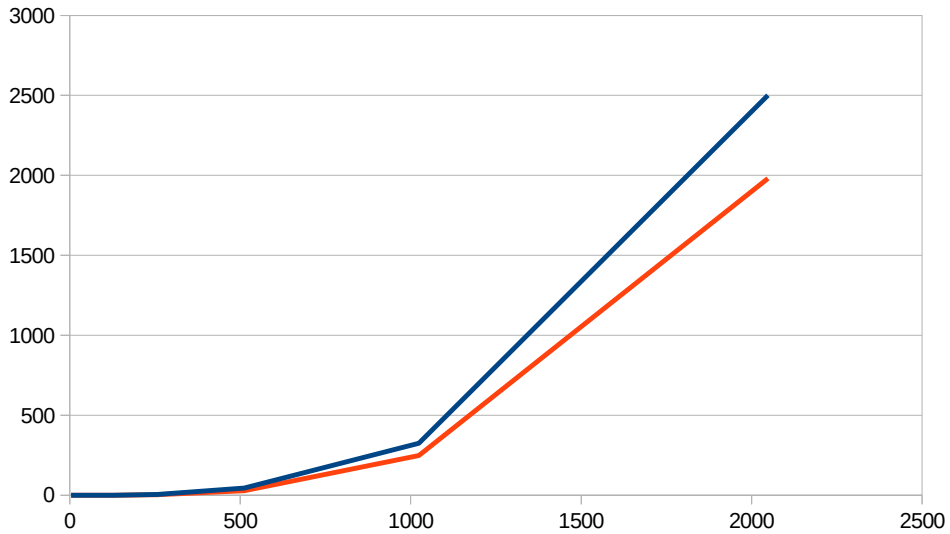
Python

How to run:

- Sequential: python matrix_mult.py matrix_a.txt matrix_b.txt output.txt
- Concurrent: python threaded_matrix_mult.py matrix_a.txt matrix_b.txt output.txt 2

The previous concurrent example will run the algorithm using 2 threads only. The number of threads must to exceed the numbers of rows of matrix_a, which is going to be the same number of rows of the result matrix (this specification was required in the assignment). Note that is possible to simulate the sequential behavior by passing “1” as the number of threads.

The following is the benchmark. The Y axis is the time required in seconds. The X axis is the size of the input matrices (both matrices A and B are square).



The blue line represents the sequential program. The red line represents the concurrent program. All the concurrent tests were executed using n/4 threads; hence, for the given test matrices: (1, 2, 4, 8, 16, 32, 64, 128, 256, 512). In other words, each thread will be responsible for multiplying 4 rows of the result matrix.

As can be inferred from the graph, there was a gain, but it was not too much significant. The gain depends on the number of threads created; and, not necessarily the maximum amount of threads will be the best approach possible. This happens because of the continuous need to change between threads and contexts, which requires heavy memory operations. Best speedup obtained was 1.546, it happened when the size of the input was 512.

C++

How to compile:

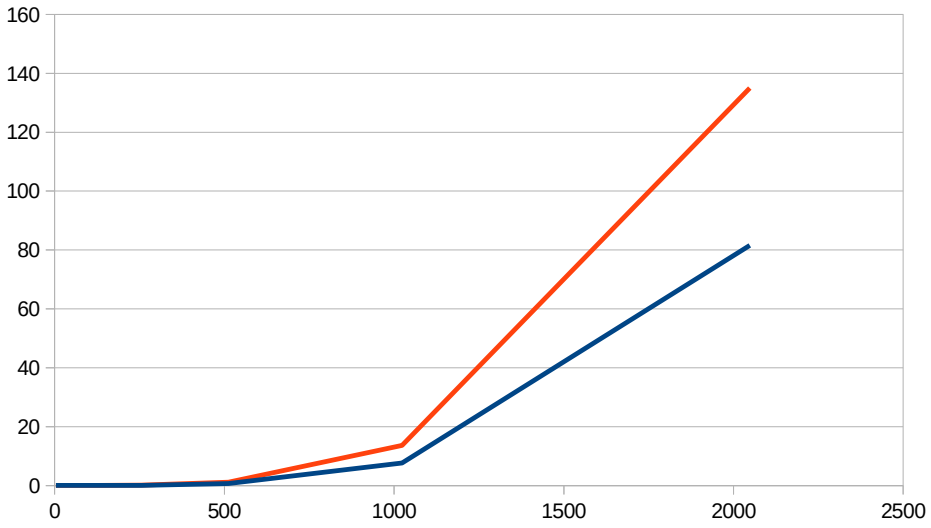
- Sequential: g++ matrix_mult.cpp -o mm
- Concurrent: g++ threaded_matrix_mult.cpp -o tmm -lpthread

How to run:

- Sequential: ./mm matrix_a.txt matrix_b.txt output.txt
- Concurrent: ./tmm matrix_a.txt matrix_b.txt output.txt 2

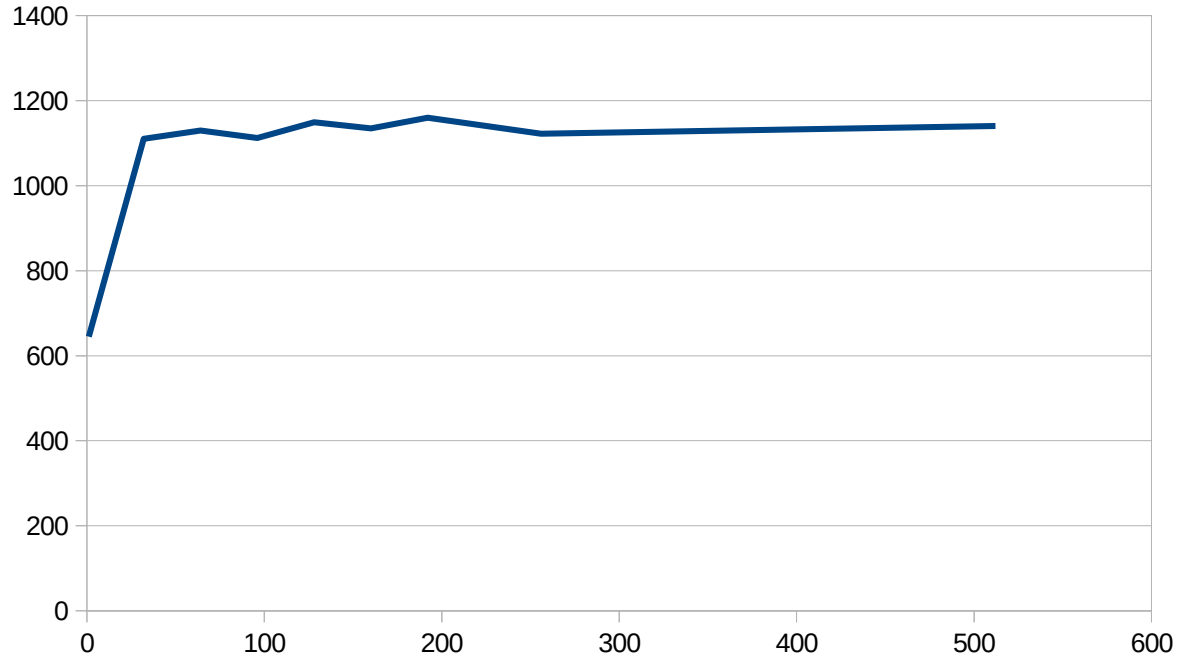
This follows the same logic as for the python case.

This assignment was really useful for me in order to have a better understanding regarding the execution speed of interpreted versus compiled language. This was not very clear to me since I used to work with small amounts of data and the difference was not, apparently, significant. However, the difference is huge! The time python needed to sequentially process the 1024 sized input was, approximately, 42 times larger than C++'s.

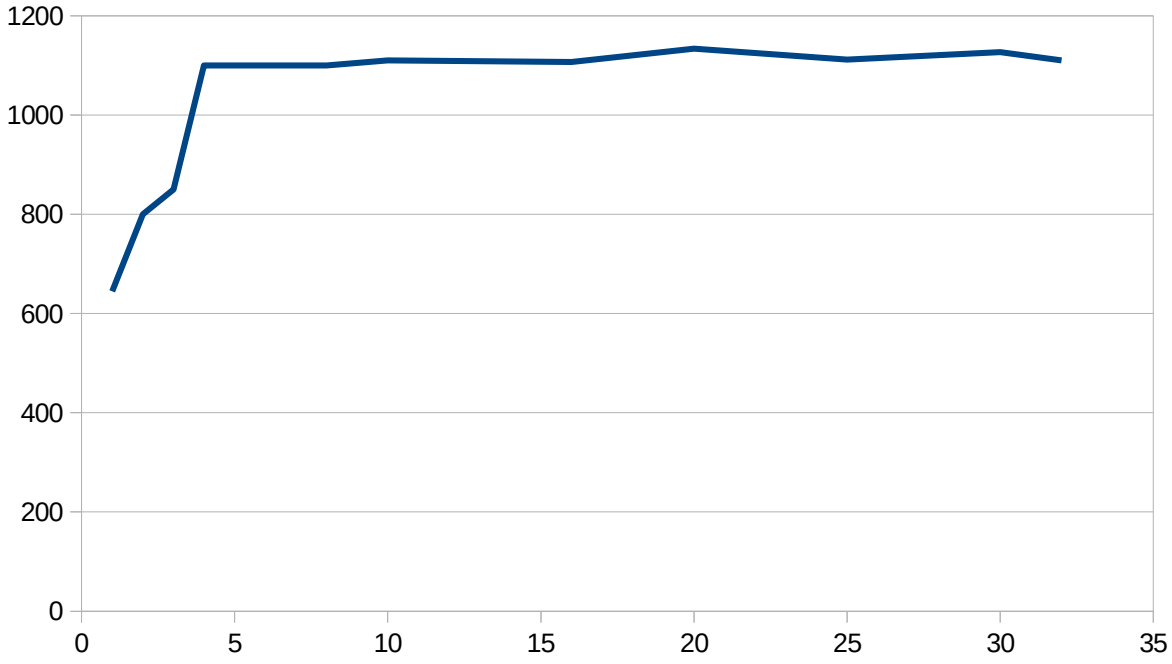


The blue line represents the sequential algorithm's execution and the red line represents the concurrent. The X axis represents the input size, while the Y axis represents the execution tie in seconds. The benchmark followed the same logic as python's: each concurrent thread would process 4 rows. In C++'s, case, however, the results did not go as expected. The senquential version of the code happened to be much better than the concurrent one. Tis probably happened because of the number of threads used. One of the most difficult parts of concurrent programming is to know how many threads to use; it is often a "magical number" that varies for each algorithm.

The number of threads, regarding this case, at least, must not be too large. As can be seen on the following graph. Executing with only 1 thread was way more efficient than using 32 or more.



Using 32 or more threads did not signifficantly changed the results. Benchmanrking for 1 to 32, we will obtain the following graph.



Which leads to the conclusion that, surprisingly, the execution with only one thread is more efficent than any concurrent execution. This may result from the operating system giving priority to other processes, or even because the compiled code is not as efficient as we wished. Remember, C++ does not have embedded threads, which may compromise its speed (it depends on the library implementation as well).