Proj1 Report

Gihad Coorey 23091788

Ethan Judge 23775697

Code explanation

The program provides command line options `-x` and `-y` for users to specify the dimensions of Matrices X and Y respectively. These are the matrices that will be multiplied for performance analysis. There are also the options `-d` to specify matrix density, `-t` to specify thread count in parallel regions, and `-h` for usage instructions. If the options are not specified, default values are used based on `#define` directives.

An example usage is `./a.out -t 4 -d 0.05 -x 100x50 -y 50x200` (assuming the object file has been compiled as `a.out` in the working directory. This will generate a 100×50 Matrix X and 50×200 Matrix Y, each with 5% density, and multiply them using 4 threads in parallel regions.

There are 2 structures defined, `Matrix` and `CompressedMatrix`, to store uncompressed matrices, and compressed matrices in the form of B and C matrices as defined in the project brief. They are also given metadata fields such as `name` and `row_sizes` to help in identifying and iterating through the data without memory corruption.

The program first allocates the matrices X and Y on the runtime heap, before populating it with (unseeded) random values from 1 to 10 in (unseeded) random locations, matching the requested density. These matrices are then compressed according to the simplified row compression format defined in the project brief, and their compressed matrices B and C written to files titled `FileB` and `FileC` respectively. The same file-writing function can also be used to write to `stdout` instead with a simple change to the function call in the `main` function.

The matrices are then multiplied using their compressed form, first sequentially and then in parallel. Both operations are timed separately using `omp_get_wtime`. For each row in Matrix X, the algorithm iterates through each element, retrieves its value and column index, and then multiplies this with the relevant row in Matrix Y, if a nonzero element exists at the right column. This product is then added to the corresponding entry in the result matrix.

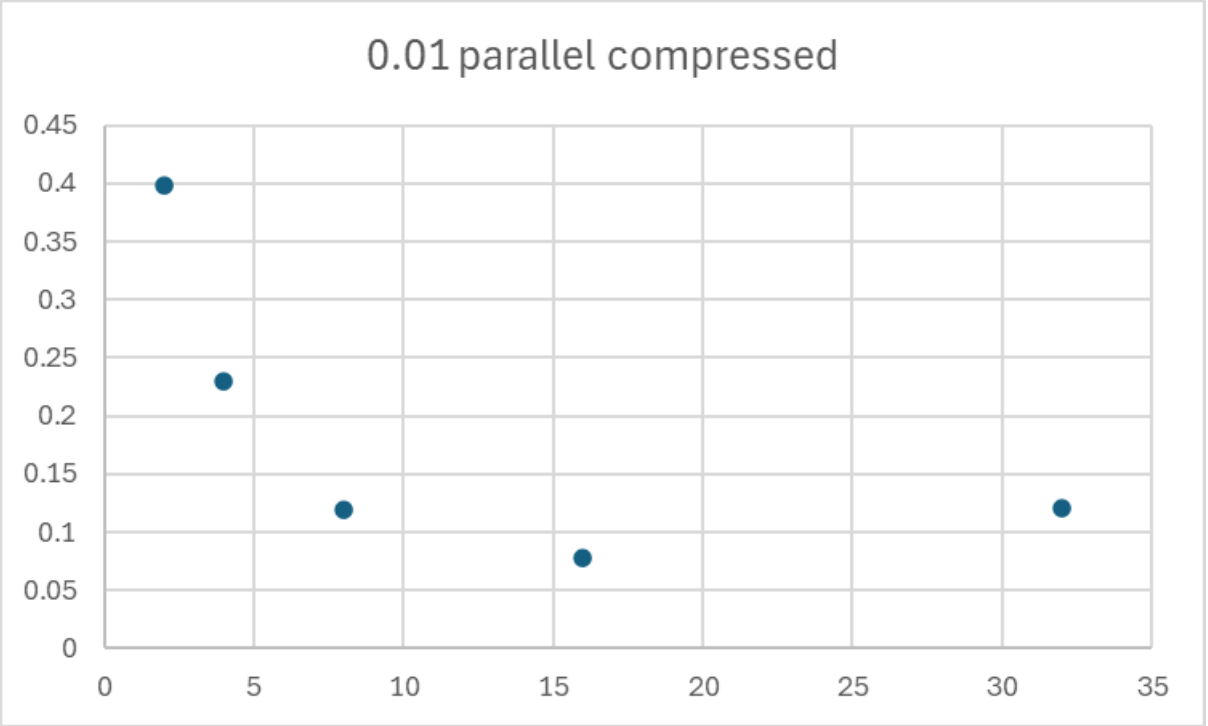
Then the original uncompressed matrices are multiplied together using the conventional multiplication algorithm, again both sequentially and in parallel for comparison. Finally, the two resulting matrices are compared to make sure the answer is the same, before the program ends execution. Note that the time taken to initialise result matrices and generate compressed matrices was not included in the timing of the algorithm. It is possible that for some sizes and densities, it will take longer to compress the matrices and then multiply them, than to just multiply them outright using the conventional algorithm.

The parallelisation for both algorithms is handled using `#pragma omp parallel for` with a `runtime` schedule specified. This allows the schedule to be changed with the runtime environment variable `OMP_SCHEDULE`, instead of having to recompile the code. Variables are `shared` by default since the only modifications made in the parallel blocks are to the result matrix, which we want to retain modifications made by any thread. the `#pragma omp atomic` directive is used when updating an entry in the result matrix to ensure no race conditions are created for the shared array.

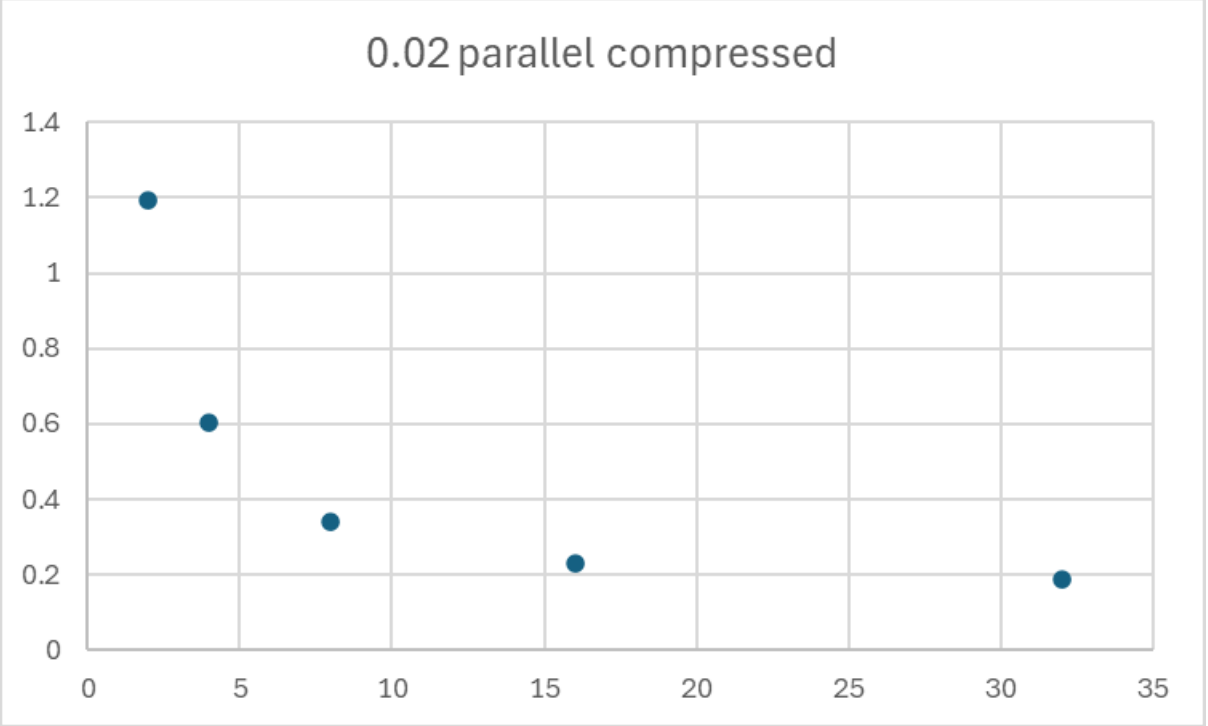
Thread optimisation - 1%, 2%, 5%

To estimate the optimal number of threads to run for all densities, a series of test runs were completed. The completion times for sequential and parallel codes for both the compressed and uncompressed matrix multiplication algorithms were recorded for each density. The thread amounts tested were `2`, `4`, `8`, `16` and `32`. The size of the matrices used in calculations were `10000` x `10000` instead of `100000` x `100000`. This is due to runtimes and memory usage becoming too large for testing purposes.

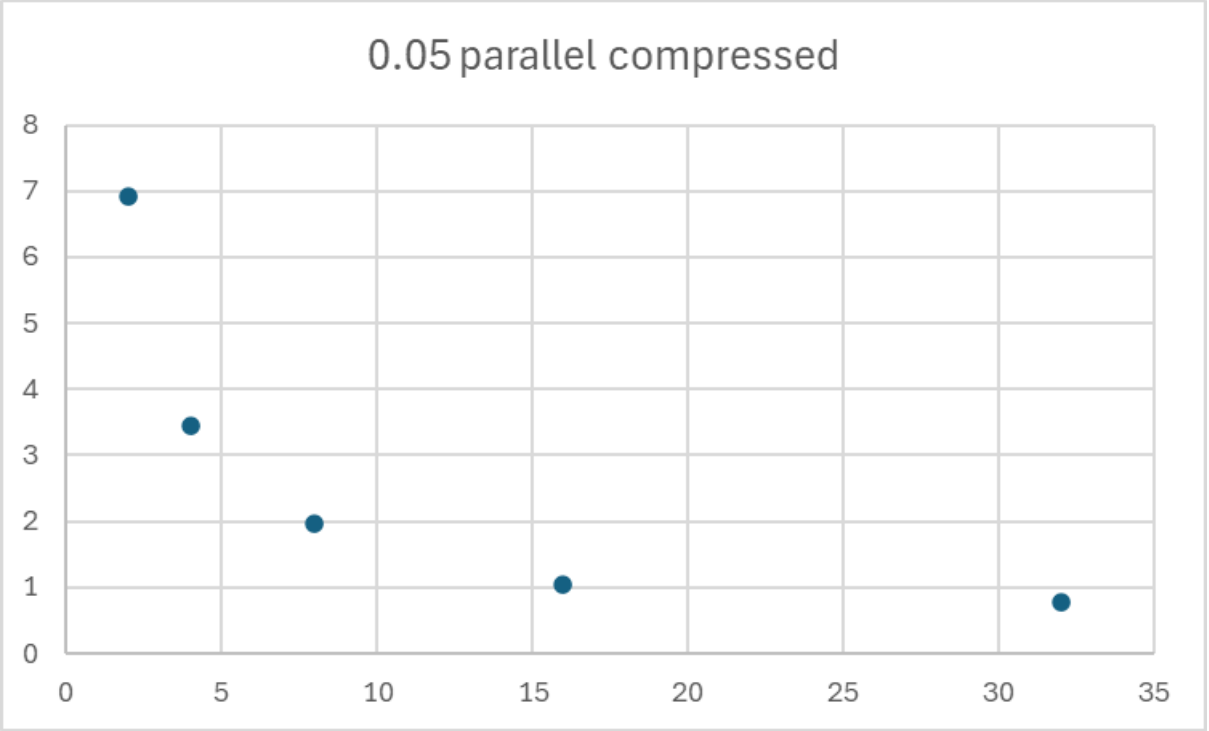
Below are graphs for each of the test conditions. They are organised by test condition rather than by density to highlight the similar trends that occur due to the utilisation of threads in different operations. The graphs are in dot form as drawing both straight or curved lines between the dots would give an inaccurate representation of the relationship between thread number and running time. For example, in 0.02 parallel compressed it could be assumed that the function tends towards an asymptote, as $\mathbf{x \to \infty}$ $\mathbf{y \to a}$ where a is some limit. But the optimal number of threads might actually be 20 (global min of the function).



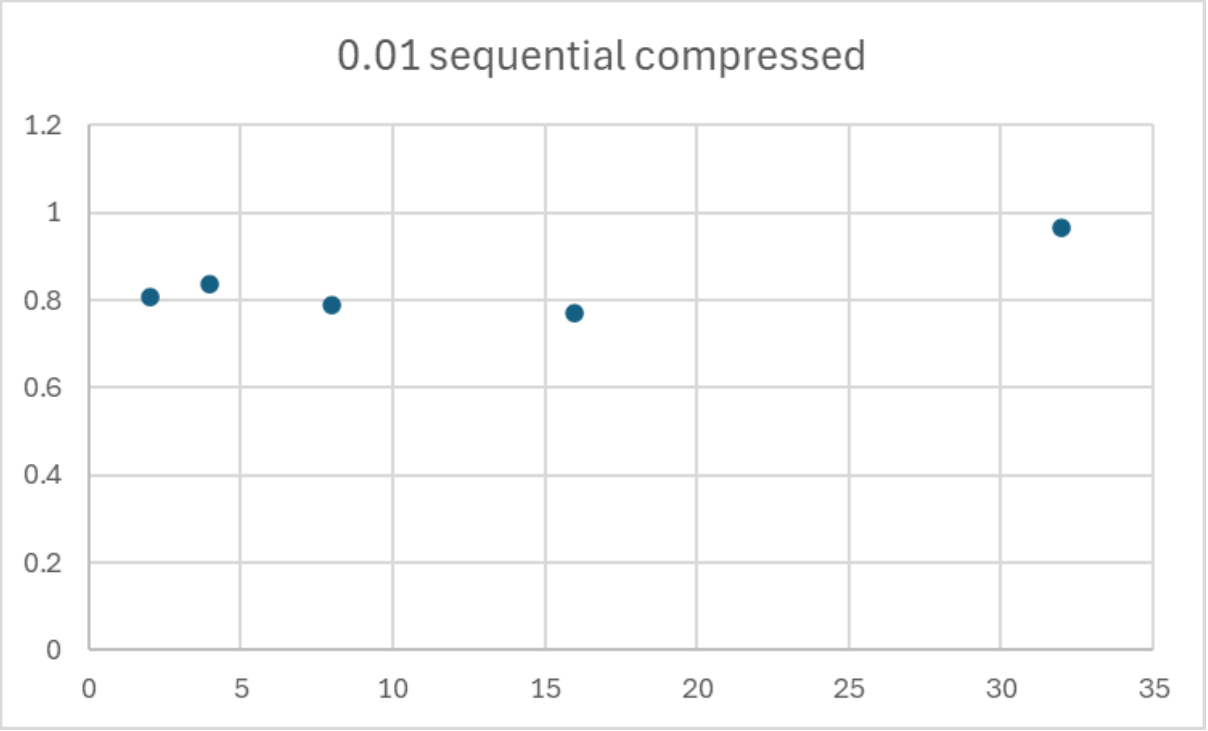
Optimal tested thread choice was 16. Optimal thread could lie between 8-32 threads.



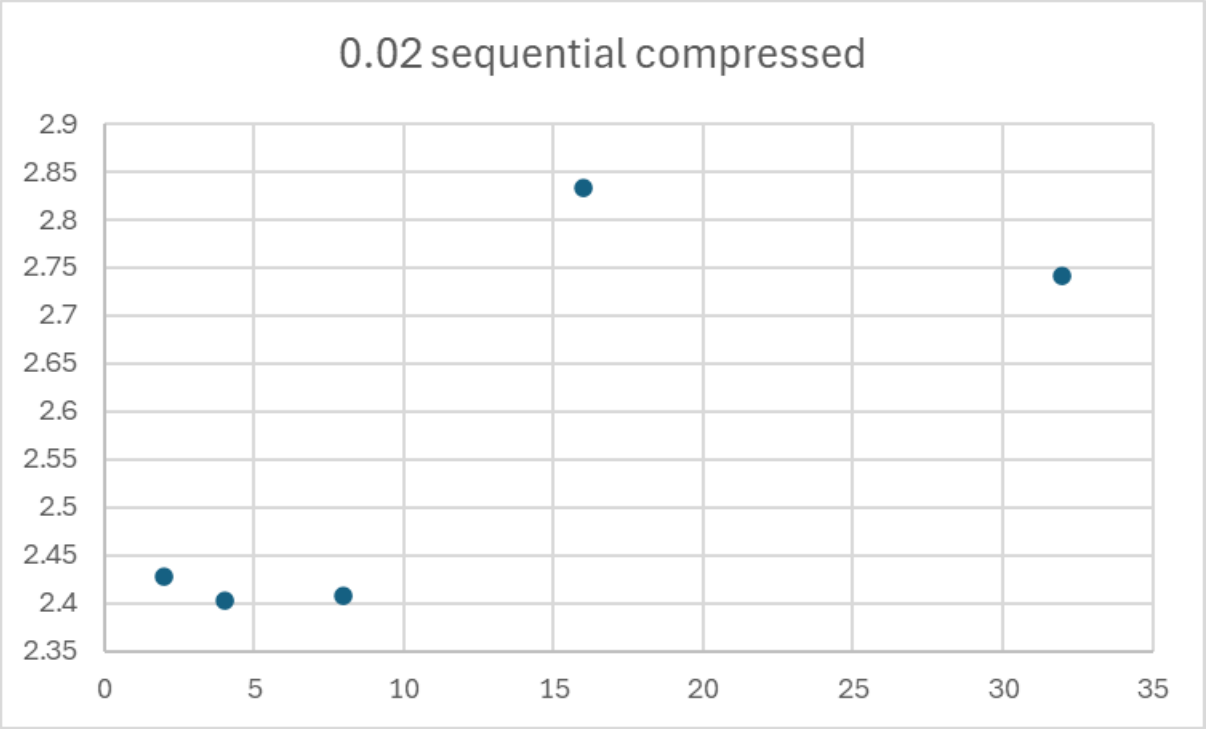
Optimal tested thread choice was 32. Optimal thread could lie between 16-∞ threads. Although it appears to be approaching an asymptote.



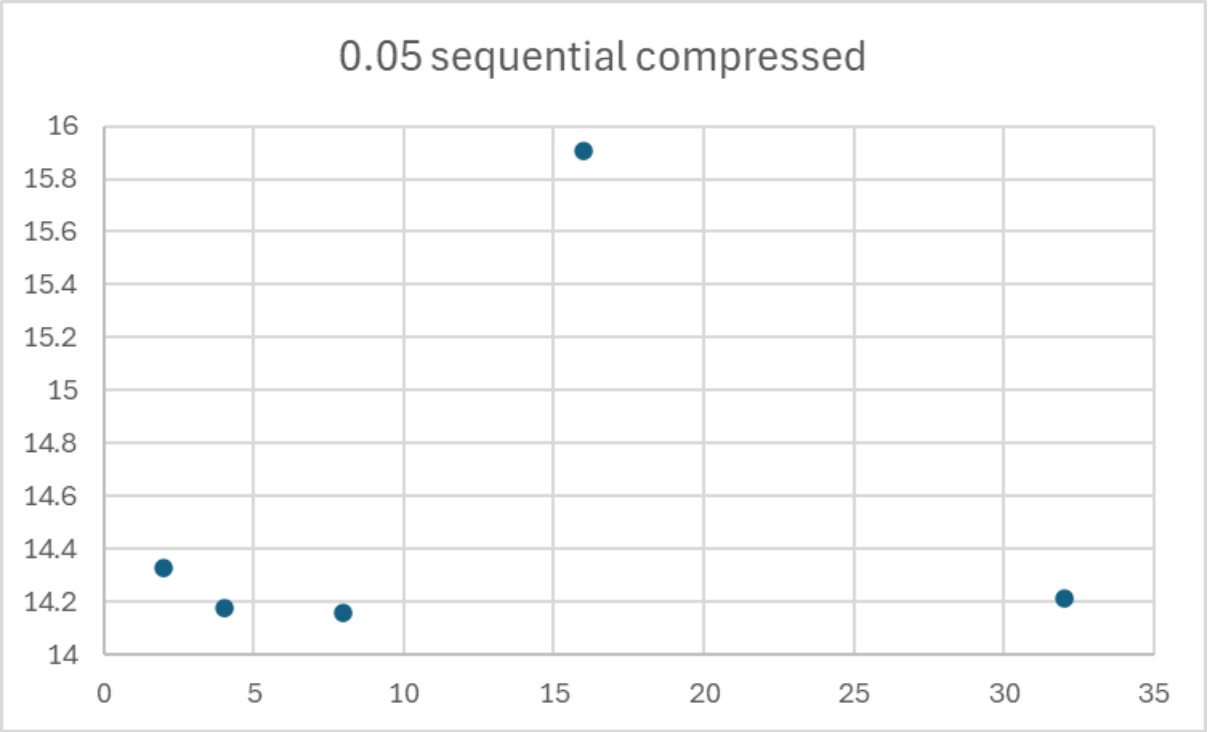
Optimal tested thread choice was 32. Optimal thread could lie between 16-∞ threads. Although it appears to be approaching an asymptote.



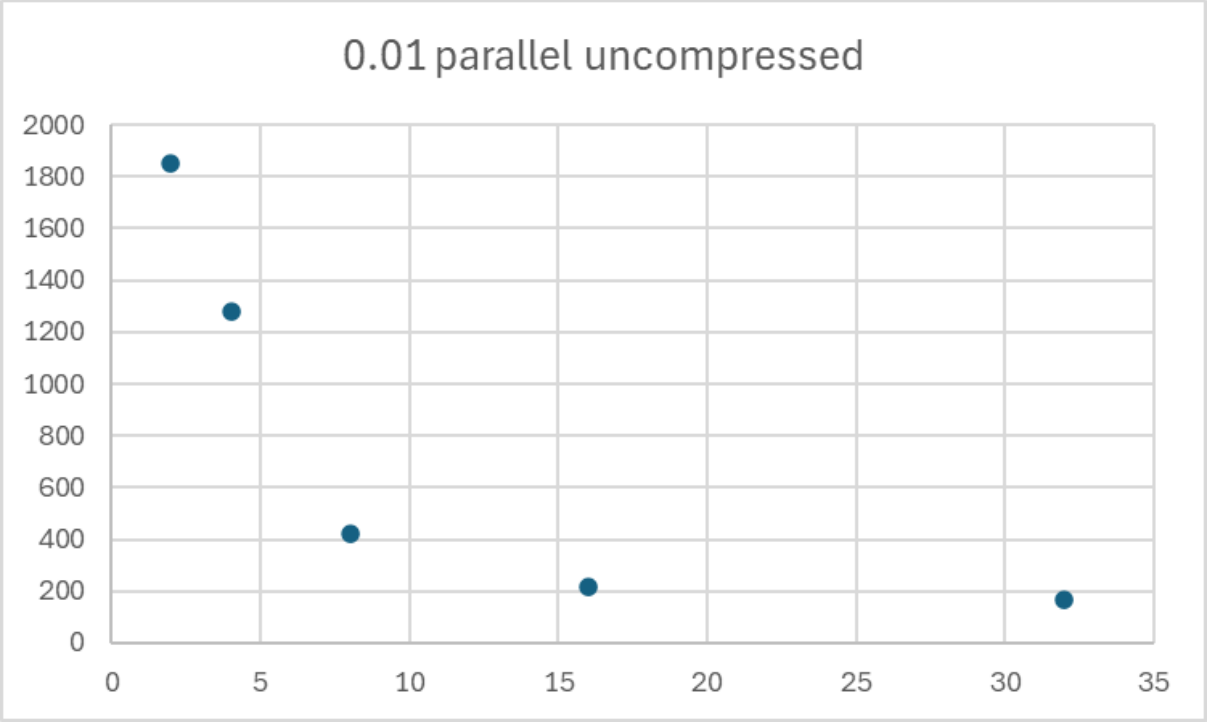
Optimal tested thread choice was 16.



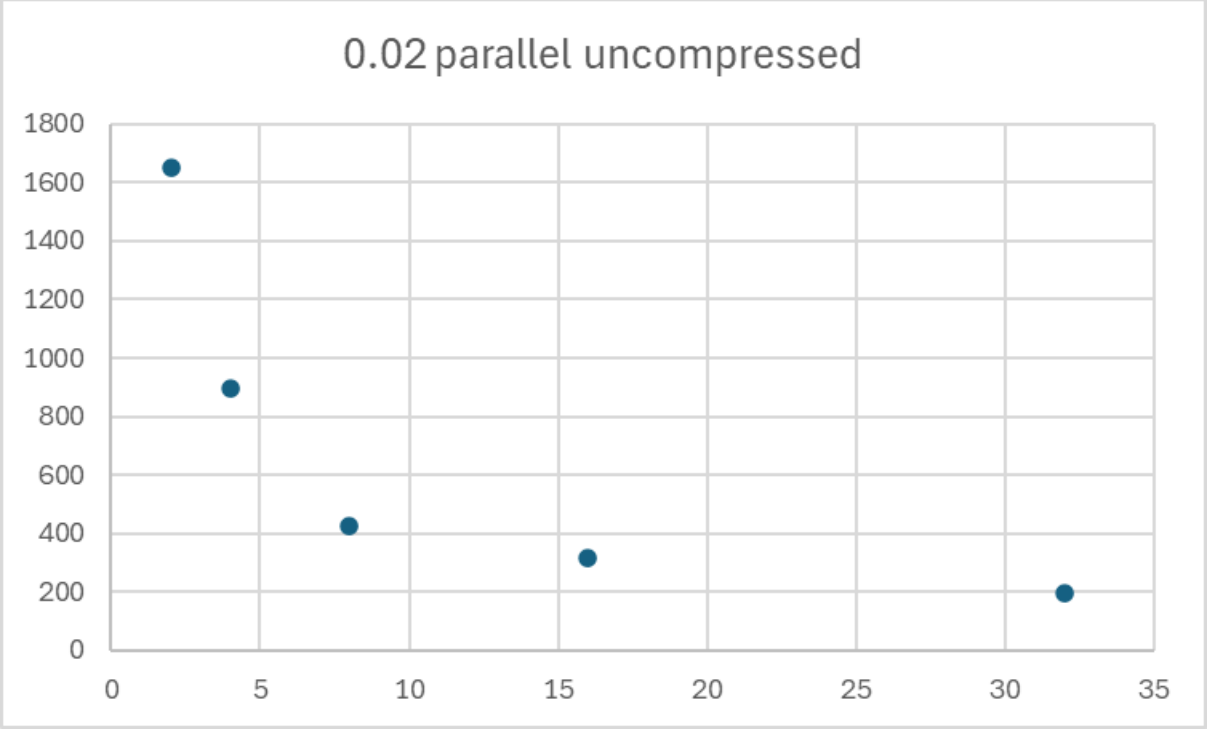
Optimal tested thread choice was 4.



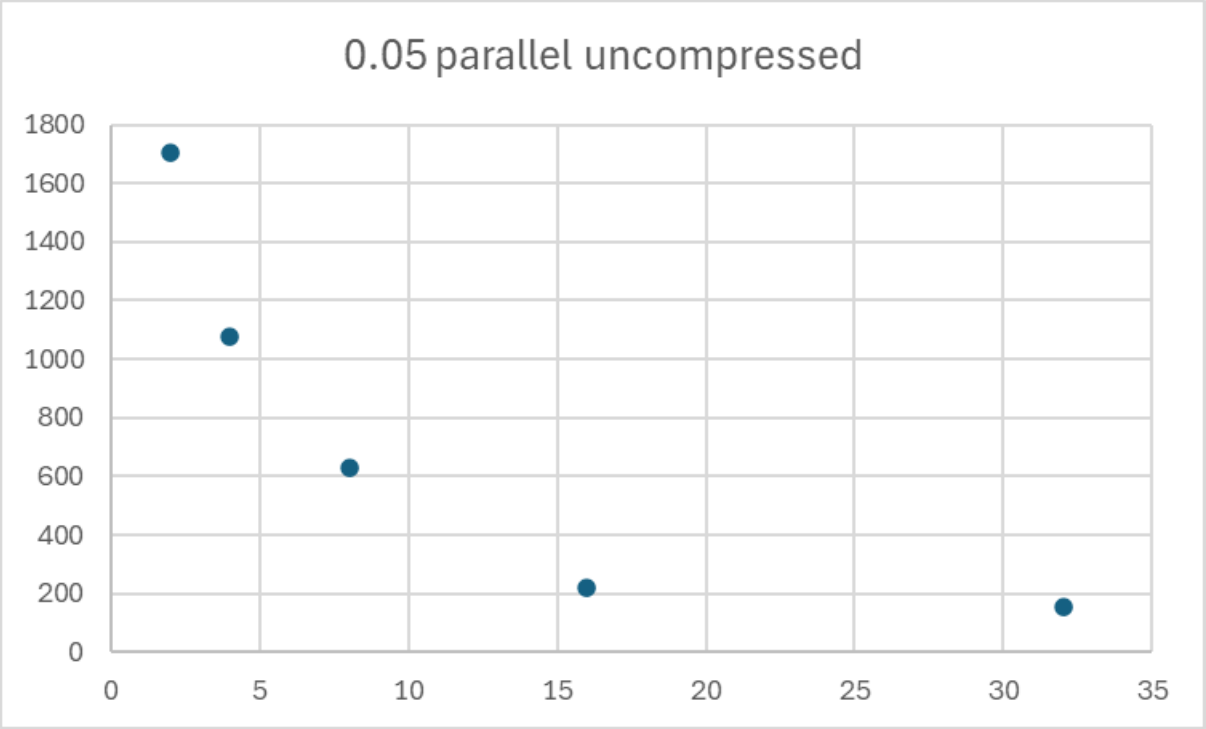
Optimal tested thread choice was 8.



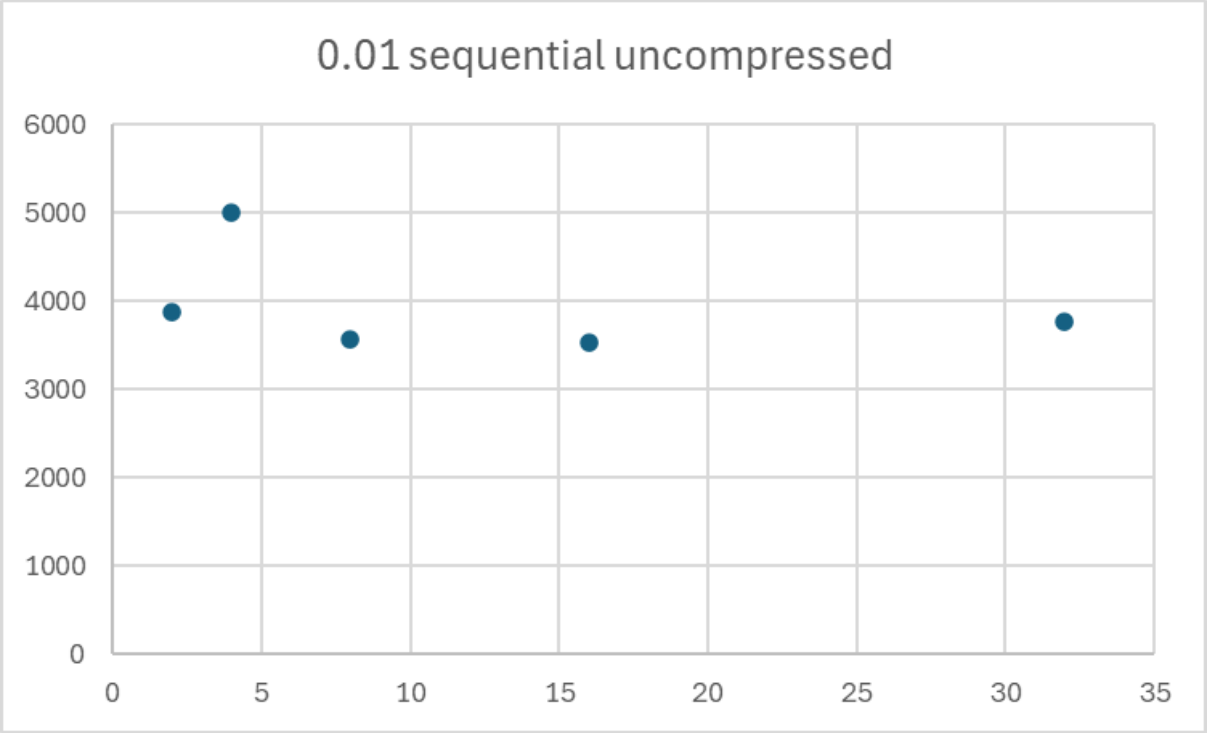
Optimal tested thread choice was 32. Optimal thread could lie between 16-∞ threads. Although it appears to be approaching an asymptote.



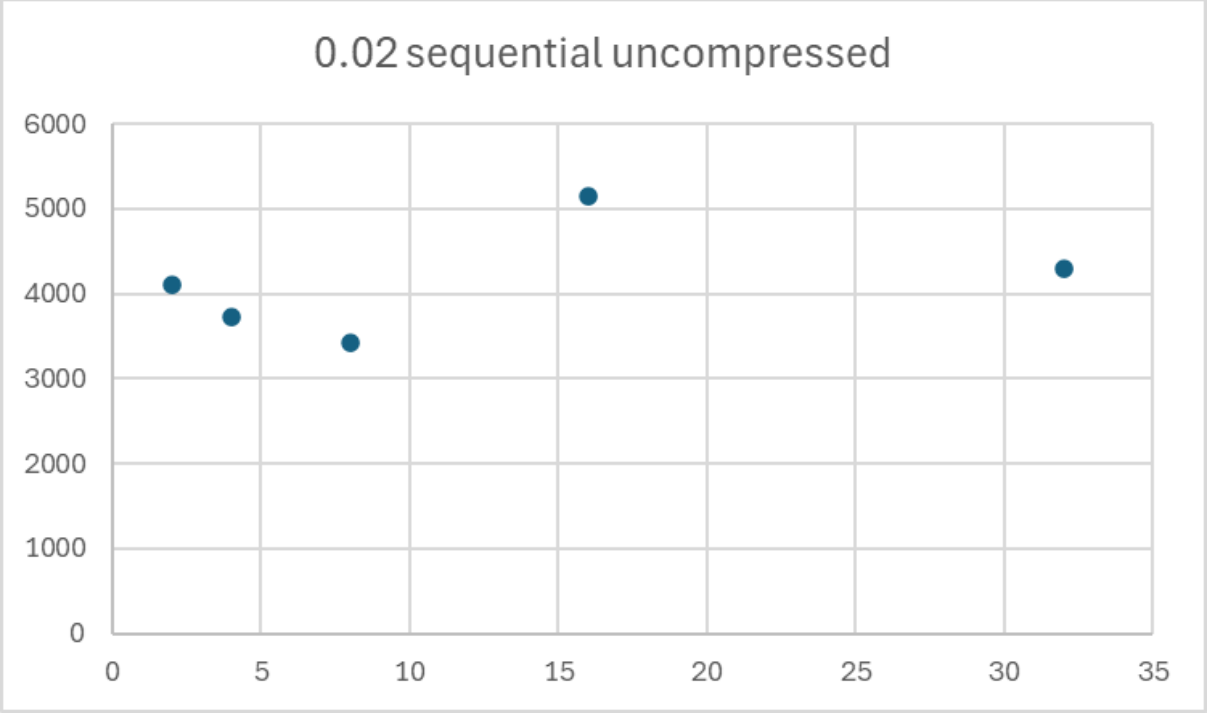
Optimal tested thread choice was 32. Optimal thread could lie between 16-∞ threads. Although it appears to be approaching an asymptote.



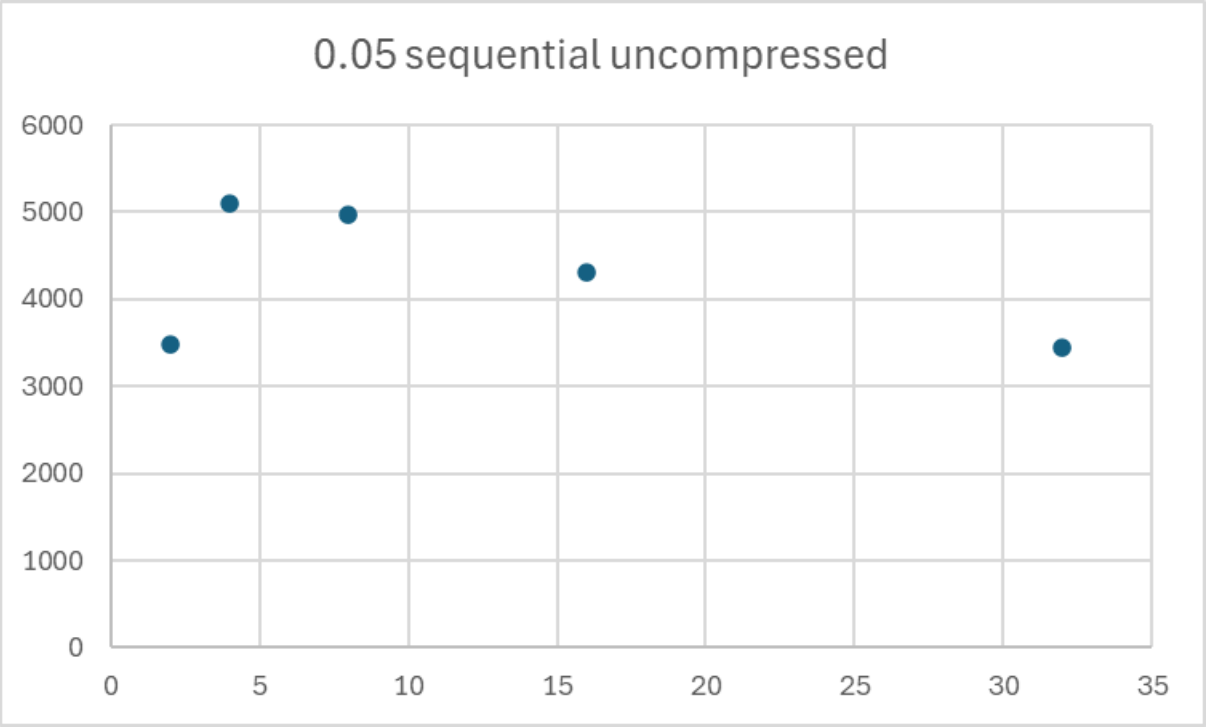
Optimal tested thread choice was 32. Optimal thread could lie between 16-∞ threads. Although it appears to be approaching an asymptote.



Optimal tested thread choice was 16.



Optimal tested thread choice was 8.

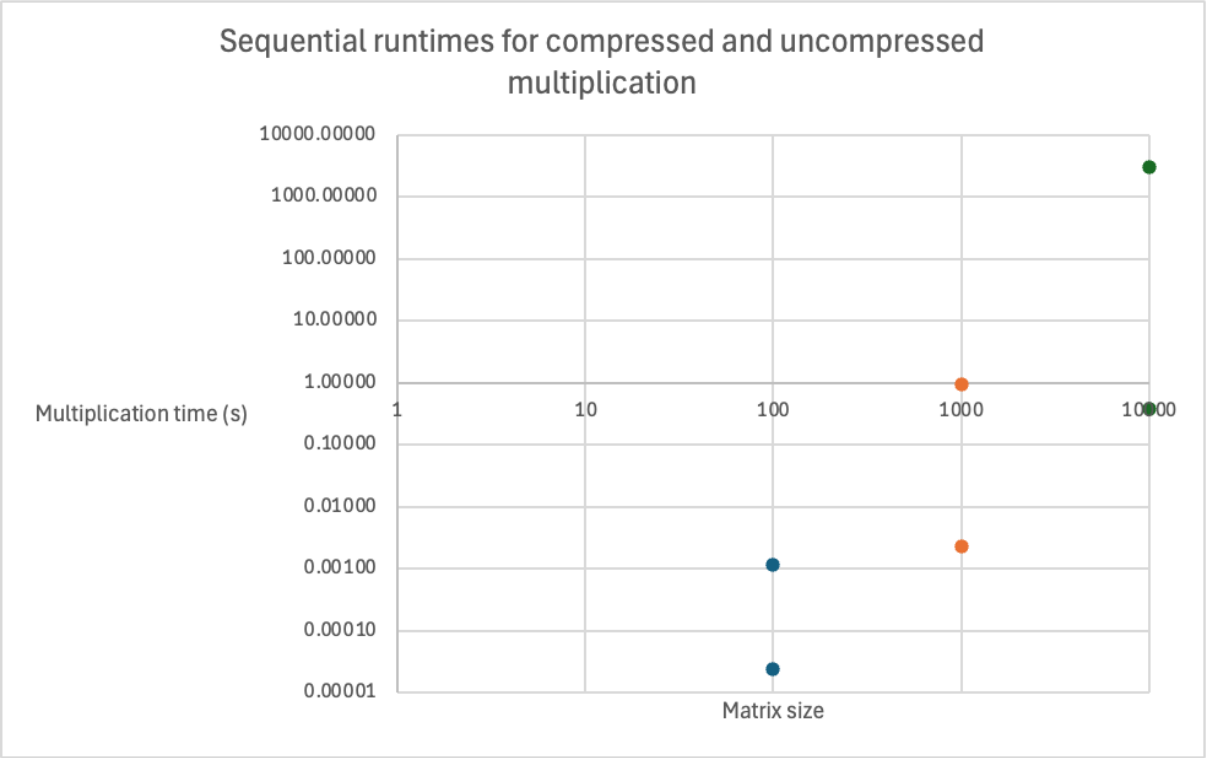


Optimal tested thread choice was 32.

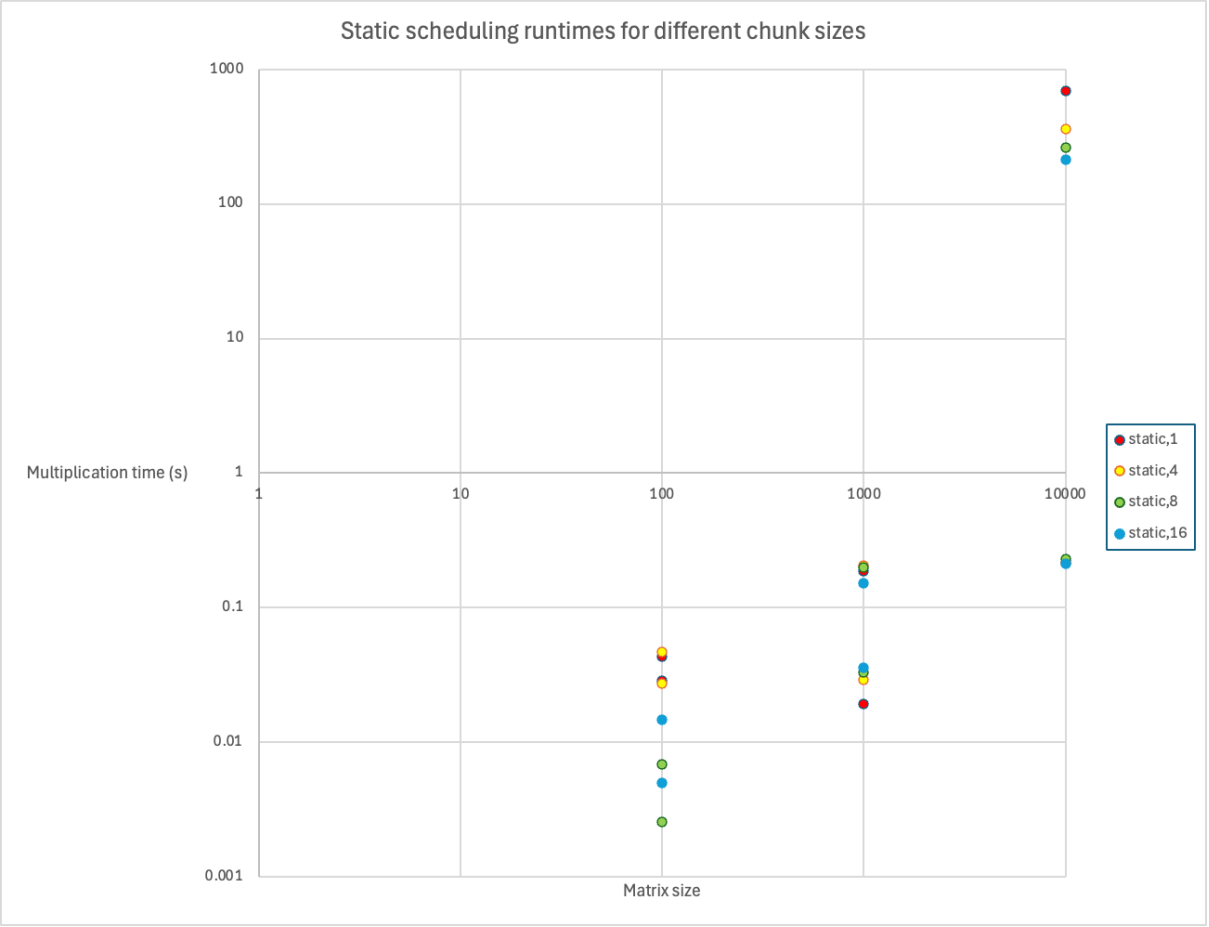
Scheduling optimisation

To test the effect of scheduling strategy, chunk sizes of `1`, `4`, `8`, and `16` were used with the `static`, `guided`, and `dynamic` strategies to run the program with 1% dense square matrices of size `100`, `1000`, `10000`, and `100000`. Timed performance tests were conducted for both the conventional multiplication algorithm and the compressed algorithm. 28 threads were used for all parallel blocks. An average time was calculated for each matrix size, algorithm, scheduling strategy, and chunk size. For all tests, the compressed multiplication algorithm was faster than the conventional algorithm, and is therefore the lower (quicker) point for a given matrix size in the following graphs.

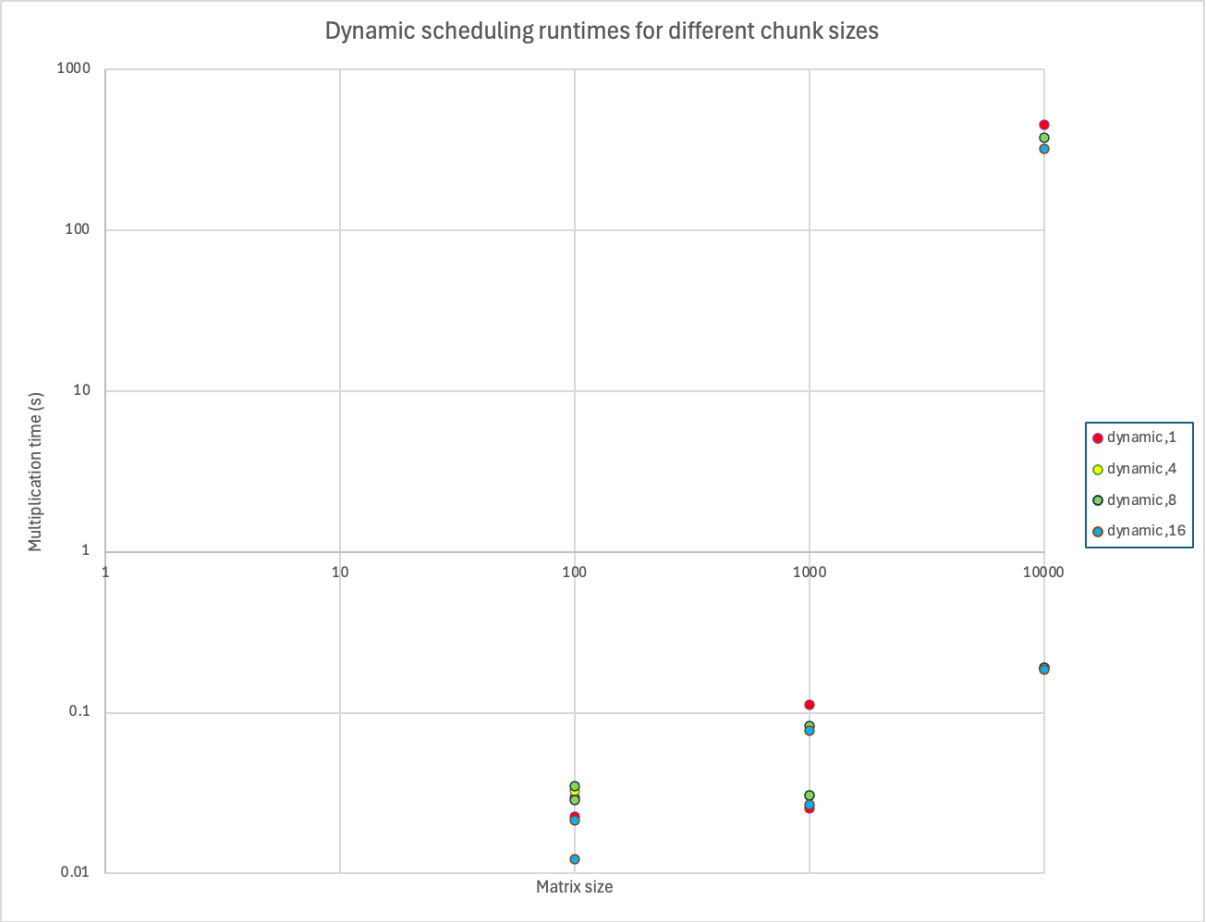
For a comparison point, the conventional and compressed multiplication algorithms were run sequentially first, with the larger size taking over 60 minutes (3600s) to complete.



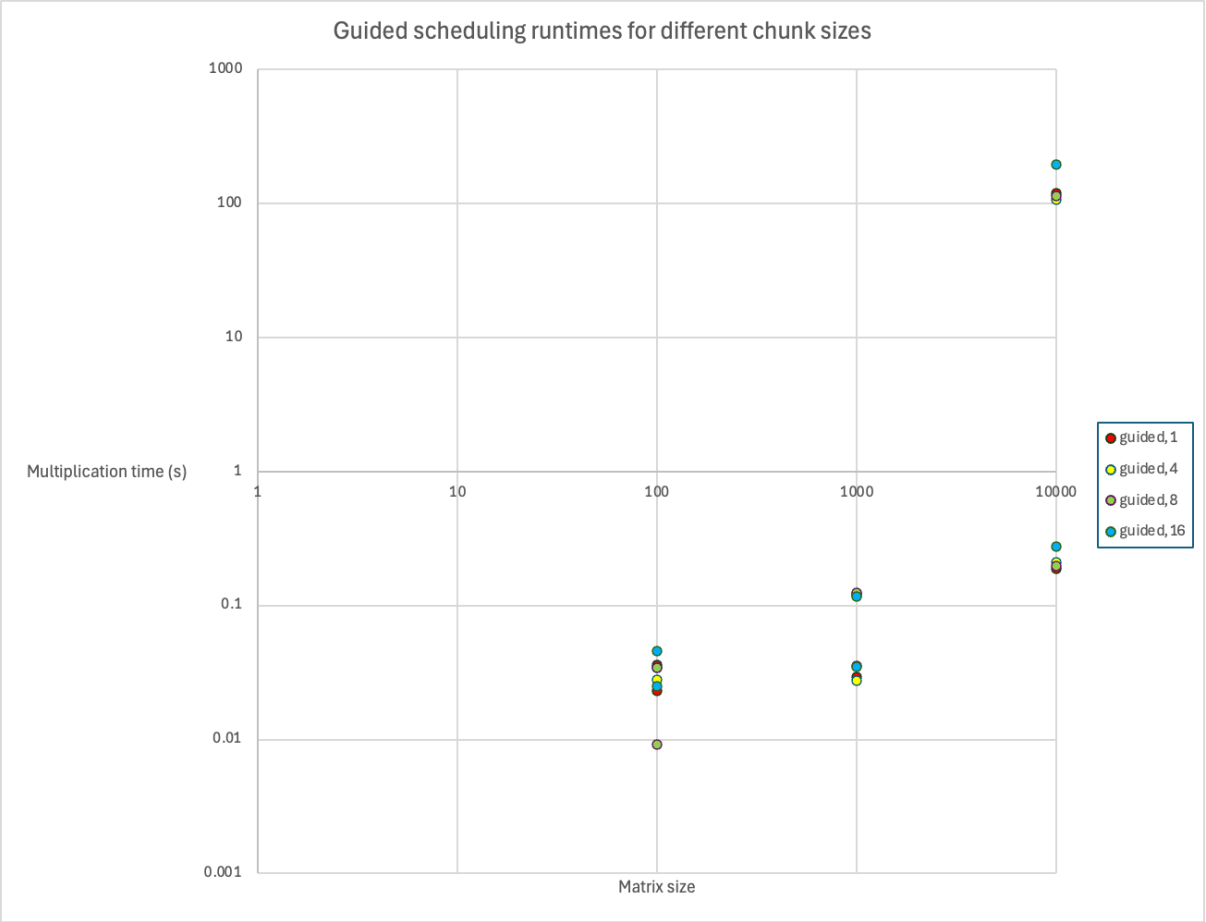
For the static schedule, optimal chunk size varied for different matrix dimensions, with performance improving consistently as chunk size increased for the largest dimension. The fastest `static` performance for the large dimension was 0.2 seconds for the compressed algorithm, and 214 seconds for the conventional.



For the `dynamic` schedule, runtimes for the smaller dimensions did not improve or degenerate noticeably with any chunk size. For the larger dimension, the compressed algorithm performed similarly at all chunk sizes, and the conventional algorithm was also less affected by chunk size compared to the `static` schedule. The fastest performance for the large dimension was 0.18s for the compressed algorithm and 320s for the conventional, both at chunk size 16.



For the `guided` schedule, the fastest runtime for the large dimension was 0.19s with a chunk size of 4 for the compressed algorithm, and 106s with chunk size 4 for the conventional. The smaller dimensions were slightly slower than the `dynamic` schedule, and more or less identical to the `static`. However, the difference between performance times for the compressed and regular algorithms were decreased.



Overall, for larger matrix sizes, the `dynamic` schedule with a larger chunk size of around 8 or 16 was found to be optimal for performance.