Matrix Multiply Report

Jaime Luengo Rozas : jl3752

March 10, 2019

1 Introduction

In this report, two types of matrix sorting are discussed and the improvement of using omp parallel.

Important:

- If a time is not labeled, it is in **ms**.
- I am not presenting any tables because I believe they don't add much more valuable info not given by the plots and explanations. Plus it is time consuming to put them in latex...

2 Hardware specs

The tests were run on a Philips 314 desktop. The hardware specs relevant to this test are the following:

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 8
- Thread(s) per core: 2
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 94
- Model name: Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
- CPU MHz: 3999.938

- CPU max MHz: 4000.0000
- CPU min MHz: 800.0000
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 8192K

3 Column Sorting

In Fig.9 we can see the influence of the number of threads over the computation time for a matrix that is 20k x 20k size. The best number of threads corresponds to the same as number of CPUs in my machine 8. We can see the big increase in time from 8 to 9 threads, since it is when at least one core is starting to do context switchin between threads. Also note that the performance of OpenMP single threads is worse than the one impleted in raw C. This is due to the overhead and extra instruction created at compile time by OpenMP.

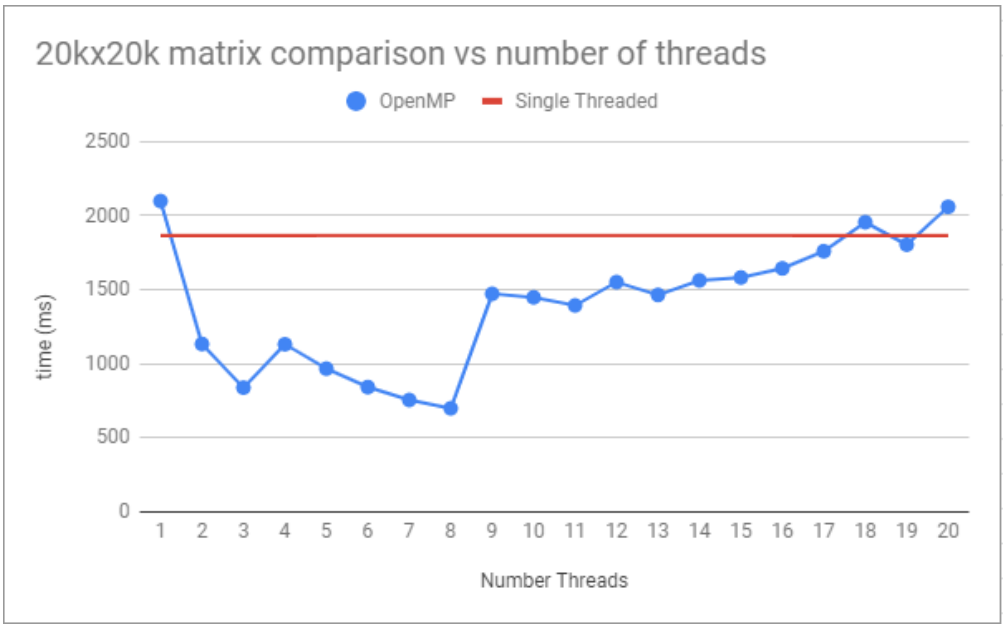

Figure 1: Comparison of executing times vs number of threads

The performance for a square matrix can be computed as following:

$$perf = \frac{n(n+1)comparisons + n^2 swaps}{cores \cdot GHz \cdot} \tag{1}$$

This performance isan estimate of the percentage of instructions strictly used for computation vs the total executed.

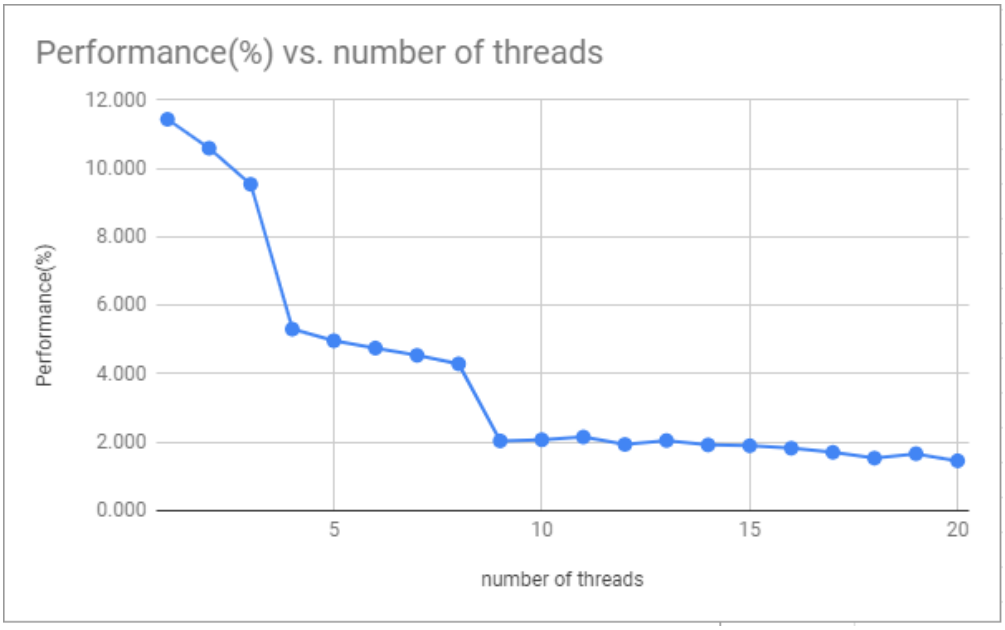

Figure 2: Performance vs number of threads for column sort

Using this formula I computed the performance graph, where from 1 to 8 threads I assume 1 to 8 cpus respectively, and from there 8 cpu are used all the time:

We can see how the highest performance with the OpenMP code is a single thread at 11.4%. However without the OpenMP package, the single threaded equivalent achieves a performance of 12.8%. This is due again to the OpenMP overhead. We can also see the jump from 3 to 4 threads, probably due to scheduling overhead, and from 8 to 9 the context switching overhead is also added.

Fig. 3 and 4. show the increase of time in both single threaded and OpenMP with 8 threads implementation when changing number of rows and number of columns. We can see how the effect is equivalent for both since, increasing number of rows, increases number of elements to search for max at $O(n^2)$ rate; and increasing number of columns also increases number of elements to swap at $O(n^2)$ rate also.

Fig.3-5 also show that there is approx. 2 orders of magnitude in execution time for small size matrixes with the single threaded program and the OpenMP 8 threads one; due to the fact that the time for scheduling and creating the executing the multithreading runtime is comparable to the execution time. As the size increases to 8K this time becomes neglectatable and OpenMP becomes 2 orders of magnitude better

4 Getting there

Getting both correct and performant OpenMP code is hard. In fact what I found myself doing in the end was analyzing small code sections within OpenMP and also reducing the amount of syncronization constructs as much as I could mantaining mutual exclusion.

To make sure my code was correct I compared always the output of my single threaded program with that of OpenMP. Testing cannot proof the non-existance of bugs for concurrent

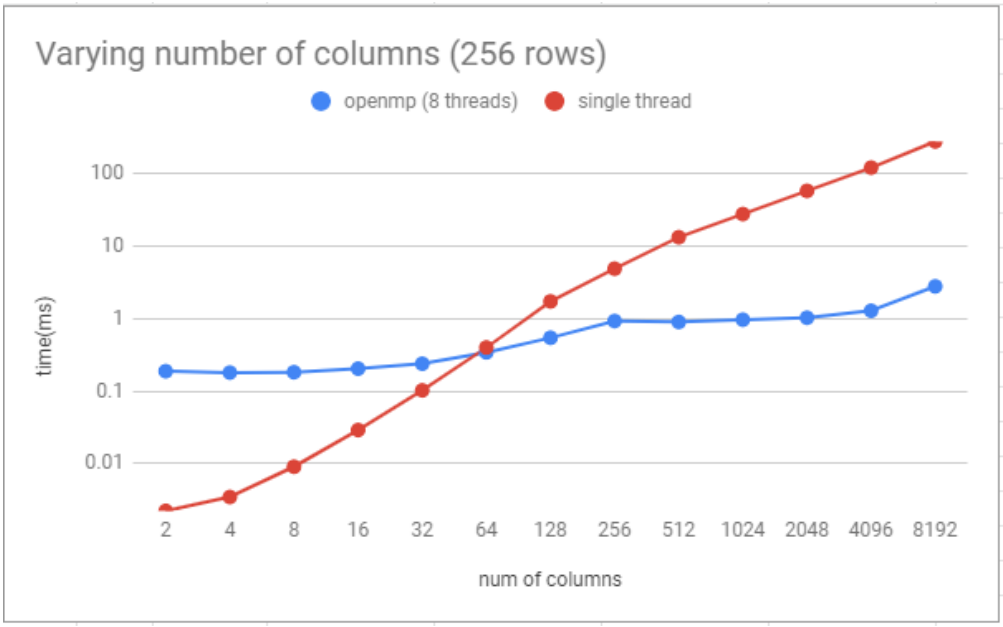

Figure 3: Comparison varying number of column for column sort

programs, however, as the size of the matrix gets large for this problem if after testing both outputs are the same for several tries, the probability that you avoided race conditions is high.

Listing 1 is an example of a first implementation I had that seems that should work correctly and efficiently, since the critical section is outside the loop.

The problem was that although the instantiation of a parallel section takes small time, if done within a large a group, the overhead becomes noticeable.

Hence I ended up wrapping everything in a single parallel section and eliminating as many implicit barriers as possibles while mantaining mutual exclusion. I followed the same approach for the blocks.

5 Getting Timing Right

In the end my biggest issue was getting timing right. Something that seems should be straight forward, and I guess that is why I overlooked it. I followed the examples given in the links:

https://www.cs.rutgers.edu/ pxk/416/notes/c-tutorials/gettime.html

http://linux.die.net/man/3/clock_gettime

Within the given clocks I decided to use **CLOCK_PROCESS_CPUTIME_ID** at the beginning of the project. My timing seemed to increase linearly with the number of threads. I overlooked this choice and tried all kinds of optimization within OpenMP. In the end I figured out this was the issue, in particular I believe that **CLOCK_PROCESS_CPUTIME_ID** accumulates the times over all threads. Very hard to understand that from the linux man page explantion:

" CLOCK_PROCESS_CPUTIME_ID

```
void openmp_sort_colum_v0(int n_threads, int m, int n, int *mat[n]){
    int maxRow = n<=m ? n : m;
    for(int i=0; i<maxRow; i++){
        int argmax_all = i, max_total = mat[i][i];
        //threading package will be instantiated maxRow times
        #pragma omp parallel shared(argmax_all, max_total) num_threads(
    n_threads)
        {
      int argmax_local = i, max_local = -1;
            #pragma omp for schedule(static,1) nowait
            for(int j=i+1;j<n; j++){
                if(max_local < mat[j][i]){
                    max_local = mat[j][i];
                    argmax_local = j;
                }
            }
            #pragma omp critical
            if(max_total < max_local){
        max_total = max_local;
        argmax_all = argmax_local;
      }
        }
        //threading package is instantiated again maxRow times
        if(i!= argmax_all) swap_rows_open_mp(n_threads, argmax_all, i, m, n,
    mat);
     }
}

void swap_rows_open_mp(int n_threads, int row_i, int row_j, int n_cols, int n,
     int *mat[n]){

     #pragma omp parallel num_threads(n_threads)
     {
        #pragma omp for schedule(static,1)
    for(int i=0; i<n_cols; i++){
      int temp = mat[row_i][i];
      mat[row_i][i] = mat[row_j][i];
      mat[row_j][i] = temp;
    }
  }
}
```

Listing 1: Correct code but inefficient

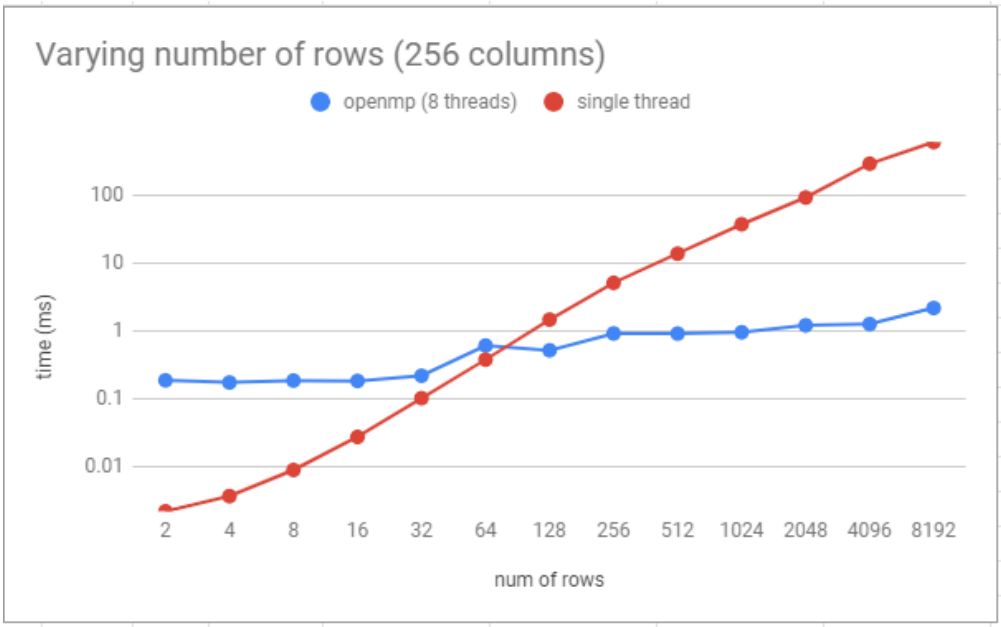

Figure 4: Comparison of varying number of rows for column sort

High-resolution per-process timer from the CPU. ”
In the I used omp_get_wtime() as recommended in class and in the manual of OpenMP.

6 Block Sorting

Fig. 6 shows the comparison between number of threads for different data placements. For cyclic placement of rows, since this doesn't take advantage of caches, it cannot beat the single threaded program at any number of threads. For row block placement, the best time was achieved by 8 threads with 16.9 s, as expected. This data placement does indeed take advantage of the L1 cache of each core.

For Fig. 7-9, the only outstanding thing is how they differ with the the previous colum sorting. Here we can see how by keep rows or columns at 256 and varying the other, as it gets larger the OpenMP program doesn't outperform that much the single threaded one. I believe it must be some syncronization issue between the threads but I would need more time to analyse it.

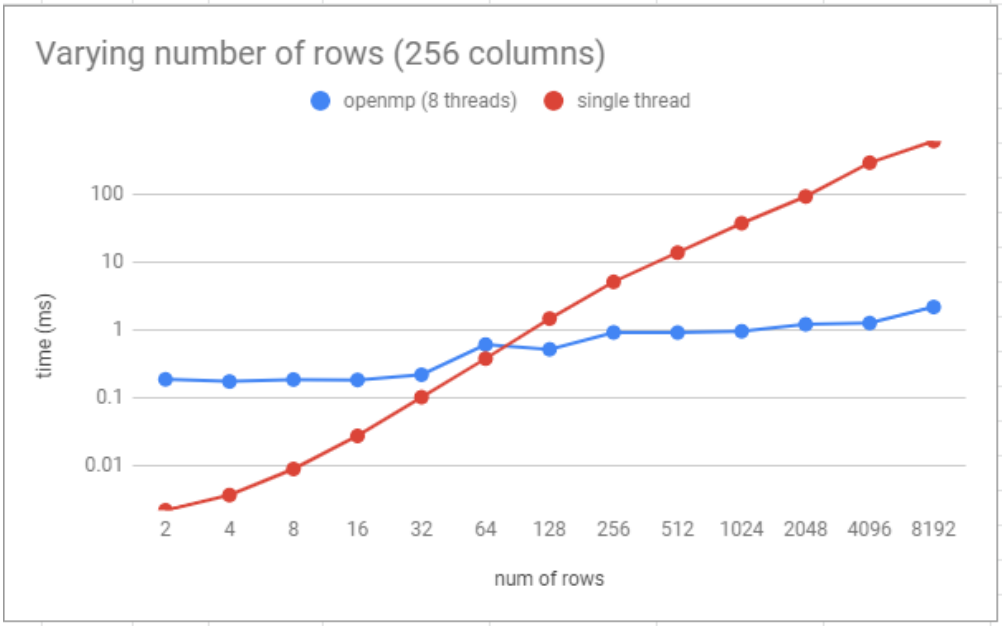

Figure 5: Comparison of varying number of rows and columns at the same time for column sort

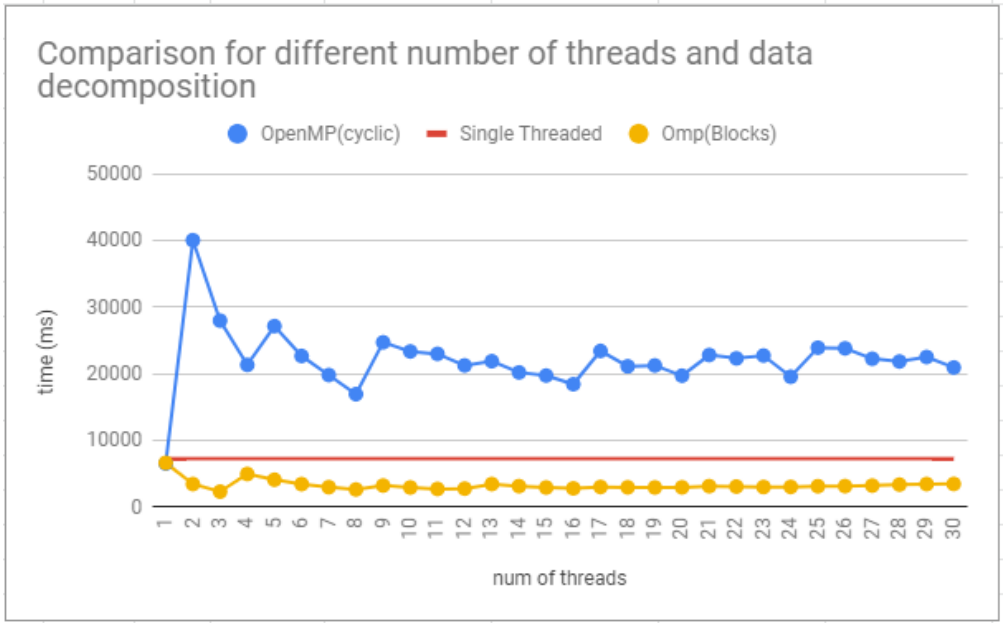

Figure 6: Comparison of executing times vs number of threads for 2 different data placements for 2Kx2K matrix for block sort

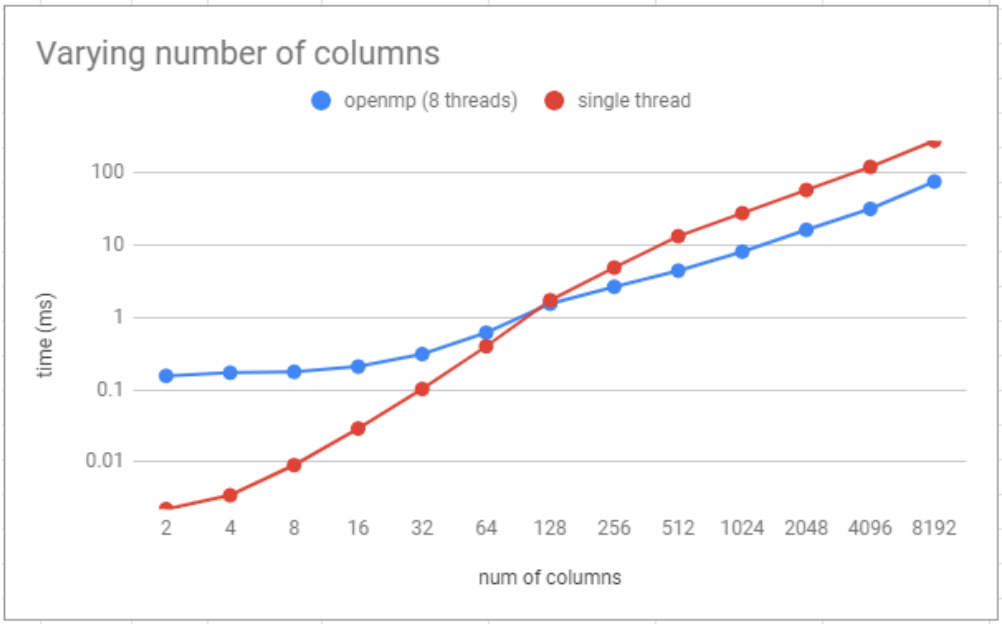

Figure 7: Comparison varying number of column for block sort

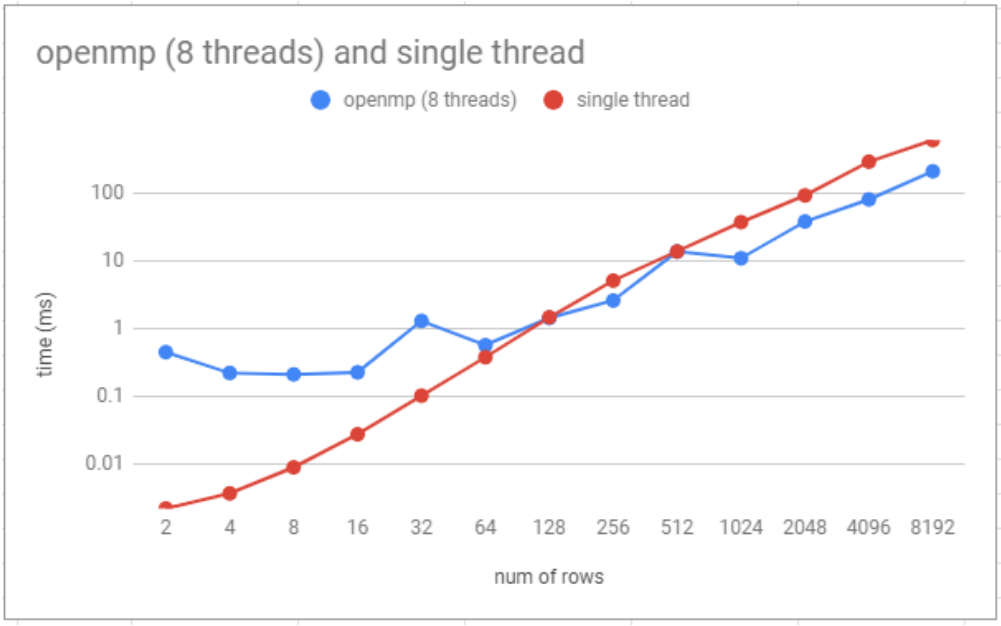

Figure 8: Comparison of varying number of rows for block sort

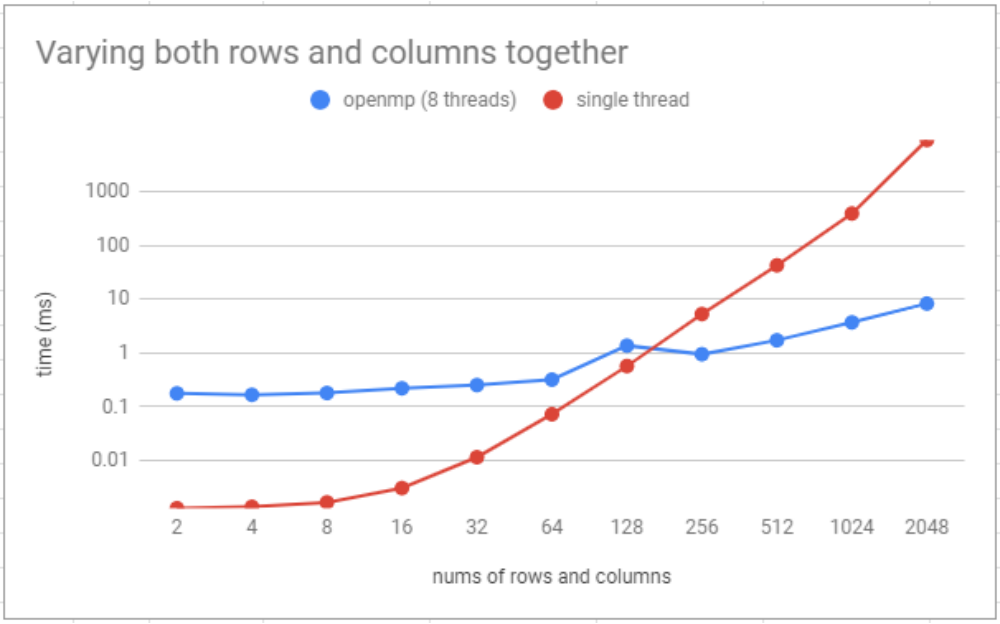

Figure 9: Comparison of varying number of rows and columns at the same time for block sort