Matrix Search Parallelization

Vitor Gonçalo Costa, m70323

Universidade de Évora
March 16, 2026

Abstract

This study aims to investigate the performance gain of a program when applying parallel programming techniques to divide operations between several threads using POSIX framework.

Presenting the Problem

We want to write a program that finds the biggest element in a **square matrix** with thousands of rows and columns, built during run time. We want to evaluate the metrics speedup, efficiency, and scalability when converting the program from sequencial to parallel computation and we should analyse how does the amount of processors and threads influences the program execution time.

The Matrix is built with the logic: $matrix[r][c] = r * m\_cols + c$, inside a nested for loop for rows and columns, where r and c represent the iteration variable for row and column respectively. This way, the output of $matrix[nrows][ncolumns]$ would always be the highest element in the matrix.

Resources Available

The machine where we will be compiling and executing the program is a MacBook Pro with an i7 dual-core processor of 2,5 GHz - This cpu has available 2 physical cores with 4 virtual cores each, which means we might be able to invoke up to 8 threads of computation.

The Sequencial Program

```c
#include <stdio.h>

const int m_rows = 30000;
const int m_cols = m_rows;
int hv = 0;
int matrix[m_rows][m_cols];

int main(void){
  printf("Populate the matrix\n");
  for(int r = 0; r < m_rows; r++){
    for(int c = 0; c < m_cols; c++){
      matrix[r][c] = r*m_cols+c;
    }
  }

  printf("Search matrix \n");
  for(int r = 0; r < m_rows; r++){
    for(int c = 0; c < m_cols; c++){
      if(matrix[r][c] > hv){
        hv = matrix[r][c];
      }
    }
  }

  printf("Done\n The biggest value found is  %d", hv);
  return 0;
}
```

The execution of the program on the right relies on a lot of memory available in order to compute such big matrix. If we pick the value of 40000 for the amount of rows and columns, and knowing that an integer requires 4 bytes of memory, we need around 6,4 GB to compute such matrix. This machine has 16 GB of LPDDR3 installed.

The current implementation takes around 5,45 seconds to be executed in this machine, and throws a bus error if m_rows is updated to 40000, which is not expected at first glance, since the system has 16GB of memory available.

This might happen due to OS imposed limits, so we allocate resources with POSIX library.

We update the above program to make use of posix_memalign() function in order to align the matrix memory with the cpu's 64 bytes cache lines. This technique is called *dynamic*

memory assignment, is usually applied to very large matrices and a requirement for *High-Performance Computation* programs.

The program was executed three times using the time command, and the average execution time was computed:

m_rows	no HPC (sec)	with HPC (sec)
30k	5,482	5,395
40k	error	8.886

Table 1: program's total time execution per amount of matrix rows.

We could now use the **time.h** library to evaluate the time spent on each operation, 1st allocate memory, 2nd building the matrix and 3rd search for the biggest value, but we save this exercise for later.

Parallel Strategies

From the three tasks presented previously, the search for the biggest value is the one that we should parallelise, therefore, a certain number of threads are spawned to split the task among them with the usage of pthread_create() and pthread_join() functions from **pthread.h** library.

With the current implementation of our program, the global variable hv, used to store the highest value found during the search operation, will be shared between threads by the time we execute a parallelised version of it. We need to take into account this *race condition* and decide on a strategy to overcome it. The usage of a pre variable lock() and after unlock() **muttex** pattern will stop the execution of threads waiting for the access to that variable, due to the nature of our problem, we update this variable to be **atomic int** instead, but this change alone actually increases the computation time up to 18,35 seconds with $m\_rows = 40k$, which again, suggests the use of **time.h** library to better understand the contribution of each parallel strategy presented in the following sections of this document. The next picture displays the time our program toke on each task after applying the implementations mentioned before:

```
1st --> Allocate aligned memory
 0.000346 sec
2nd --> Populate the matrix
 4.552563 sec
 3rd --> Search matrix
 3.964743 sec
Done
 The biggest value found is  1599999999./BetterSequencialMatrixSearch  7.22s user 1.55s system 98% cpu 8.886 total
```

From the previous image we can be sure the 3rd task is the one to parallelise. We should also notice that the changes performed to the program actually increase it's time of execution, the view operation performed on this type of variable is more resource consuming than when performed on a variable of type integer. This can be proved by analysing the time for each task on the first version of our program, populating the matrix toke around 4.5 seconds while the search matrix task toke less than 4 seconds for 40k rows. The wall-clock time of the above figure is 20.460 seconds, this representes the real time a users waits for a program to finish,

Columns Split

Since the matrix in the spotlight is a square one it doesn't really matter if we perform a split by columns or rows. What matters is the speed one of the about to be spawned threads takes to find the highest value present in the matrix. This way, threads will only perform read on hv global var and not write, which is a more time consuming computer operation.

A new function, parallel_search_cols(void *args), was defined in order to share the work between threads, it expects the start and end column indices as arguments in order to split the matrix in as many slots as needed. A second function, run_parallel_search(int

num_threads), was also defined to make the program more useful by allowing to define the amount of threads to be created.

The best execution times obtained on this scenario were with the creation of 600 to 800 threads, presenting a total cpu computing time of around 8,5 seconds on the 3rd task. The next figure computes the sum up time taken by all threads per its cardinality for both strategies presented in this document.

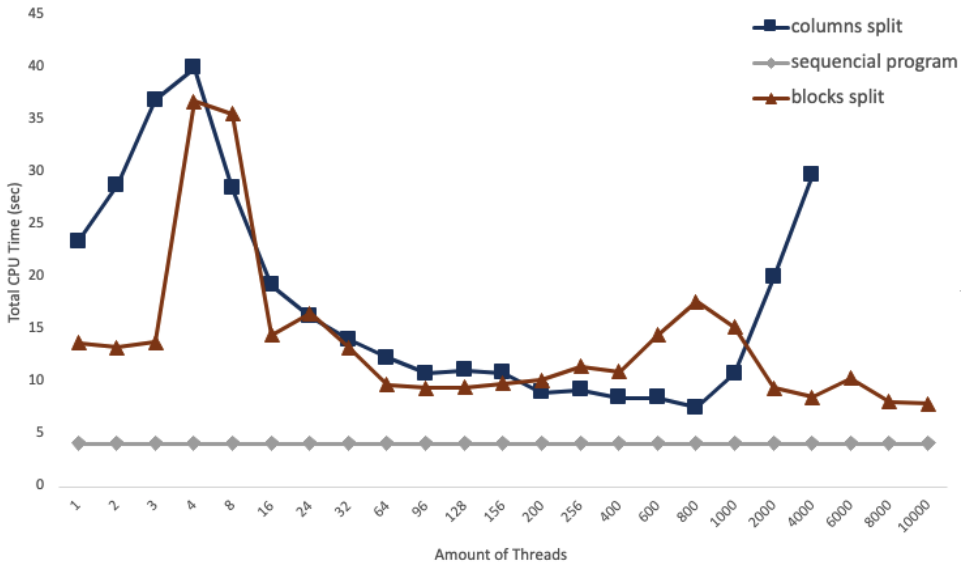

Figure 1: Time taken by all threads on the search matrix task.

A performance analysis is computed later this document, but we shall advance that the wall-clock time for 800 threads is around 8,21 seconds (around 1 millisecond per thread), which is better than the sequencial program in table 1. Next we present the strategy to split the matrix by blocks.

Blocks Split

We updated the function run_parallel_search (int num_threads), in order to divide the matrix in blocks of the same size. First it splits the matrix in two and calculate the amount of blocks for each side by the square root of the amount of blocks to be created. If we pick 400 threads for example, means we have 400 blocks to be worked out during the 3rd task. Each of this blocks has 2000 rows and columns, $\sqrt{400} = 20$,which means 20 blocks for each side of the first split, and then we can validate the amount of items in the blocks with: $20 * (2000 * 2000) * 2 = 160k$.

We also defined a new struct object to carry the start/end row and column for each block assigned to a created thread, this helps to define the block's rows and columns boundaries. The function parallel_search_block() receives this list of arguments by the time is invoked by pthread_create() and performs the comparison of matrix index with the atomic int hv variable.

From figure 1 we can advance that, in general, both strategies have similar performances, but the second solution appears to scale better than the first one. Some of the spikes present in the graph for this solution could be explained by the size of sub-matrixes to assign on threads. The amount of threads used in those cases do not divide the matrix in perfectly square sub-matrixes, all threads access the atomic hv var on each iteration, therefor some weird behaviours could happen when trying to access it's address.

Performance Analysis

In order to evaluate the performance of our program when parallelised with both of previous strategies we compute the wall-clock cpu time obtained per threads, available in table 2.

The **speedup** metric is determined by the quotient between the best time obtained by a sequential program and a parallelised version of it: $Speedup = \frac{t^*}{t_p}$.

The **efficiency** of a parallel version of a program can then be calculated by the fraction $efficiency = \frac{speedup}{p}$.

Finally, the **scalability** of a program measures how well a parallel program handles increasing levels of parallelism.

# Threads	Columns	Blocks
4	16.627	16.626
8	12.812	17.125
16	10.572	10.372
32	9.265	9.466
49	8.664	8.422
64	8.28	8.520
400	7.749	10.884
800	7.916	10.613
1000	8.126	10.560

Table 2: Wall-CPU Time (in seconds) taken for each strategy per threads used.

Combining values from tables 1 and 2, and using the formulas presented before we can compute the performance metrics for both strategies:

Strategy	#Threads	Speedup	Efficiency (%)
Columns	400	1.1467	0.007
Blocks	400	0.8164	0.2
Columns	49	1.026	2.1
Blocks	49	1.0526	2.1

Table 3: Performance of parallel strategies.

We can conclude that the parallelisation of our initial program didn't got a significant performance gain. Probably due to the nature of our problem, since the biggest value is always in the last entry of the matrix, threads will have to loop through the entire batch of items to compare values. But also because of the implementation applied, as mentioned before, the function run_parallel_search (int num_threads) could be improved in order to access the atomic variable only once per thread.

In order to evaluate scalability we have to compute the previous metrics for both strategies when the matrix size increases from 29k to 40k rows and columns. We choose to use 64 threads for this experience.

Conclusions

Modern CPUs can spawn thousands of virtual threads to perform calculations.

The action of populating the matrix could also update the value of global variable hv instead of the 3rd task. Search the matrix could have a local variable to store the highest value found and compare with the global var hv at the end.

The clock() function from time.h library is useless to compute wall-clock time and therefore should be ignored on metrics evaluations.