A parallel implementation of the Jacobi method

Giacomo Carfì 52051

July 8, 2022

Contents

1 Introduction

In this report we will analyze a possible parallel implementation of Jacobi's method: an iterative method that computes the solution of a system of linear equations Ax=b where x is a vector of length n, A is the matrix of coefficients of dimension n*n, b is the vector of known terms of length n. The method at each iteration computes a new approximation of the values of the vector x using the formula in Equation: 1.

$$x_i^{k+1} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x_j^k) \quad \forall i.i = 1, ..., n \tag{1}$$

2 Analysis and Implementation

In this section we are going to perform an analysis of possible parallel implementations of Jacobi's method and how they were implemented with native C++ threads and using FastFlow along with the sequential version. The following pseudocode explains the algorithm used: 1.

The program starts by acquiring as input from the command line the length of the matrix **n**, the number of iterations **k** and a number, 0 or 1, to enable or disable the printing of the matrix, the vector of known terms b, and vector x. In parallel versions, the number of threads **nw** to be used is also given as input. Then a matrix of dimension n*n and a vector x called *acutal_x* of dimension n are generated both with values sampled from a uniform distribution of real numbers. The vector b is computed by performing $b = A * actual\_x$ and then the vector $actual\_x$ is discarded. A new vector x is generated with values all equal to 0 which is used a starting vector for Jacobi's method. A seed is set to make the experiments repeatable.

Algorithm 1 Jacobi Method

```
1:  function JACOBI(ITERATIONS, A, B)
2:      $x \leftarrow$ initialize with all zeroes
3:      for $it = 0$ to $Iterations$ do
4:          for $i = 0$ to $A.length$ do
5:              $sum \leftarrow 0$
6:              for $j = 0$ to $A.length$ do
7:                  if $j \neq i$ then
8:                      $sum \leftarrow sum + (A[i][j] * x[j])$
9:              $new\_x[i] \leftarrow (b[i] - sum)/A[i][i]$
10:         $x \leftarrow new\_x$
```

2.1 Analysis

Since the data to be processed does not come from a stream, and in output we produce data of the same size as the input, the parallel pattern used is a **data parallel pattern**. In particular, this is a **map** data parallel computation that is repeated for a given number of iterations. The general way this pattern works is to divide the data into chunks, each chunk is given to a worker who computes the partial result and all partial results are combined to generate the final output. The purpose of using this parallel pattern is to **minimize the completion time** T_c, in this case of the single iteration, compared to the time spent in executing the iteration sequentially.

A preliminary experiment was conducted to determine the completion time of the sequential version of the program by fixing the number of iterations to 100 and using different matrix sizes. For small matrices the grain of computation becomes small making parallel computation disadvantageous since the time spent on the setup of parallel computation would be greater than the time spent on the actual computation of the method. To demonstrate this the experiments are performed with matrices of size *500*500, 5000*5000, 20000*20000.* Performing experiments with even larger matrices would lead to memory problems since they could not be contained entirely in memory in the machine on which the experiments are performed. After running the method sequentially, we can define the **ideal completion time** $T_{id}(nw)$ as $T_{id}(nw) = T_{seq}/nw$ where T_{seq} is the completion time of the sequential program and nw is the number of threads used. Given the capabilities of the machine provided, the maximum number of threads used is 32. Table 1 shows the results where the time is expressed in microseconds.

Ideal Completion Time			
Nw	**500**	**5000**	**20000**
1	37605	3956879	64037051
2	18802	1978439	32018525
4	9401	989219	16009262
8	4700	494609	8004631
16	2350	247304	4002315
32	1175	123652	2001157

Table 1: With $nw = 1$ we denote the sequential completion time spent by the program for matrices of size 500*500, 5000*5000, 20000*20000. For $nw > 1$ we denote the ideal completion time that a parallel computation should achieve using nw workers. The time is expressed in microseconds.

Given the nature of the problem, execute a parallel implementation by running the fork and the join of the threads at each iteration would lead to large overheads. The time spent on thread setup can be quantified by compiling and running the file *test_threadsetup.cpp* passing as an argument the number of threads to be created. The time spent setting up 2 threads is about 160μs while the time spent setting up 32 threads is about 1324μs. This time is not negligible considering that for matrices of size 500*500 the cost of a single iteration is about 376μs. Also, using a threadpool would require waiting time in filling the queue of available tasks. So no threadpool is used, the best choice would be to keep threads active until there are no more iterations left to run.

An initial implementation of the parallel version then could be done by performing a splitting of the matrix A by rows, assigning a row of the matrix, vector x, and vector b to each thread in a cyclic manner. However, this implementation does not make use of the **spatial locality principle** of the caches since each thread would read non-contiguous addresses from memory and lead to overheads.

To reduce these overheads a second implementation may be done by splitting the matrix into chunks of contiguous rows of size approximately n/nw where n is the lenght of the matrix. This implementation makes use of the spatial locality principle but could brings some overhead related to load balancing since the last chunk might have larger size than the others and thus the last thread would execute more work than the others. Also, chunks that are too large in size may not be contained entirely in the caches.

Additional overheads in the completion time of the parallel version of the program may be due to the migration of thread execution from one core to another to avoid overheating the cores. This creates delays in moving the working set from one cache to another. So the final parallel implementation of the program using Native C++ Thread is then done by splitting the matrix into chunks and by pinning the execution of the threads to the cores. We then provide more specific information for the various possible implementations and later we make a comparison between them.

2.2 Sequential Implementation

The sequential implementation follows the pseudocode shown in the algorithm 1. The sequential version was used to establish a baseline of the program's performance in comparison with parallel versions.

2.3 Native C++ Threads Implementation

Various implementations have been compared to shows overhead for the parallel version of the program using C++ Native Threads. In the first implementation the matrix A is divided by rows, each thread is then assigned one row, the entire vector b and the entire vector x in a cyclic way. Each thread performs the computation by reading the values from the given vector x and writing the new values to a temporary vector called new_x, this is done to prevent threads from overwriting old values of x that might be needed for other threads' computation. In the second implementation the matrix A is divided by chunks of contiguous rows of size approximately n/nw. Each chunk is assigned to a thread, the rest of the implementation remains the same. The final implementation is the same as the second parallel version, dividing the matrix by chunks, but by pinning the execution of the threads on the cores. All the implementation makes use of the **std::barrier** class to perform thread synchronization. At the end of their computation, the threads remain in a waiting state in the barrier, when the last thread finishes its computation, the *on_completion* method 4 is called so that the new computed values in *new_x* are copied into the vector x and the counter of remaining iterations is decreased. At this point a new iteration is executed.

The first implementation is shown in pseudocode in the Algorithm 3, the second version is shown in the Algorithm 2.

2.4 FastFlow Implementation

The implementation with FastFlow uses the **ff::ParallelFor** object that is used in the library to implement the map data parallel pattern. ParallelFor is used with *Spinwaits* rather than *locks* to speed up computation especially in cases where the grain is small by setting the appropriate flags. In fact in case of very short waits, spinning may be preferable to blocking as it avoids context switch overhead. Experiments with FastFlow are performed by setting the value for chunksize to 0 so that the chunk size assigned to each thread is approximately n/nw. Since FastFlow uses ThreadPinning and since the parameter for the chunksize passed is 0, this implementation follows the final implementation done with Native C++ Threads.

2.5 Other implementation details

The methods for creating the matrix A, vector x, for computing vector b, and for printing the matrix and vectors to the console were implemented in the *utility.h* library under utils directory, which is

Algorithm 2 Parallel Chunks

```
$delta \leftarrow n/nw$
for $i = 0$ to $nw$ do
    $ranges[i] \leftarrow$ assigns pairs of indices at distance delta from each other

function ThreadBody(tid, ranges[$l$])
    while $Iterations > 0$ do
        for $i = ranges.first$ to $ranges.second$ do
            $sum \leftarrow 0$
            for $j = 0$ to $A.length$ do
                if $j \neq i$ then
                    $sum \leftarrow sum + (A[i][j] * x[j])$
            $new\_x[i] \leftarrow (b[i] - sum)/A[i][i]$
        waits on the barrier
```

Algorithm 3 Parallel Cyclic

```
function ThreadBody(tid)
    while $Iterations > 0$ do
        for $i = tid$ to $n, i+ = nw$ do
            $sum \leftarrow 0$
            for $j = 0$ to $A.length$ do
                if $j \neq i$ then
                    $sum \leftarrow sum + (A[i][j] * x[j])$
            $new\_x[i] \leftarrow (b[i] - sum)/A[i][i]$
        waits on the barrier
```

Algorithm 4 Barrier Callback

```
function on_completion()
    $iterations \leftarrow iterations - 1$
    $x \leftarrow new\_x$
```

linked in every version of the program used. Also in this directory *utimer.cpp* library is present, which is used to compute execution times.

3 Measures

Let's define T_{seq} as the completion time spent by the sequential version of the program while T_{par} indicates the completion time spent by the parallel version. To measure the performance of the various implementations and compare the parallel versions with the sequential one, the measures considered are:

1. **Speedup** $Sp(nw) = \frac{T_{seq}}{T_{par}(nw)}$ It is the ratio between the best known sequential time and parallel execution time, where nw is the parallelism degree. It measures how good is the parallelization with respect to the best sequential computation.

2. **Scalability** $Sc(nw) = \frac{T_{par}(1)}{T_{par}(nw)}$ It is the ratio between the parallel execution time with parallelism degree 1 and parallel execution time with parallelism degree nw. It measures how efficient is the parallel implementation in achieving better performances on larger parallelism degrees.

3. **Efficiency** $\varepsilon(nw) = \frac{Sp(nw)}{nw}$ It is the ratio between the ideal execution time and the actual execution time. The efficiency measures the ability of the parallel application in making a good usage of the available resources.

The completion time considered, whether parallel or sequential, does not take into account the time spent to generate the random matrix A, random vector x, and the computation of b. The completion time spent in the parallel version using Native C++ Threads also considers the time spent on thread setup which includes the time spent creating threads, waiting for them to terminate, and deleting the resources assigned to them.

4 Experiments

The choice made for the experiments is to make a comparison between the various implementations to show the overheads and how they are mitigated. Experiments are performed by setting the number of iterations k to **100**. They are done with matrices of different sizes, and each experiment is repeated 5 times by averaging the completion times obtained to get more reliable results.

The complete list of experiments is therefore:

1. Sequential
   - 1.1. Experiments with matrices of size 500*500, 5000*5000, 20000*20000.
2. Native C++ Threads
   - 2.1. Experiments with matrices of size 500*500, 5000*5000, 20000*20000. The matrix is divided into chunks of continuous rows of size about n/nw.
   - 2.2. Experiments with matrices of size 500*500, 5000*5000, 20000*20000. The matrix divided by rows, each row assigned to a thread cyclically.
   - 2.3. Experiments with matrices of size 500*500, 5000*5000, 20000*20000. The matrix is divided into chunks of continuous rows of size about n/nw with use of thread pinning.
3. FastFlow
   - 3.1. Experiments with matrices of size 500*500, 5000*5000, 20000*20000. The matrix is divided into chunks of continuous rows of size about n/nw.

Individual experiments are performed by compiling and running the respective files and passing appropriate values as arguments. Information on how to run the experiments are:

1. ./sequential.out *matrix_size number_of_iterartions check_flag*
2. ./parallel_chunks_barrier.out *matrix_size number_of_iterartions number_of_threads check_flag*
3. ./parallel_row_cyclic_barrier.out *matrix_size number_of_iterartions number_of_threads check_flag*
4. ./parallel_chunks_barrier_threadpinned.out *matrix_size number_of_iterartions number_of_threads check_flag*
5. ff_parallel.out *matrix_size number_of_iterartions number_of_threads check_flag*

Where check_flag is an integer, if check_flag is equal to 0 then the matrix, vector b, and computed vector x are not printed on the console, otherwise they are printed.

Files are compiled using the **-O3** flag to achieve the best possible sequential completion time and the **-pthread** flag for parallel versions. FastFlow, utimer and utility libraries are included using appropriate flags. A makefile is created to perform the compilation of individual files, and to compile all files one after the other with the *make all* rule. Another rule is *make clean* to delete compiled .out files. A bash script called *execute.sh* was created to run all the experiments one after another. The completion times for the sequential version of the program are saved in the file called *sequential.csv*, the completion times for the parallel versions are saved in the file called *parallel.csv*, both under the results directory. An additional script in python called *compute_stats.py* was created to generate plots related to completion time. The script also computes speedup, scalability and efficiency for the various experiments and creates respective csv files.

The experiments were conducted on servers provided by the University of Pisa that use an Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz 32 cores CPU.

5 Results

The results show that considering the size of the matrix is very important. For small matrices of dimension 500*500, the grain of the computation is small, making the completion time spent in the parallel version of Jacobi's method, when many threads are used, to be affected by overheads such as overhead due to the time spent in thread setup. Using this size for the matrix, the best completion time is achieved using only 8 threads instead of 32.

For matrix sizes 5000*5000 and 20000*20000 the parallel version computation shows benefits on completion time. The Tables 2, 3, 4 shows the average completion times for sequential and parallel versions of the program, where time is expressed in microseconds.

The parallel version that makes use of cyclic scheduling by assigning a row of the matrix to each thread, performs worse in most cases than the chunks partitioning of the matrix as expected, this is most likely because the spatial locality principle is not exploited. The parallel version that uses chunks of rows and fixes the computation of threads on cores brings some very small benefits on computation. The pictures in Figure 1 show plot of the completion times spent.

Matrix Size 500*500					
Nw	**Seq** (μs)	**Chunks** (μs)	**Chunks TP** (μs)	**Cyclic** (μs)	**FF** (μs)
1	**37605**	39616	40507	39679	38265
2	–	**21293**	21871	21709	21524
4	–	**13337**	13465	14191	17117
8	–	9922	**9690**	10423	11826
16	–	10806	18181	11330	**10687**
32	–	17092	19088	18530	**13996**

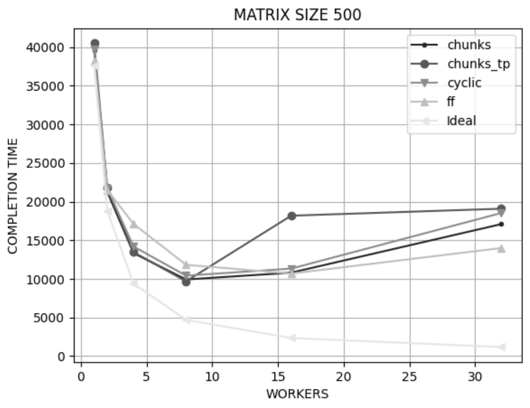
Table 2: The completion time obtained on matrices of size 500*500 is shown. The time is expressed in microseconds.

Matrix Size 5000*5000					
Nw	**Seq** (μs)	**Chunks** (μs)	**Chunks TP** (μs)	**Cyclic** (μs)	**FF** (μs)
1	**3956879**	4073578	4052814	4013054	3960447
2	–	**2030399**	2178349	2113809	2054614
4	–	**1072207**	1109690	1165945	1485872
8	–	**548132**	561634	597758	759381
16	–	312132	**303105**	320360	394984
32	–	250330	**243633**	276213	264906

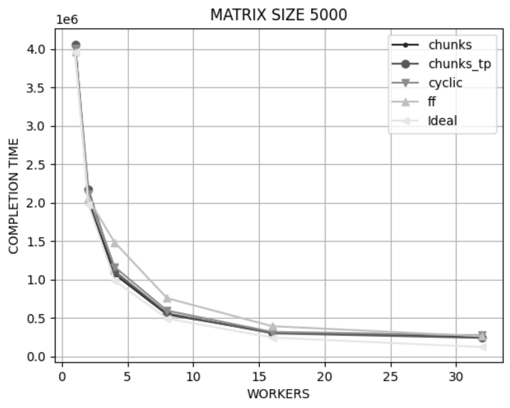
Table 3: The completion time obtained on matrices of size 5000*5000 is shown. The time is expressed in microseconds.

Matrix Size 20000*20000					
Nw	**Seq** (μs)	**Chunks** (μs)	**Chunks TP** (μs)	**Cyclic** (μs)	**FF** (μs)
1	**64037051**	64438852	64459108	64100651	63498428
2	–	33278299	34656469	34817614	**32062025**
4	–	**17334195**	17356837	17676812	23593877
8	–	**8614508**	8647586	9241722	11846922
16	–	4492213	**4410797**	4756648	6035434
32	–	3538047	3515908	**3514639**	3710581

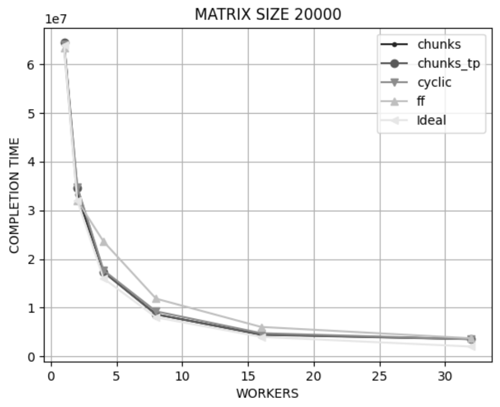
Table 4: The completion time obtained on matrices of size 20000*20000 is shown. The time is expressed in microseconds.



(a) Completion Time Matrix Size 500.



(b) Completion Time Matrix Size 5000.



(c) Completion Time Matrix Size 20000.

Figure 1: The plots show the completion times of each experiment for different sizes of the A matrix. The ideal completion time is also included in the plot to show the differences with the actual completion time achieved.

Tables 5, 6, 7 show the speedup obtained for the various experiments. Here we can clearly see the benefits gained from parallel computing, especially when the grain of computation is increased. The ideal speedup is obtained when **Sp(nw)=nw** where *nw* is the number of worker used. This is also called **Linear Speedup**. Linear speedup is achieved more or less up to the use of 16 threads with matrix sizes 5000*5000 and 20000*20000. When we use 32 cores non negligible overheads are present. One of the possible overheads, in addition to the time spent on thread setup, is given by the **cache coherence protocol** which is activated by the system to make the vector *new_x* coherent between the various caches. In fact, whenever a thread writes to the vector, even if each thread modifies only one part of the vector without accessing the parts where the other vectors are operating, the new_x vector is made consistent with respect to all other threads by the cache coherence protocol by copying the vector to the caches of all cores that are performing the computation of a thread, this invevitably generates overheads. The Figures in 2 also show plots related to the speedup.

Tables 8, 9, 10 provide information on the scalability obtained from the various parallel models. As for the speedup we have ideal scalability when Sc(nw)=nw.

Tables 11, 12, 13 provide information on the efficiency of the program, obtained from the various parallel models. Parallel implementations with FastFlow and using thread pinning get the best efficiency most of the time, although it may not correspond to the best completion time.

Nw	Chunks	Chunks TP	Cyclic	FF
1	0.949	0.928	0.948	0.983
2	1.766	1.719	1.732	1.747
4	2.82	2.793	2.65	2.197
8	3.79	3.881	3.608	3.18
16	3.48	2.068	3.319	3.519
32	2.2	1.97	2.029	2.687

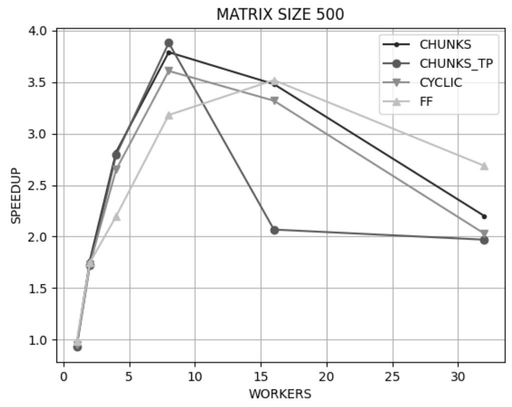
Table 5: Speedup Matrix size 500.

Nw	Chunks	Chunks TP	Cyclic	FF
1	0.971	0.976	0.986	0.999
2	1.949	1.816	1.872	1.926
4	3.69	3.566	3.394	2.663
8	7.219	7.045	6.62	5.211
16	12.677	13.054	12.351	10.018
32	15.807	16.241	14.325	14.937

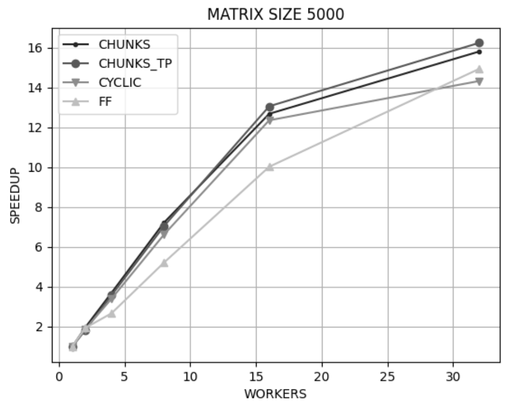
Table 6: Speedup Matrix size 5000.

Nw	Chunks	Chunks TP	Cyclic	FF
1	0.994	0.993	0.999	1.008
2	1.924	1.848	1.839	1.997
4	3.694	3.689	3.623	2.714
8	7.434	7.405	6.929	5.405
16	14.255	14.518	10.610	14.721
32	18.1	18.214	18.220	17.258

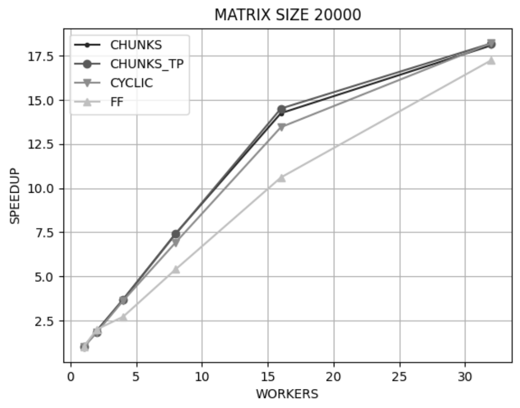
Table 7: Speedup Matrix size 20000.



(a) Speedup Matrix Size 500.



(b) Speedup Matrix Size 5000.



(c) Speedup Matrix Size 20000.

Figure 2: The plots show the speedup achieved for different sizes of the A matrix and for different parallelism degree.

Nw	Chunks	Chunks TP	Cyclic	FF
1	1.0	1.0	1.0	1.0
2	1.861	1.852	1.828	1.778
4	2.97	3.008	2.796	2.235
8	3.993	4.18	3.807	3.236
16	3.666	2.228	3.502	3.580
32	2.318	2.122	2.141	2.734

Table 8: Scalability Matrix size 500.

Nw	Chunks	Chunks TP	Cyclic	FF
1	1.0	1.0	1.0	1.0
2	2.006	1.860	1.898	1.928
4	3.799	3.652	3.442	2.665
8	7.432	7.216	6.714	5.215
16	13.051	13.371	12.527	10.027
32	16.273	16.635	14.529	14.95

Table 9: Scalability Matrix size 5000.

Nw	Chunks	Chunks TP	Cyclic	FF
1	1.0	1.0	1.0	1.0
2	1.936	1.860	1.841	1.980
4	3.717	3.714	3.626	2.691
8	7.48	7.454	6.936	5.360
16	14.345	14.614	13.476	10.521
32	18.213	18.334	18.238	17.113

Table 10: Scalability Matrix size 20000.

Nw	Chunks	Chunks TP	Cyclic	FF
1	0.949	0.928	0.948	0.983
2	0.883	0.860	0.866	0.874
4	0.705	0.698	0.662	0.549
8	0.474	0.485	0.451	0.398
16	0.218	0.129	0.207	0.220
32	0.069	0.062	0.063	0.084

Table 11: Efficiency Matrix size 500.

Nw	Chunks	Chunks TP	Cyclic	FF
1	0.971	0.976	0.986	0.981
2	0.974	0.908	0.936	0.946
4	0.922	0.892	0.848	0.913
8	0.902	0.881	0.828	0.893
16	0.792	0.816	0.772	0.848
32	0.494	0.508	0.448	0.422

Table 12: Efficiency Matrix size 5000.

Nw	Chunks	Chunks TP	Cyclic	FF
1	0.994	0.993	0.999	1.000
2	0.962	0.924	0.920	0.986
4	0.924	0.922	0.906	0.940
8	0.929	0.926	0.866	0.937
16	0.891	0.907	0.841	0.920
32	0.566	0.569	0.569	0.453

Table 13: Efficiency Matrix size 20000.

6 Conclusions

An analysis of possible parallel implementations of Jacobi's method has been proposed. Considering various possible overheads, the analysis led to the choice of an implementation using chunks and thread pinning for FastFlow and Native C++ Threads. Experiments are performed considering all possible implementations to show the overheads and how they are mitigated. The experiments were conducted using matrices of different sizes, since for small matrices the computation of the method in parallel version would have high overheads. For larger matrices, linear speedup can be more or less maintained by using up to a maximum of 16 threads. Efficiency also remains good using up to 16 threads by obtaining values around 80%. Using 32 threads, splitting the matrix by rows and by chunks brings similar results on completion time and speedup for matrices of size 20000*20000. The use of 32 threads brings benefits in completion time although there are obvious overheads. We therefore conclude that parallelization of Jacobi's method brings benefits on completion time.