COSC 6365
Performance analysis of Jacobi and Gauss-Seidel iterative methods

Radmir Sultamuratov, Yerbol Palzhanov

12 December, 2021

1 Introduction

In numerical analysis, Jacobi and Gauss - Seidel methods are iterative methods to linear algebraic system of equations. For both methods sufficient (but not necessary) condition for the methods to converge is that the matrix A in the system A $* x =$ b is strictly diagonally dominant. We implement OpenMP/C library to analyze performance of parallelization of these two methods on the server `STAMPEDE2`.

2 Overview of methods

2.1 Jacobi iterative method [1]

Jacobi iterative method (JIM) is one of the easiest iterative methods for solving a system of linear equations

$$A * \vec{x} = \vec{b}$$

For any equation, the i^{th} equation

$$\sum_{j=1}^{N} a_{ij} x_j = b_i$$

we solve for the value x_i while assuming that the other entries of $\vec{x} = (x_1, x_2, x_3, \cdots, x_N)^T$ remain fixed and hence we obtain

$$x_i = \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{N} a_{ij} x_j \right) / a_{ii}$$

This suggests an iterative method by

$$x_i^{(k)} = \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{N} a_{ij} x_j^{(k-1)} \right) / a_{ii}$$

where $x_i^{(k)}$ means the value of k^{th} iteration for unknown x_i with $k = 1, 2, 3, \cdots$, and $\vec{x}^{(0)}$ is an initial guess vector, e.g., we can guess that

$$\vec{x}^{(0)} = (0, 0, 0, \cdots, 0)^T$$

This is so called Jacobi's method. Note that the order in which the equations are examined is irrelevant. **That's why we can compute them parallelly each equation in each iteration.**

2.2 Gauss-Seidel iterative method[2]

For Gauss-Seidel method if we proceed as with the Jacobi method, but now assume that the equation are examined one at a time in sequence, and that previously computed results are used as soon as they are available, we obtain the Gauss-Seidel method:

$$x_i^{(k)} = \left(b_i - \sum_{j<i} a_{ij} x_j^{(k)} - \sum_{j>i} a_{ij} x_j^{(k-1)} \right) / a_{ii}$$

Gauss - Seidel method is **sequential** by nature in general. However, Gauss - Seidel method can be **efficiently parallelized for some sparse matrices.** For example while solving `PDEs` by Finite Element Methods, the stiffness matrix appears to be sparse tri-diagonal or *some*-diagonal matrix, which still leaves for parallelization of Gauss-Seidel method[3].

3 Methodology

3.1 Choice of matrix and parameters

Again we implement Jacobi and Gauss-Seidel methods to solve a system of linear algebraic equations $A * \vec{x} = \vec{b}$ for $\vec{x} \in R^N$. The matrix should be diagonally dominant in order to guarantee convergence of the methods. For that reason we for both methods choose matrix $A = \{a_{ij}\} \in R^{N \times N}$ and vector $\vec{b} \in R^N$ in the following way

$$a_{ij} = \begin{cases} (i+j)/N & i \neq j \\ N^2 & i = j \end{cases}$$

$$\vec{b} = A * \vec{1}$$

where N is the array size. For array size we considered N $\in \{32, 128, 512, 2048, 8192\}$.

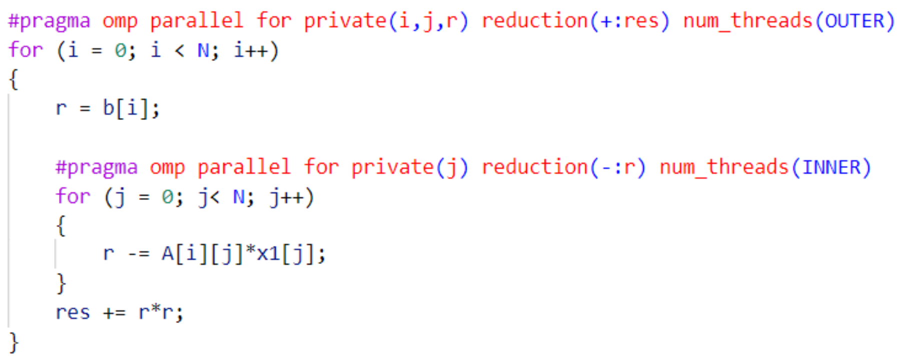
3.2 Parallelization of Jacobi Method

In the sequential version of code of Jacobi methods we have written we have two single loops and two nested loops. All those loops can be parallelized which we did in the following way.

Main iteration loop

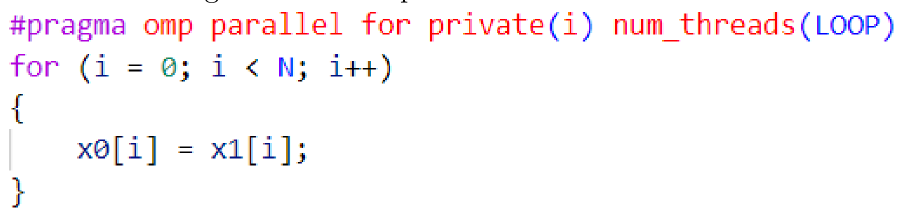
```
#pragma omp parallel for private(i,j,sum) num_threads(OUTER)
for (i = 0; i < N; i++)
{
    sum = 0;

    #pragma omp parallel for reduction(+:sum) num_threads(INNER)
    for (j = 0; j < N; j++)
    {
        if (i != j)
        {
            sum += A[i][j]*x0[j];
        }
    }
x1[i] = (b[i]-sum)/A[i][i];
}
```

Residual calculation loop



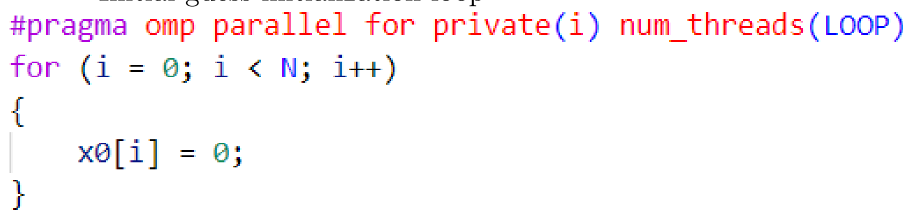
```
#pragma omp parallel for private(i,j,r) reduction(+:res) num_threads(OUTER)
for (i = 0; i < N; i++)
{
    r = b[i];

    #pragma omp parallel for private(j) reduction(-:r) num_threads(INNER)
    for (j = 0; j< N; j++)
    {
        r -= A[i][j]*x1[j];
    }
    res += r*r;
}
```

Passage of vector loop



```
#pragma omp parallel for private(i) num_threads(LOOP)
for (i = 0; i < N; i++)
{
    x0[i] = x1[i];
}
```

Initial guess initialization loop



```
#pragma omp parallel for private(i) num_threads(LOOP)
for (i = 0; i < N; i++)
{
    x0[i] = 0;
}
```

We have three parameters OUTER, INNER, LOOP as number of threads we split the outer, inner and single loops into. Also we have parameter SIZE which is the size N of the arrays. We ran the codes using TAU with the following values of the parameters

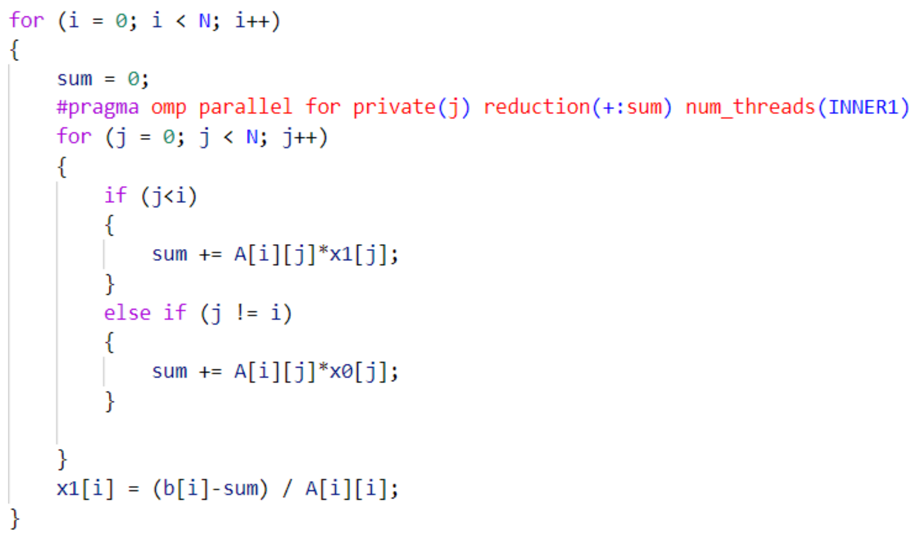
OUTER	1 2 4 8 12
INNER	1 2 4 8 12
LOOP	1 2 4 8
SIZE	32 128 512 2048 8192

3.3 Parallelization of Gauss-Seidel Method

The case of the Gauss-Seidel method is more difficult in terms of parallelization. The Gauss-Seidel method code we have written as the Jacobi method code also has two nested and two single loops. Residual calculation, passage of vector and initial guess initialization loops are identical with the Jacobi code so we added the same pragma directives with the same parameters. However the main iteration loop is very much different.

The outer loop has to be sequential as it has dependency for the vector $x1$. Also we decided to use if statement to combine two loops with usage of $x0$ and $x1$ into one loop so the inner loop could be parallelized.

Main iteration loop (Gauss-Seidel)



```
for (i = 0; i < N; i++)
{
    sum = 0;
    #pragma omp parallel for private(j) reduction(+:sum) num_threads(INNER1)
    for (j = 0; j < N; j++)
    {
        if (j<i)
        {
            sum += A[i][j]*x1[j];
        }
        else if (j != i)
        {
            sum += A[i][j]*x0[j];
        }

    }
    x1[i] = (b[i]-sum) / A[i][i];
}
```

We have parameters INNER1 for the inner loop of the main iteration, INNER2, OUTER2 for residual calculation nested loops, and LOOP for single loops. And we have parameter SIZE here as well. We ran the codes using TAU with the following values of the parameters

INNER1	1 2 4 8 16 48 96
INNER2	1 2 4 8 12
OUTER2	1 2 4 8 12
LOOP	1 2 4 8
SIZE	32 128 512 2048 8192

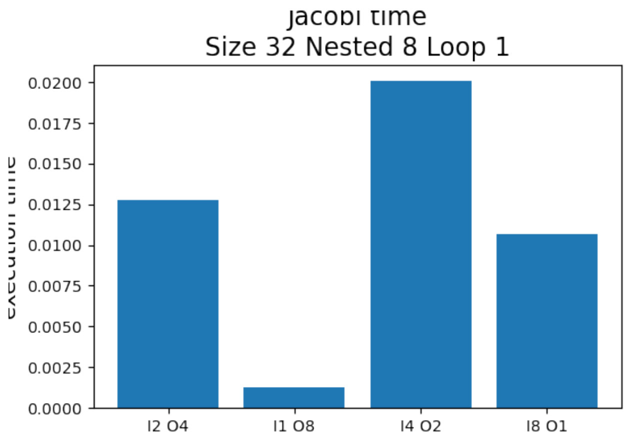
3.4 Metrics

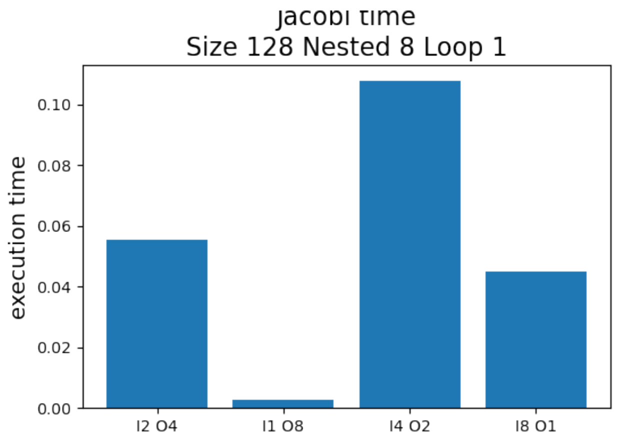
For all cases we collected reports on execution time and total cache (L1,L2,L3) misses using TAU metrics `PAPI_L1_TCM; PAPI_L2_TCM ; PAPI_L3_TCM`.

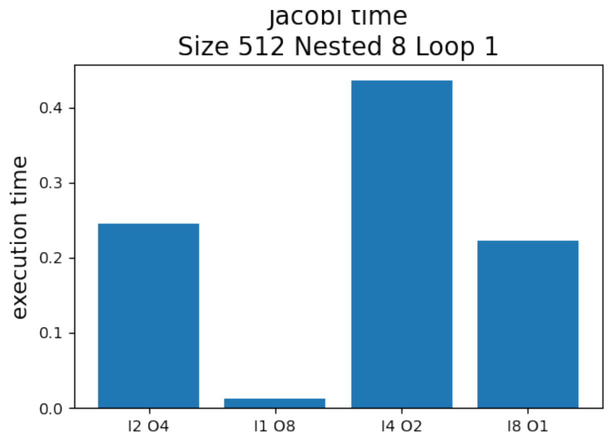
4 Results and Analysis on Jacobi Method

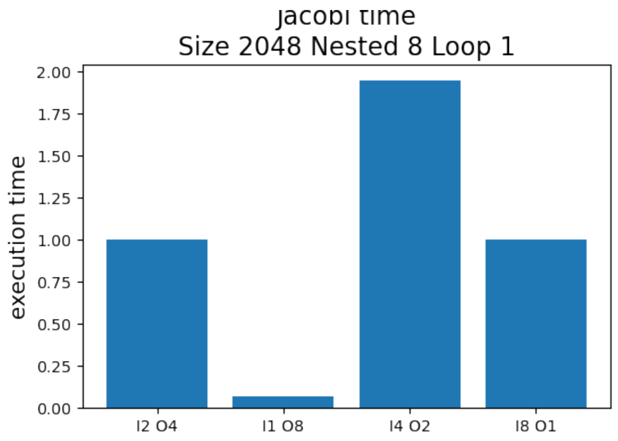
4.1 Execution time

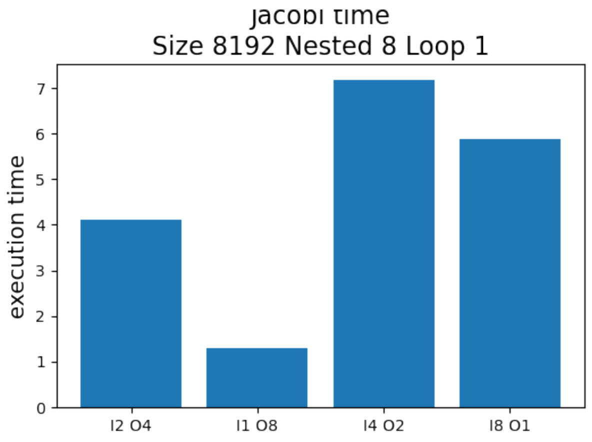
Implementation of Jacobi Iterative Method (JIM) has two nested loops while re-calculating x in each iteration. Graphs below report the execution time of JIM using 8 threads with different matrix sizes. Allocating more threads for outer loop reduces the execution time.



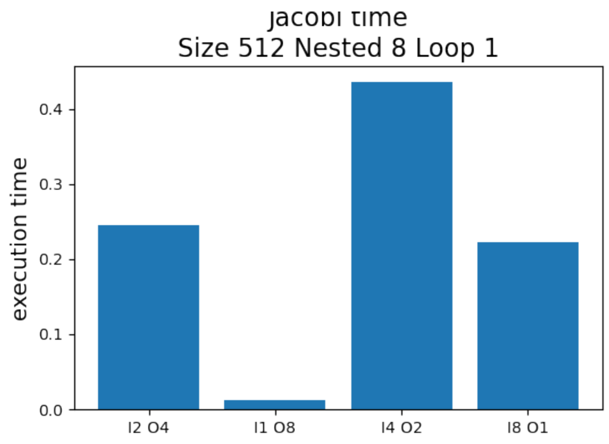


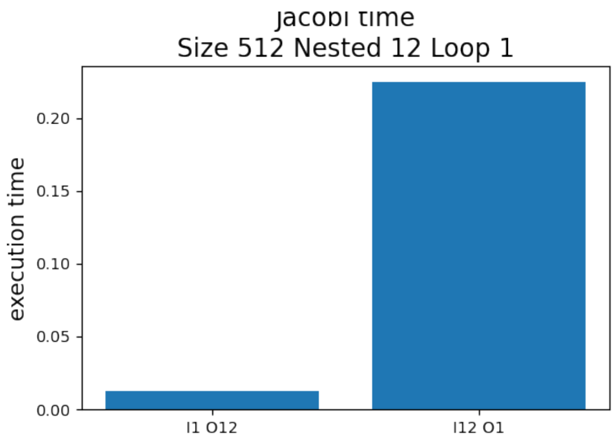


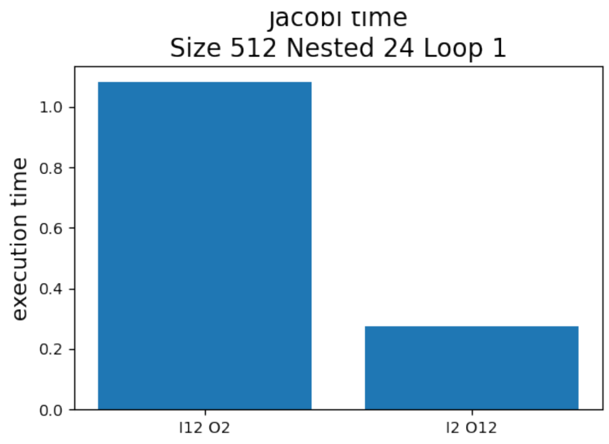


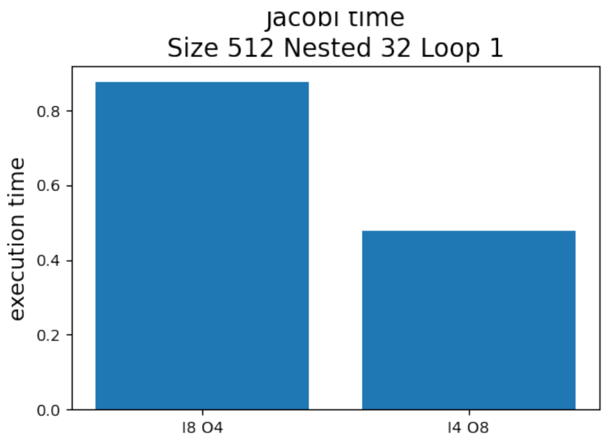


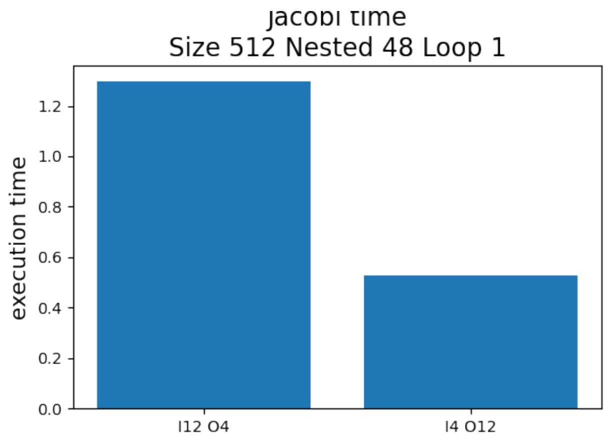
Same kind of correlation can be reported with different thread numbers while fixing matrix size. For example, $N = 512$



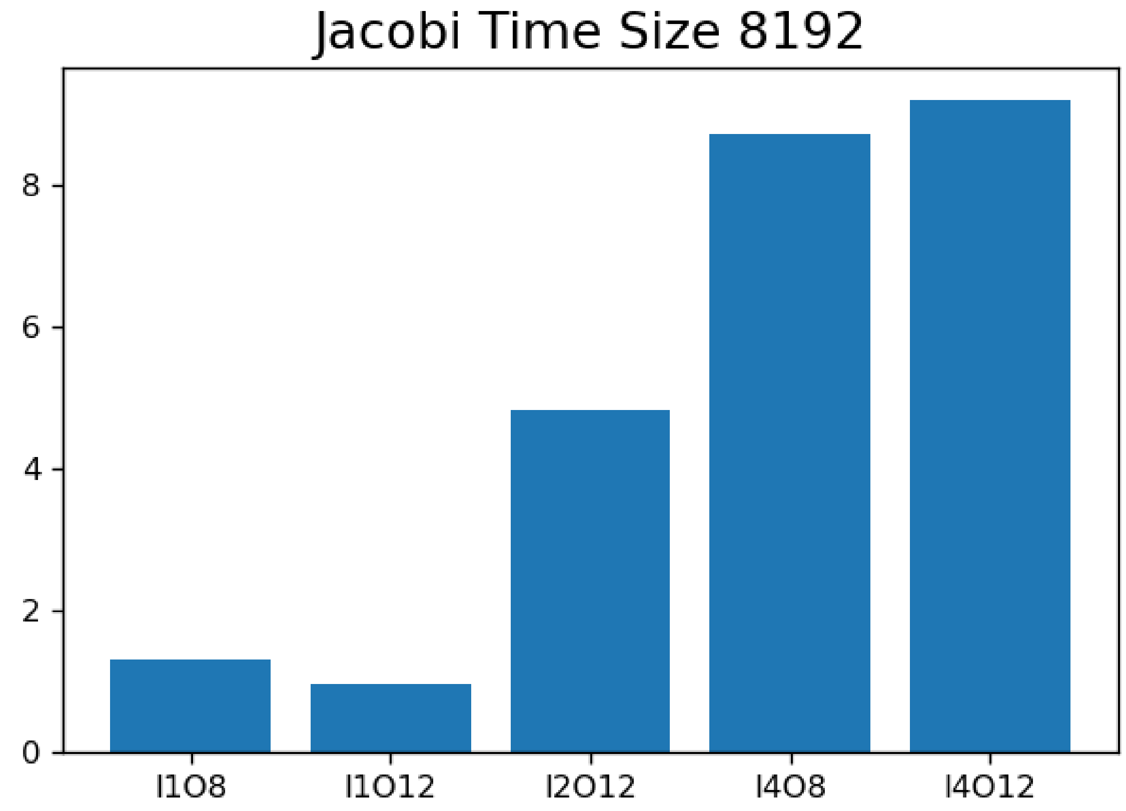






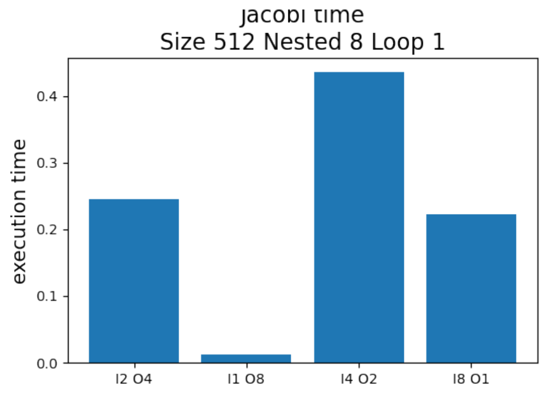


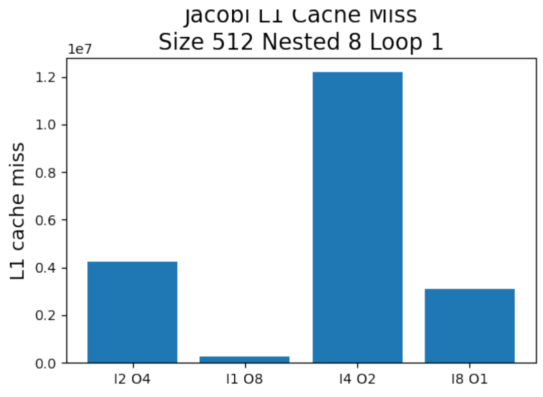
The graph below is a combination of best performances of JIM with 8192 $\times$ 8192 matrix and different thread distributions. Allocating more threads for outer loop makes execution faster, however we see the opposite for inner loop threads.

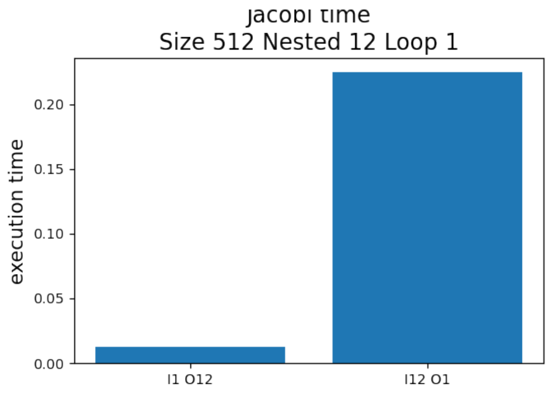


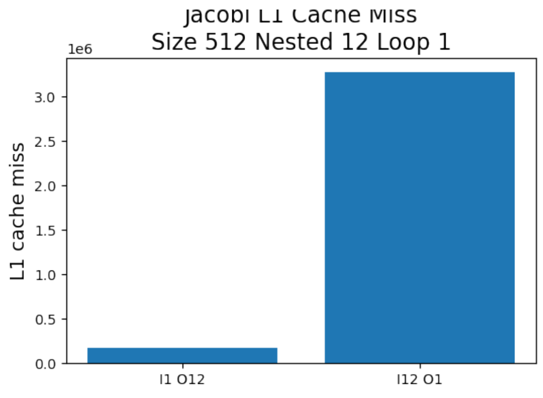
4.2 L1 cache misses

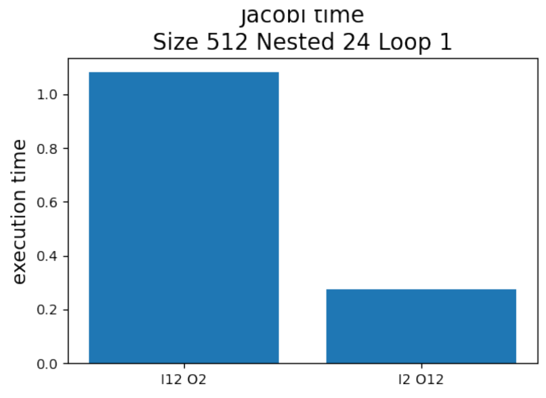
Correlation between execution time and `L1 cache miss` is clear: less cache misses faster the execution.

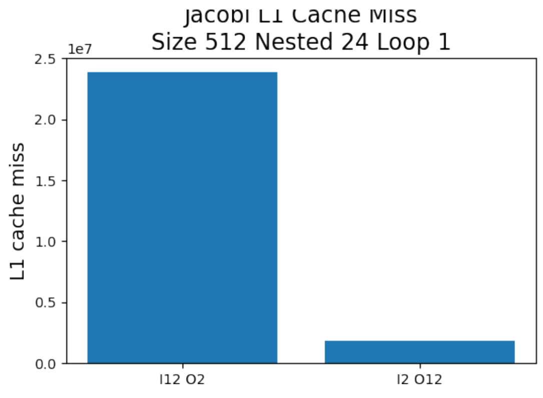


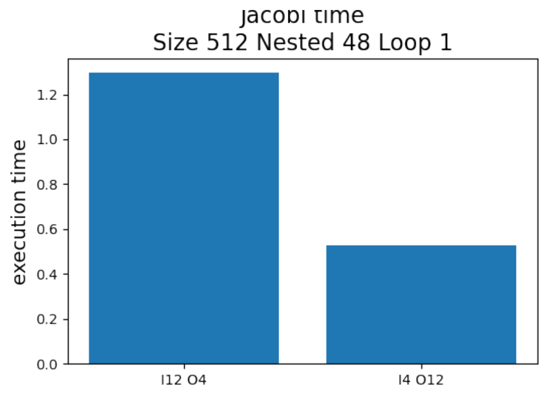


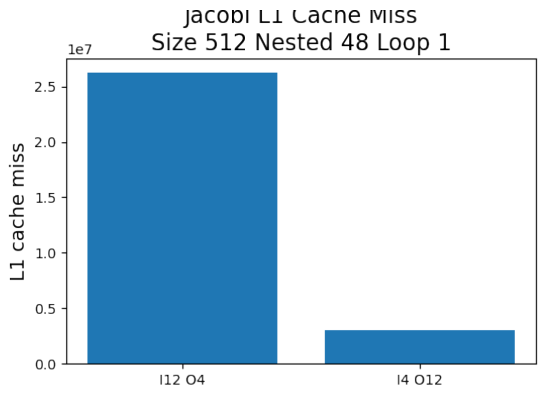






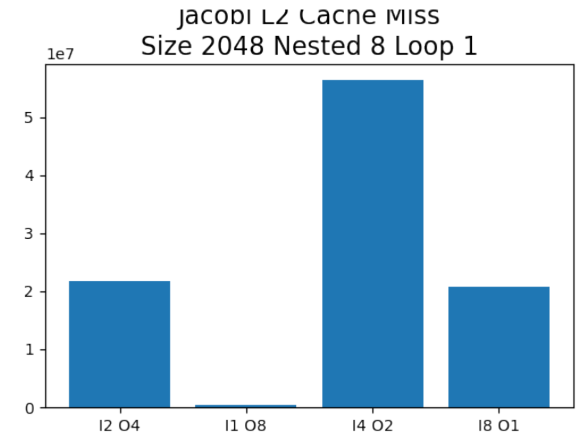


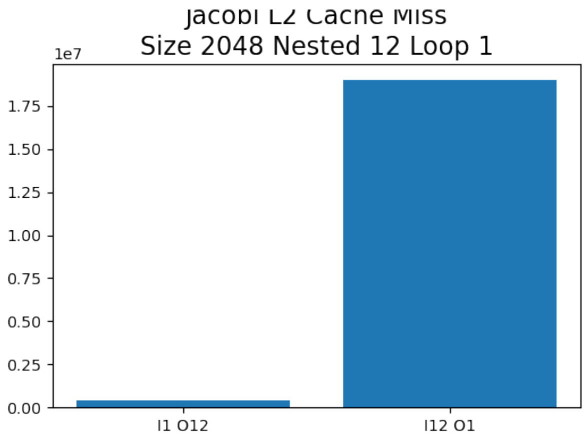


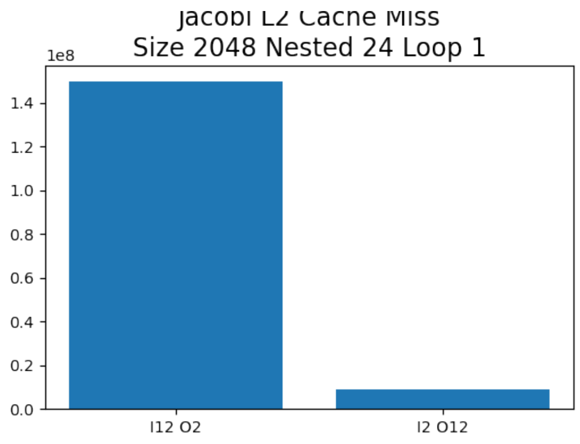


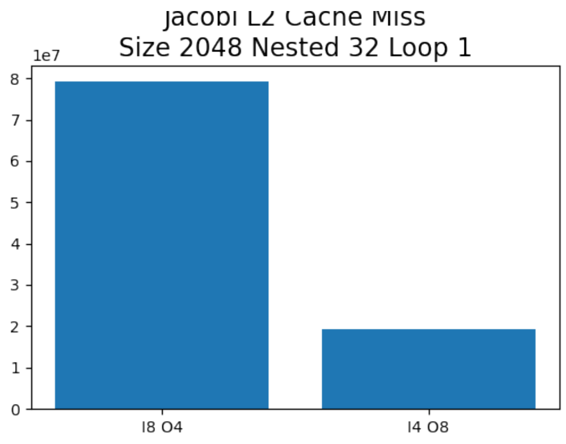
4.3 L2 cache misses

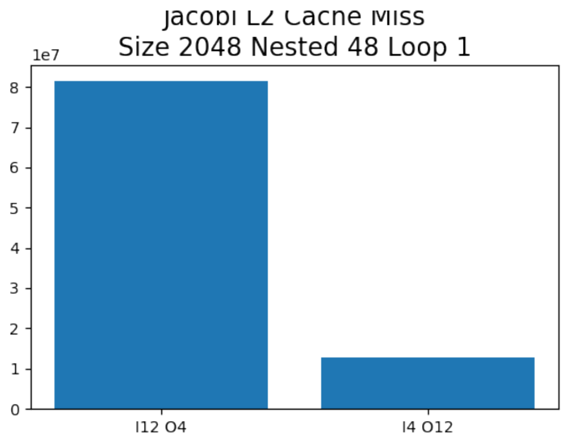
$N = 2048$

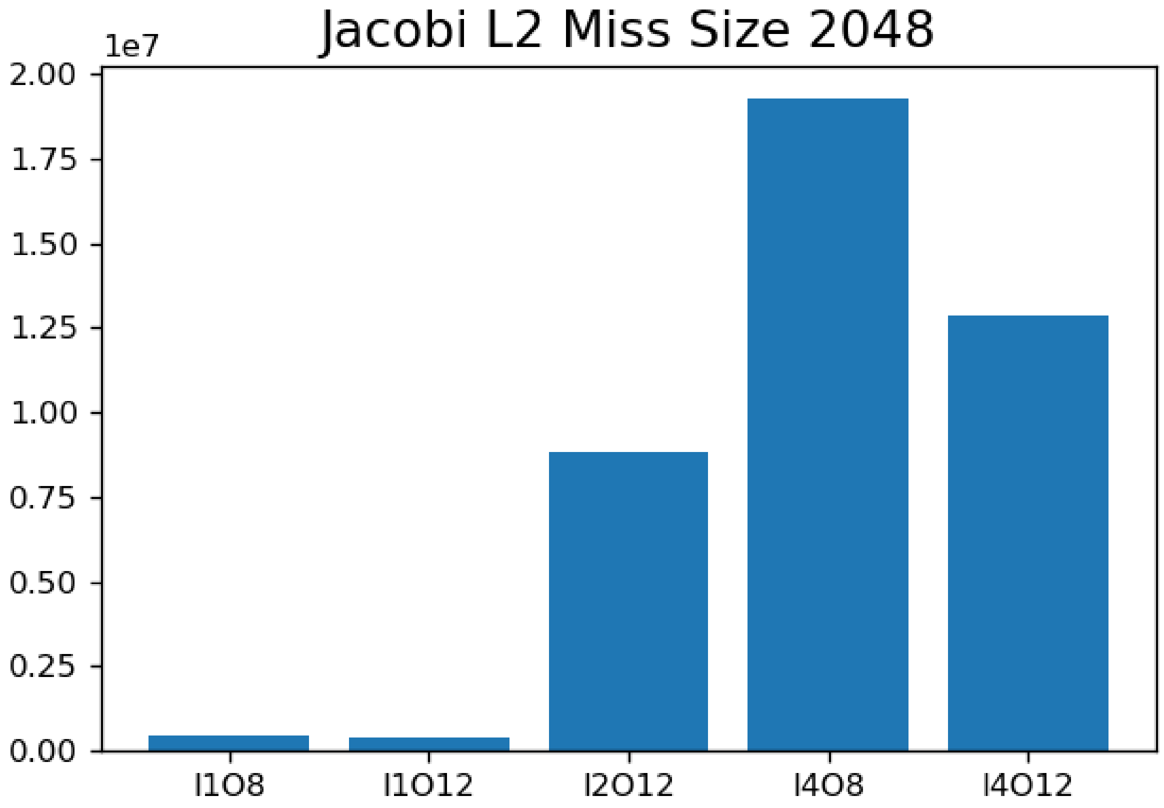












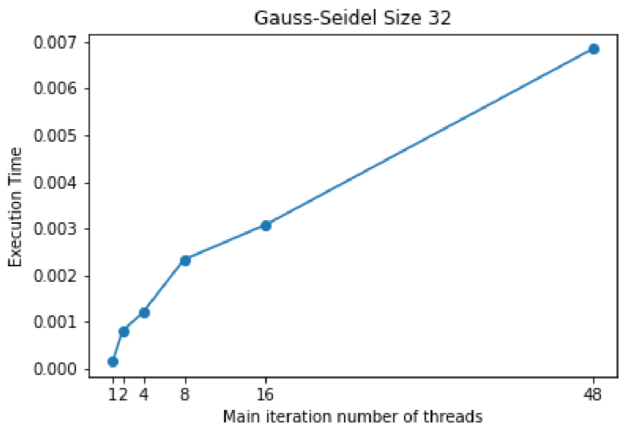
5 Conclusion on Jacobi Method

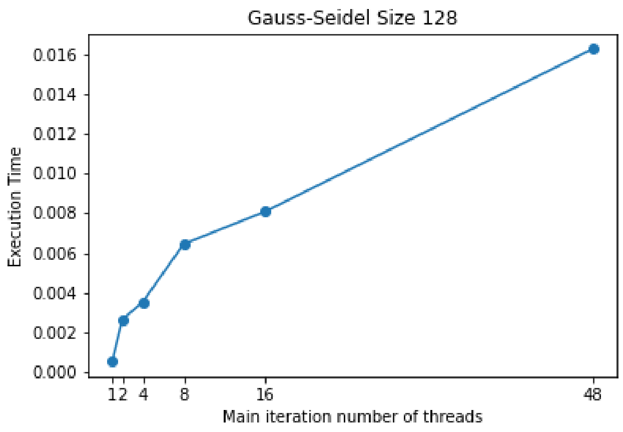
Overall, JIM has better performance in terms of execution time and cache misses, while outer loops are parallelized. It's hard to explain why parallelization of the inner loops does not give positive effect but we think that we should do something with memory bandwidth between cores. Since inner loop parallelisation splits the columns of the array A it may cause false sharing of the data.

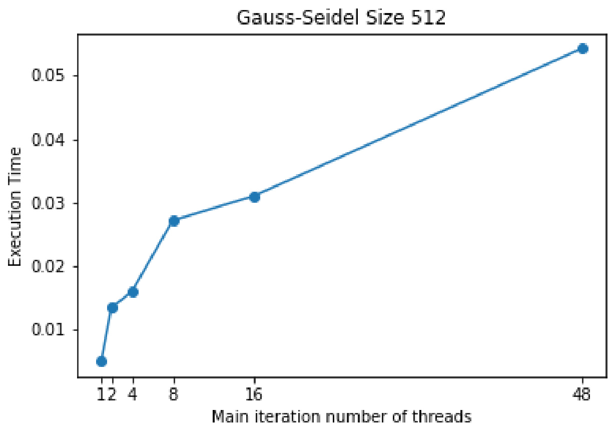
6 Results and Analysis on Gauss-Seidel Method

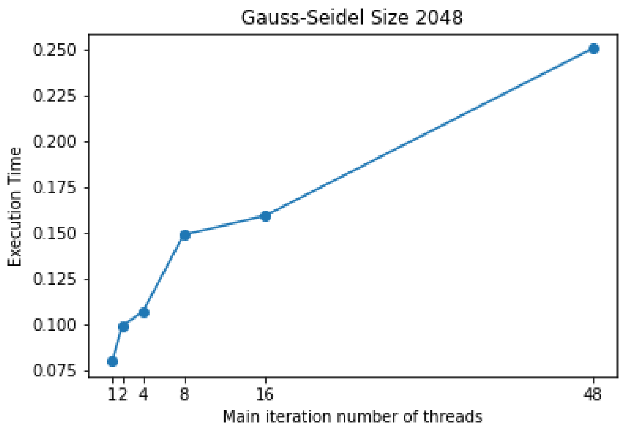
6.1 Execution time

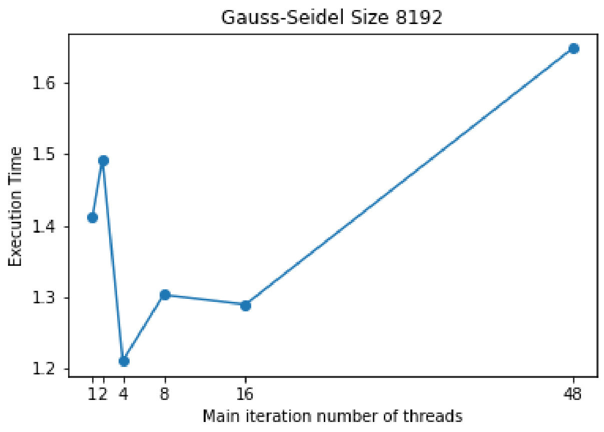
With Gauss - Seidel method we observe similar trends as in JIM. Since we can't parallelize outer loop, it uses new values immediately after it is available, we parallelized only inner loop and parallelizing inner caused some sort of slow down as in JIM.



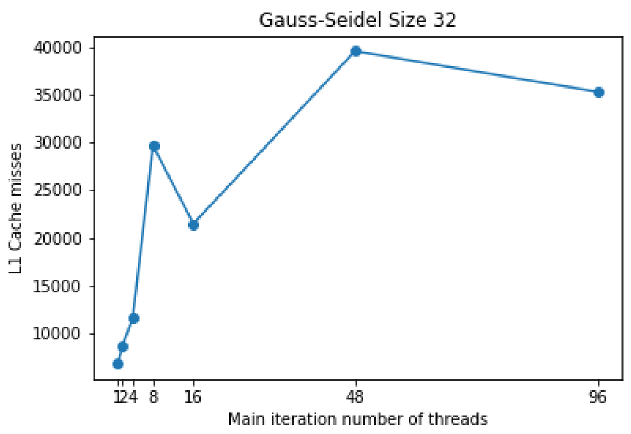


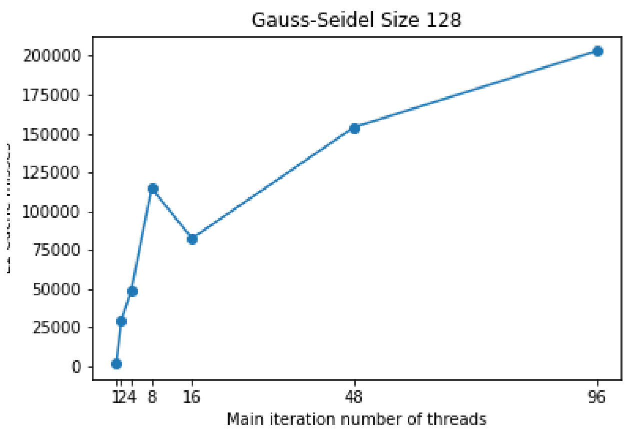


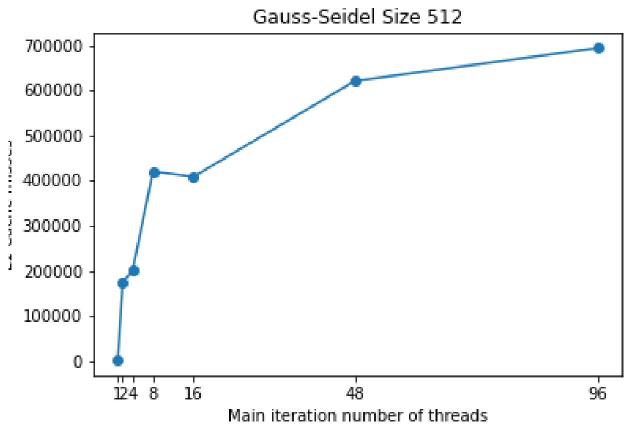


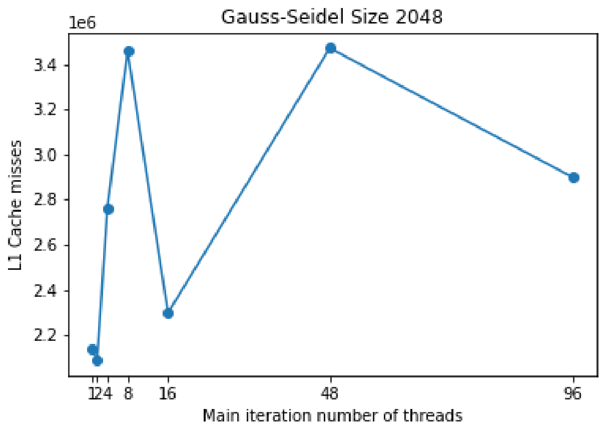


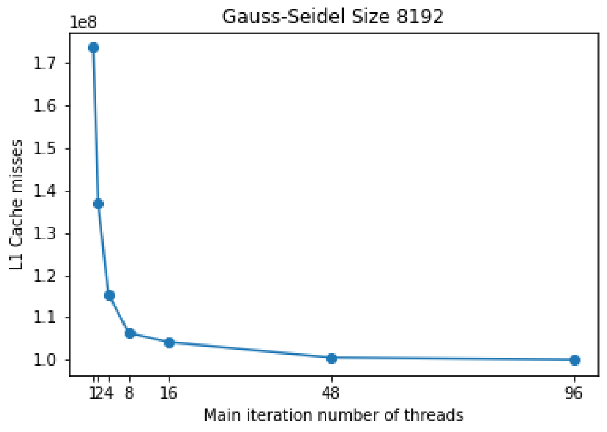
6.2 L1 cache misses











7 Conclusion on Gauss-Seidel Method

The Guass-Seidel implementation have the same pattern as the Jacobi method in sense that inner loop parallelization does not show better performance. However we also see that parallelization has better performance for larger array sizes (8192). Also for larger array size we see that there are less L1 cache misses with larger number of threads.

8 Final thoughts

In practice, parallelization using MPI is more common. It can be efficiently parallelized with condition that the devised schemes should achieve data locality, minimize the number of communications, and maximize the overlapping between the communications and the computations[4].
In general both methods have high data - dependency and Gauss - Seidel method is sequential by nature. However considering the fact that in practice we don't solve SLAE with random numbers , they always model some events and have some structure, we can always optimize(parallelize) our method depending on structure of matrix.

References

[1] https://en.wikipedia.org/wiki/Jacobi_method

[2] https://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel_method

[3] A Parallel Gauss–Seidel Method for Block Tridiagonal Linear Systems, Pierluigi Amodio and Francesca Mazzia, SIAM Journal on Scientific Computing 1995 16:6, 1451-1461

[4] Parallel implementations of the Jacobi linear algebraic systems solver , Athanasios Margaris, Stavros Souravlas, Manos Roumeliotis, BCI 2007

Appendix

All source codes are available on the project's github webpage
https://github.com/radmir-s/hpc-jacobi-gaussseidel-project.