Parallel Implementation of the Conjugate Gradient Method

Under the guidance of Dr. Surya Prakash

Niranjan Joshi - 160001026
Prathamesh Naik -160001037

Notes about our implementation

- We've created 2 libraries my_library.hpp (which contains the main implementation) and my_testing_library.hpp (which contains methods for testing)
- We've created 3 driver programs -
    - solver.cpp (solves the problem given in source code)
    - solver-file.cpp (solves the problems specified in input.txt and writes the answers to output.txt)
    - tester.cpp (used for testing, generates random systems with different combinations of parameters and records the test results in a file)
- For compiling tester.cpp: g++ tester.cpp -fopenmp -std=c++1z (requires c++ 17)
- For others: g++ solver-file.cpp -fopenmp and g++ solver.cpp -fopenmp
- Please refer to "Complete Test Results.xlsx" for test results of sizes till 1000. For

Introduction

The conjugate gradient method is one of the most effective algorithms for the numerical solution of large systems of linear equations whose matrix is symmetric and positive-definite. Since some sparse systems are too large to be handled by a direct implementation, it is usually implemented as an iterative algorithm.

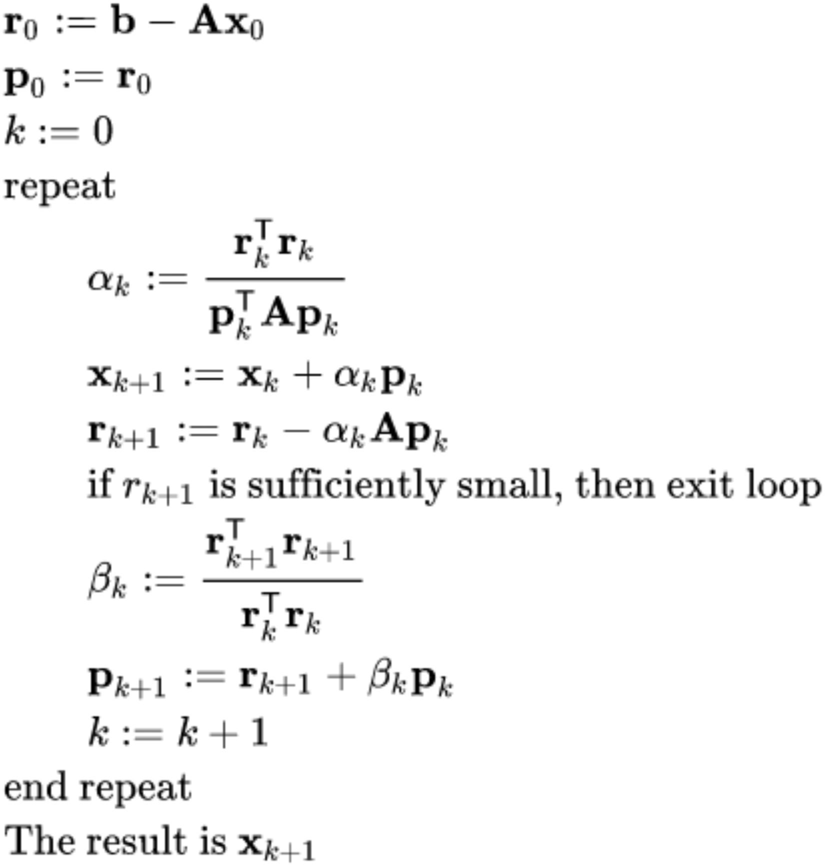
Such systems often arise while solving partial differential equations like the Poisson equation. This method is also used in solving various unconstrained optimization problems like energy minimization.

The conjugate gradient method is especially useful when the matrix is sparse because its time and space complexity depends on the number of non-zero elements, rather than the total size.

In this project, we will implement a parallel version of the iterative conjugate gradient method and do a performance analysis.

Scope

- Studying the Conjugate Gradient method
- Implementing it as an iterative algorithm and using sparse matrix storage format
- Parallelizing it using OpenMP
- Extending the method for the non positive definite case
- Complexity analysis
- Performance analysis- Execution time vs size,sparsity,threads (speedup)



```
$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{p}_0 := \mathbf{r}_0$
$k := 0$
repeat
    $\alpha_k := \dfrac{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}{\mathbf{p}_k^\mathsf{T}\mathbf{A}\mathbf{p}_k}$
    $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k\mathbf{p}_k$
    $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$
    if $r_{k+1}$ is sufficiently small, then exit loop
    $\beta_k := \dfrac{\mathbf{r}_{k+1}^\mathsf{T}\mathbf{r}_{k+1}}{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}$
    $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$
    $k := k + 1$
end repeat
The result is $\mathbf{x}_{k+1}$
```

The algorithm for solving **`Ax = b`** where **`A`** is a real, symmetric, positive-definite matrix. The input vector **`x₀`** can be an approximate initial solution or **`0`**.

Parallelizing

The above sequential algorithm will be parallelized by using:

1. Parallel sparse matrix-vector product
2. Parallel dot product of vectors
3. Parallel L-2 norm
4. Parallel copying of vectors
5. Parallel addition of vectors
6. Parallel algorithm for checking symmetric.

Sparse representation

The compressed sparse row (CSR) format stores a sparse `m × n` matrix `M` in row form using three (one-dimensional) arrays `(A, IA, JA)`. Let `NNZ` denote the number of nonzero entries in `M`.

1. The array `A` is of length `NNZ` and holds all the nonzero entries of `M` in row-major order.
2. The array `IA` is of length `m + 1`. `IA[i]` contains the number of non zero elements till row `i-1`.
3. The third array, `JA`, contains the column index in `M` of each element of `A` and hence is of length `NNZ` as well.

In case the matrix is not symmetric positive definite, we can detect that, compute its transpose A^T, and then solve the system $A^TA=A^Tb$ instead. For this we implemented sparse matrix transpose and sparse matrix multiplication.

Complexity analysis

1. Parallel sparse matrix-vector product `(res = A*x): O(n)`
2. Parallel dot product of vectors `(a`^{T}`b) :O(log(n))` (Due to reduction)
3. Parallel copying of vectors `(a := b) : O(1)`
4. Parallel addition of vectors `(c = a + alpha*b) : O(1)`
5. Checking Symmetry `(A`^{T}`==A) : O(log(n))`
6. Conjugate Gradient (solving for **x**): `O(n* iterations)` (iterations ≤ 100)

Experimentation

A tester function takes in parameters such as size, sparse proportion, number of threads and whether to assume positive definiteness. Next, the solver is run 10 times (configurable) using appropriately generated random matrix systems and the average execution time and the standard deviation is returned. This is done so as to minimize the effect of random errors.

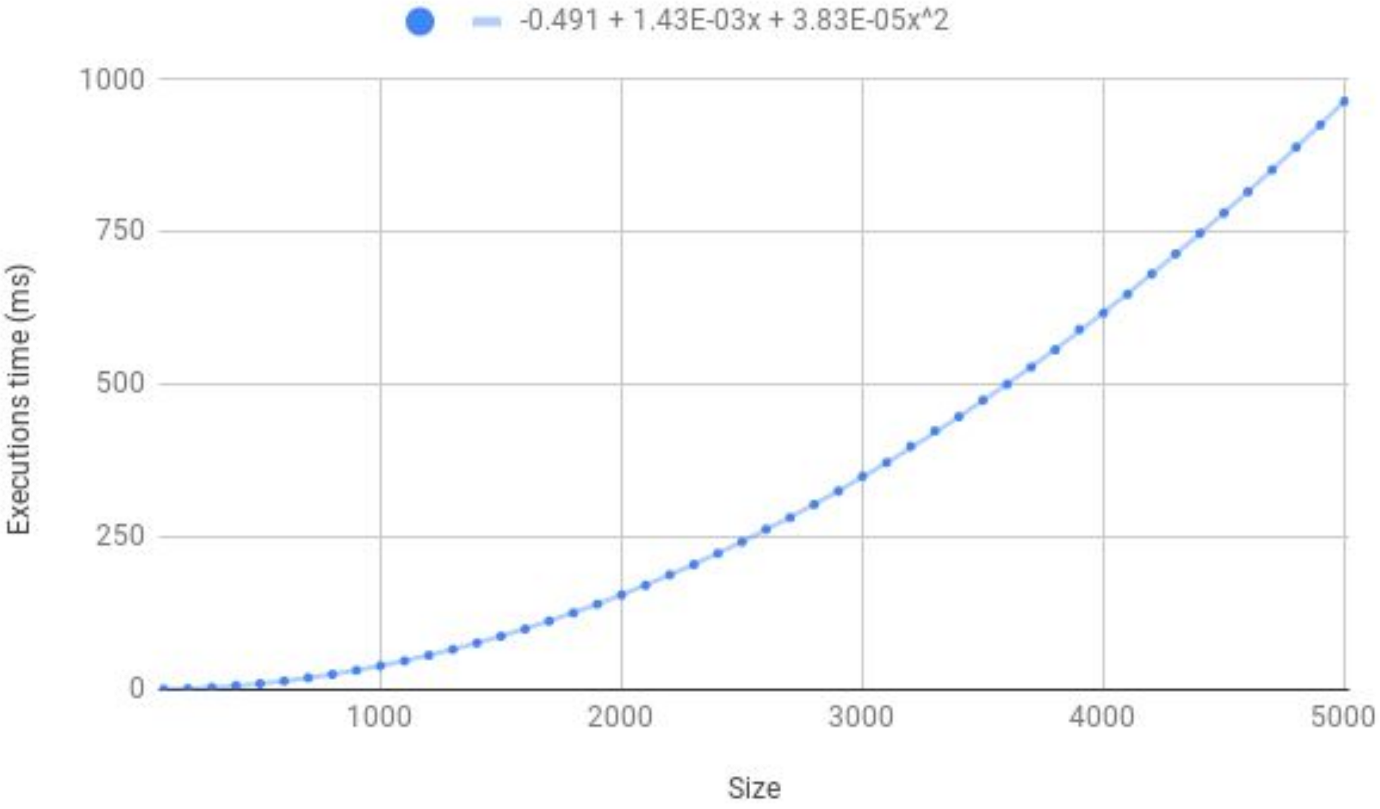
The following methods are used:

- Generating random matrix of given size and sparse proportion
- Generating random feasible b vector
- Converting a matrix to CSR format
- Generating a random symmetric positive definite matrix

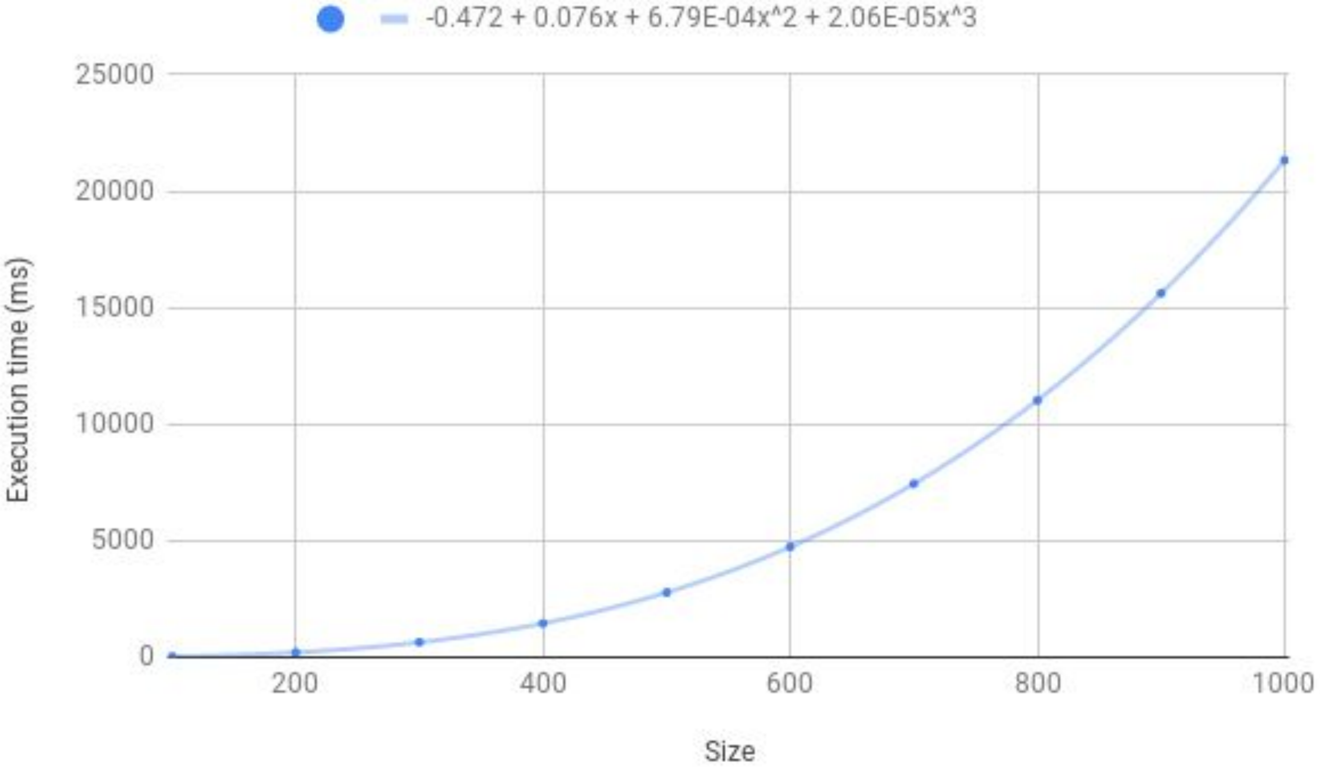
Testing loop

```
std::ofstream output("comprehensive_test_results" + GetCurrentTimeForFileName() + ".txt");
for (int psd = 0; psd < 2; psd++)
{
      for (int num_threads = 1; num_threads < 5; num_threads++)
      {
            for (int size = 100; size <= 1000; size += 100)
            {
                auto [avg, std_dev] = tester(size, psd, num_threads);
                output << psd << " " << num_threads << " " << size << " " << avg << " " << std_dev << endl;
                cout << psd << " " << num_threads << " " << size << " " << avg << " " << std_dev << endl;
            }
      }
}
```

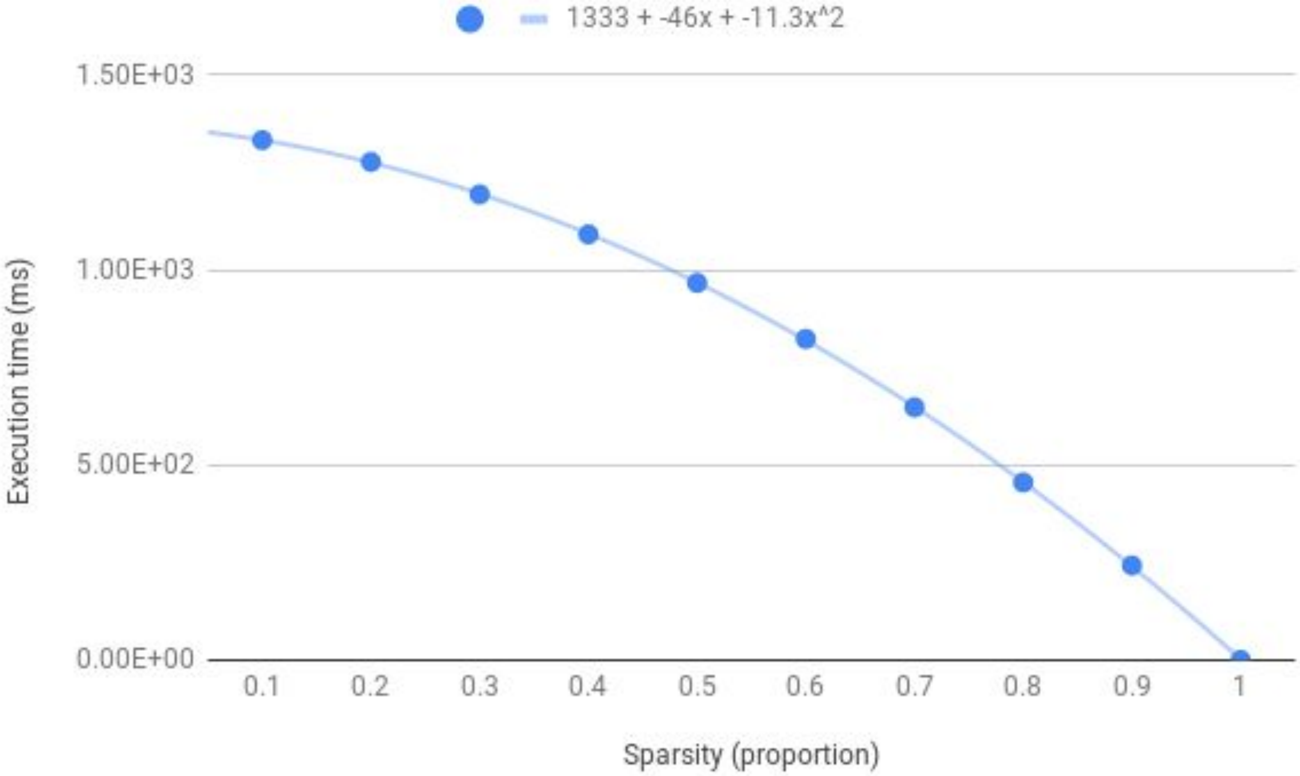
Execution Time v/s Input Size for SPD sparse matrix.



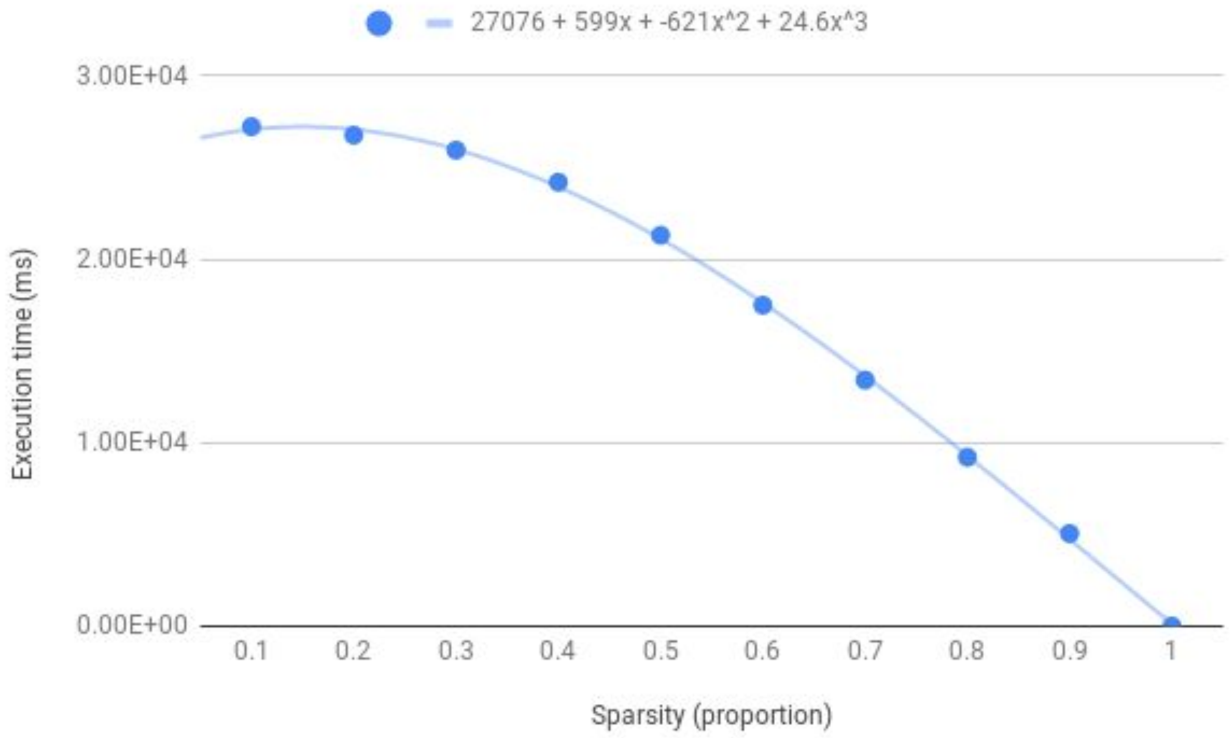
Execution Time v/s Input Size for Non-SPD sparse matrix.



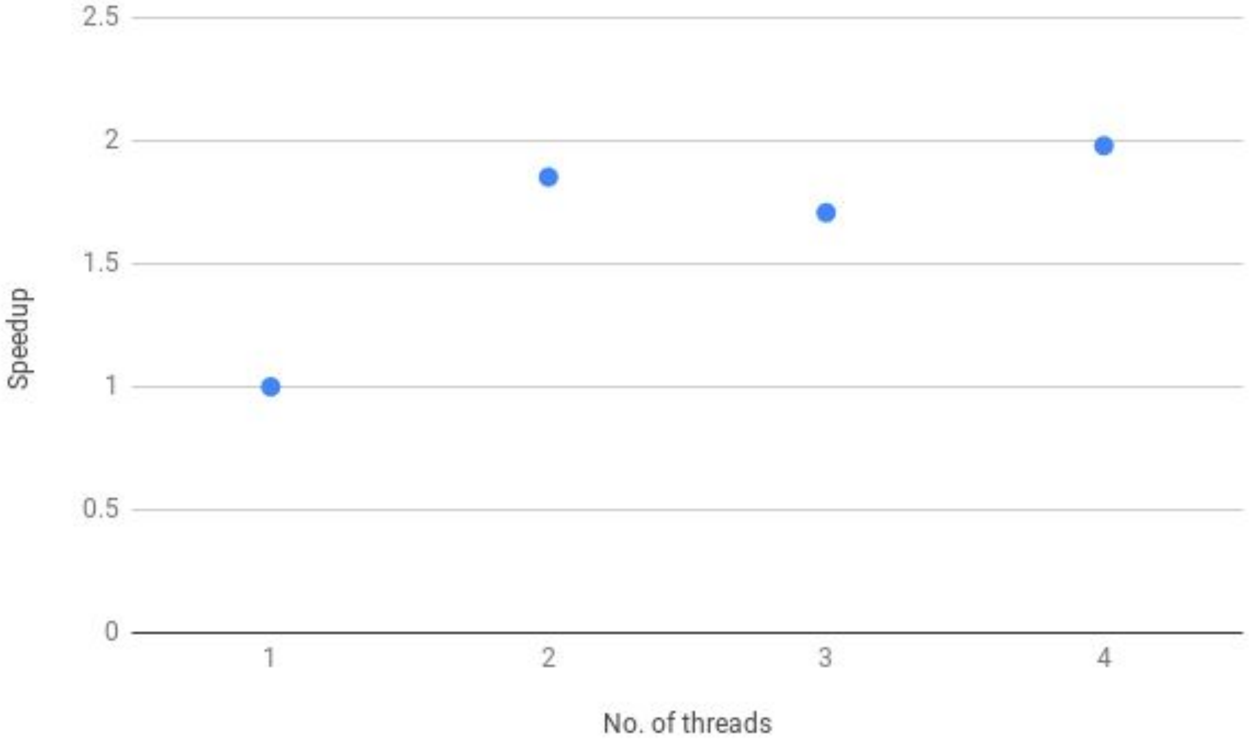
Execution Time v/s Sparsity Proportion for SPD sparse matrix.



Execution Time v/s Sparsity Proportion for Non-SPD sparse matrix.



Speedup v/s No. of threads for SPD matrix, input size of 5000



Conclusion

In this project, we implemented and analysed the parallel algorithm for the conjugate gradient method while using sparse matrix representation. Our implementation works for both Symmetric Positive Definite and Non-Positive Definite matrices. From the plots we can conclude that the execution time depends on the input size, sparsity proportion of the sparse matrix, type of the matrix and number of available threads.

1. For SPD matrix the execution time has quadratic relation with the input size. The theoretical complexity of parallel algorithm is linear (given $O(n)$ processors) with input size but due to limitations on the number of cores (4 in most laptops), actual observed complexity is quadratic.
2. For Non-SPD matrix the execution time has cubic relation with input size. This additional time is due to the computation of A^TA.
3. Execution time decreases for any type of the matrix as the sparsity proportion increases. This is because our implementation scales as the number of nonzero elements (rather than all elements).
4. Maximum speedup attained is **2** for **4** threads and input size of **5000**.

References

- Wikipedia. Conjugate gradient method. (https://en.wikipedia.org/wiki/Conjugate_gradient_method)
- Caraba, E. . A Parallel Implementation Of The Conjugate Gradient Method. (pdfs.semanticscholar.org/eaed/aec18d8931d99abe7fbd5b48e3de6997b7be.pdf)
- Rudi Helfenstein, Jonas Koko, “Parallel preconditioned conjugate gradient algorithm on GPU”, Journal of Computational and Applied Mathematics, (http://www.sciencedirect.com/science/article/pii/S0377042711002196)
- R. P. Bycul, A. Jordan and M. Cichomski, "A new version of conjugate gradient method parallel implementation", Proceedings. International Conference on Parallel Computing in Electrical Engineering (https://ieeexplore.ieee.org/document/1115282)