CS 359 Parallel Computing Project Report

Parallel Implementation of the Conjugate Gradient Method

Under Guidance of

Dr. Surya Prakash

By:

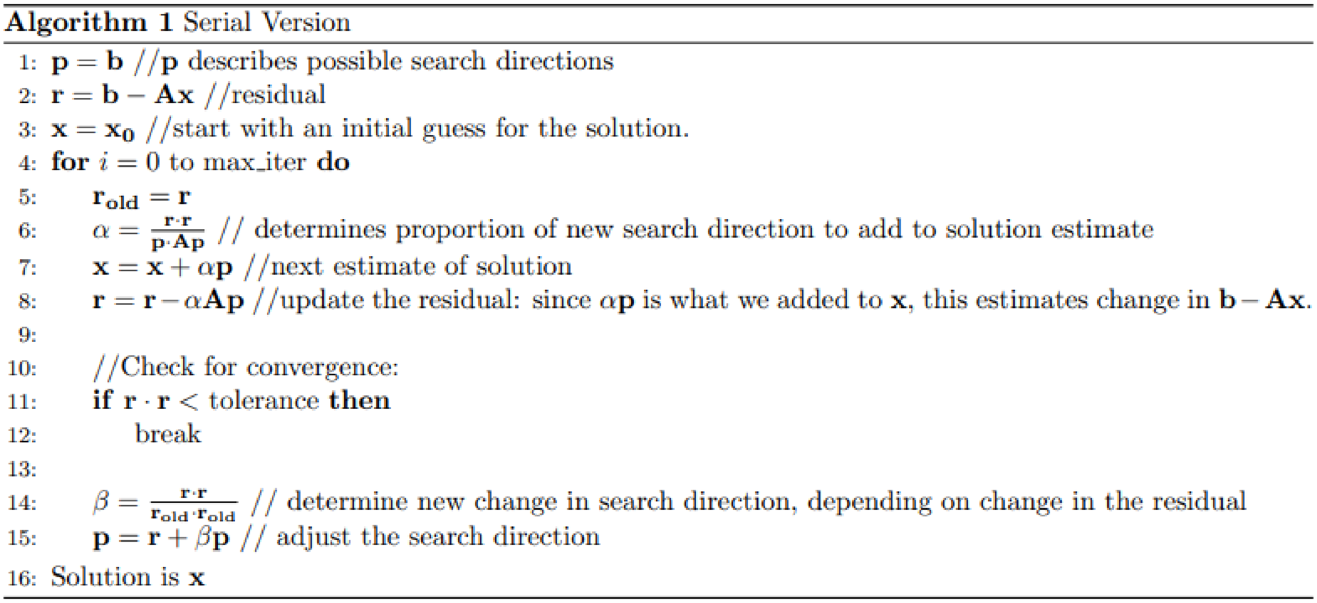
Aditi Ganvir - cse210001016
Princy Sondarva - cse210001068

Introduction:

The conjugate gradient (CG) method is an iterative algorithm for numerical solution of systems of linear equations which when written as matrix equations have a symmetric and positive-definite matrix. The approach is often used for large matrices where a direct method would take too much time. 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. The solution of the matrix equations is often the most time-consuming portion of a scientific simulation code and often the matrix equations need to be solved many times as part of a self-consistent algorithm. In this project, We demonstrate use of a the OpenMP (shared memory) to parallelize this algorithm.

Conjugate Gradient Method:

The CG algorithm is in general used to search for the minimum value of a multi-dimensional function. For the case of matrix solving, we are solving the matrix equation $Ax = b$ for x, where A is a symmetric and positive-definite matrix and b is the right hand side vector. In this case, we are searching for the minimum value of the residual $r = b - Ax$. In each iteration, the estimate of the solution x is updated by moving a certain amount α along the search direction vector p. The coefficient α is determined by the current iteration's values of A and p. The search direction is adjusted depending on the change in the norm of the residual which is also used as the convergence check. A pseudo-code is shown below:



```
Algorithm 1 Serial Version
$\mathbf{p} = \mathbf{b}$ //$\mathbf{p}$ describes possible search directions
$\mathbf{r} = \mathbf{b} - \mathbf{Ax}$ //residual
$\mathbf{x} = \mathbf{x_0}$ //start with an initial guess for the solution.
for $i = 0$ to max_iter do
    $\mathbf{r_{old}} = \mathbf{r}$
    $\alpha = \frac{\mathbf{r} \cdot \mathbf{r}}{\mathbf{p} \cdot \mathbf{Ap}}$ // determines proportion of new search direction to add to solution estimate
    $\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$ //next estimate of solution
    $\mathbf{r} = \mathbf{r} - \alpha\mathbf{Ap}$ //update the residual: since $\alpha\mathbf{p}$ is what we added to $\mathbf{x}$, this estimates change in $\mathbf{b} - \mathbf{Ax}$.

    //Check for convergence:
    if $\mathbf{r} \cdot \mathbf{r} <$ tolerance then
        break

    $\beta = \frac{\mathbf{r} \cdot \mathbf{r}}{\mathbf{r_{old}} \cdot \mathbf{r_{old}}}$ // determine new change in search direction, depending on change in the residual
    $\mathbf{p} = \mathbf{r} + \beta\mathbf{p}$ // adjust the search direction
Solution is $\mathbf{x}$
```

References: https://en.wikipedia.org/wiki/Conjugate_gradient_method

Parallelizing:

The above sequential algorithm is 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 Matrix:

Sparse matrices are a specialized way of representing matrices that contain a significant number of zero elements. They're crucial in various computational fields, particularly in optimizing storage and computation for large-scale systems. When using the conjugate gradient method, which deals with solving systems of linear equations, sparse matrices offer significant advantages in terms of memory efficiency and computational speed due to their ability to focus on non-zero elements exclusively.

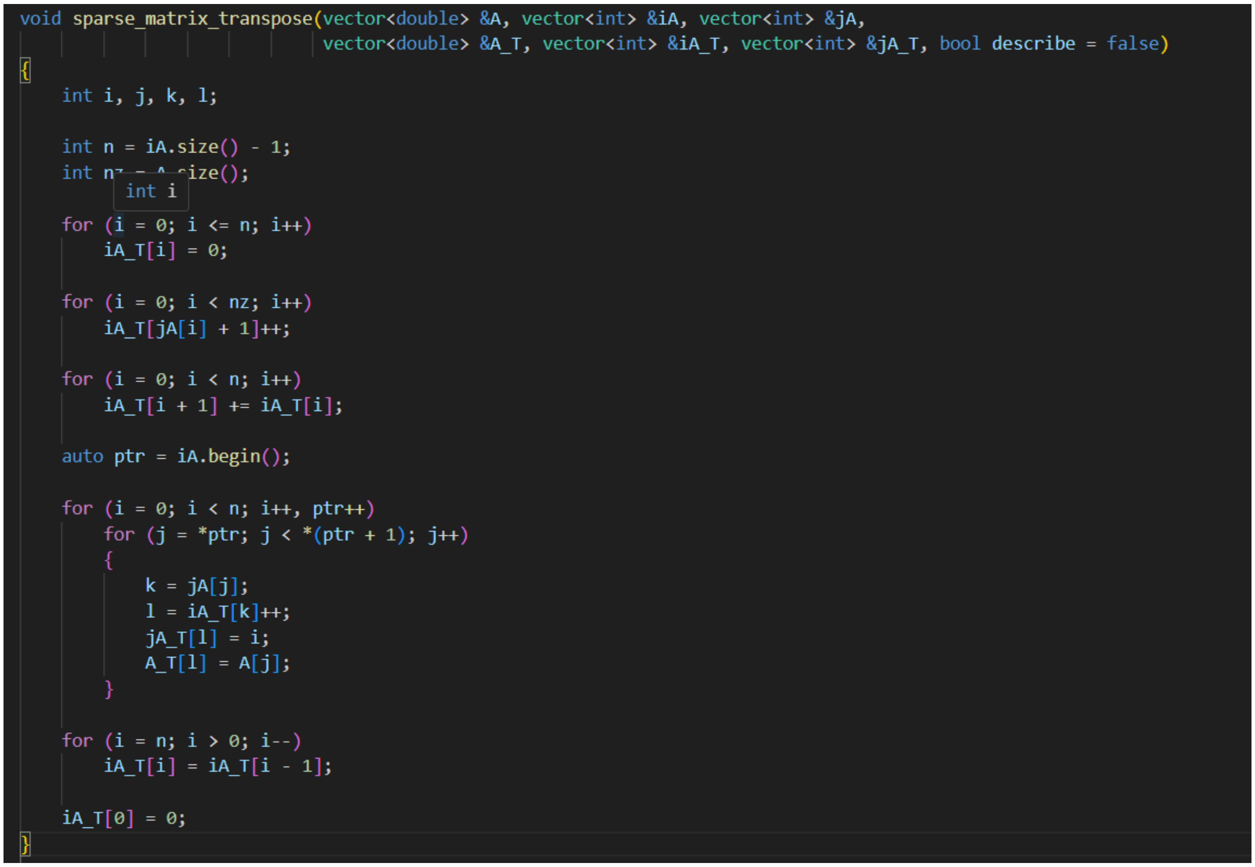
The compressed sparse row (CSR) format is one such representation for sparse matrices. It stores the matrix in row form using three arrays: A, IA, and JA.

A: This array contains all the non-zero elements of the matrix M in row-major order. It's of length NNZ, where NNZ represents the total number of non-zero entries in M.

IA: The IA array, of length m + 1, holds the count of non-zero elements until each row i-1. It essentially keeps track of the starting index for each row in the A array.

JA: The JA array, also of length NNZ, stores the column index of each element in the A array, indicating the column position of the corresponding non-zero element in the matrix.

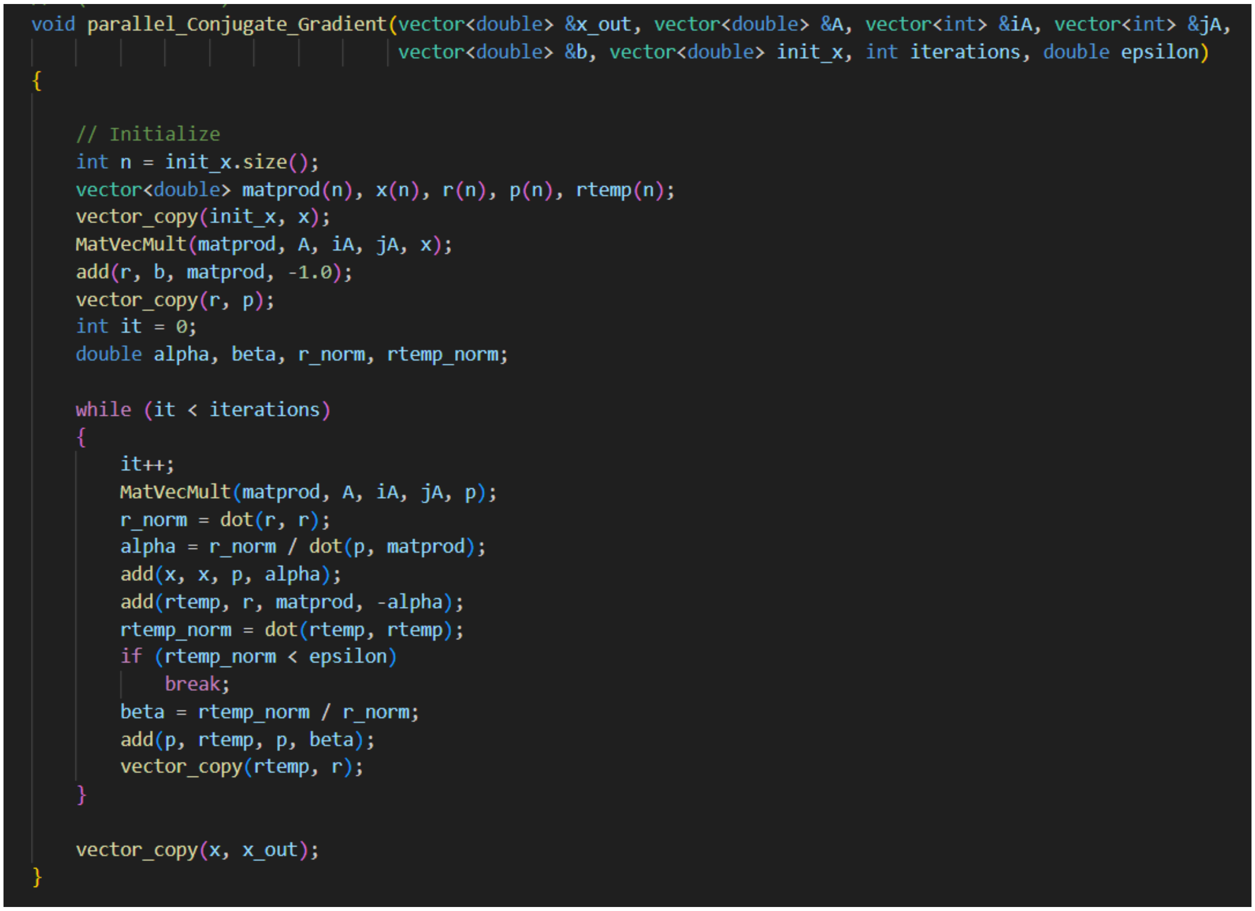
In case the matrix is not symmetric positive definite, we can detect that, compute its transpose A^T, and then solve the system A^TA=A^Tb instead. For this we implemented sparse matrix transpose and sparse matrix multiplication. For which the code is shown below:



```
void sparse_matrix_transpose(vector<double> &A, vector<int> &iA, vector<int> &jA,
                             vector<double> &A_T, vector<int> &iA_T, vector<int> &jA_T, bool describe = false)
{
    int i, j, k, l;

    int n = iA.size() - 1;
    int nz = A.size();

    for (i = 0; i <= n; i++)
        iA_T[i] = 0;

    for (i = 0; i < nz; i++)
        iA_T[jA[i] + 1]++;

    for (i = 0; i < n; i++)
        iA_T[i + 1] += iA_T[i];

    auto ptr = iA.begin();

    for (i = 0; i < n; i++, ptr++)
        for (j = *ptr; j < *(ptr + 1); j++)
        {
            k = jA[j];
            l = iA_T[k]++;
            jA_T[l] = i;
            A_T[l] = A[j];
        }

    for (i = n; i > 0; i--)
        iA_T[i] = iA_T[i - 1];

    iA_T[0] = 0;
}
```

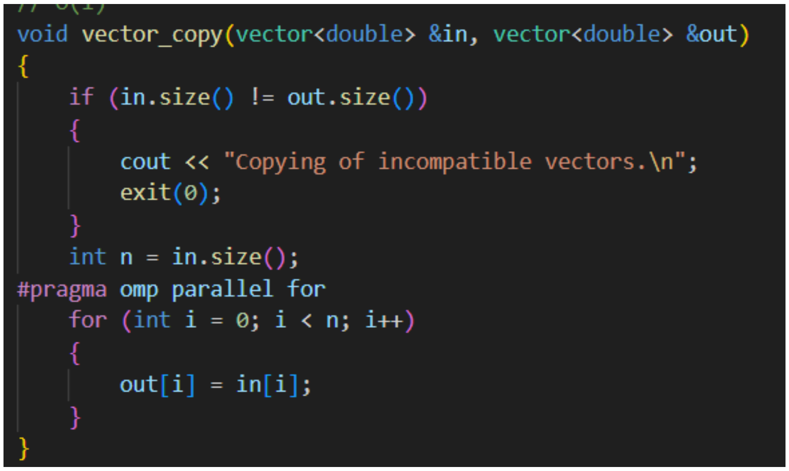
Parallel Code:

Parallel_Conjugate_Gradient:



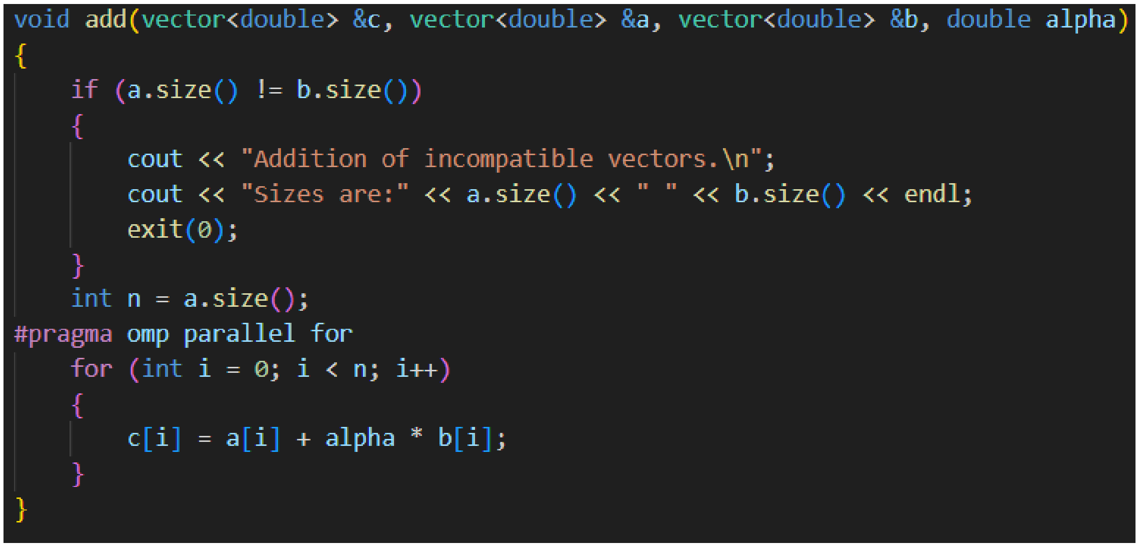
```
void parallel_Conjugate_Gradient(vector<double> &x_out, vector<double> &A, vector<int> &iA, vector<int> &jA,
                                 vector<double> &b, vector<double> init_x, int iterations, double epsilon)
{

    // Initialize
    int n = init_x.size();
    vector<double> matprod(n), x(n), r(n), p(n), rtemp(n);
    vector_copy(init_x, x);
    MatVecMult(matprod, A, iA, jA, x);
    add(r, b, matprod, -1.0);
    vector_copy(r, p);
    int it = 0;
    double alpha, beta, r_norm, rtemp_norm;

    while (it < iterations)
    {
        it++;
        MatVecMult(matprod, A, iA, jA, p);
        r_norm = dot(r, r);
        alpha = r_norm / dot(p, matprod);
        add(x, x, p, alpha);
        add(rtemp, r, matprod, -alpha);
        rtemp_norm = dot(rtemp, rtemp);
        if (rtemp_norm < epsilon)
            break;
        beta = rtemp_norm / r_norm;
        add(p, rtemp, p, beta);
        vector_copy(rtemp, r);
    }

    vector_copy(x, x_out);
}
```

Vector Copy:



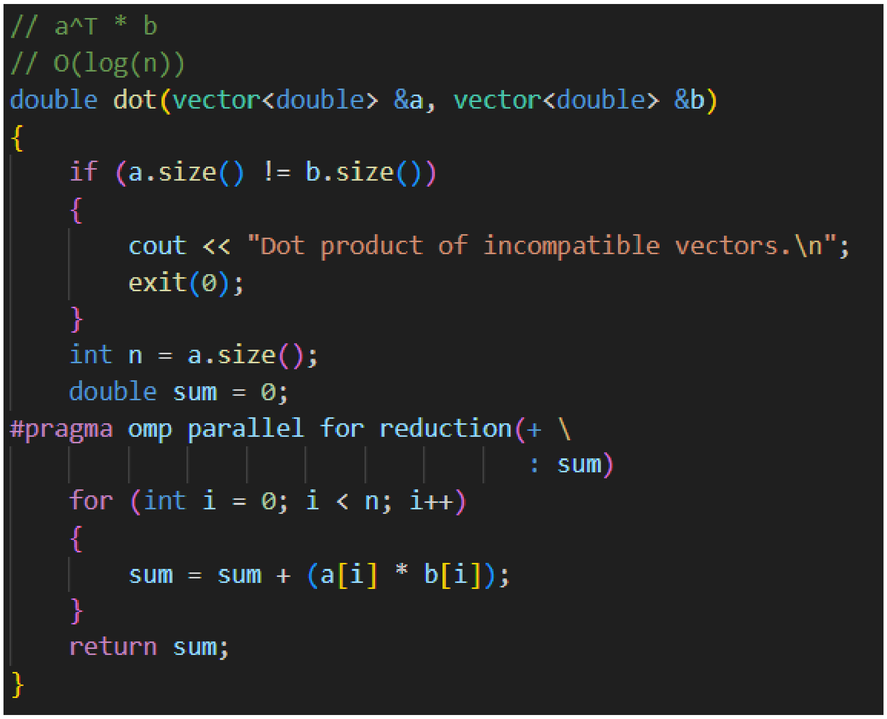
```
void vector_copy(vector<double> &in, vector<double> &out)
{
    if (in.size() != out.size())
    {
        cout << "Copying of incompatible vectors.\n";
        exit(0);
    }
    int n = in.size();
#pragma omp parallel for
    for (int i = 0; i < n; i++)
    {
        out[i] = in[i];
    }
}
```

Add:



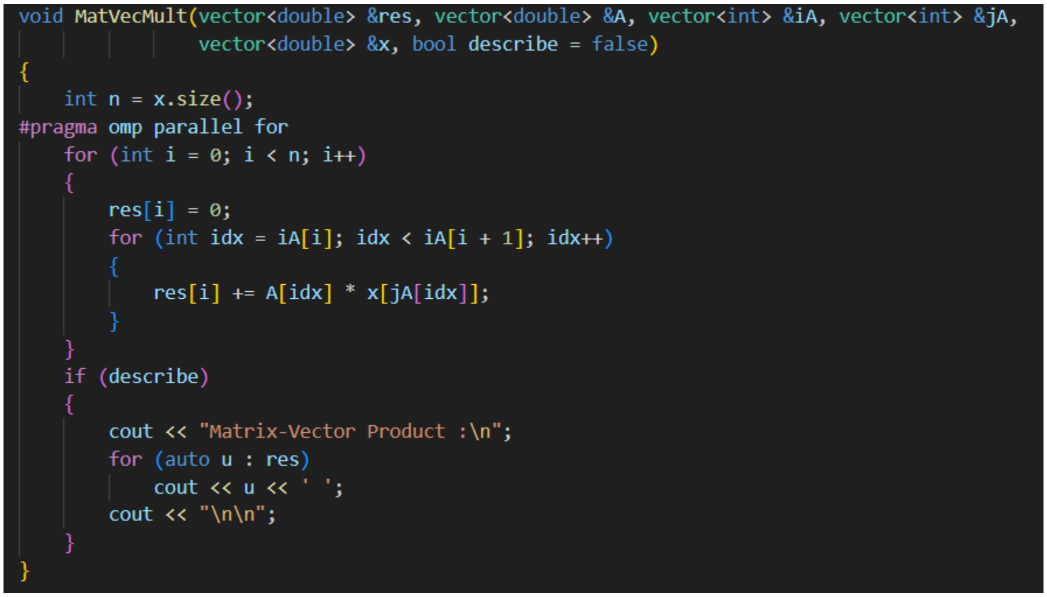
```
void add(vector<double> &c, vector<double> &a, vector<double> &b, double alpha)
{
    if (a.size() != b.size())
    {
        cout << "Addition of incompatible vectors.\n";
        cout << "Sizes are:" << a.size() << " " << b.size() << endl;
        exit(0);
    }
    int n = a.size();
#pragma omp parallel for
    for (int i = 0; i < n; i++)
    {
        c[i] = a[i] + alpha * b[i];
    }
}
```

Dot_Product:



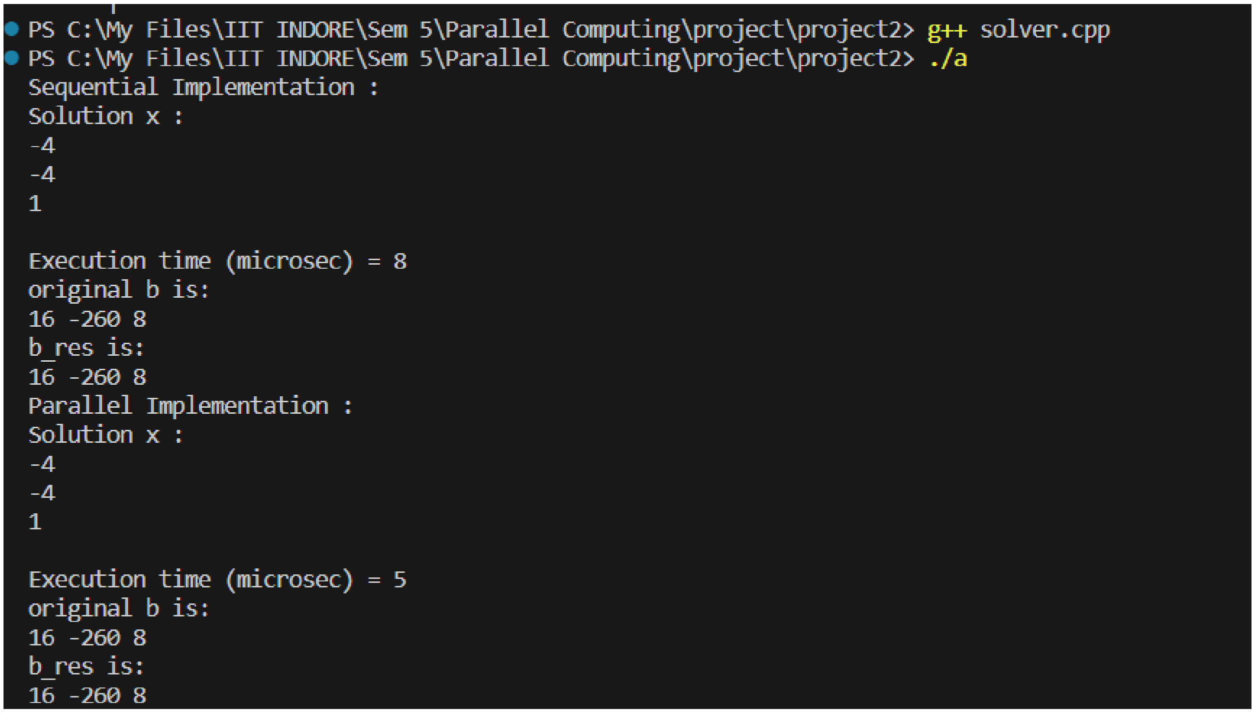
```
// a^T * b
// O(log(n))
double dot(vector<double> &a, vector<double> &b)
{
    if (a.size() != b.size())
    {
        cout << "Dot product of incompatible vectors.\n";
        exit(0);
    }
    int n = a.size();
    double sum = 0;
#pragma omp parallel for reduction(+ \
                                    : sum)
    for (int i = 0; i < n; i++)
    {
        sum = sum + (a[i] * b[i]);
    }
    return sum;
}
```

Matrix_Vec_Multiplication:



```
void MatVecMult(vector<double> &res, vector<double> &A, vector<int> &iA, vector<int> &jA,
                vector<double> &x, bool describe = false)
{
    int n = x.size();
#pragma omp parallel for
    for (int i = 0; i < n; i++)
    {
        res[i] = 0;
        for (int idx = iA[i]; idx < iA[i + 1]; idx++)
        {
            res[i] += A[idx] * x[jA[idx]];
        }
    }
    if (describe)
    {
        cout << "Matrix-Vector Product :\n";
        for (auto u : res)
            cout << u << ' ';
        cout << "\n\n";
    }
}
```

Output:



```
PS C:\My Files\IIT INDORE\Sem 5\Parallel Computing\project\project2> g++ solver.cpp
PS C:\My Files\IIT INDORE\Sem 5\Parallel Computing\project\project2> ./a
Sequential Implementation :
Solution x :
-4
-4
1

Execution time (microsec) = 8
original b is:
16 -260 8
b_res is:
16 -260 8
Parallel Implementation :
Solution x :
-4
-4
1

Execution time (microsec) = 5
original b is:
16 -260 8
b_res is:
16 -260 8
```

Complexity Analysis:

1. Parallel sparse matrix-vector product (res = A*x): O(n)
2. Parallel dot product of vectors (aTb) :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 (AT==A) : O(log(n))
6. Conjugate Gradient (solving for x): O(n* iterations) (iterations ≤ 100)

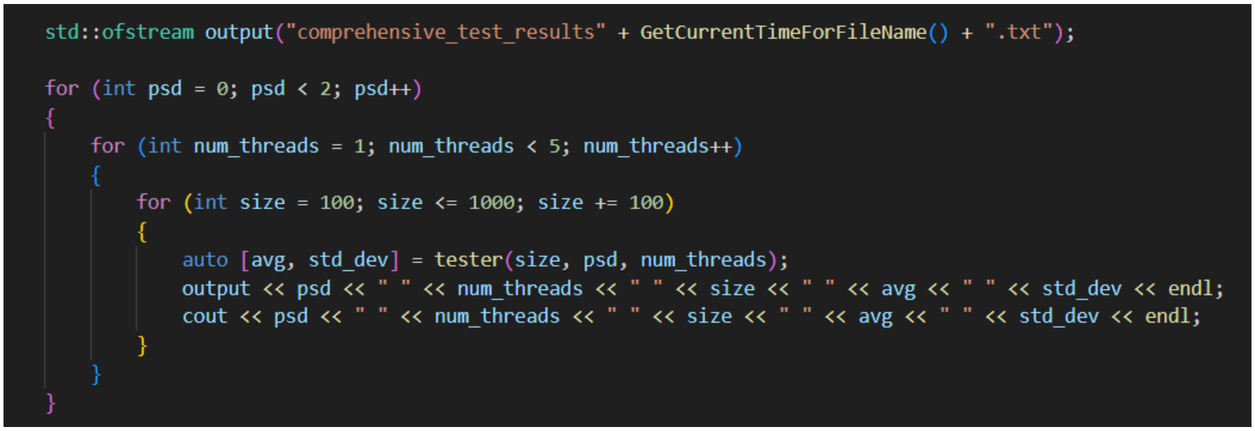
Experimentation:

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

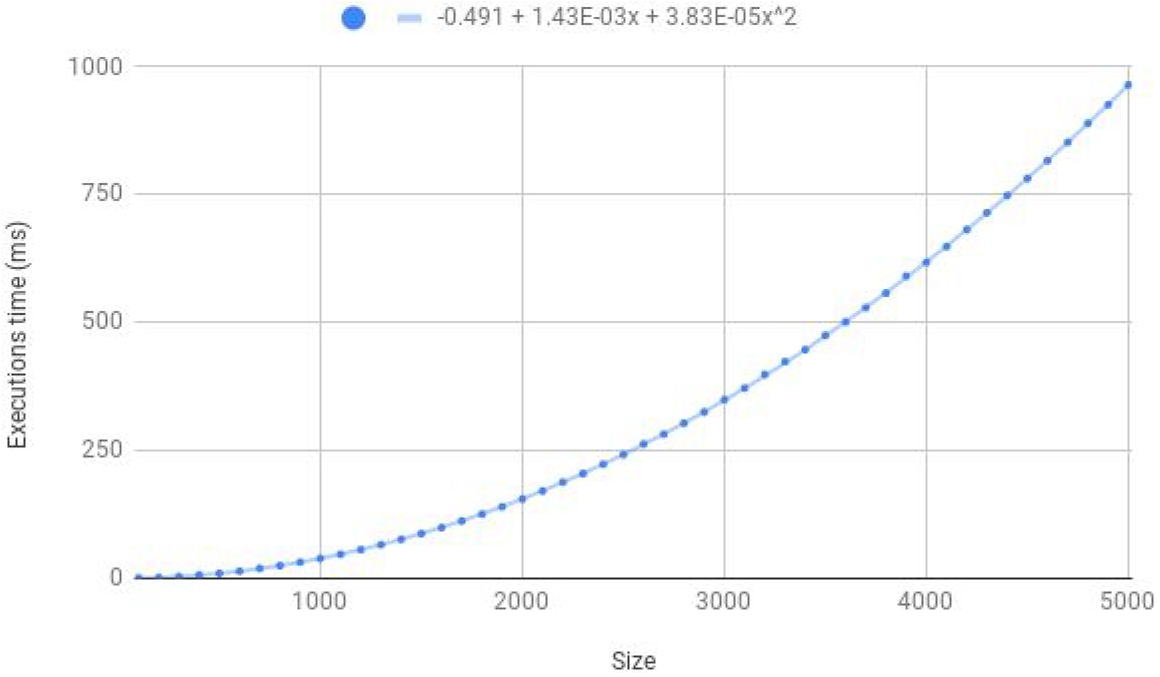
Testing loop:



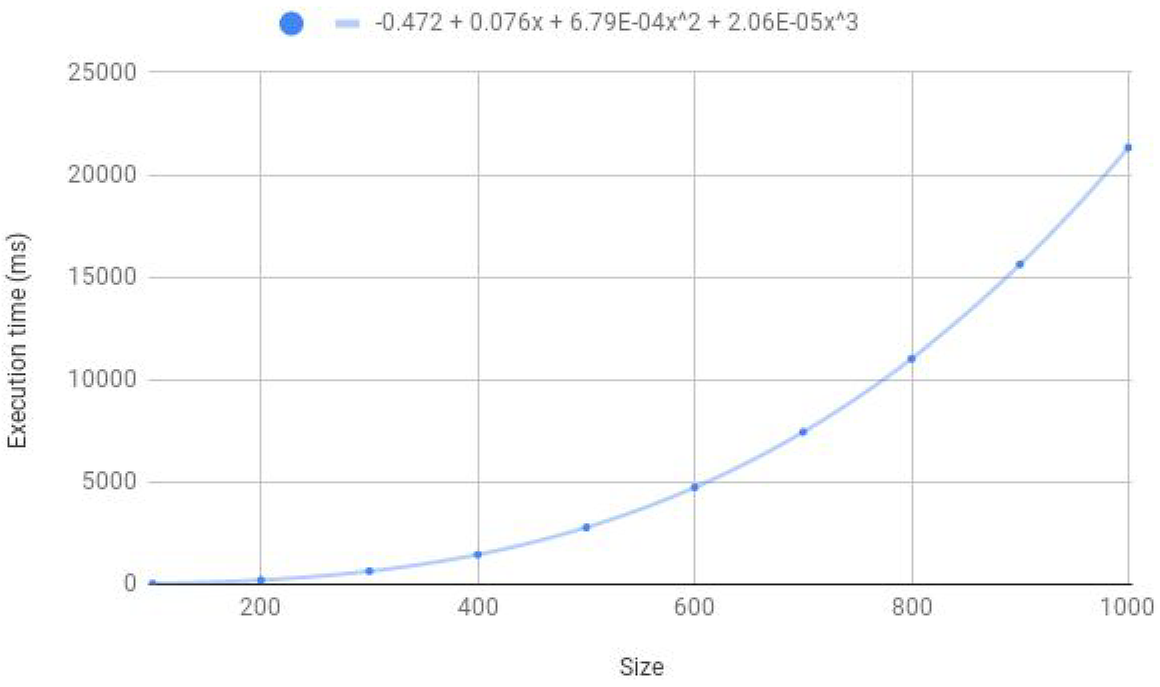
```cpp
std::ofstream output("comprehensive_test_results" + GetCurrentTimeForFileName() + ".txt");

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;
        }
    }
}
```

Results Analysis:

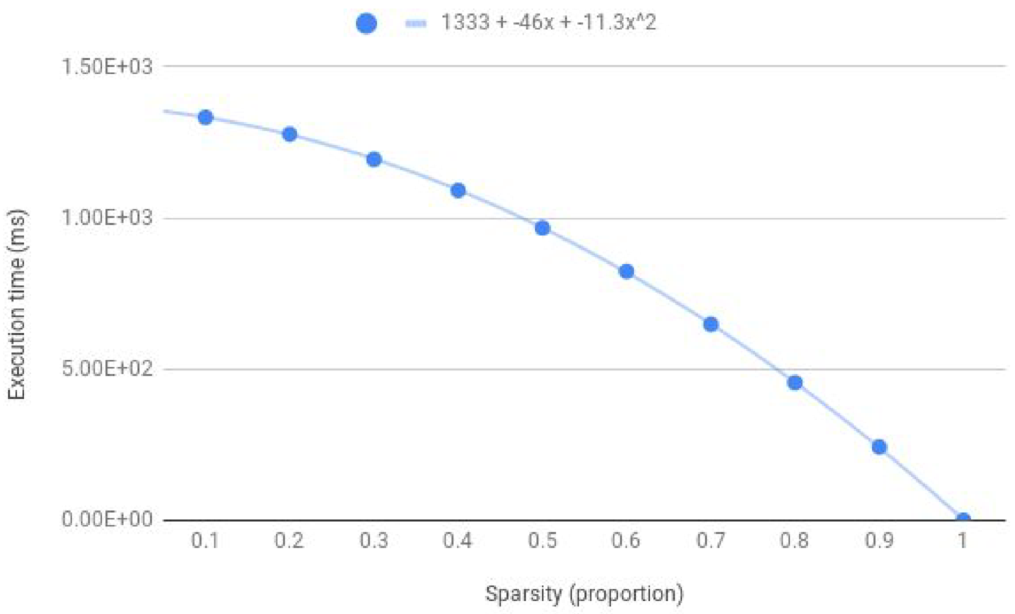
Execution Time v/s Input Size for SPD sparse matrix



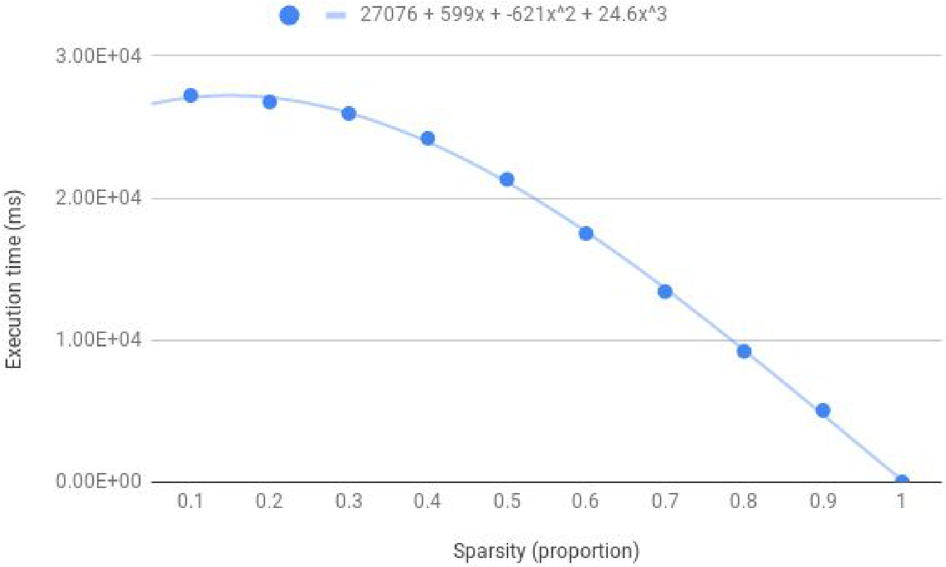
Execution Time v/s Input Size for Non-SPD sparse matrix



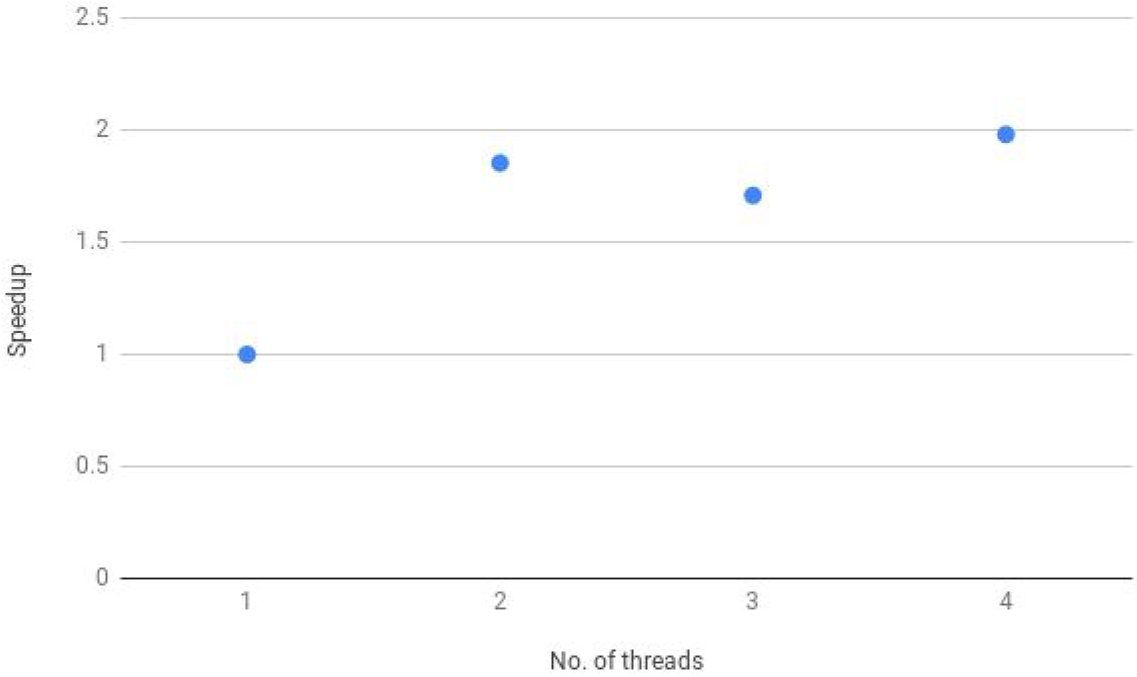
Execution Time v/s Sparsity Proportion for SPD sparse matrix



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 analyzed 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 the SPD matrix the execution time has a quadratic relation with the input size. The theoretical complexity of parallel algorithms 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 matrices the execution time has a cubic relation with input size. This additional time is due to the computation of A T A.
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.

Github Repo Link : https://github.com/aditiganvir28/Conjugate_Gradient_CS359

References:

● Wikipedia. Conjugate gradient method. (https://en.wikipedia.org/wiki/Conjugate_gradient_method)
● Caraba, E. . A Parallel Implementation Of The Conjugate Gradient Method. (https://www.semanticscholar.org/paper/A-Parallel-Implementation-Of-The-Conjugate-Gradient-Caraba/eaedaec18d8931d99abe7fbd5b48e3de6997b7be?p2df)
● 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)