CONJUGATE GRADIENT METHOD

CS359 - Parallel Computing

B Harshavardhan - 150001003
K Ganesh Raj - 150001012

<u>Introduction:</u> Conjugate means orthogonal and this method works on minimising a Quadratic equation using its gradient. The Conjugate Gradient Method is the most prominent iterative method for solving sparse systems of linear equations.

In most cases of interest the coefficient matrix is sparse, i.e., the number of zeroes is so large that storing the matrix as a two-dimensional array is impractical or impossible. So many of the sparse linear system solvers for distributed memory systems use iterative methods.

An iterative method makes an initial guess at a solution to the system, and then tries to repeatedly improve the guess. Thus, iterative methods produce a sequence of vectors, x0,x1,... each of which is an approximation to the solution to the linear system. For programming assignment 2, you should use C and MPI to implement the iterative method known as the method of conjugate gradients.

<u>Description:</u>

Consider a linear system

$$Ax = b$$

The matrix is positive definite if for every nonzero vector x, the scalar

$$x \cdot Ax > 0.$$

The method of conjugate gradients can also be viewed as solving a minimization problem:

Suppose q(x) = $1/2x.Ax - x.b$

where A is a symmetric positive definite n×n matrix, x and b are n-dimensional column vectors, and · denotes the dot product. In order to find the minimum value of q,

we can compute its gradient which turns out to be ∇q(x) = Ax−b, and it can be shown that the solution to Ax − b = 0 is the unique minimum of q. So minimizing q and solving the linear system Ax = b are equivalent problems.

After making an initial guess at a solution to the linear equation (or the minimum of q), the conjugate gradient method proceeds by choosing "search directions"

p1,p2,p3………..

Each of these directions is just an n-dimensional vector. Once a direction is chosen, a new estimate for the solution is obtained by minimizing q along the new direction. That is, if pk is the new search direction, and $x_k - 1$ is the previous estimate of a solution, then the new estimate, x_k will be chosen by choosing the scalar α_k that minimizes

$$min\ q(x_{k-1} + \alpha p_k)$$

So we need to see how to choose the search directions. A key ingredient in the choice is the residual. If x_k is one of our estimates, then the corresponding residual is

$$r_{k=}\ b - Ax_k$$

It can be viewed as a measure of how close xk is to the solution to the original linear system. Having chosen search directions p1,p2,...,pk−1, and computed estimates x0,x1,...,xk−1 and residuals r0,r1,...,rk−1, the method chooses the new search direction, pk, from the orthogonal complement of

$$Span\{Ap_{1.}\ Ap_{2,}\ ......\}$$

More explicitly, pk is chosen from this subspace so that it minimizes the distance

$$\left\|p_k - r_{k-1}\right\|$$

in the usual Euclidean norm. It can be shown that if

$$\beta_k = \frac{r_{k-1} \cdot r_{k-1}}{r_{k-2} \cdot r_{k-2}},$$

then p_k will minimize $\|p_k - r_{k-1}\|$, when

$$p_k = r_{k-1} + \beta_k p_{k-1}.$$

With this choice of p_k, it can also be shown that if

$$\alpha_k = \frac{r_{k-1} \cdot r_{k-1}}{p_k \cdot Ap_k},$$

then

$$x_k = x_{k-1} + \alpha_k p_k$$

will minimize $q(x_{k-1} + \alpha p_k)$.

The Method : So from the above conclusions the Conjugate Gradient method can be written as :

```
$k = 0;\ x_0 = 0;\ r_0 = b$
while $(\|r_k\|^2 > \text{tolerance})$ and $(k < \text{max\_iter})$
    $k{+}{+}$
    if $k = 1$
        $p_1 = r_0$
    else
        $\beta_k = \frac{r_{k-1} \cdot r_{k-1}}{r_{k-2} \cdot r_{k-2}}$      // minimize $\|p_k - r_{k-1}\|$
        $p_k = r_{k-1} + \beta_k p_{k-1}$
    endif
    $s_k = Ap_k$
    $\alpha_k = \frac{r_{k-1} \cdot r_{k-1}}{p_k \cdot s_k}$          // minimize $q(x_{k-1} + \alpha p_k)$
    $x_k = x_{k-1} + \alpha_k p_k$
    $r_k = r_{k-1} - \alpha_k s_k$
endwhile
$x = x_k$
```

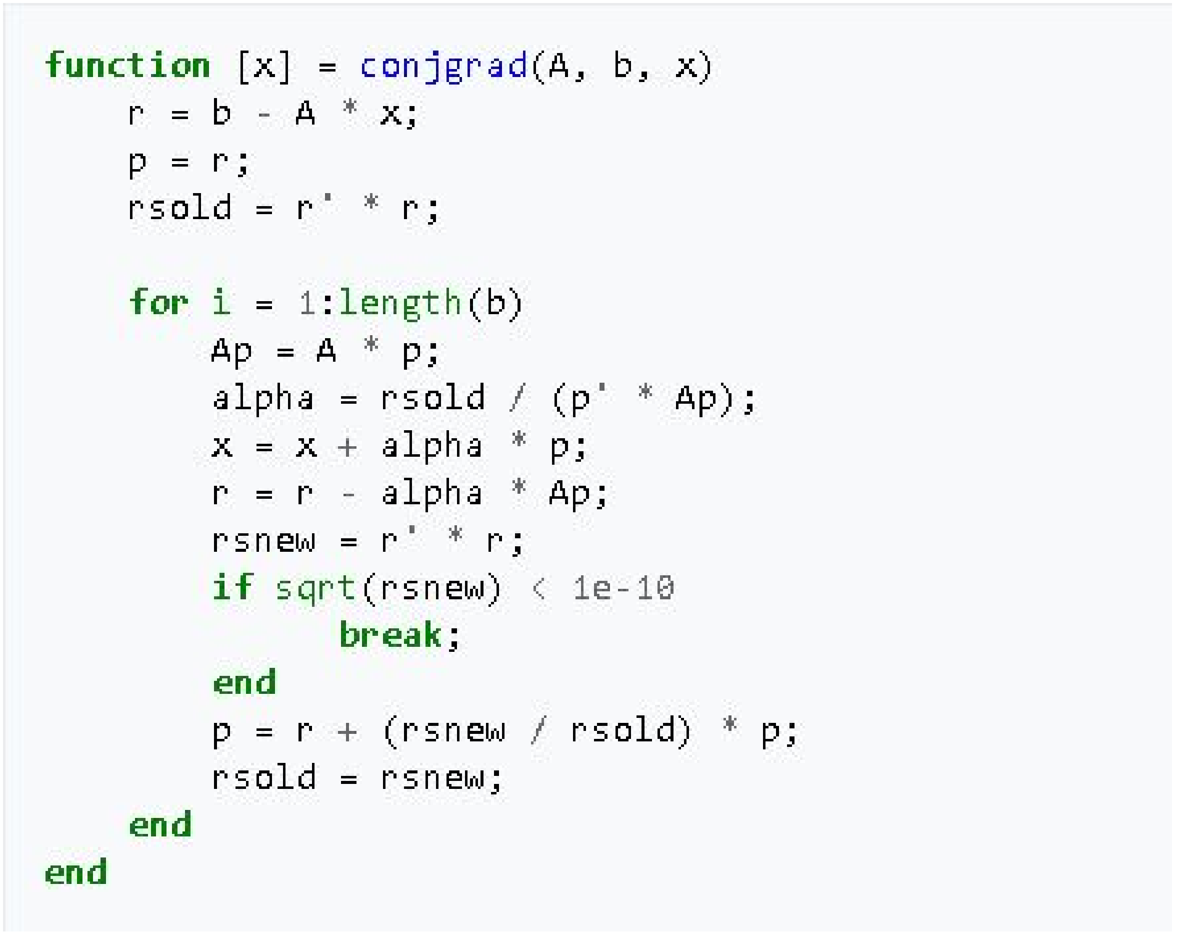
Characteristics of this method :

- ❏ This method only for positive definite systems.
- ❏ Symmetric matrices are preferable.
- ❏ In general the solution is obtained in at most *n* iterations.
- ❏ The solutions are accepted only if the error is less than the tolerance limit.
- ❏ The iterations are ended when the error becomes less than the tolerance limit.
- ❏ This is both direct and an iterative method.
- ❏ The rate of convergence depends on the square root of the condition number.
- ❏ The consecutive search directions are orthogonal.
- ❏ The error $||x - x_k||$ decreases at a rapid rate sometime approaching the answer in number of iterations much smaller than *n.*

Parallelisation :

This method can be parallelised to make it efficient by analysing functions that involved in the method and parallelising them individually.

The basic algorithm for implementing Conjugate Gradient method is:



```
function [x] = conjgrad(A, b, x)
    r = b - A * x;
    p = r;
    rsold = r' * r;

    for i = 1:length(b)
        Ap = A * p;
        alpha = rsold / (p' * Ap);
        x = x + alpha * p;
        r = r - alpha * Ap;
        rsnew = r' * r;
        if sqrt(rsnew) < 1e-10
              break;
        end
        p = r + (rsnew / rsold) * p;
        rsold = rsnew;
    end
end
```

The data types used are:

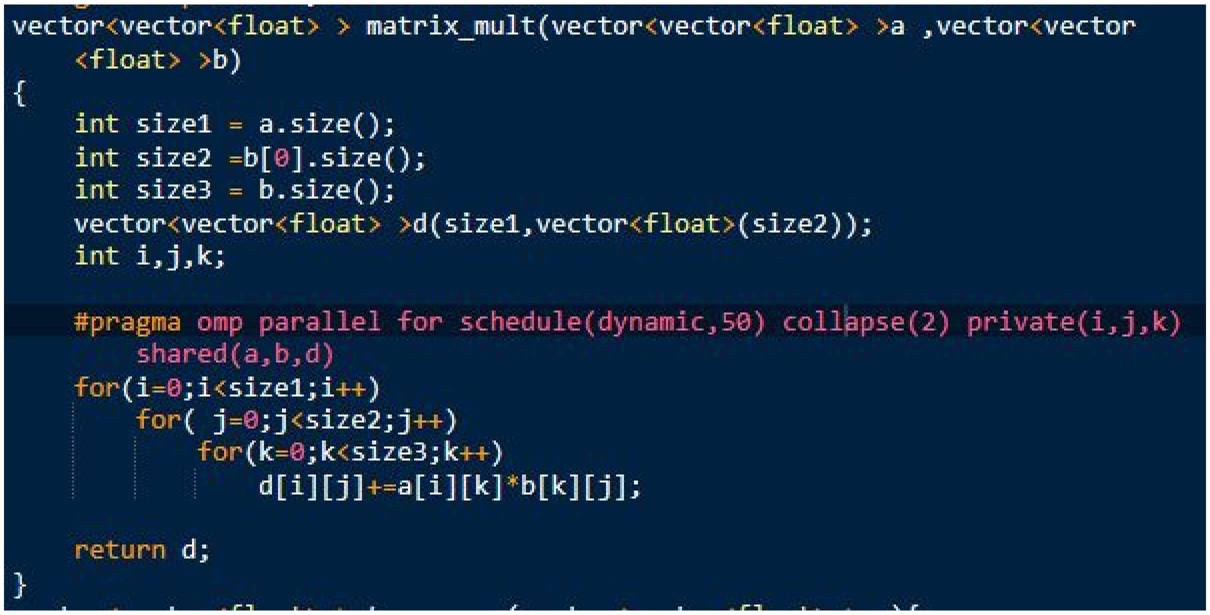
Vector -   wherever matrices and vectors where required.
Double float - for scalars and norm calculations.

The functions that can be parallelised are;

- Matrix Multiplication
- Matrix addition
- Matrix subtraction.
- Euclidian norm of matrix
- Transpose of a matrix
- Matrix division and multiplication by a scalar.

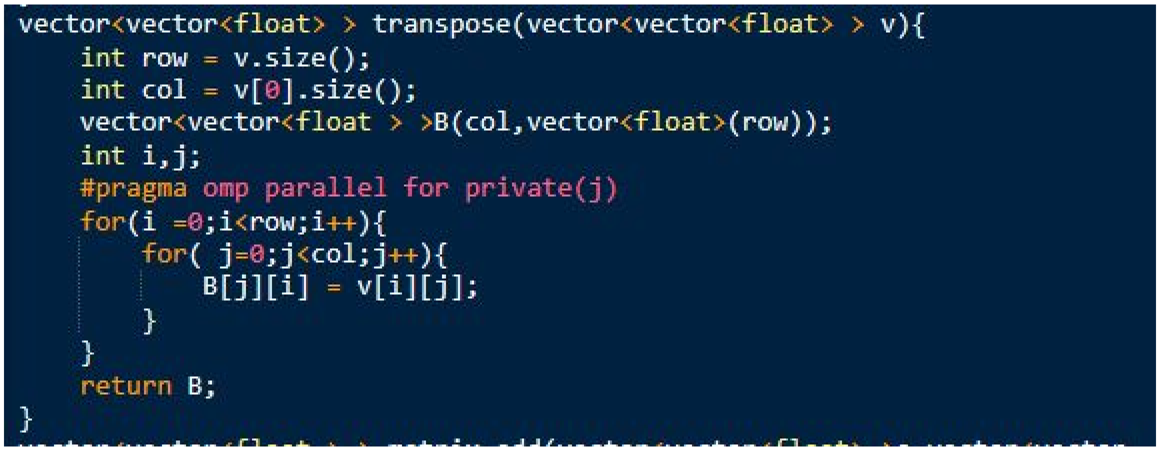
The code snippets of the final parallelised functions are given below:

matrix multiplication :



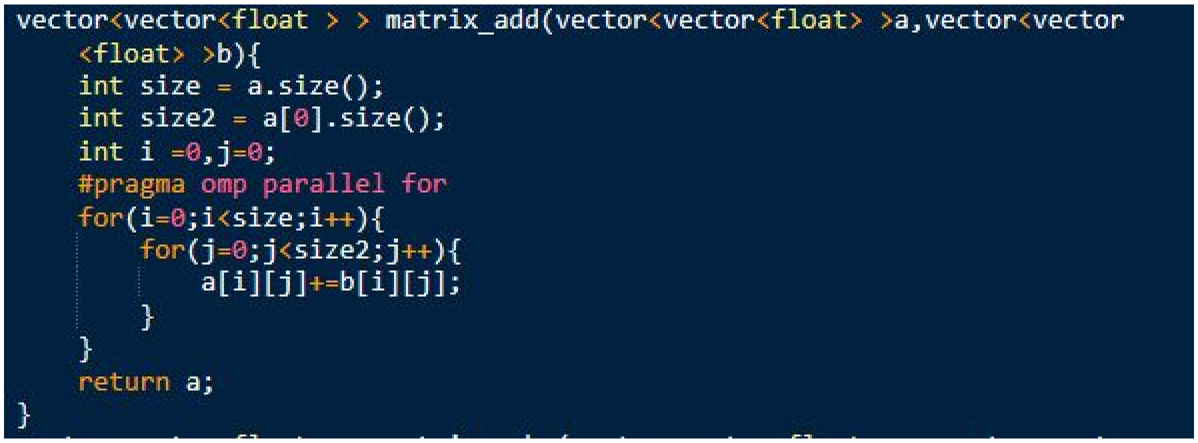
```
vector<vector<float> > matrix_mult(vector<vector<float> >a ,vector<vector
    <float> >b)
{
    int size1 = a.size();
    int size2 =b[0].size();
    int size3 = b.size();
    vector<vector<float> >d(size1,vector<float>(size2));
    int i,j,k;

    #pragma omp parallel for schedule(dynamic,50) collapse(2) private(i,j,k)
        shared(a,b,d)
    for(i=0;i<size1;i++)
        for( j=0;j<size2;j++)
            for(k=0;k<size3;k++)
                d[i][j]+=a[i][k]*b[k][j];

    return d;
}
```

Matrix transpose :



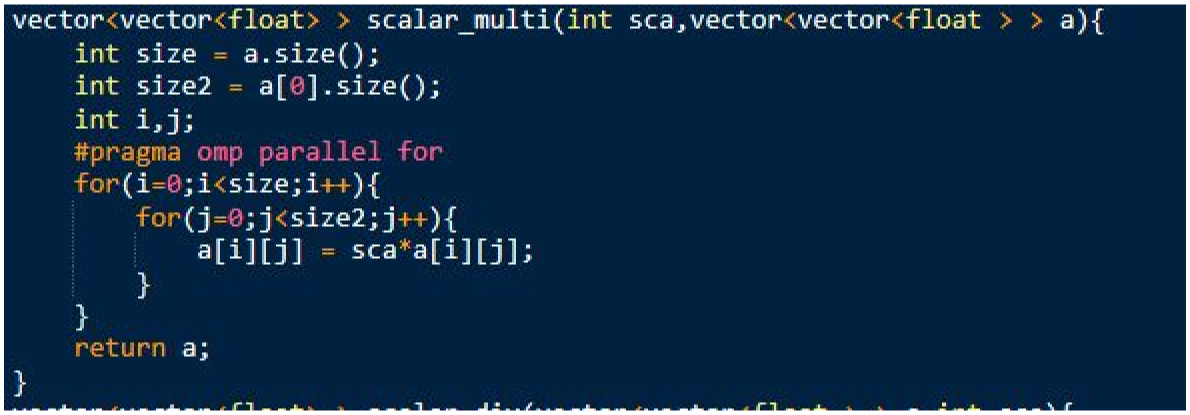
```
vector<vector<float> > transpose(vector<vector<float> > v){
    int row = v.size();
    int col = v[0].size();
    vector<vector<float > >B(col,vector<float>(row));
    int i,j;
    #pragma omp parallel for private(j)
    for(i =0;i<row;i++){
        for( j=0;j<col;j++){
            B[j][i] = v[i][j];
        }
    }
    return B;
}
```

Matrix addition :



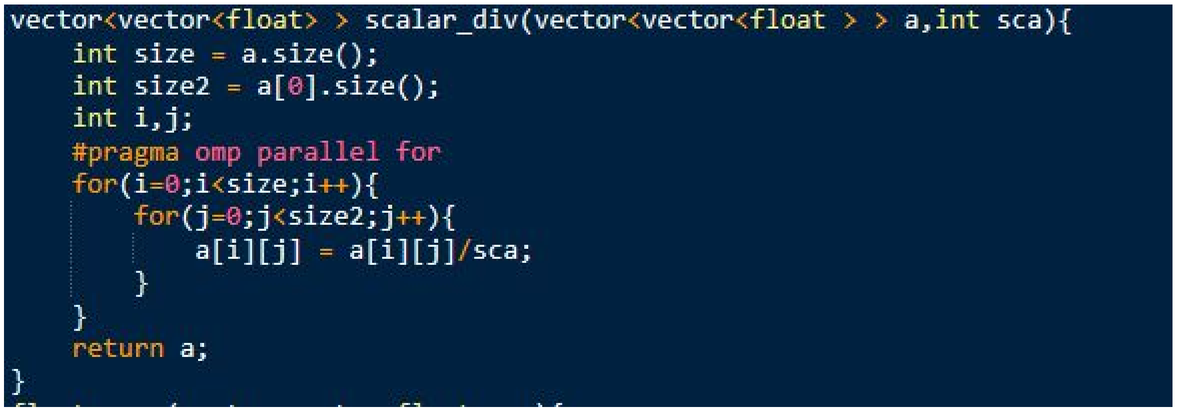
```
vector<vector<float > > matrix_add(vector<vector<float> >a,vector<vector
    <float> >b){
    int size = a.size();
    int size2 = a[0].size();
    int i =0,j=0;
    #pragma omp parallel for
    for(i=0;i<size;i++){
        for(j=0;j<size2;j++){
            a[i][j]+=b[i][j];
        }
    }
    return a;
}
```

Scalar and Matrix multiplication :



```
vector<vector<float> > scalar_multi(int sca,vector<vector<float > > a){
    int size = a.size();
    int size2 = a[0].size();
    int i,j;
    #pragma omp parallel for
    for(i=0;i<size;i++){
        for(j=0;j<size2;j++){
            a[i][j] = sca*a[i][j];
        }
    }
    return a;
}
```

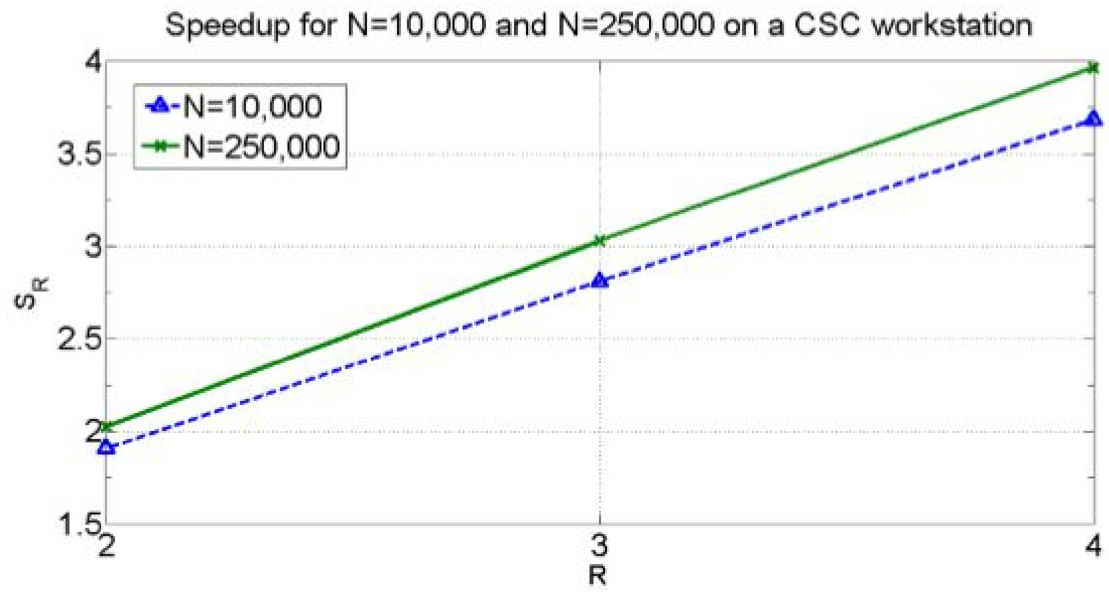
Scalar and matrix division :



```
vector<vector<float> > scalar_div(vector<vector<float > > a,int sca){
    int size = a.size();
    int size2 = a[0].size();
    int i,j;
    #pragma omp parallel for
    for(i=0;i<size;i++){
        for(j=0;j<size2;j++){
            a[i][j] = a[i][j]/sca;
        }
    }
    return a;
}
```

Conclusion : The difference in times between parallel and sequential algorithms can be used to calculate Speedup and efficiency.

The values of Speedup when plotted for a range of values would appear like this :



The speedup depends on a lot of factors like data creation overhead, individual processor performance …. etc. The above graph is an approximate estimation.

References:

- Parallel CG method - KTH stockolm
- Painless CG method - Carnegie Mellon University
- The CG method - University of Oslo