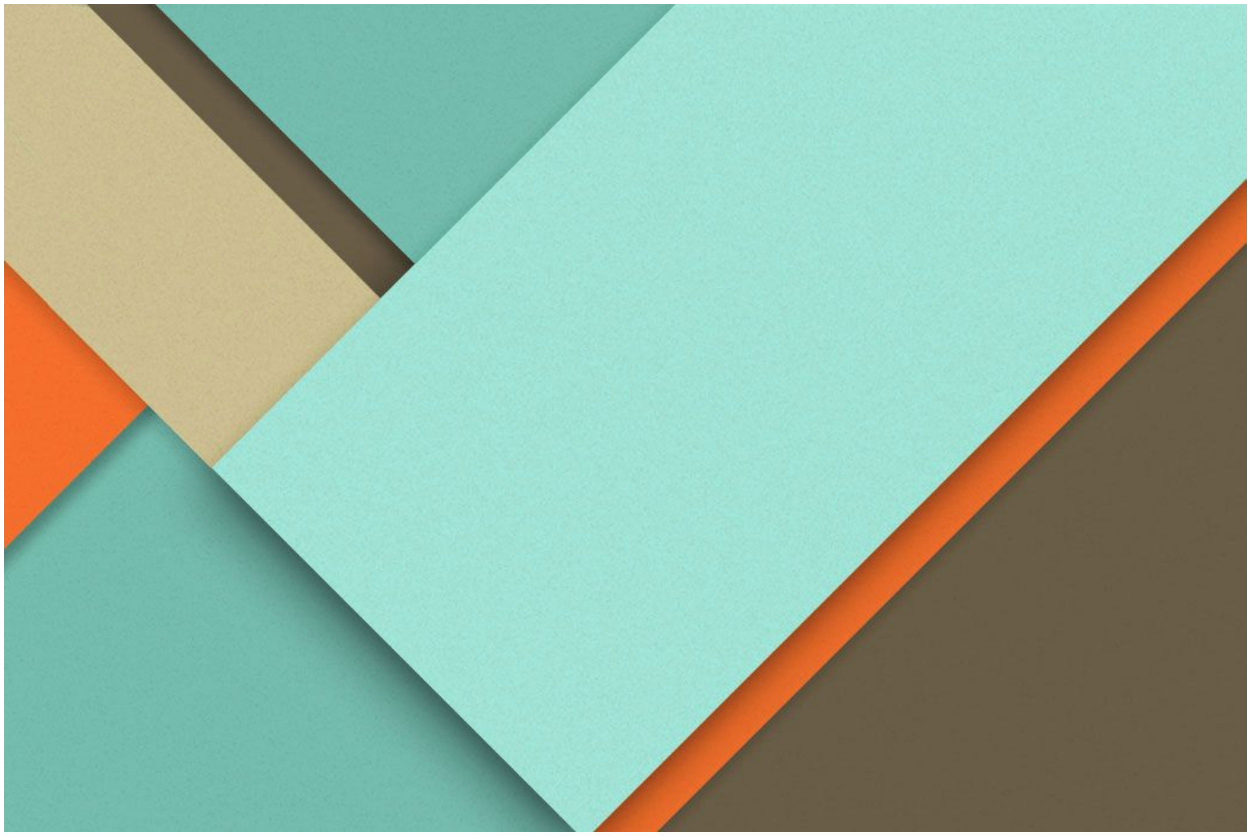


Gaussian Elimination

—

Jhalak Gupta
170001024

Ashwini Jha
170001012

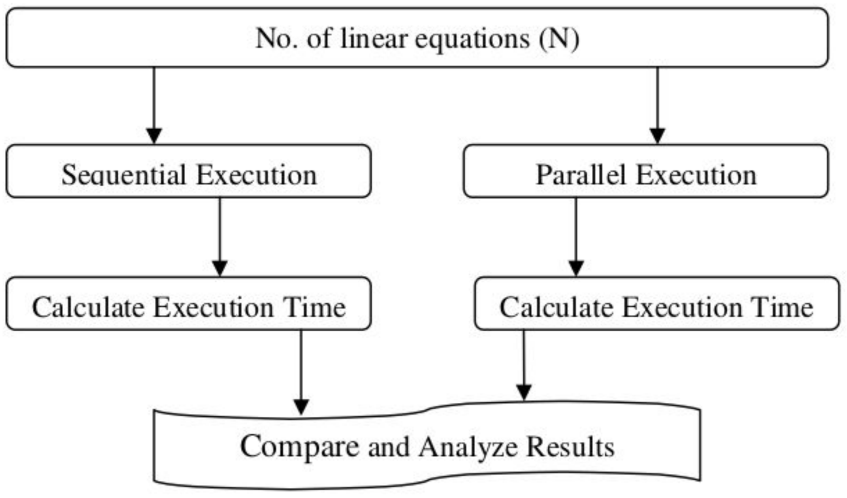
Introduction:

Given a system Ax = b, we can utilize several different methods to obtain a solution. If a unique solution is known to exist, and the coefficient matrix is full, a direct method such as Gaussian Elimination is usually selected.

Gaussian elimination, also known as row reduction, is an algorithm in linear algebra for solving a system of linear equations. It is usually understood as a sequence of operations performed on the corresponding matrix of coefficients. This method can also be used to find the rank of a matrix, to calculate the determinant of a matrix, and to calculate the inverse of an invertible square matrix. The method is named after Carl Friedrich Gauss (1777–1855). Some special cases of the method - albeit presented without proof - were known to Chinese mathematicians as early as circa 179 AD.

A matrix can always be transformed into an upper triangular matrix and in fact, one that is in row echelon form. Once all of the leading coefficients (the leftmost nonzero entry in each row) are 1, and every column containing a leading coefficient has zeros elsewhere, the matrix is said to be in reduced row echelon form.

We describe the techniques and algorithm involved in achieving good performance by reducing execution time through OpenMP Parallelization on multi-core. We tested the algorithms by writing the program using OpenMP on the multi-core system and measure their performances with their execution times.



The parallel algorithm, which is used to solve a dense system of linear equations using Gaussian elimination with partial pivoting, is developed. Gauss method is based on the transformation of linear equations, which do not change the solution. It includes the transformations: Multiplication of any equation by a nonzero constant, Permutation of equations, Addition of any system of the equation to other equation. This algorithm consists of two phases:

1. In the first phase, the Pivot element is identified as the largest absolute value among the coefficients in the first column. Then Exchange the first row with the row containing that element. Then eliminate the first variable in the second equation using transformation. When the second row becomes the pivot row, search for the coefficients in the second column from the second row to the n-th row and locate the largest coefficient. Exchange the second row with the row containing the largest coefficient. Continue this procedure till (n-1) unknowns are eliminated.

2. The second phase in known as backward substitution phase which is concerned with the actual solution of the equations and uses the back substitution process on the reduced upper triangular system.

Sequential algorithm

```
Input: Given Matrix a [1: n, 1: n+1]
Output: x [1: n]
// Traingularization process
for k = 1 to n-1
Find pivot row by searching the row with greatest absolute value among the
elements of
column k
swap the pivot row with row k
for i = k+1 to n
m i,k = a i,k / a k,k
for j = k to n+1
a i,j = a i,j - m i,k * a k,j
End loop [j]
End loop [i]
End loop [k]
// Back substitution process
xn = a n,n+1 / a n,n
for i = n-1 to 1 step -1 do
sum = 0
for j = i+1 to n do
sum = sum + a i,j * xj
```

```
End loop [j]
xi = ( a i,n+1 - sum )/a i,i
End loop[i]
```

In the sequential algorithm of Gauss elimination we found that the innermost loops indexed by i, and j can be executed in parallel without affecting the result. In the parallel algorithm, we insert the #pragma directive to parallelize the loops.

Parallel algorithm

```
Input: a [1: n, 1: n+1] // Read the matrix data
Output: x [1: n]
Set the numbr of threads
Strat clock
// Traingularization process
for k = 1 to n-1
Insert #pragma directive to parallelize
for i = k+1 to n
m i,k = a i,k / a k,k
for j = k to n+1
a i,j = a i,j - m i,k * a k,j
End loop [j]
End loop [i]
End loop [k]
// Back substitution process
xn = a n,n+1 / a n,n
for i = n-1 to 1 step -1 do
sum = 0
for j = i+1 to n do
sum = sum + a i,j * xj
End loop [j]
xi = ( a i,n+1 - sum )/a i,i
End loop[i]
Stop clock
Display time taken and solution vector
```

Sequential Implementation:

```
static void serial(vector<double> M, vector<double> b, const int size) {
          vector<pair<int,int> > swaps;
          vector<double> x(size);
          {
            vector<double> temp(size);
            Timer t("\n\nSERIAL\t\t");
              for(int i = 0; i < size - 1; i++) {

         //checking for divide by zero error: possible in sparse matrix
                  if( M[i * size + i] == 0 ){
                        cout<<"DIVIDE BY ZERO DETECTED\n";
                        int index = i;
                        //int maxVal = 0;
                        for(int j = i + 1; j < size; j++){
                              if(M[j * size + i] != 0) index = j;
                        }
                        if(index == i){
                              cout<<"THERE IS NO UNIQUE SOLUTION\n";
                              return;
                        }
                        else{
                              cout<<"DIVIDE BY ZERO ERROR SOLVED\n";
                        }
                        swaps.push_back(make_pair(i,index));
                        swap(b[i],b[index]);
                        //swapping rows
                        for(int j = 0; j < size; j++){
                              temp[j] = M[index * size + j];
                        }

                        for(int j = 0; j < size; j++){
                              M[index * size + j] = M[i * size + j];
                        }

                        for(int j = 0; j < size; j++){
                              M[i * size + j] = temp[j];
                        }
```

```
       }

       for(int j = i + 1; j < size; j++) {

         double k = -M[j * size + i] / M[i * size + i];
         b[j] += b[i] * k;

         for(int l = i; l < size; l++) {
           M[j * size + l] += M[i * size + l] * k;
         }
       }
     }

     x[size - 1] = b[size - 1] / M[size * size - 1];
     for(int i = size - 2; i >= 0; i--) {
       double sum = 0;
       for(int j = i + 1; j < size; j++) {
         sum += M[i * size + j] * x[j];
       }
       x[i] = (b[i] - sum) / M[i * size + i];
     }

   }
   cout<<"AUGMENTED MATRIX AFTER ROW OPERATIONS:\n";

   int size2 = min(size,7);
   cout<<"\n";
   for(int i=0;i<size2;i++){
    cout<<"\t";
    for(int j=0;j<size2;j++){

       cout<<M[i*size + j]<<"\t";
    }
    cout<<"\t|\t"<<b[i]<<"\t\t"<<endl;
   }
   cout<<"\n\nSOLUTION VECTOR: \n";
   for(int i=0;i<size2;i++){
    cout<<"\tx"<<i+1<<":\t"<<x[i]<<endl;
   }
  return;
}
```

Parallel Implementation:

```
static void parallel(vector<double> M, vector<double> b, const int size) {
      vector<double> x(size);
          {
            Timer t("\n\nPARALLEL\t");
              double k;
              int i, j, l;

              for(i = 0; i < size - 1; i++) {
                  #pragma omp parallel for shared(M, b) private(k, j, l)
                  for(j = i + 1; j < size; j++) {
                      double k = -M[j * size + i] / M[i * size + i];
                      b[j] += b[i] * k;

                      #pragma omp simd
                      for(l = i; l < size; l++) {
                          M[j * size + l] += M[i * size + l] * k;
                      }
                  }
              }


          }
          x[size - 1] = b[size - 1] / M[size * size - 1];
        for(int i = size - 2; i >= 0; i--) {
            double sum = 0;

            #pragma omp simd
            for(int j = i + 1; j < size; j++) {
                sum += M[i * size + j] * x[j];
            }
            x[i] = (b[i] - sum) / M[i * size + i];
        }


          cout<<"AUGMENTED MATRIX AFTER ROW OPERATIONS:\n";

          int size2 = min(size,7);
          cout<<"\n";
          for(int i=0;i<size2;i++){
            cout<<"\t";
            for(int j=0;j<size2;j++){

                  cout<<M[i*size + j]<<"\t";

            }

            cout<<"\t|\t"<<b[i]<<"\t\t"<<endl;
          }
          cout<<"\n\nSOLUTION VECTOR: \n";
          for(int i=0;i<size2;i++){
            cout<<"\tx"<<i+1<<":\t"<<x[i]<<endl;
          }
    }
```

NOTE: **This code may not give correct results for Sparsely populated Matrix as Input!**

Parallel Implementation with Sparsely populated matrix:

```
static void parallel(vector<double> M, vector<double> b, const int size) {
      vector<pair<int,int> > swaps;
      vector<double> x(size);
          {
            vector<double> temp(size);
            Timer t("\n\nPARALLEL\t");
              double k;
              int i, j, l;
              for(i = 0; i < size - 1; i++) {
       //checking for divide by zero error: possible in sparse matrix
                  if( M[i * size + i] == 0 ){
                        cout<<"DIVIDE BY ZERO ERROR DETECTED\n";
                        int index = i;
                        #pragma omp parallel for
                        for(int j = i + 1; j < size; j++){
                              if(M[j * size + i] != 0) index = j;
                        }
                        if(index == i){
                              cout<<"THERE IS NO UNIQUE SOLUTION\n";
                              return;
                        }
                        else{
                              cout<<"DIVIDE BY ZERO ERROR SOLVED\n";
                        }
                        swaps.push_back(make_pair(i,index));
                        swap(b[i],b[index]);
                        //swapping rows
                        #pragma omp parallel for
                        for(int j = 0; j < size; j++){
                              temp[j] = M[index * size + j];
                        }
                        #pragma omp parallel for
                        for(int j = 0; j < size; j++){
                              M[index * size + j] = M[i * size + j];
                        }
                        #pragma omp parallel for
                        for(int j = 0; j < size; j++){
```

```
             M[i * size + j] = temp[j];
          }
       }
       #pragma omp parallel for shared(M, b) private(k, j, l)
       for(j = i + 1; j < size; j++) {
         double k = -M[j * size + i] / M[i * size + i];
         b[j] += b[i] * k;

         #pragma omp simd
         for(l = i; l < size; l++) {
           M[j * size + l] += M[i * size + l] * k;
         }
       }
     }
   }
   x[size - 1] = b[size - 1] / M[size * size - 1];
  for(int i = size - 2; i >= 0; i--) {
    double sum = 0;

    #pragma omp simd
    for(int j = i + 1; j < size; j++) {
      sum += M[i * size + j] * x[j];
    }
    x[i] = (b[i] - sum) / M[i * size + i];
  }
   cout<<"AUGMENTED MATRIX AFTER ROW OPERATIONS:\n";
   int size2 = min(size,7);
   cout<<"\n";
   for(int i=0;i<size2;i++){
    cout<<"\t";
    for(int j=0;j<size2;j++){
       cout<<M[i*size + j]<<"\t";
    }
    cout<<"\t|\t"<<b[i]<<"\t\t"<<endl;
   }
   cout<<"\n\nSOLUTION VECTOR: \n";
   for(int i=0;i<size2;i++){
    cout<<"\tx"<<i+1<<":\t"<<x[i]<<endl;
   }
}
```

Complexity Analysis:

SEQUENTIAL ALGORITHM:

Time Complexity: $O(n^3)$

No of Processors: 1

Work Complexity: $O(n^3)$

Cost Complexity: $O(n^3)$

PARALLEL ALGORITHM:

Time Complexity: $O(n)$

No. of Processors: $O(n^2)$

Work Complexity: $O(n^3)$

Cost Complexity: $O(n^3)$

PARALLEL ALGORITHM (with sparse matrix as input):

Time Complexity: $O(n^3)$

No of Processors: $O(n^2)$

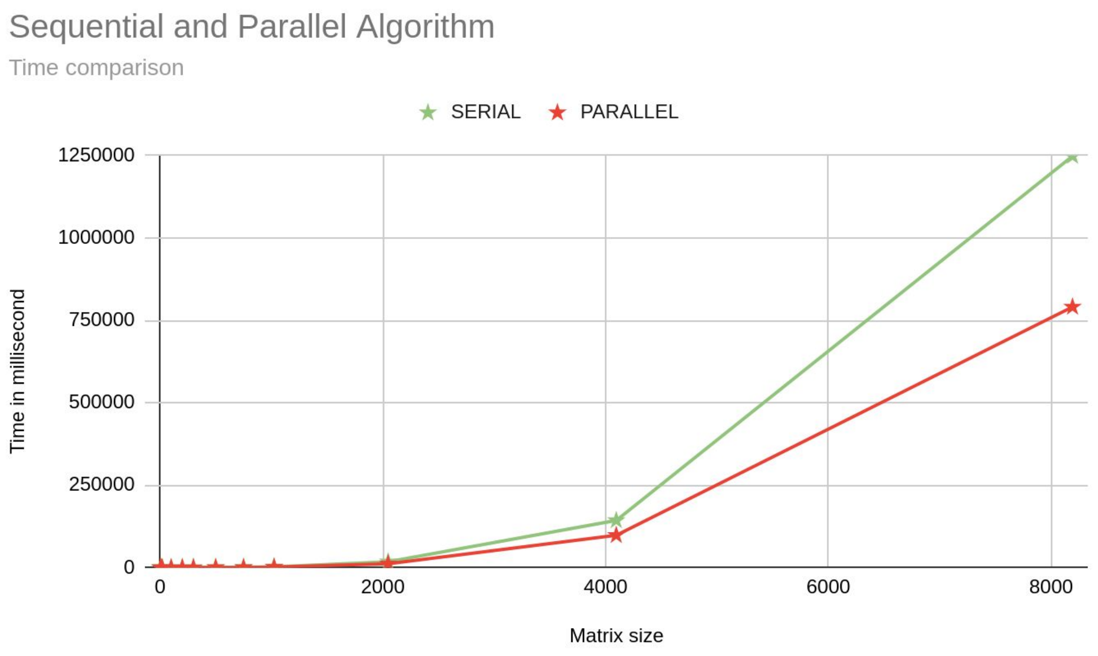
Work Complexity: $O(n^3)$

Cost Complexity: $O(n^3)$

Results:

Comparison of Parallel and Sequential Algorithm:

Matrix Size	**Sequential Algorithm** (in micro seconds)	**Parallel Algorithm** (in micro seconds)
5	5.96	344.97
10	25.199	1.47E+03
20	145.826	2734.25
100	5478.42	8436.7
200	21709.2	43198.1
300	60715.2	66552.6
500	257319	179137
750	908294	558812
1024	2.30E+06	1.59E+06
2048	1.78E+07	1.15E+07
4096	1.43E+08	9.81E+07
8192	1.25E+09	7.91E+08



Conclusion:

(1) We have observed that parallelizing serial algorithm using OpenMP has increased the performance.

(2) For multi-core system, OpenMP provides a lot of performance increase and parallelization can be done with careful small changes.

(3) The parallel algorithm takes more time in case of small input size with comparison to the sequential algorithm, but as the size of input increases, time taken by parallel algorithm is reduces as compared to sequential.

(4) The parallel algorithm is approximately twice faster than the sequential and the speedup is linear.

References:

1. https://cs.wmich.edu/elise/courses/cs626/s12/cs6260_presentation_1.pdf
2. https://www.researchgate.net/publication/276197438_Performance_Analysis_of_Parallel_Algorithms_on_Multi-core_System_using_OpenMP?enrichId=rgreq-008e60ebcf05ed5cf084be09804db03f-XXX&enrichSource=Y292ZXJQYWdlOzI3NjE5NzQzODtBUzo2NDcxOTA3MzMxNDQwNjRAMTUzMTMxMzcwMzE4OA%3D%3D&el=1_x_3&_esc=publicationCoverPdf
3. http://homepages.math.uic.edu/~jan/mcs572/parallelLU.pdf
4. https://www.geeksforgeeks.org/gaussian-elimination/
5. https://en.wikipedia.org/wiki/Gaussian_elimination
6. https://www3.nd.edu/~zxu2/acms60212-40212-S12/Gaussian-openMP.pdf