PARALLEL COMPUTING PROJECT REPORT

CS 359

Mihir Patel - 210001050
Abhinav Kumar - 210001002

ABSTRACT

Dense matrix computations play a crucial role in various scientific applications, forming the foundation for key program kernels across scientific domains. One essential application is solving large systems of linear equations arising from diverse fields such as circuit simulation, structural analysis, and power networks. To address the linear equation problem $AX = B$, various methods can be employed, with LU factorization standing out as a direct scheme.

LU decomposition proves to be a versatile technique, offering solutions not only for linear systems but also for finding the inverse of a matrix, computing determinants, and solving systems of linear equations. The conventional Doolittle algorithm sequentially decomposes a matrix into the product of an upper triangular matrix and a lower triangular matrix.

To enhance the efficiency of this algorithm, particularly when dealing with a substantial number of unknowns, the primary challenge lies in implementing a parallelized version that demonstrates improved speedup.

SEQUENTIAL ALGORITHM DESIGN

The algorithm employed computes the lower triangular matrix (L) and upper triangular matrix (U) from a given matrix A. Matrix A is generated using the rand() function, which provides random values. The initial values of the upper and lower triangular matrices are set to zero.

To introduce variability in the generated random values, the srand() function is utilized with a seed parameter. The seed is initialized by the gettimeofday() function. srand() serves the purpose of randomizing the values produced by rand() by using the provided seed.

PARALLEL ALGORITHM DESIGN

To enhance the algorithm's speed, parallel implementation was deemed essential, requiring the distribution of computational tasks among CPU processors. Analyzing the legacy sequential code revealed sections where execution time was significant, prompting the transformation into parallel code. The chosen approach for parallelization is data parallelism.

In the analysis, it was observed that data independence exists between the computation of row elements in any column of the lower triangular matrix and any other row element in that column. Similarly, calculating the column elements of any row in an upper triangular matrix is independent of any other column element in that row. Leveraging this independence, data parallelism is employed. Rows or columns are divided, and their elements are calculated concurrently to reduce execution time compared to serial execution.

The implementation strategy builds upon an existing sequential C implementation of the algorithm. OpenMP directives are introduced, and necessary modifications are made. The parallelization of the Doolittle algorithm incorporates block-wise decomposition. Parallel threads handle the decomposition of matrix blocks by distributing the workload among them, effectively parallelizing the for loops.

SEQUENTIAL ALGORITHM

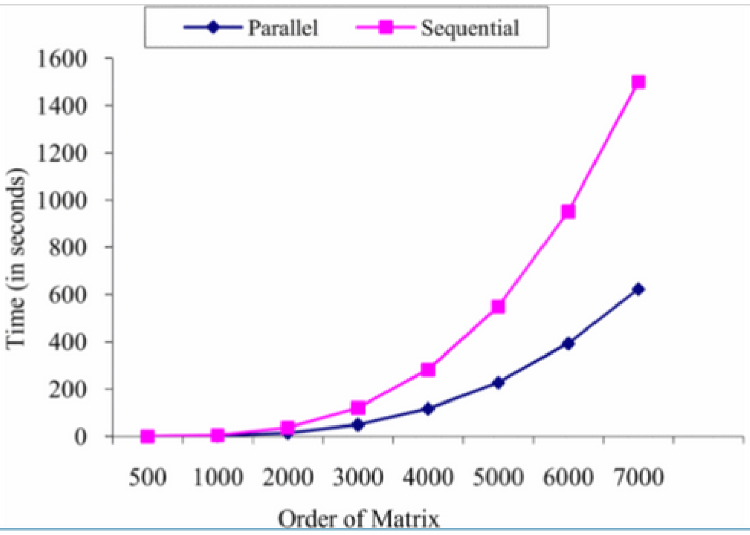
```
for(k=0;k<n;k++) {
   lwr[k][k]=1;
   for(j=k;j<n;j++){
     ld sum=0;
     for(int s=0;s<=k-1;s++){
       sum+=lwr[k][s]*upr[s][j];
     }
     upr[k][j]=a[k][j]-sum;
   }
for(i=k+1;i<n;i++){
     ld sum=0;
     for(int s=0;s<=k-1;s++) {
       sum+=lwr[i][s]*upr[s][k];
     }
     lwr[i][k]=(a[i][k]-sum)/upr[k][k];
   }
  }
```

PARALLEL ALGORITHM

```
#pragma omp parallel shared(lwr, upr, a, nthreads)
private(tid, i, k, j)
{
  tid = omp_get_thread_num();
  if (tid == 0)
    nthreads = omp_get_num_threads();
  for (k = 0; k < n; k++)
  {
    lwr[k][k] = 1;
#pragma omp for schedule(static, chunk)
    for (j = k; j < n; j++)
    {
      ld sum = 0;
      for (int s = 0; s <= k - 1; s++)
      {
        sum += lwr[k][s] * upr[s][j];
      }
      upr[k][j] = a[k][j] - sum;
    }
#pragma omp for schedule(static, chunk)
    for (i = k + 1; i < n; i++)
    {
      ld sum = 0;
      for (int s = 0; s <= k - 1; s++)
      {
        sum += lwr[i][s] * upr[s][k];
      }
      lwr[i][k] = (a[i][k] - sum) / upr[k][k];
    }
  }
}
```

TIME COMPLEXITY ANALYSIS

STATISTICS



n	Sequential Runtime	Parallel Runtime
2	0.000001	0.001365
3	0.000002	0.002003
5	0.000003	0.004033
10	0.00024	0.006001
100	0.006044	0.052817
500	0.212859	0.170541
750	0.618581	0.32374
1000	1.41235	0.587288

Theoretically:

- Sequential -> O(n ^ 3)
- Parallel -> O(n ^ 3 / p)

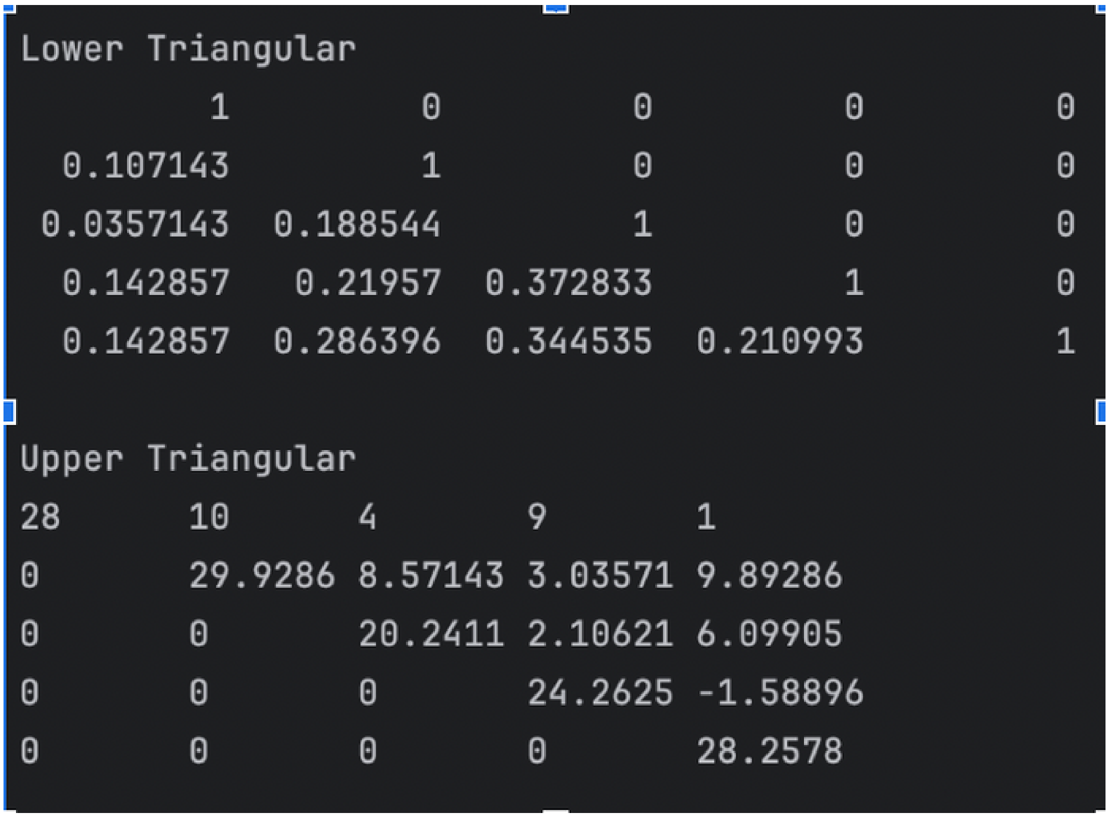
RESULT VALIDATION

EXAMPLE MATRIX

$$\begin{bmatrix} 28 & 10 & 4 & 9 & 1 \\ 3 & 31 & 9 & 4 & 10 \\ 1 & 6 & 22 & 3 & 8 \\ 4 & 8 & 10 & 27 & 3 \\ 4 & 10 & 10 & 8 & 33 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0.10714286 & 1 & 0 & 0 & 0 \\ 0.03571429 & 0.18854415 & 1 & 0 & 0 \\ 0.14285714 & 0.21957041 & 0.37283339 & 1 & 0 \\ 0.14285714 & 0.28639618 & 0.34453484 & 0.21099286 & 1 \end{bmatrix}$$

$$\times \begin{bmatrix} 28 & 10 & 4 & 9 & 1 \\ 0 & 29.92857143 & 8.57142857 & 3.03571429 & 9.89285714 \\ 0 & 0 & 20.24105012 & 2.10620525 & 6.09904535 \\ 0 & 0 & 0 & 24.26246905 & -1.58896357 \\ 0 & 0 & 0 & 0 & 28.25779268 \end{bmatrix}$$

CODE RESULT FOR BOTH SEQUENTIAL AND PARALLEL ALGORITHMS



```
Lower Triangular
         1         0         0         0         0
  0.107143         1         0         0         0
 0.0357143  0.188544         1         0         0
  0.142857   0.21957  0.372833         1         0
  0.142857  0.286396  0.344535  0.210993         1

Upper Triangular
28      10      4       9       1
0       29.9286 8.57143 3.03571 9.89286
0       0       20.2411 2.10621 6.09905
0       0       0       24.2625 -1.58896
0       0       0       0       28.2578
```

EXPERIMENTAL RESULTS

We implemented both Serial Doolittle Algorithm and the Parallel Doolittle Algorithm in the Quad core machine with the following configuration:

Processor: M1 Processor
RAM: 8 GB
OS: macOS
Compiler: GCC, OpenMP

In this work Doolittle algorithm was implemented that uses the block wise decomposition in parallel using OpenMp. The or loops are parallelized in a manner that blocks of matrices are decomposed by dividing the work among parallel threads. Algorithm was evaluated for input matrix of sizes 100, 1000,1500,2000.
The results on a multi-core processor show that the proposed Parallel Doolittle Algorithm achieves good performance (speedup) compared to the sequential algorithm.

CONCLUSION

In this work we have presented parallel implementation of the Doolittle Algorithm using OpenMP allowing the users to utilize the multiple cores present in the modern CPUs. The experimental results on a multi-core processor show that the proposed Parallel Doolittle Algorithm achieves good performance (speedup) compared to the sequential algorithm. For large order matrices Parallel Doolittle Algorithm can be readily used. In future, to further reduce the execution time of the Doolittle Algorithm we can utilize GPGPU (General Purpose Computing on Graphical Processing Unit) [6] and parallelize using CUDA, OpenCL.

REFERENCES

https://ieeexplore.ieee.org/document/7154772