IIT DELHI

COL380: Introduction to Parallel and Distributed Programming (Self Study)

Assignment-1 Report

Prof. Rijurekha Sen

Souvagya Ranjan Sethi (2019CS10405)
Department of Computer Science & Engineering

June 18, 2023

1 Directory Structure

The Zip folder contains 3 folders.

- **Sequential**
  This folder contains a file **main.cpp** which is the sequential code for the implementation of LU decomposition using gaussian elimination using partial pivoting. the algorithm for the implementation is same as provided in the problem statement.

  It also contains a **Makefile** for compiling the code and running the executable. You can change the program arguments in the make file in the run object.

```cpp
int LUD(double** A, double** L, double** U, int* P, int n){
    for(int k = 0; k < n; k++){
        int max_row = k;
        double max_val = 0;
        for(int i = k+1; i < n; i++){
            if(abs(A[i][k]) > max_val){
                max_val = abs(A[i][k]);
                max_row = i;
            }
        }
        if (max_val == 0 and k != n-1){
            cout << "Singular Matrix" << endl;
            return -1;
        } else if(max_row == k && k != n-1){
            continue;
        } else{
            swap(P[k], P[max_row]);
            for(int i = 0; i < n; i++){
                swap(A[k][i], A[max_row][i]);
            }
            for(int i = 0; i < k; i++){
                swap(L[k][i], L[max_row][i]);
            }
            U[k][k] = A[k][k];
            for(int i = k+1; i < n; i++){
                L[i][k] = A[i][k]/U[k][k];
                U[k][i] = A[k][i];
            }
            for(int i = 0; i < n; i++){
                for(int j = 0; j < n; j++){
                    A[i][j] = A[i][j] - L[i][k]*U[k][j];
                }
            }
        }
    }
    return 0;
}
```

- **OpenMP**

  This folder contains a `main.cpp` which contains the parallel implementation of the LU decomposition using the OpenMP threads. It also contains a `Makefile` for compiling and running the executable.

  Since the Gaussian Implementation updates the Matrix A at each step of the algorithm. We need to ensure that false sharing donot arise because of different threads different values of K. So, the threads parallelisation is applied to the update operation of L,U,A at step of K. This ensures that all the threads read the same value of matrix A.

```
  #pragma omp parallel for num_threads(t)
    for(int i = k+1; i < n; i++){
      L[i][k] = A[i][k]/U[k][k];
      U[k][i] = A[k][i];
    }

  #pragma omp parallel for num_threads(t)
    for(int i = 0; i < n; i++){
      for(int j = 0; j < n; j++){
        A[i][j] = A[i][j] - L[i][k]*U[k][j];
      }
    }
```

  In the second pragma loop, since the Threads operate on different values of matrix A and update them, false sharing is prevented. In the first pragma loop, threads are divided on i, and each operate to update different values of Lower and Upper Matrix.

  The program partitions the data column wise with contiuous blocks allocated to each thread. So, this ensures that the nearby data items are stored and can be fetched from the local cache of the thread. The parallel work is syncronized as each thread computes its work and updated its respective value and then proceeds to the next loop of k.

  We know that SpeedUp is the ratio of the ratio of the time it takes to run a program on one processor to the time it takes to run the same program on multiple processors.

$$S_p = \frac{\text{Execution time using 1 processor}(t_1)}{\text{Execution time using p processors}(t_p)}$$

  Efficiency is the ratio of SpeeUp to the number of processors used.

$$\epsilon_p = \frac{S_p}{p}$$

  Note: Since for n=8000, the time taken to run the overall code was taking too much time. I have used n=1000 and listed all the observations.

No. of Threads used	Time for LUD(in ns)	Parallel Efficiency	Total time
1	7.52239	1	4m 30.339s
2	4.21969	0.89	4m 25.771s
4	2.1802	0.86	4m 26.590s
8	0.958011	0.98	4m 25.980s
16	0.5275	0.89	4m 21.590s

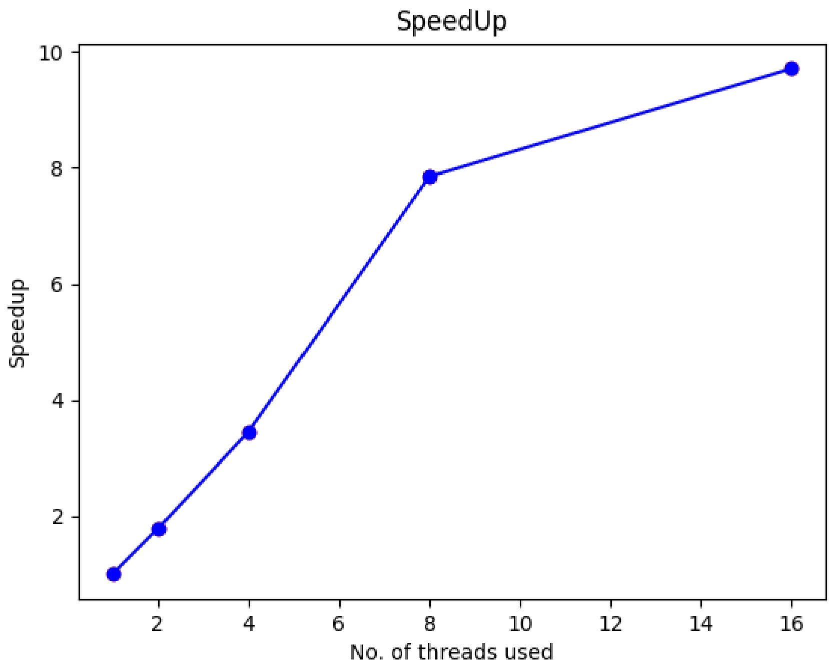

Figure 1: SpeedUp vs Number of Threads used

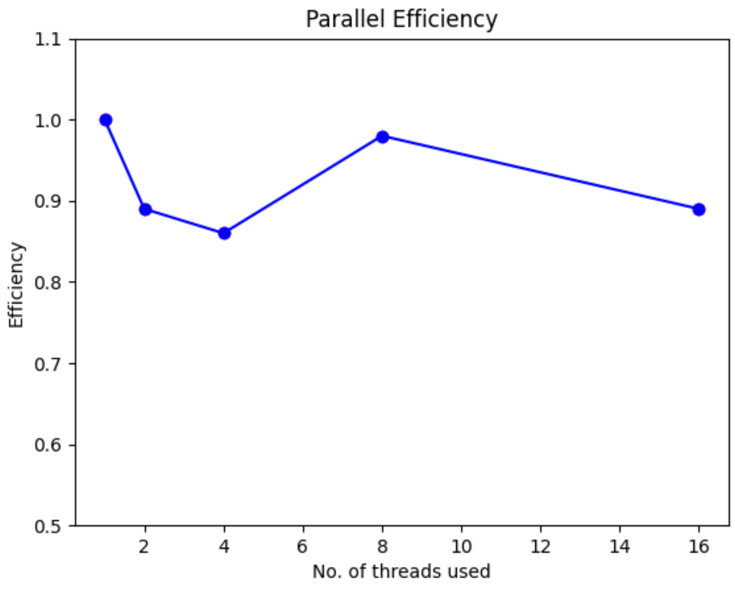

Figure 2: Effiicicency vs Number of threads Used

- **PThreads**
  The observations are similar to the Openmp threads.

```cpp
struct LUThreadArgs {
    double** A;
    double** L;
    double** U;
    int* P;
    int n;
    int t;
    int k;
    pthread_barrier_t* barrier;
};

// Function to perform the LU decomposition for a given k value (thread
   function)
void* LUThread(void* args) {
    LUThreadArgs* threadArgs = static_cast<LUThreadArgs*>(args);
    double** A = threadArgs->A;
    double** L = threadArgs->L;
    double** U = threadArgs->U;
    int* P = threadArgs->P;
    int n = threadArgs->n;
    int t = threadArgs->t;
    int k = threadArgs->k;
    pthread_barrier_t* barrier = threadArgs->barrier;

    for (int i = k + 1; i < n; i++) {
        L[i][k] = A[i][k] / U[k][k];
        U[k][i] = A[k][i];
    }
    pthread_barrier_wait(barrier);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] -= L[i][k] * U[k][j];
        }
    }
    pthread_exit(nullptr);
}

int LUD(double** A, double** L, double** U, int* P, int n, int t) {
    for (int k = 0; k < n; k++) {
        int max_row = k;
        double max_val = 0;
        for (int i = k + 1; i < n; i++) {
            if (std::abs(A[i][k]) > max_val) {
                max_val = std::abs(A[i][k]);
                max_row = i;
            }
        }
        if (max_val == 0 && k != n - 1) {
            std::cout << "Singular Matrix" << std::endl;
            return -1;
        } else if (max_row != k) {
            std::swap(P[k], P[max_row]);
            for (int i = 0; i < n; i++) {
                std::swap(A[k][i], A[max_row][i]);
            }
            for (int i = 0; i < k; i++) {
                std::swap(L[k][i], L[max_row][i]);
            }
        }
        U[k][k] = A[k][k];

        pthread_t threads[t];
        LUThreadArgs threadArgs[t];
        for (int i = 0; i < t; i++) {
            threadArgs[i].A = A;
            threadArgs[i].L = L;
            threadArgs[i].U = U;
            threadArgs[i].P = P;
            threadArgs[i].n = n;
            threadArgs[i].t = t;
            threadArgs[i].k = k;
            pthread_create(&threads[i], nullptr, LUThread, &threadArgs[i]);
        }
        for (int i = 0; i < t; i++) {
            pthread_join(threads[i], nullptr);
        }
    }
    return 0;
}
```