COL380 - Introduction to Parallel and Distributed Programming

Assignment 1
LU Decomposition using OpenMP and pthreads

Pradyumna Meena(2016CS10375)

Manav Rao(2016CS10523)

Parallel Code Structure / Design Choices

Common notions

- There were no loop carried dependencies in any of the loop. Hence data and operations can be divided easily between the threads.
- Pointers to the matrices were shared but there are no explicit writes to any of those hence avoiding any locks and critical sections.
- Many operations were not parallelizable hence need to be executed by one thread only to avoid any possibility of race condition.

For OpenMP

- First step is to find the maximum and swap rows. These operations are executed only by the master thread.

```
#pragma omp master
    {

        for(int i = col;i<n;i++){
            if(fabs(*(mat[i] + col))>fabs(maxima.value)){
                maxima.value = *(mat[i] + col);
                maxima.idx = i;
            }
        }

        idx = maxima.idx;
        if(maxima.value==0.0){
            cout << "Singular Matrix" << endl;
        }

        int temp1 = perm[col];
        perm[col] = perm[idx];
        perm[idx] = temp1;

        double* add = mat[col];
        mat[col] = mat[idx];
        mat[idx] = add;

        *(upper[col] + col) = *(mat[col] + col);
        maxima.value = 0.0;
    }
```

- Once the swapping is done, all threads are synchronized at this point. Next step is again parallelized using threads.

```
#pragma omp barrier
#pragma omp for schedule(static)
    for(int i = 0;i<col;i++){
        double temp2 = *(lower[col] + i);
        *(lower[col] + i) = *(lower[idx] + i);
        *(lower[idx] + i) = temp2;
    }

#pragma omp for schedule(static)
    for(int i = col+1;i<n;i++){
        *(lower[i] + col) = (*(mat[i] + col))/(*(upper[col] + col));
        *(upper[col] + i) = *(mat[col] + i);
    }
```

Doing these operations in a single for loop requires a barrier after because some

threads will be doing different amounts of work because of a conditional statement.

- The next for loop ends with an explicit barrier since it marks the end of the parallel region.

```
#pragma omp for schedule(static)
    for(int i = col+1;i<n;i++){
        for(int j = col+1;j<n;j++){
            *(mat[i] + j) = *(mat[i] + j) - (*(lower[i] + col))*(*(upper[col] + j));
        }
    }
```

For pthreads

- Structs were used for passing multiple arguments to each thread at the time of creation.

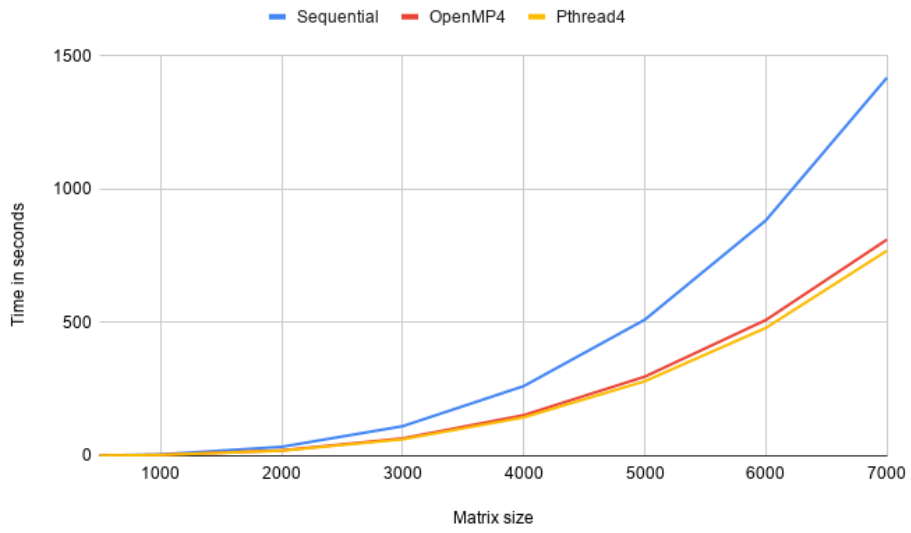
```
struct threadArgs1{
    int col;
    int idx;
    int end;
    int start;
    vector<double*> *array;
};

struct threadArgs2{
    int col;
    int idx;
    int end;
    int start;
    vector<double*> *lower;
    vector<double*> *upper;
    vector<double*> *mat;
};
```

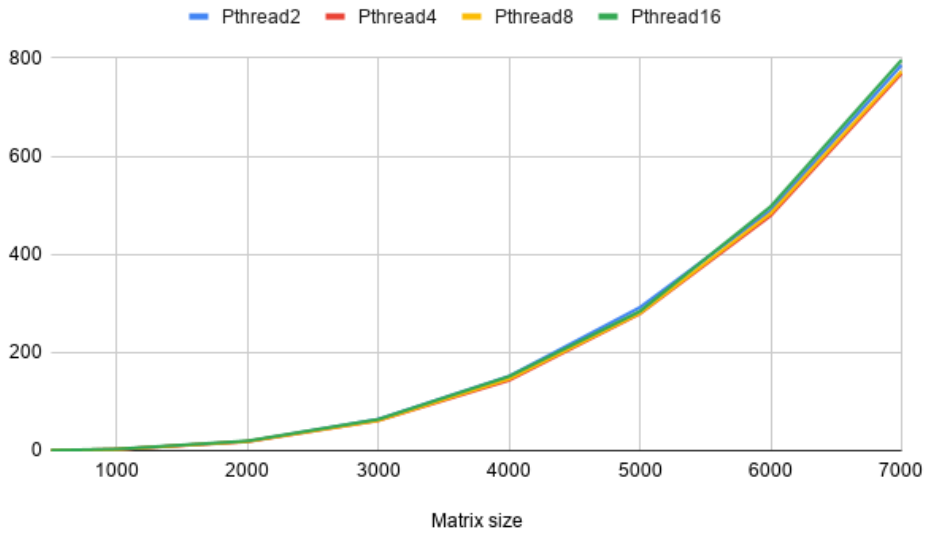
- Separate functions for each of the three loops in the algorithm.
    - *func1 - swapping*
    - *func2 - modifies lower and upper matrices*
    - *func3 - modifies the matrix itself*
- Set of iterations were equally divided between the threads except the last thread which in some cases might have fewer iterations.

Results

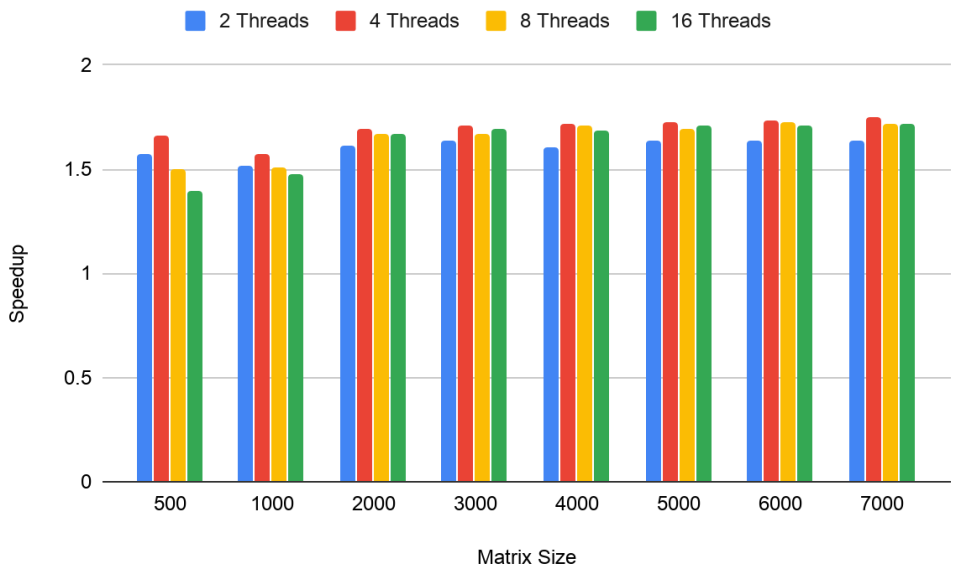
- Speed Comparison between Sequential code and OpenMP code with 4 threads

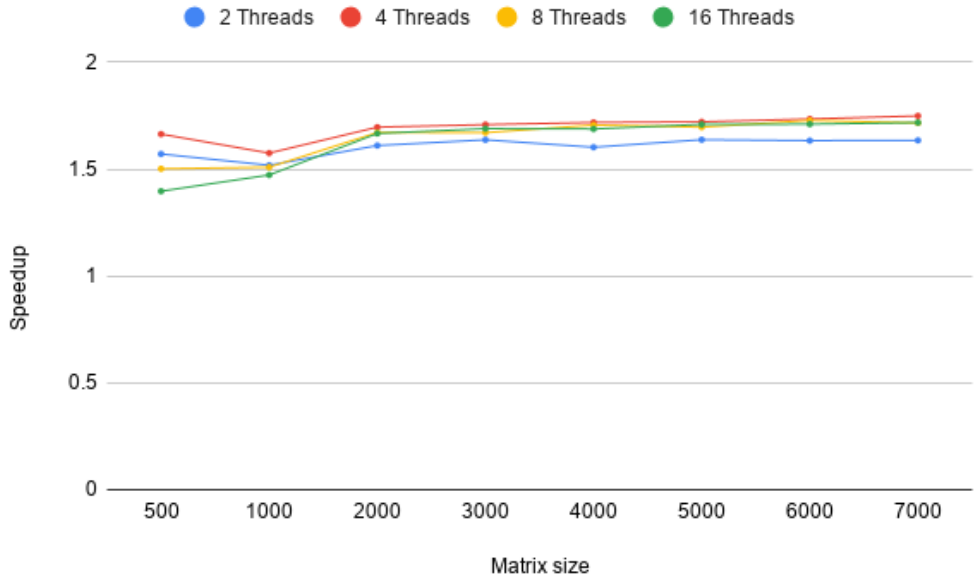


- Speed Comparison between Pthreads with varying thread count

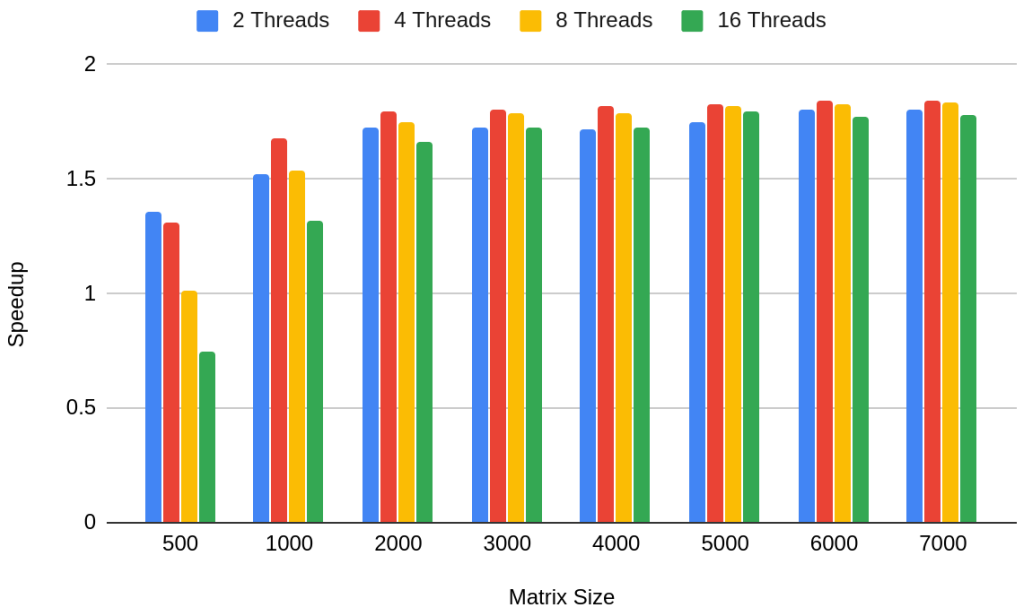


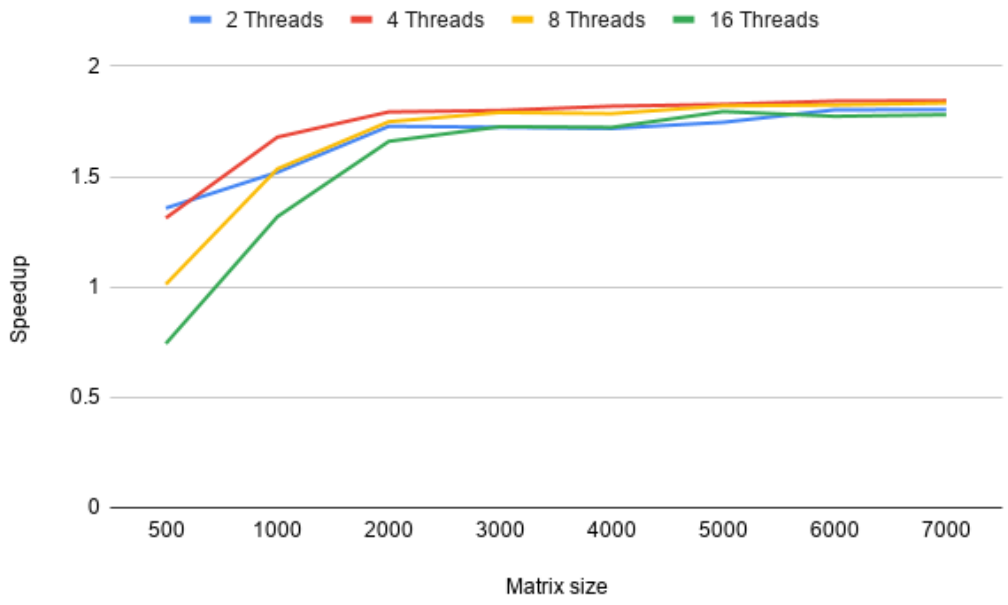
- Speedup with OpenMP





- Speedup with pthreads





Discussions

- *Machine specifications:*
    - Processor - 2.7 GHz Intel Core i5
    - Number of cores - 2

- *Speedup:*
  We can see the saturation in the speedup graphs which can be explained by both Amdahl's and Gustafson's law that we cannot keep on increasing speedup as we increase the number of processors. Speedup tends to a limit as the number of processors approaches infinity.

- *double** vs vector<vector<double>>:*
  Anu update at the i^{th} position of a vector is unaffected by what is happening at any other position in the vector. This is a very important observation with respect to this assignment since there is no dependency across a vector i.e. i^{th} value depends on $(i-1)^{th}$ or $(i+1)^{th}$ value. But if we use a vector then it would be a shared resource and hence any update at any position requires to be exclusive i.e. locks and critical sections. This is where pointers will turn out to be very fruitful as at any point during execution we are just reading the pointer value and not changing it.