COL380: Parallel and Distributed Programming: Assignment 1 Report

Karan Tanwar, Ritvik Vij

21-01-2020

This report comprises the working of LU Factorisation of Matrix using OpenMP and Pthreads.

1 OpenMP

1.1 Working Outline

The serial code of LU Factorisation was given to us. Using OpenMP, we parallelized the some of the code sections which eventually resulted in the increased speed-up of the Factorisation.
There were careful observations made for a better understanding of which sub-section of the code, to parallelize.

1.2 Implementation Layout and Design Choices

1. We used **pointer to 2-D arrays** as the data structure to define the matrices. This choice was better than dynamic allocation in vectors as it increases the time of execution and eventually decomposition of matrices.

2. we could use array of pointers, because this would reduce the complexity of swapping entire column of input matrix from O(n) to O(1). This is beneficial only for the input matrices not so 'big' in size

3. We observed, not every section(for-loop) should be parallelized because the speed-up was finally bottle necked by the thread fork and join **overhead**.

4. Every pointer was freed to **avoid memory corruption** or segmentation fault.

1.3 Parallelization Strategy

This section explains how the program partitions the data, work and exploits parallelism.

1. Firstly, it was observed that outer loop cannot be parallelised because the input matrix is being updated at every iteration.

2. Now for finding the max (i.e. var colmax), parallelization was not done because, firstly, the amount of data being divided was not evaluated before runtime, and secondly, the instructions were not that heavy to be data parallized.

3. **Implicit barrier were removed** till the last nested loop because there were no data dependencies.

4. Swaping of k entries of 2 rows of L matrix is also parallised by those available threads again, with removing the implicit barrier (because we dont have to wait for some threads to complete their task before other threads perform action on other data).

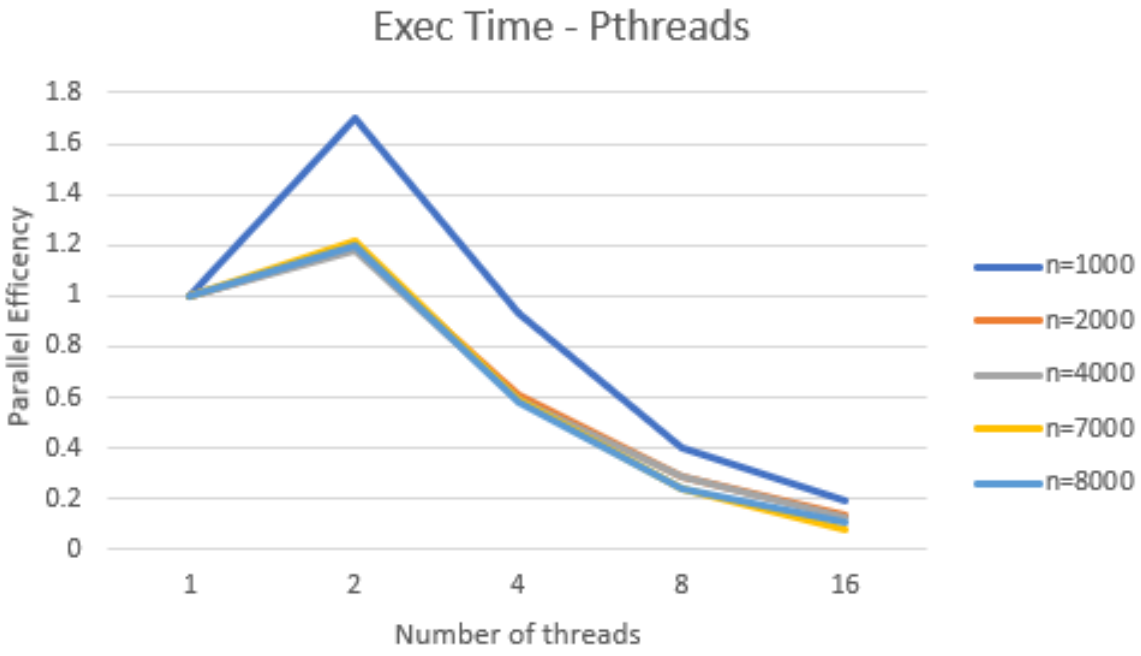
5. **U[k][k] is the same value as colmax** calculated in this certain coloumn. Thus we can use colmax, instead of U[k][k] in code shown below. This enables us to **fully exploit the power of nowait**. But the threads were synchronised before last nested loop.

```
u[k][k] = colmax;
#pragma omp for
for(long i=k;i<n;i++){
  // l[i][k] = A[i][k]/u[k][k];
  l[i][k] = (double)(A[i][k]/colmax);
  u[k][i] = A[k][i];
}
```

6. Here, in larger nested loop, #pragma omp for **collapse** is used to increase the total number of iterations that will be partitioned across the available number of OMP threads by **reducing the granularity of work** to be done by each thread.

1.4 Parallel Efficiency of Program Execution

This section includes graph of parallel execution v/s number of processors.

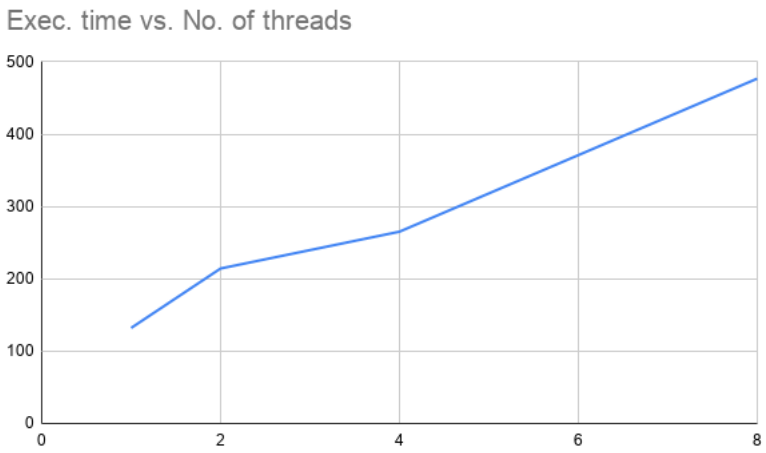


1.5 Factors affecting Parallelism

1. This was observed that the code is running at the order of $O(n^3)$ because of the last code section with 3 nested loops.

```
for i = k+1 to n
  for j = k+1 to n
    a(i,j) = a(i,j) - l(i,k)*u(k,j)
```

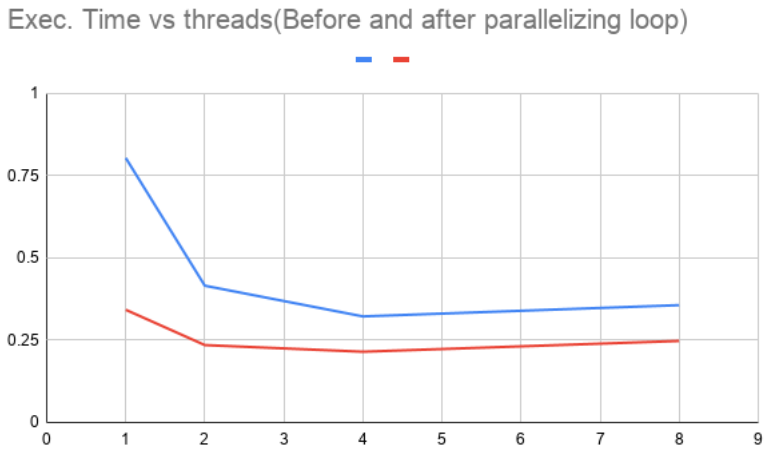
2. After removing the parallel pragma from this loop, we observed **exponential increase in execution time** comparing with the previous correct version of code.
Also, the time of execution is not decreasing by increasing threads. This can be seen in the graph plot for input matrix size = 8000.(For threads = 1, n is taken as 7000).



This concludes, for larger size of matrices, parallelizing last nested loop is the deciding factor for efficiency.

3. The loop mentioned below calculates the maximum element and its position in each column.

```
for k = 1 to n
  max = 0
  for i = k to n
    if max < |a(i,k)|
      max = |a(i,k)|
      k' = i
  if max == 0
    error (singular matrix)
```



The blue graph indicates the execution time after parallelizing this loop too. This clearly shows that it was not beneficial to parallelise the same.

2 Pthread

2.1 Working Outline

The serial code of LU Factorisation was given to us. Using pthreads, we parallelized the some of the code sections which eventually resulted in the increased speed-up of the Factorisation.
Pthreads is a short abreveation for Posix Threads which is an interface to create user level threads for parallelization.

2.2 Implementation Layout and Design Choices

1. A major change in the design choice was to keep the Input and Output Matrices as **global variables** instead of pass by reference. This choice was made due to pthread's design interface to have a void* function with a void* argument as the portion of the code being implemented by a single thread.

2. An array of a struct was created, one struct for each thread. An array was created so that the threads don't compete for the same address space.

3. Three different functions were made, each with portions of the decomposition code which weren't conflicting so could be parallelized together. **pthread_create() was called 3 different times for each of these function**.

2.3 Parallelization Strategy

This section explains how the program partitions the data, work and exploits parallelism.

1. Firstly, it was observed that outer loop cannot be parallelized because the input matrix is being updated at every iteration.

2. T threads were forked from main thread.

3. According to the number of threads mentioned by user, firstly, the first two for loop is parallelized in the function func multi1. The implicit barrier was removed reason being, that the next instructions being exececuted were neither loading nor modifying the input matrix, rather next set of instructions were operating on the output matrices

4. **Swaping of k entries of 2 rows** of L matrix is also parallised by those available threads again(the second loop in the function multi1).The range of computation for a thread with thread id num is (num)*(sizeofmatrix)/numberofthreads) to (num+1)*(sizeofmatrix)/(numberofthreads).

5. Now parallising the next loop requires the barrier because after this loop, L and U matrices values are loaded and we would want them to be updated before using them to modify the input matrix.

6. U[k][k] is the same value as colmax calculated in this certain coloumn. Thus we can use colmax, instead of U[k][k] in code shown below. A new func is called for the existing threads to implement the code snippet beyond this.

```
u[k][k] = colmax;
core = -1;
for(int i=0;i<NUM_OF_THREADS;i++){
   td[i].loop_var = k;
   td[i].swap_var = k_;
   core++;
   td[i].core = core;
   int rc=pthread_create(&threads[i], NULL, multi2, (void *)&td[i]);
```

7. This loop updates L[i][k] and U[k][i], which are different values being updated as the previous for loop was doing. This was the reason for **removing the implicit barrier**.

8. Here, in larger nested loop, a new function multi3 is being called by the same threads used in all of the above implementation. We have parallelized only one of the indices amongst the row counter and the column counter, parallelizing both of them would cause a race condition i.e. one thread accessing memory that has been updated already by another thread.

2.4 Parallel Efficiency of Program Execution

This section includes graph of parallel execution v/s number of processors.

3 Conclusion

1. This was observed that our implementation of **OpenMP was faster than Pthreads** for larger input matrices because OpenMP was performing many optimizations(such as collapse) that weren't handled in the case of pthreads.

2. Execution time for 2 and 4 number of threads were **nearly comparable** for the input size upto 8000.

3. False data sharing might be fatal for parallelized because, since there would be data dependencies or race conditions. Threads synchronisation was an important aspect of parallelizing the program which was taken care after using nowait in for loops.