COL 380: ASSIGNMENT 1

Parallel Implementation of LU Decomposition of Matrices

Vatsal Jingar 2020CS50449

Stitiprajna Sahoo 2020CS10394

Piya Bundela 2021CS10118

15 February 2024

Contents

1 Introduction

The assignment revolves around implementing LU decomposition using row-pivoting, which is a fundamental numerical technique used in linear algebra, with a focus on its parallel implementation. The idea behind the LU decomposition is to express a matrix A as the product of a lower triangular matrix L and an upper triangular matrix U. This decomposition facilitates solving systems of linear equations and performing matrix inversion efficiently.
The task involves developing two parallel implementations of LU decomposition using Gaussian elimination to factor a dense N x N matrix. In this assignment we are trying to explore how LU decomposition can be parallelized efficiently using shared-memory programming models like **OpenMP** and **Pthreads**.

1.1 What is LU decomposition

We are trying to express **A** as the product of **L** and **U**, like **A = LU**. We will achieve this by performing a series of row operations on A, transforming it into L and U through a process called Gaussian elimination. The algorithm is based on row pivoting, which helps us avoid situations where our calculations might produce really large or small numbers, which could throw off our results. By being strategic about our pivot selection, partial pivoting makes sure our LU decomposition is as precise and stable as possible. To compute A = LU, we compute a permutation matrix **P** such that **PA=LU**
In the following sections, we explain the pseudocode and the significant 3 approaches that we implemented.

1.2 Pseudocode

```
initialize pi as a vector of length n
initialize u as an n x n matrix with 0s below the diagonal
initialize l as an n x n matrix with 1s on the diagonal and 0s above the diagonal
for i = 1 to n
  pi[i] = i
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)
  swap pi[k] and pi[k']
  swap a(k,:) and a(k',:)
  swap l(k,1:k-1) and l(k',1:k-1)
  u(k,k) = a(k,k)
  for i = k+1 to n
    l(i,k) = a(i,k)/u(k,k)
    u(k,i) = a(k,i)
  for i = k+1 to n
    for j = k+1 to n
      a(i,j) = a(i,j) - l(i,k)*u(k,j)
```

The pseudocode outlines the LU decomposition algorithm with row pivoting, a numerical technique for factoring a dense $N \times N$ matrix A into lower and upper triangular matrices, L and U, respectively, along with a permutation matrix P. The algorithm begins by initializing the permutation vector π, which tracks the row exchanges, and the matrices L and U. Subsequently, it iterates through each column k of A to perform pivoting. Within each iteration, the row with the maximum absolute value in column k' is identified, and if no nonzero pivot element is found, the matrix is deemed singular. Otherwise, rows in A, π, and L are swapped according to the pivot. The diagonal element of U is set to the pivot element, and the k-th column of L and the k-th row of U are computed to update the matrices. Finally, the trailing submatrix of A is adjusted to eliminate elements below the diagonal. The resulting permutation vector π represents the permutation matrix P, while L and U represent the lower and upper triangular matrices, respectively. This process ensures numerical stability by minimizing round-off errors during the decomposition process.

2 Sequential Code

2.1 Workflow of the Code:

The sequential implementation of LU decomposition in `C++`, consists of the following sections:

1. `inputMatrix(N)`: It generates the input matrix A of size $N \times N$ representing the matrix to be decomposed.
2. `initOutputs()`: It initializes the output matrices L and U, along with the permutation vector π. Each element of L and U is set to the corresponding element in the input matrix, while π is initialized with the identity permutation, ensuring a consistent starting point for the LU decomposition process. It also creates a $temp\_A$ matrix which is a copy of A, for computation purposes in LU decompose.
3. `LUdecompose()`: This is the main decomposition function:
   - It iterates over each column k of the matrix A using a single loop that ranges from 0 to $N-1$, where N is the size of the matrix.
   - Within each iteration, it searches for the row with the maximum absolute value in column k to identify the pivot element. This is achieved by looping over the rows of the matrix $temp\_A$, starting from the current column index k.
   - After identifying the pivot element with index k', the function swaps the rows of matrices $temp\_A$ and L, and the permutation vector π accordingly. This is done to ensure that the pivot element becomes the diagonal element of the upper triangular matrix U.
   - It updates the lower triangular matrix L and the upper triangular matrix U based on k'. This involves using nested loops to iterate over the rows and columns of the matrices. For each element to be updated, the corresponding element of $temp\_A$ is divided by the pivot element ($temp\_A[k'][k]$), and the result is stored in L or U.
   - After completing the LU decomposition, the function computes the residual matrix $temp\_A$ by eliminating elements below the diagonal. This involves subtracting the product of elements of L and U from the corresponding elements of $temp\_A$. The operation is performed using nested loops that iterate over the rows and columns of $temp\_A$, excluding the diagonal and elements above it. Loop indices are managed to exclude elements below the diagonal, ensuring that only the residual elements are updated.
4. `verifyLU()`: This function verifies the LU decomposition by computing the matrix product PA and comparing it with the LU decomposition result. It computes the residual matrix, calculates the $L_{2,1}$ norm of the residual and prints it to console.

The `main()` function orchestrates the entire process by calling the input function, initialization function, LU decomposition function, and optionally, the verification function.

2.2 Data Layout Choice:

- We store L, U, P, $temp\_A$ as 2-D arrays and π as a 1-D array. The choice to store them as arrays instead of vectors is the contiguous allocation of array where as vector may not be allocated contiguously due to dynamic allocation. Contiguous allocation will improve percentage of cache hits (locality) which will reduce overall latency. This was the basic memory optimizations that were done in sequential codes itself. This reduced the latency of sequential code overall.

2.3 Bottlenecks of the code:

- **Nested Loops**: The nested loops for iterating over rows and columns contribute to the sequential nature of the algorithm, leading to cubicaly increasing execution time with matrix size.
- **Matrix Updates**: Updates to the temporary matrix and computation of elements of L and U are inherently sequential due to dependencies on previously computed values.
- **Data Access Patterns**: Sequential access to matrix elements may result in cache inefficiencies and slower memory access times.
- **Single-threaded Execution**: The code runs on a single thread, limiting its ability to utilize multi-core processors efficiently.

3 Parallel Implementation:

3.1 Non Parallelisable Sections of the Code:

- **Column Processing** The core operation of LU decomposition involves processing each column of the matrix sequentially, which is the outermost for-loop of LU decompose function. The decomposition of one column depends on the results of previous columns. Therefore, processing columns in parallel is not feasible without introducing quite complex synchronization or data dependencies, which would compromise the correctness of the decomposition.
- **Dependency Resolution** Operations such as finding pivot elements have loop-carried dependencies among columns which necessitate a sequential execution order to ensure that the decomposition is performed accurately. We tried performing this step in parallel by introducing critical sections, but it added more overhead.

3.2 Parallelization Opportunities:

We believe that parallelization of the following computations would lead to better performance by harnessing the inherent parallelism in the LU decomposition algorithm, without harming the algorithm integrity. -

- **Swapping rows of** $temp\_A$ **and L**: This can be assigned to different threads to swap some contiguous columns of each row.
- **Computing Elements of** L **and** U: Assigning different columns to different threads to compute elements concurrently can speed up the computation process, as each iteration of the loop is independent of the others.
- **Updating** $temp\_A$: This operation contributes to a major fraction of the execution time of the program. Assigning different rows or columns of $temp\_A$ to different threads for concurrent updates can reduce overall latency. We observed that the greatest percentage of LU decomposition code was done in executing the nested loop. And the iterations of this loop was independent from each other, so they were highly parallelisable. So we put our main focus on efficiently parallelising this loop.

4 Pthreads Code:

4.1 Basic Implementation:

- An array of structs is assigned to store metadata for the pthreads like `thread-id`, `k` and `k'`.
- At every iteration of the outermost loop, the threads are created and function `parallel-compute` is assigned to each thread.
- The thread gets its boundaries of the loop by the `getbounds` function. This function distributes equal and contiguous chunks of the loop-range in sequential order to the threads. The boundary calculation code (as below) ensures load balancing by distribution equal iterations among threads:

```
int q = num_iter/num_threads;
int c = int(ceil(num_iter/(1.0*num_threads)));
int r = num_iter%num_threads;
int base = r*c;
if(id<r) return {id*c, (id+1)*c};
else return {base+(id-r)*q, base+(id+1-r)*q};
```

- Each threads does the L, U, and *temp_A* update for the corresponding bounds:

```
for(int ind=left; ind<right; ind++){
    L[ind][k] = (temp_A[ind][k])/U[k][k];
    U[k][ind] = temp_A[k][ind];
}
pthread_barrier_wait(&barrier_1);

for(int i=left; i<right; i++){
    for(int j=k+1; j<N; j++){
        temp_A[i][j] -= L[i][k]*U[k][j];
    }
}
```

  For the *temp_A* update nested loop, we only distribute the outer loops among threads. Inner loop is completely performed by a thread, because it involves updating different entries of the same row which if distributed among threads will cause frequent cases of false sharing.

- Note that the updated value of $temp_A$ should be calculated with the updated values of L and U respectively. This gives us a **synchronisation point**, where the computation of L and U should occur *before* updates to *temp_A*. To do so, we created a **barrier** using `pthread-barrier-wait`.
- Finally the main thread waits to join all the threads before proceeding to the next iteration of outermost loop.

4.2 Optimisations:

1. We tried to implement swapping of elements of the k^{th} and pivot row of L by distributing elements among pthreads as:

```
pair<int,int> bounds_2 = getBounds(id, PTHREAD_COUNT, k);
int l = bounds_2.first;
int r = bounds_2.second;
for(int j=l; j<r; j++){
    swap(L[k][j], L[temp_k][j]);
}
```

   But this added to the time of execution of the code. This increases latency can account to the increases cases of false sharing, as we are updating elements of the same row of L, which are contiguously laid. Assuming `cache-line-size = 8 doubles`, when k is not a multiple of 8, the cache lines for different threads will overlap. And as swap is a write operation, it will cause multiple cache invalidations due to false sharing and create overhead. So we made swap sequential for the best implementation.

2. In the update of *temp_A*, each thread updates `temp_A[i][j] -= L[i][k]*U[k][j]` in the loop where `i = left_bound to right_bound`. Here, each thread accesses the k^{th} element of different rows, which will cause cache miss every time it accesses `L[i][k]`, *as different rows will not come on the same cache line, as N is huge.*
   To reduce cache misses, we created arrays `l,u` of size `N` which would store the k^{th} column of L and k^{th} row of U respectively (this update will take place as we update the corresponding rows and columns in the upper for loop, so no cache miss there).

```
for(int ind=left; ind<right; ind++){
    L[ind][k] = (temp_A[ind][k])/U[k][k];
    U[k][ind] = temp_A[k][ind];
    l[ind] = L[ind][k];
    u[ind] = U[k][ind];
}
pthread_barrier_wait(&barrier_1);

for(int i=left; i<right; i++){
    for(int j=k+1; j<N; j++){
        temp_A[i][j] -= l[i]*u[j];
    }
}
```

   Now, `l, u` will ensure that the threads get contiguous elements to update, and cache miss will reduce to a huge extent. This optimisation step reduced the time of our pthread execution by reducing cache misses. This has been observed in perf reports:

```
Performance counter stats for './pth_impl':

       77,550,459      cache-references
       13,272,607      cache-misses              #   17.115 % of all cache refs

      2.748195993 seconds time elapsed

      5.772736000 seconds user
      0.981581000 seconds sys
```

Figure 1: Perf report of pthreads code without l,u arrays

```
Performance counter stats for './pth_impl':

       75,377,041      cache-references
        8,289,750      cache-misses              #   10.998 % of all cache refs

      2.414325772 seconds time elapsed

      5.538847000 seconds user
      0.860043000 seconds sys
```

Figure 2: Perf report of pthreads code with l,u arrays

3. **Loop unrolling:** We reduce the number of iterations for each thread by loop unrolling, by having more instructions in a single iteration:

```
for(int i=left; i<right; i++){
    for(int j=k+1; j<N; j+=8){
        temp_A[i][j] -= l[i]*u[j];
        if(j+1 < N) temp_A[i][j+1] -= l[i]*u[j+1];
        if(j+2 < N) temp_A[i][j+2] -= l[i]*u[j+2];
        if(j+3 < N) temp_A[i][j+3] -= l[i]*u[j+3];
        if(j+4 < N) temp_A[i][j+4] -= l[i]*u[j+4];
        if(j+5 < N) temp_A[i][j+5] -= l[i]*u[j+5];
        if(j+6 < N) temp_A[i][j+6] -= l[i]*u[j+6];
        if(j+7 < N) temp_A[i][j+7] -= l[i]*u[j+7];
    }
}
```

This would reduce the number of loop control instructions, and also increase cache utilisation. When `u[j]` will be fetched from memory to cache, `u[j], u[j+1], ... u[j+7]` will be brought to the cache on the same cache line. Similarly for *temp_A*, and so the whole loop iteration can be performed without any cache miss or any loop control execution in between. This reduces latency by a few milliseconds. Reduction in cache misses due to loop unrolling in pthreads is observed in perf reports:

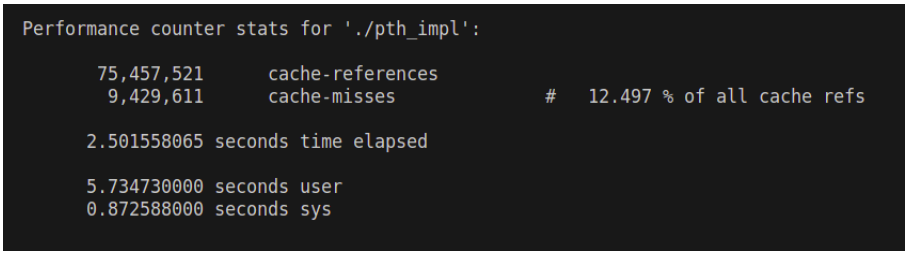

Figure 3: Perf report of pthreads code without loop unrolling

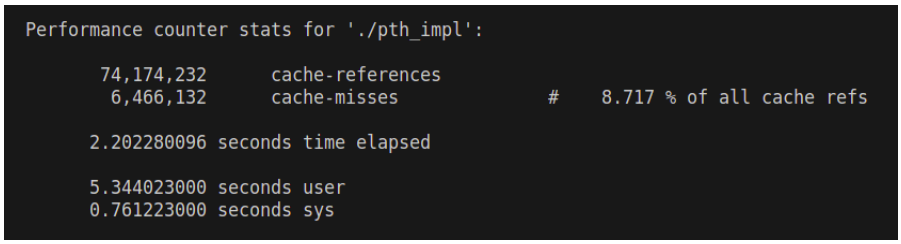

Figure 4: Perf report of pthreads code with loop unrolling, cache miss rate reduced from 12 to 8 %

4. As the heaviest step of our computation was the nested loop that updated $temp_A$, we made $temp_A$ as a array of pointers to 1-D arrays (each element was the pointer to its corresponding row). This would make the rows non-contiguously aligned in the memory. This step would reduce the instance of false sharing, as threads were assigned different rows, and different rows will always be on different cache lines as they are not contiguously aligned (due to pointers). This step also reduced the latency by a few milliseconds.

5. Barriers have some overhead associated with them in form of waiting and waking up of threads. So, we tried to remove barrier from pthreads by slightly changing the code-workflow so that synchronization is not required:

```
for(int i=left; i<right; i++){
    L[i][k] = temp_A[i][k]/U[k][k];
    U[k][i] = temp_A[k][i];
    double x = (temp_A[i][k]/U[k][k]);
    for(int j=k+1; j<N; j++){
        temp_A[i][j] -= (x*temp_A[k][j]);
    }
}
```

This code doesn't require $temp_A$ to be updated with L and U, as L, U are itself functions of previous columns and rows of $temp_A$. This removes the requiremenmt of synchronisation point/barrier. This step reduced the time of pthread execution by $100ms$

4.3 Performance:

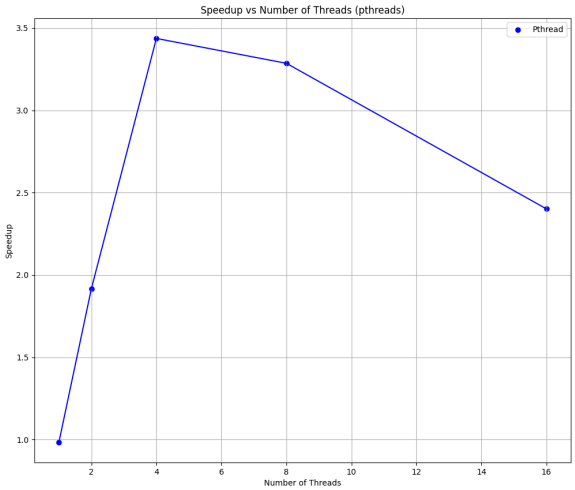

Figure 5: Speedup of pthread code for N = 1000 vs Number of threads

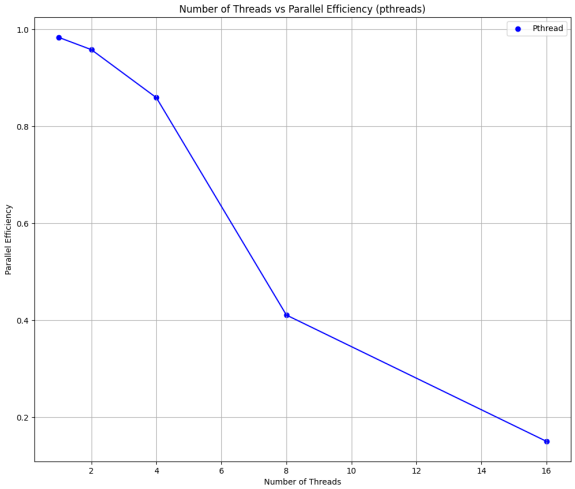

Figure 6: Efficiency of pthread code for N = 1000 vs Number of threads

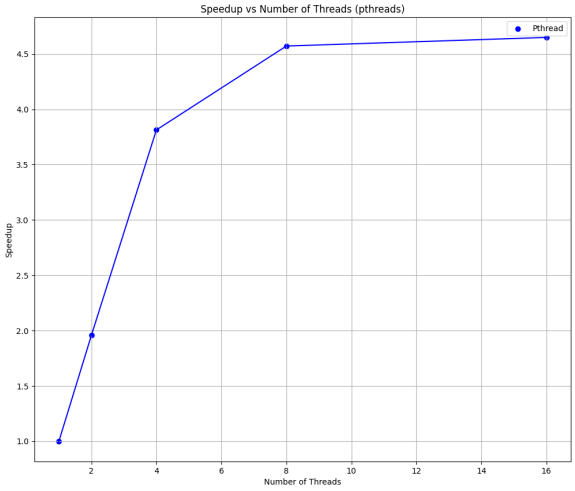

Figure 7: Speedup of pthread code for N = 7000 vs Number of threads

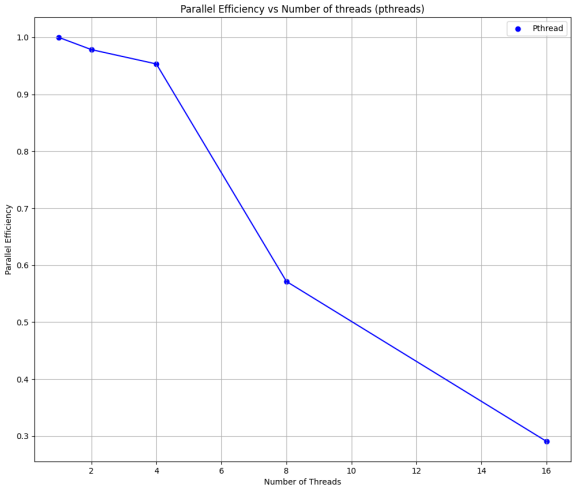

Figure 8: Efficiency of pthread code for N = 7000 vs Number of threads

5 OpenMP Code:

5.1 Basic Implementation:

- We used `pragma omp for` to parallelise both the for-loops:

```
#pragma omp parallel for num_threads(PTHREAD_COUNT) schedule(static)
for (int i = k + 1; i < N; i++){
    L[i][k] = temp_A[i][k] / U[k][k];
    U[k][i] = temp_A[k][i];
}

#pragma omp parallel for num_threads(PTHREAD_COUNT) schedule(static)
for (int i = k + 1; i < N; i++){
    for (int j = k + 1; j < N; j+=8){
        temp_A[i][j] -= L[i][k]*U[k][j];
    }
}
```

- For the *temp_A* update nested loop, we only distribute the outer loops among threads. Inner loop is completely performed by a thread, because it involves updating different entries of the same row which if distributed among threads will cause frequent cases of false sharing.
- There is a **synchronisation point** between the 2 for-loops, so we omit `no-wait` clause.

5.2 Optimisations:

1. We tried to implement swapping of elements of the k^{th} and pivot row of L by making the for loop parallel:

```
#pragma omp parallel for num_threads(PTHREAD_COUNT) schedule(static)
for (int i = 0; i < k; i++){
    swap(L[k][i], L[temp_k][i]);
}
```

   But this added to the time of execution of the code. This increases latency can account to the increases cases of false sharing, as we are updating elements of the same row of L, which are contiguously laid. Similar to pthreads, here also it was kep sequential.

2. In the update of *temp_A*, each thread updates `temp_A[i][j] -= L[i][k]*U[k][j]` in the loop where `i = left_bound to right_bound`. Here, each OMP thread accesses the k^{th} element of different rows (as we parallelise the outer loop only), which will cause cache miss every time it accesses `L[i][k]`, *as different rows will not come on the same cache line, as N is huge.*
   To reduce cache misses, we created arrays `l,u` of size `N` which would store the k^{th} column of L and k^{th} row of U respectively (this update will take place as we update the corresponding rows and columns in the upper for loop, so no cache miss there).

```
#pragma omp parallel for num_threads(PTHREAD_COUNT) schedule(static)
for (int i = k + 1; i < N; i++){
    L[i][k] = temp_A[i][k] / U[k][k];
    U[k][i] = temp_A[k][i];
    l[i] = L[i][k];
    u[i] = U[k][i];
}
#pragma omp parallel for num_threads(PTHREAD_COUNT) schedule(static)
for (int i = k + 1; i < N; i++){
    for (int j = k + 1; j < N; j++){
        temp_A[i][j] -= l[i]*u[j];
    }
}
```

Now, `l, u` will ensure that the OMP threads get contiguous elements to update, and cache miss will reduce to a huge extent. This optimisation step reduced the time of our openMP execution by reducing cache misses. This has been observed in perf reports:

```
Performance counter stats for './omp_impl':

       35,549,525      cache-references
        6,246,445      cache-misses             #   17.571 % of all cache refs

      5.442389676 seconds time elapsed

     29.578534000 seconds user
      0.107642000 seconds sys
```

Figure 9: Perf report of openMP code without l,u arrays

```
Performance counter stats for './omp_impl':

       32,843,436      cache-references
        4,496,298      cache-misses             #   13.690 % of all cache refs

      3.157068522 seconds time elapsed

     17.519511000 seconds user
      0.055142000 seconds sys
```

Figure 10: Perf report of openMP code with l,u arrays, cache miss rate reduced from 17.5 to 13.7 %

3. **Loop unrolling:** We reduce the number of iterations assigned to each thread by loop unrolling, by having more instructions in a single iteration:

```
#pragma omp parallel for num_threads(PTHREAD_COUNT) schedule(static)
for (int i = k + 1; i < N; i++)
    for(int j=k+1; j<N; j+=8){
        temp_A[i][j] -= l[i]*u[j];
        if(j+1 < N) temp_A[i][j+1] -= l[i]*u[j+1];
```

```
            if(j+2 < N) temp_A[i][j+2] -= l[i]*u[j+2];
            if(j+3 < N) temp_A[i][j+3] -= l[i]*u[j+3];
            if(j+4 < N) temp_A[i][j+4] -= l[i]*u[j+4];
            if(j+5 < N) temp_A[i][j+5] -= l[i]*u[j+5];
            if(j+6 < N) temp_A[i][j+6] -= l[i]*u[j+6];
            if(j+7 < N) temp_A[i][j+7] -= l[i]*u[j+7];
        }
    }
```

This would reduce the number of loop control instructions, and also increase cache utilisation. When `u[j]` will be fetched from memory to cache, `u[j], u[j+1], ... u[j+7]` will be brought to the cache on the same cache line. Similarly for $temp\_A$, and so the whole loop iteration can be performed without any cache miss or any loop control execution in between. This reduces latency of openMP code by a few milliseconds. The perf reports show reduced cache misses due to loop unrolling:

```
Performance counter stats for './omp_impl':

       56,685,261      cache-references
        6,791,216      cache-misses              #   11.981 % of all cache refs

      4.494274791 seconds time elapsed

     24.536416000 seconds user
      0.068879000 seconds sys
```

Figure 11: Perf report of openMP code without loop unrolling

```
Performance counter stats for './omp_impl':

       52,628,473      cache-references
        1,868,158      cache-misses              #    3.550 % of all cache refs

      2.027425869 seconds time elapsed

     10.727365000 seconds user
      0.071621000 seconds sys
```

Figure 12: Perf report of openMP code with loop unrolling, cache miss rate reduced from 12 to 3.5 %

4. As the heaviest step of our computation was the nested loop that updated $temp_A$, we made $temp_A$ as a array of pointers to 1-D arrays (each element was the pointer to its corresponding row). This would make the rows non-contiguously aligned in the memory. This step would reduce the instance of false sharing, as we only parallelised the outer for-loop and thus openmp threads would always get different rows, and different rows will always be on different cache lines as they are not contiguously aligned (due to pointers). This step also reduced the latency by a few milliseconds.

5. When the total number of iterations is too-small, we observed that executing them with sequential code was much faster than using openmp, as parallelisation involved some overhead of threads. and the total number of iterations varied with k. So we put a conditional `if-clause` to parallelise the for loop only if the number of iterations were above a certain threshold:

```
#pragma omp parallel for num_threads(PTHREAD_COUNT) schedule(static)\
                                                    if (N - k - 1 > 100)
for (int i = k + 1; i < N; i++){
    L[i][k] = temp_A[i][k] / U[k][k];
    U[k][i] = temp_A[k][i];
    l[i] = L[i][k];
    u[i] = U[k][i];
}
#pragma omp parallel for num_threads(PTHREAD_COUNT) schedule(static)\
                                                    if (N - k - 1 > 100)
for (int i = k + 1; i < N; i++){
    for (int j = k + 1; j < N; j++){
        temp_A[i][j] -= l[i]*u[j];
    }
}
```

6. **Guided scheduling in OpenMP**

   In the OpenMP LU decomposition implementation,we have used schedule(guided) which dynamically adjusts workload chunk sizes, improving load balancing by allocating larger chunks initially and gradually reducing them. Without a specified schedule, static scheduling is used by default, potentially leading to load imbalance and suboptimal performance. `schedule(guided)` thus yields better performance due to improved load balancing.

5.3 Performance:

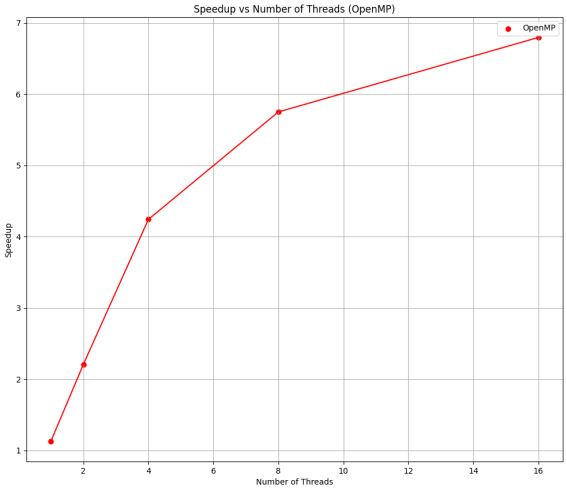

Figure 13: Speedup of OpenMP code for N = 1000 vs Number of threads

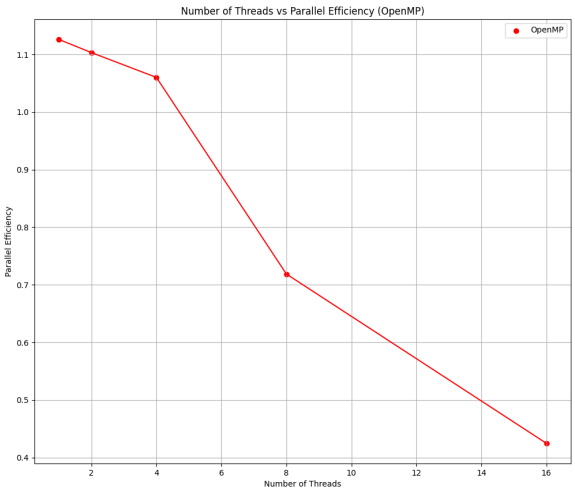

Figure 14: Speedup of openMP code for N = 1000 vs Number of threads

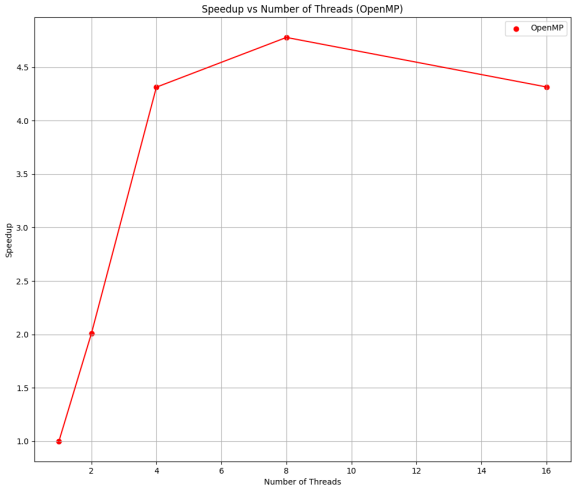

Figure 15: Speedup of OpenMP code for N = 7000 vs Number of threads

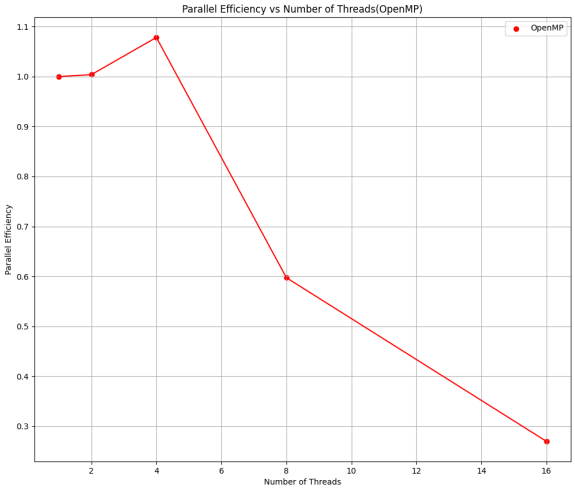

Figure 16: Speedup of openMP code for N = 7000 vs Number of threads

6 Comparision of Performance:

6.1 Timing Measurements:

The tables shows the timing in milliseconds (ms) for the best pthread and openmp executions of the LUdecompose function using 1,2,4,8 and 16 threads for problem sizes $N = 1000$ and $N = 7000$.

Threads	Pthreads	OpenMP
1	608	598
2	312	271
4	174	141
8	182	104
16	249	88

Table 1: Timing in ms for $N = 1000$, where sequential execution time was 598 ms

Threads	Pthreads	OpenMP
1	1185170	1185170
2	595532	590308
4	310767	274770
8	259231	248019
16	254905	274638

Table 2: Timing in ms for $N = 7000$, where sequential execution time was 1185170 ms

6.2 Speedup and Efficiency graphs:

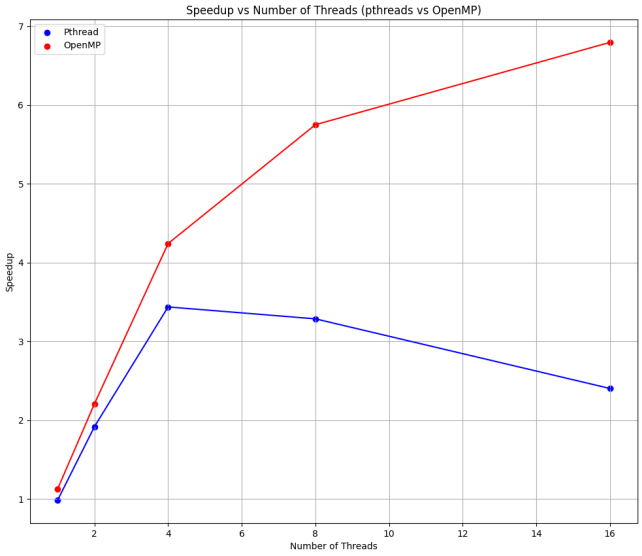

Figure 17: Speedup of OpenMP and Pthreads on N = 1000 vs number of threads

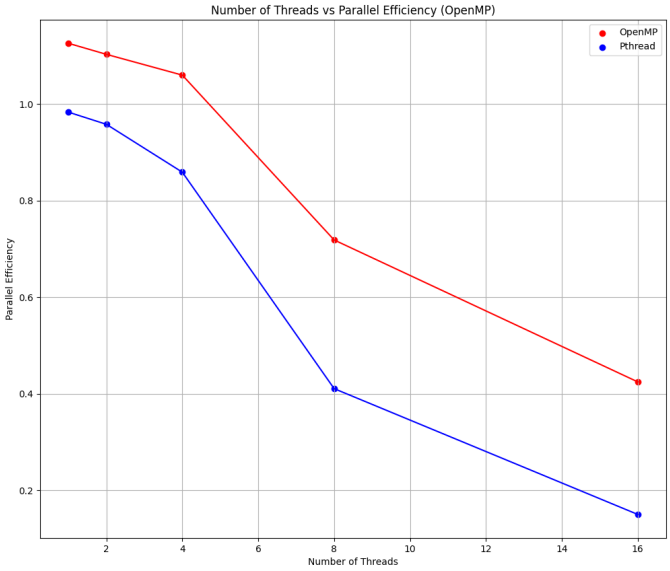

Figure 18: Efficiency of OpenMP and Pthreads on N = 1000 vs number of threads

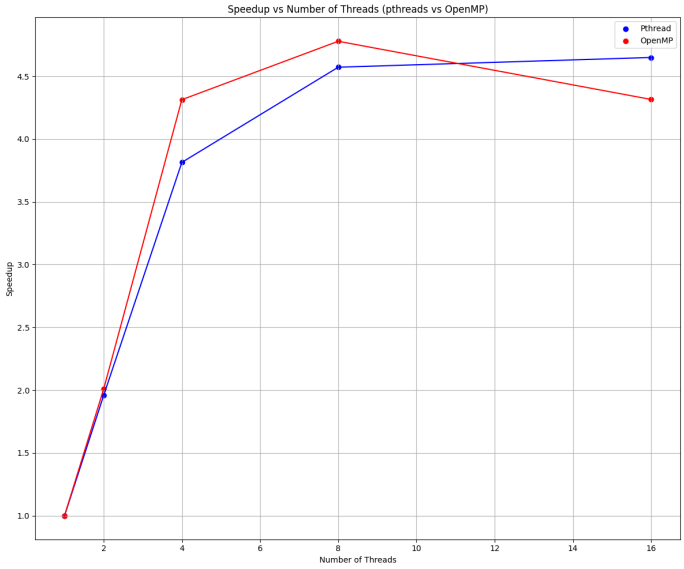

Figure 19: Speedup of OpenMP and Pthreads on N = 7000 vs number of threads

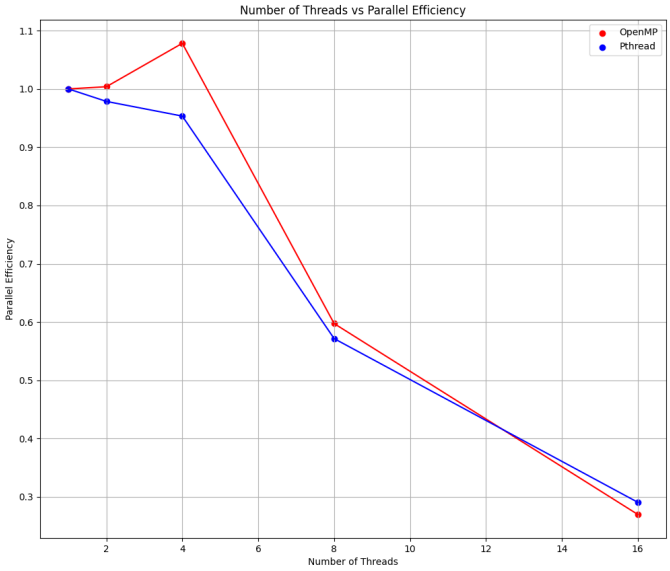

Figure 20: Efficiency of OpenMP and Pthreads on N = 7000 vs number of threads

6.3 Karp-flatt Metric to evaluate parallelisation:

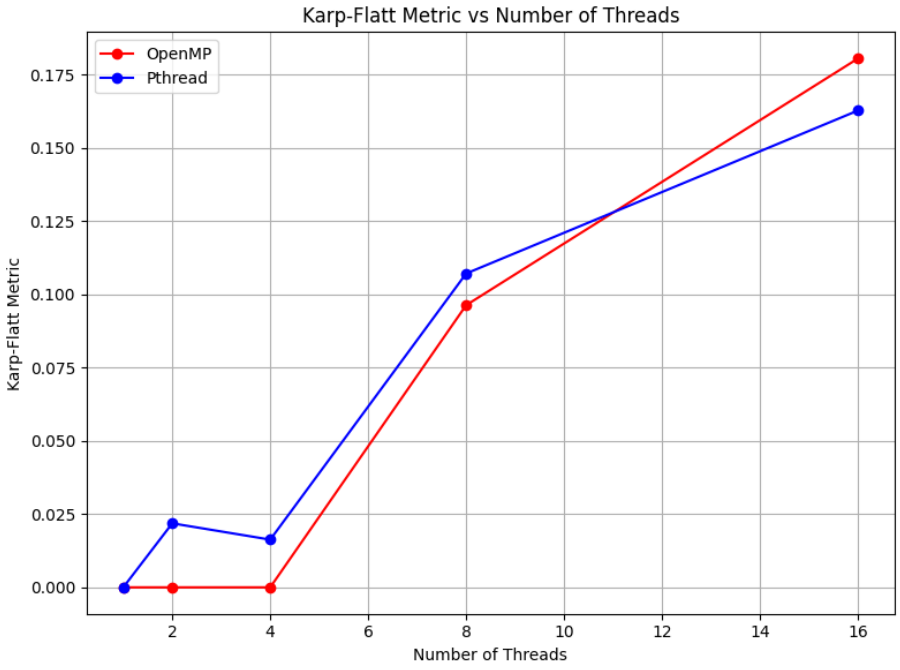

Figure 21: Karp-flatt of OpenMP and Pthreads on N = 7000 vs number of threads

Karp-flatt metric gives a measure of the serial fraction of the code. As we can see in the graph above, the karp-flatt metric is increasing in a linear manner for `num_threads = [1,2,4,8,16]`. As `num_threads` is increasing exponentially, so we can observe that karp-flatt metric of our code is increasing in a logarithmic manner (`O(log(num_threads))`) with the number of threads, which is bearable.