Assignment 1 COL380
LU decomposition

Team - bratva
Pratham, Mihir, Kushagra

February 15, 2024

§1 Overview of the Code

We have parallelized the code to perform LU decomposition of the given matrix utilizing `pthread` and `openMP` libraries in C++ to perform LU decomposition of a given matrix.

§1.1 Pthread

- `Global Variables`:

```
int thread_count;
int n;
double **A, **L, **U;
int *pi;
```

We have defined the code with global variables for thread count, matrix size, and pointers to matrices for input, lower triangular, upper triangular, and permutation matrices.

- `Struct for thread paramters`

```
struct ThreadArgs {
    int thread_index;
    int k;
};

struct ThreadArgs* thread_args =
(struct ThreadArgs*)malloc(thread_count*sizeof(struct ThreadArgs));
```

Since the function assigned to threads accepts only 1 void pointer argument, we provide it an argument of the type `(void*)(struct Threadargs*)`. This was done in order to allow each thread to use variables that are specific to it, so in the code we have done. The struct simply specifies to each thread its thread index, also called the rank of the thread, and the value of iteration of the outer loop, i.e, k, (denoting the iteration number of the outermost loop of the algorithm)

- `luDecomposition`

This function is responsible for LU decomposition and is run in the main thread itself. Before the first for loop, we allocate storage for the `pthread_t` as

```
pthread_t* thread_handles;
thread_handles = (pthread_t*)malloc(thread_count*
sizeof(pthread_t));
```

In **each iteration of the outermost for loop on k**, the main thread first finds the value of k' (as defined in the pseudo-code) and does the relevant swaps between k and k'. Next it finds the values of $L[i][k]$ and $U[i][k]$ accordingly. The part of the code that we parallelize is

```
for (int i = k+1; i < n; ++i) {
    for (int j = k+1; j < n; ++j) {
        A[i][j] = A[i][j] - (L[i][k] * U[k][j]);
    }
}
```

In order to do so, we start the threads with the **parallel** function after suitably updating their `ThreadArgs` values

```
for (int thread = 0; thread < thread_count; thread++){
    (*(thread_args+thread)).thread_index = thread;
    (*(thread_args+thread)).k = k;
    pthread_create(&thread_handles[thread], NULL,
    parallel, (void*)(thread_args+thread));
}
```

After all threads finish executing **parallel** function, we join them. In the end we free memory allocated to `thread_handles` and `thread_args`.

- `parallel`

As mentioned above, the $O(n^2)$ loop is parallelized by **dividing the rows of A almost equally among the m threads**. That is the range of i in the for loop

```
for (int i = k+1; i < n; ++i)
```

is divided in m intervals $[l_0, r_0], [l_1, r_1]$....., $[l_{m-1}, r_{m-1}]$ where $l_0 = k+1$, $r_{m-1} = n$ and $r_i - l_i = \lfloor \frac{n-k}{m} \rfloor$ for $i < m-1$ and $r_{m-1} - l_{m-1} = (n-k) - (m-1)\lfloor \frac{n-k}{m} \rfloor$, and thread number x executes the following code

```
for (int i = l_x; i <= r_x; ++i) {
    for (int j = k+1; j < n; ++j) {
        A[i][j] = A[i][j] - (L[i][k] * U[k][j]);
    }
}
```

Since the nested for loops did not carry any data dependency between (i_1, j_1) and (i_2, j_2) for $(i_1, j_1) \neq (i_2, j_2)$ (only A is written to in the loops and the update of $A[i_2][j_2]$ does not depend on $A[i_1][j_1]$), it is possible to parallelize in the way above **without causing any race condition** among the threads.

Also since matrices A,L,U were declared as a double pointer type, only row-entries would be in consecutive locations in the memory, and entries in different rows would be in different blocks. As the parallel threads are accessing different rows of the matrix A, the cache blocks modified/read at an instance by different threads would be mostly different and there is **no instance of false sharing**. In case of matrices L and U, we are only performing read operations, therefore no case of false sharing arises for these.

§1.2 OpenMP

Only the following code snippet is modified from the sequential code in the `luDecomposition function` which is responsible for LU decomposition and utilizes OpenMP directives for parallelization. The primary

parallelization is implemented within this function for the OpenMP case.

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

The workload is automatically divided among threads by OpenMP, allowing parallel computation of LU decomposition.

§2 Performance analysis

Speedup represents the factor of reduction in time taken to run the program in comparison to the sequential program, and is calculated using the formula given by:

$$\text{Speedup} = \frac{t_{series}}{t_{parallel}}$$

As can be seen from the tables below, we observe that as the number of threads increases, the speedup increases by a factor of thread_count until the number of threads reaches a certain value, which in our case is coming out to be 8. And on further increasing the number of threads, we do not see any more speedup because at a given time, one core can only run one thread.

Even after having 8 cores in the processor, we get a maximum speedup of only around 3.7 because of core allocation by the OS. Our operating system allocates only a small portion of the cores in the processor to our program, as many other processes are being simultaneously run in the background. Therefore, it is highly dependent on the OS as to how many cores we get for our parallel program and we cannot control it. The data below was obtained by executing our program on **MacBook Air M1 2020 which has 4 performance CPU cores and 4 efficiency CPU cores**

Using multiple threads should ideally reduce the time taken to run the program by the factor of min(number of threads, allocated cores) but this is not the case in reality, as there is also, a lot of overhead involved in creating and joining the threads. Similarly, if we consider the program with number of threads equal to 1, i.e, we create and join 1 thread which involves unnecessary overhead of thread creation, it will be slower than purely sequential code where we don't create any threads, rather the main thread performs the complete LU decomposition. Therefore, by 1 thread, we actually mean the sequential program, without using pthreads.

parallel efficiency of the program for a given number of threads, is given by the formula:

$$p_{efficiency} = \frac{speedup}{number\_threads}$$

§2.1 Speedup for parallel program using pthreads

Number of Threads	Run Time (seconds)	Speedup	Efficiency
1 (sequential)	8.29786	1.000	1.000
2	4.25067	1.952	0.976
4	2.34542	3.538	0.885
8	2.20316	3.759	0.470
16	2.20388	3.757	0.235

Table 1 pthread with input size $n = 2000$

Number of Threads	Run Time (seconds)	Speedup	Efficiency
1 (sequential)	63.1088	1.000	1.000
2	33.1897	1.900	0.950
4	18.9703	3.325	0.831
8	17.3016	3.646	0.456
16	17.475	3.614	0.226

Table 2 pthread with input size $n = 4000$

Number of Threads	Run Time (seconds)	Speedup	Efficiency
1 (sequential)	488.198	1.000	1.000
2	297.006	1.644	0.822
4	185.273	2.635	0.659
8	151.03	3.235	0.404
16	155.149	3.145	0.196

Table 3 pthreads with input size $n = 8000$

§2.2 Speedup for parallel program using OpenMP

Number of Threads	Run Time (seconds)	Speedup	Efficiency
1 (sequential)	8.29786	1.000	1.000
2	5.16293	1.605	0.803
4	2.88782	2.878	0.720
8	2.46932	3.363	0.420
16	2.4504	3.389	0.212

Table 4 openMP with input size $n = 2000$

Number of Threads	Run Time (seconds)	Speedup	Efficiency
1 (sequential)	63.1088	1.000	1.000
2	38.6492	1.631	0.815
4	22.5961	2.793	0.698
8	18.6556	3.382	0.423
16	18.8248	3.350	0.209

Table 5 openMP with input size $n = 4000$

Number of Threads	Run Time (seconds)	Speedup	Efficiency
1 (sequential)	488.198	1.000	1.000
2	300.872	1.622	0.811
4	183.64	2.657	0.664
8	149.528	3.264	0.408
16	155.087	3.146	0.197

Table 6 openMP with input size $n = 8000$

§3 pthreads vs OpenMP

Number of Threads	Run Time (seconds) with pthreads	Run Time (seconds) with OpenMP
2	4.25067	5.16293
4	2.34542	2.88782
8	2.20316	2.46932
16	2.20388	2.4504

Table 7 pthreads vs OpenMP for $n = 2000$

Number of Threads	Run Time (seconds) with pthreads	Run Time (seconds) with OpenMP
2	33.1897	38.6492
4	18.9703	22.5961
8	17.3016	18.6556
16	17.475	18.8248

Table 8 pthreads vs OpenMP for $n = 4000$

Number of Threads	Run Time (seconds) with pthreads	Run Time (seconds) with OpenMP
2	297.006	300.872
4	185.273	183.64
8	151.04	149.528
16	155.149	155.087

Table 9 pthreads vs OpenMP for $n = 8000$

§4 Plots

§4.1 Plots for the time taken vs. Number of threads

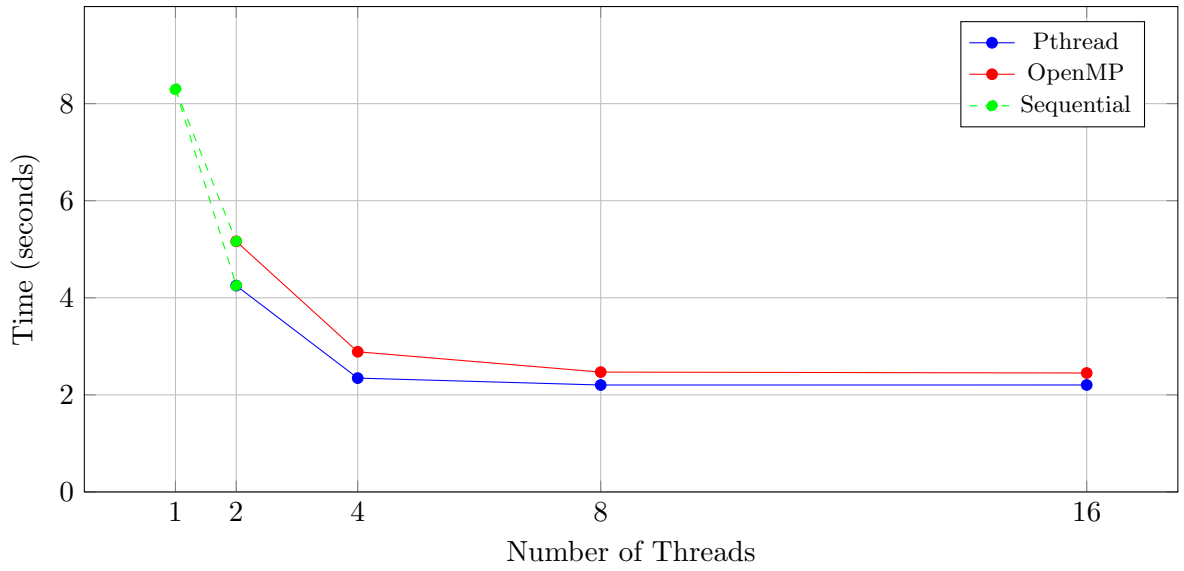

Figure 1: Comparison of Execution Time - Size 2000

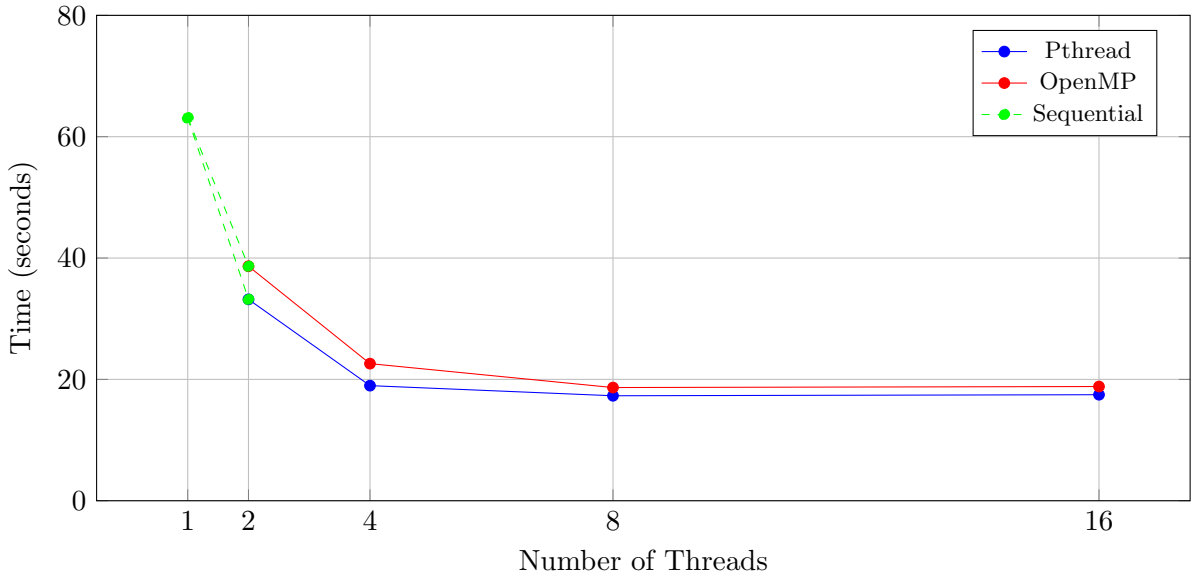

Figure 2: Comparison of Execution Time - Size 4000

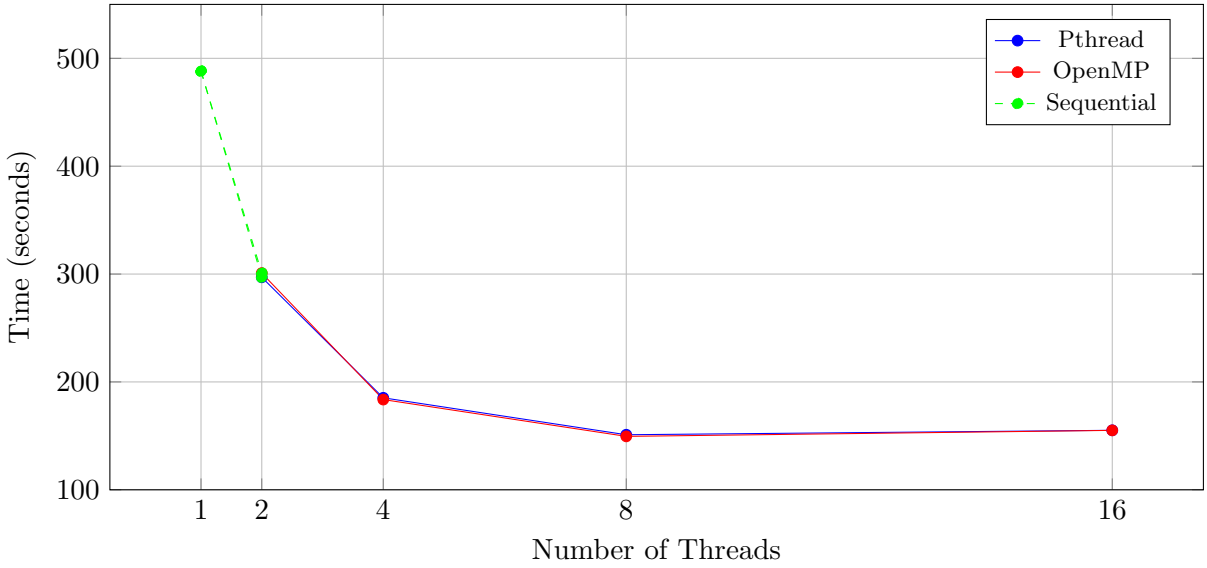

Figure 3: Comparison of Execution Time - Size 8000

We note that as the size of the input increases (from 2000 to 8000), the execution time generally increases for all implementations (pthread and OpenMP).

Both pthread and OpenMP show reductions in execution time as the number of threads increases, indicating the benefits of parallelism. However, the time starts to increase after 8 threads as our machine has 8 cores, and the parallelism benefits saturates, which is associated with more computational overhead.

We also observe the times taken by OpenMP and pthread to be similar - however, the time taken by pthread is slightly lower for every n, number_of_threads.

§4.2 Plots for the Speedup vs. Number of threads

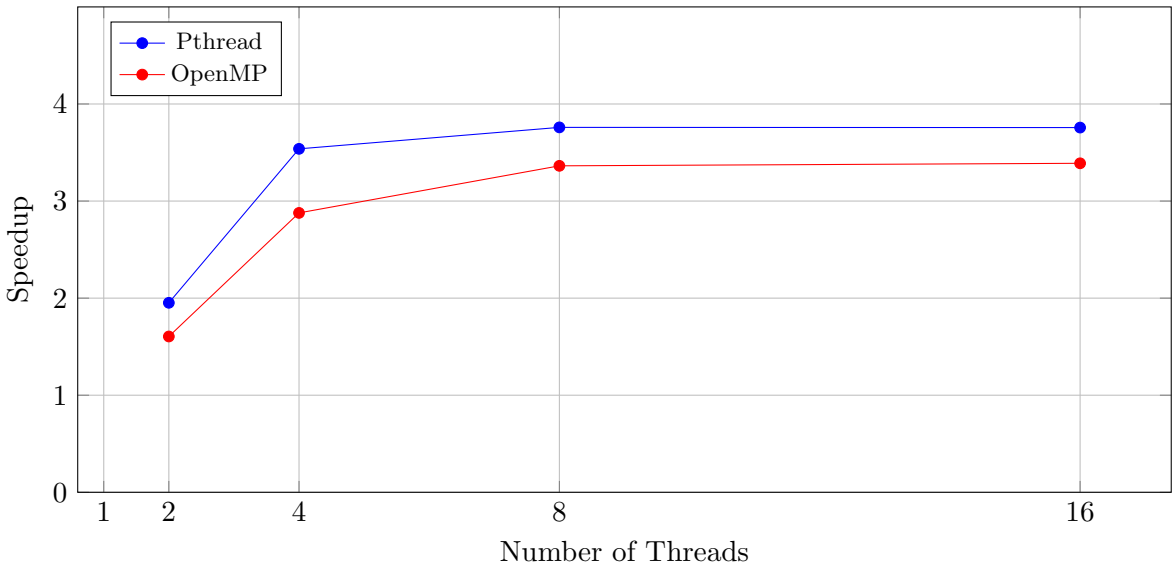

Figure 4: Speedup Comparison - Size 2000

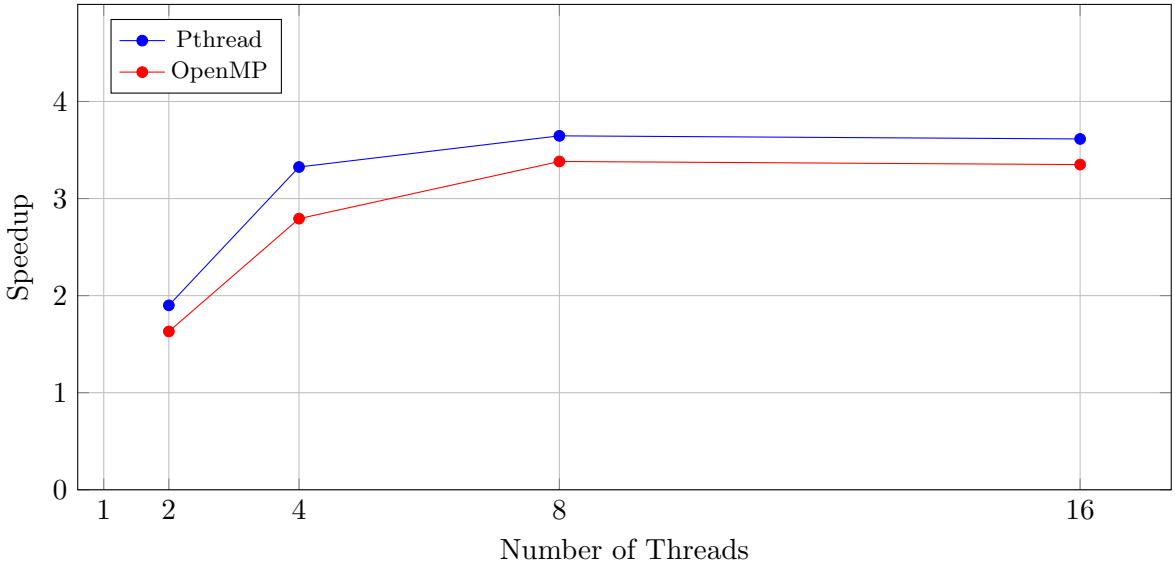

Figure 5: Speedup Comparison - Size 4000

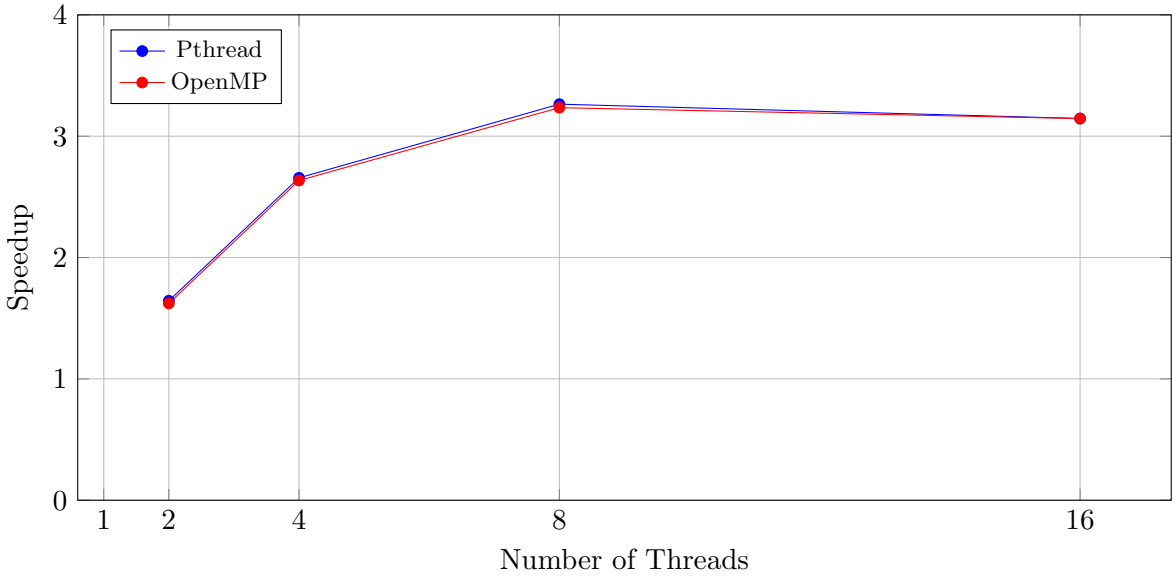

Figure 6: Speedup Comparison - Size 8000

We observe from these plots that the speedup is maximum when the number of threads is around 8 and we get an effective utilisation of around **4** cores.

However, the speedup tends to saturate or diminish as the number of threads becomes large, especially evident in the larger input sizes (4000 and 8000).

§4.3 Plots for the Efficiency vs. Number of threads

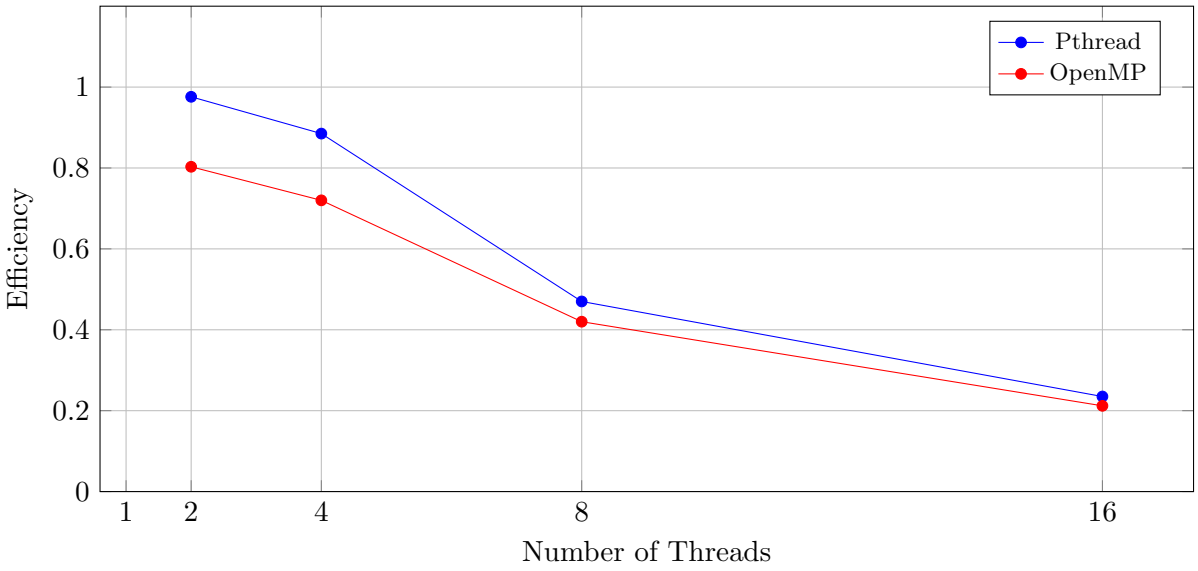

Figure 7: Efficiency Comparison - Size 2000

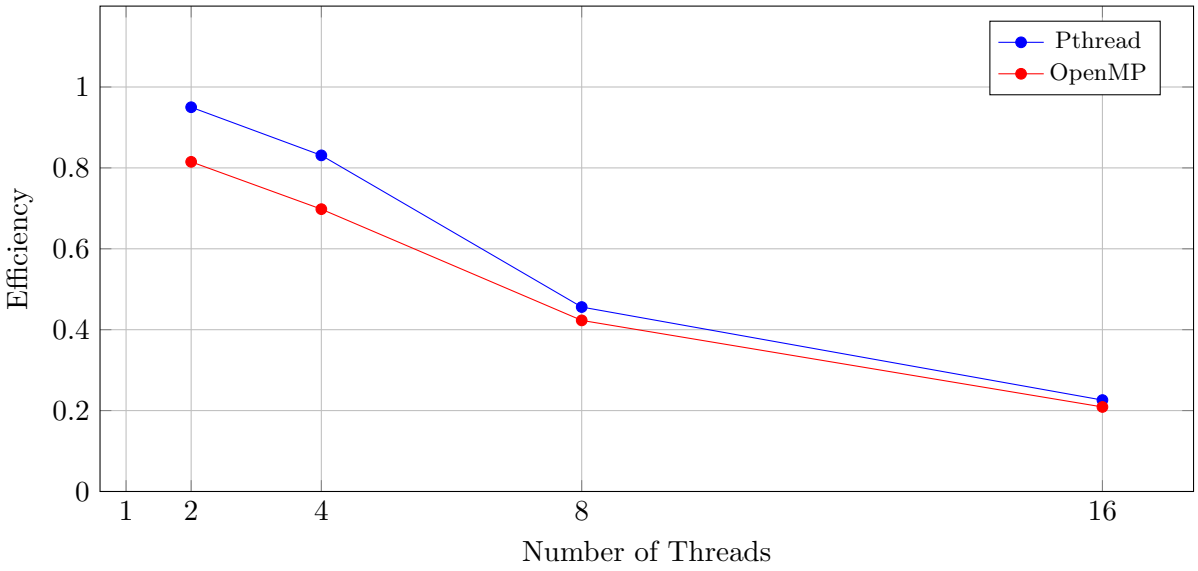

Figure 8: Efficiency Comparison - Size 4000

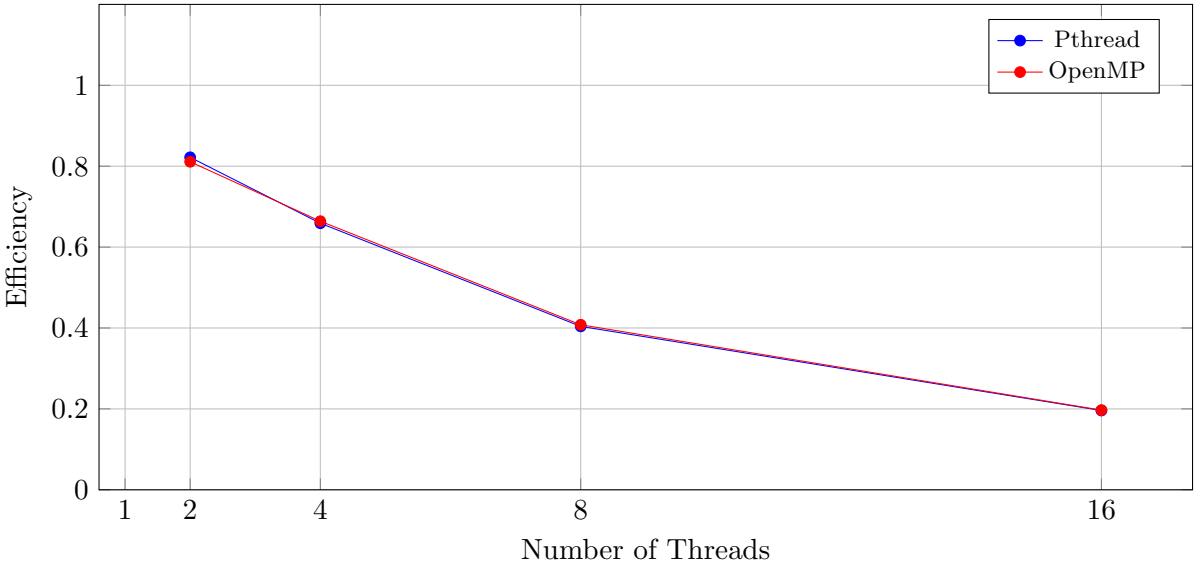

Figure 9: Efficiency Comparison - Size 8000

We observe that the efficiency decreases as the number of threads increases for both pthread and OpenMP implementations, because the utilisation per core decreases as we increase the number of threads. (due to overhead)

We also observe that the efficiency falls rapidly (steep) until the number of threads is 8, after which it falls slowly because we get the maximum utilisation at 8 threads.

Based on the above three analyses (on time taken, speedup and efficiency), we see pthreads gives slightly better results such as efficiency and speedup compared to OpenMP. This happens because OpenMP is a high level abstraction so we get extra overhead in OpenMP, in contrast to pthreads where we explicitly parallelise the code, and hence the result.