# OpenMP and pthreads Assignment

Yukti Makhija (2019BB10067)

## 1) Lower-Upper Decomposition

LU Decomposition is a technique to factorise a matrix ***A*** into the product of one upper (***U***) and one lower (***L***) triangular matrix. The pseudocode provided in the assignment is for LU decomposition with partial pivoting (it uses Gaussian Elimination)which only allows us to permute the rows. A permutation vector ***P*** is updated after the rows are interchanged. At the end the following condition should be satisfied:

$$PA = LU$$

Pseudocode

**Initialization**

**Find Max**

**Row Pivoting**

**Update *L* and *U***

**Update *A***

```
1  inputs: a(n,n)
2  outputs: π(n), l(n,n), and u(n,n)
3
4  initialize π as a vector of length n
5  initialize u as an n x n matrix with 0s below the diagonal
6  initialize l as an n x n matrix with 1s on the diagonal and 0s above the diagonal
7  for i = 1 to n
8      π[i] = i
9
10 for k = 1 to n
11     max = 0
12     for i = k to n
13     if max < |a(i,k)|
14         max = |a(i,k)|
15         k' = i
16     if max == 0
17         error (singular matrix)
18
19     swap π[k] and π[k']
20     swap a(k,:) and a(k',:)
21     swap l(k,1:k-1) and l(k',1:k-1)
22     u(k,k) = a(k,k)
23
24     for i = k+1 to n
25         l(i,k) = a(i,k)/u(k,k)
26         u(k,i) = a(k,i)
27
28     for i = k+1 to n
29         for j = k+1 to n
30             a(i,j) = a(i,j) - l(i,k)*u(k,j)
31
32 Here, the vector π is a compact representation of a permutation matrix p(n,n),
33 which is very sparse. For the ith row of p, π(i) stores the column index of
34 the sole position that contains a 1.
```

All the major steps have been labelled in figure above.

**Initialization**

Matrix ***A*** is initialised with random double values and the 0s were positioned above and below the diagonals for ***L*** and ***U***. This can be achieved with two nested for-loops (complexity = $\mathbf{O(n^2)}$). $\boldsymbol{\pi}$ (compact representation of ***P***) is initialised with the index values (possible in one for loop) (complexity = **O(n)**).

**Find Max**

We iterate over all the columns and find the max absolute value in $k^{th}$ column below the diagonal. The value is stored in **max** and the row ($i^{th}$) in which it was found is stored in **k'**. One for loop is required for each column (two-nested loops). (complexity = $\mathbf{O(n^2)}$)

**Row Pivoting**

When the absolute max value has been identified in the kth column, we swap the rows k and k' (where it was found). Depending on the implementation the complexity would vary. If we store ***A*** , ***L***, ***U*** as 2-D matrices or use an array to store column pointers then this would require one for-loop. But if we use a pointer for each row then pointers for both the rows can be swapped in a single command.

**Update *L* and *U***

The values below the diagonal in L need to be scaled. This is possible with one for-loop (complexity = **O(n)**).

**Update *A***

This is the most important stage in the algorithm where Gaussian Elimination takes place and requires maximum time (>95%). There are three-nested for loops (complexity = $\mathbf{O(n^3)}$) and during parallelization it would become important to determine the optimal order to reduce the runtime. This depends on concepts like cache hit/miss and shared memory. If we are storing the data as an array of n column pointers then the second loop (in i) should come before the $j^{th}$ loop as all the elements in a row can be loaded together. But if we are using column pointers then $j^{th}$ loop should come before $i^{th}$ loop. A significant reduction in runtime was observed after arranging the loops in the correct order.

**Verification**

***L*** and ***U*** matrices obtained can be verified by finding the $L_{2,1}$ norm of $|\boldsymbol{PA - LU}|$ **= Residual matrix.** This value should be close to zero. This is not a part of the implementation and doesn't need to be parallelized.

## 2) Sequential Results

Time taken by different steps in the sequential implementation was measured for (n=4000).

| | **Time (seconds)** | **Percentage of total time (%)** |
|---|---|---|
| Initialization | 0.4 | 0.25 |
| Find Max | 0.15 | 0.10 |
| Row Pivoting | 3.9 | 2.48 |
| Update ***L*** and ***U*** | 0.92 | 0.58 |
| Update ***A*** | **152.03** | **96.59** |
| **Total** | **157.4** | 100 |

## 3) Implementation Details

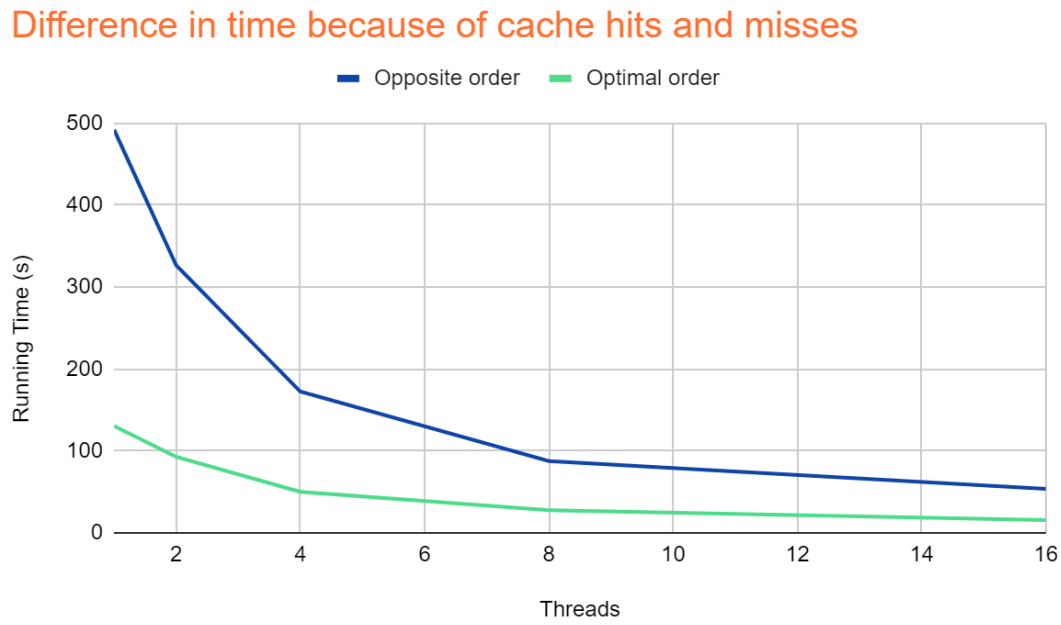
### Optimizations

The following optimizations were performed to increase the performance:

a) Choosing an appropriate data structure:
We had the option of choosing between 2-D arrays and n pointers to n-element data vectors. Representing them as n pointers is more efficient because it allows continuous memory access and negligible cache misses (because large chunks of continuous values can be accessed together). Now we have to make another decision about whether we want to store n-row pointers or n-columns pointers. Based on the pseudocode, column pointers are better. In *L and U Update* and *Find Max* sections of the code, loops iterate over elements of a column. If we store column pointers, then we will be able to access all the elements in a column at once (without cache miss).

As discussed above, in *Update A* step, the order of the loops is very important. In my case, since I had stored n-column pointers to n-element vectors, I ensured that the outer loop iterates on the columns and the inner loop iterates on the rows. I have compared the performance increase observed by changing the order in the following graph.

| n = 4000 | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| Opposite order | 492.125 | 326.641 | 172.375 | 87.589 | 53.763 |
| Optimal order | 130.326 | 92.772 | 50.059 | 27.56 | 15.488 |




b) Initialization: drand48_r() was used to randomly initialise ***A*** of size (n x n) with doubles in the range **(0,1).** This required two nested loops, and the outer one was parallelized. ***L, U***, and ***P*** can be initialised in the same loop.

c) Find Max: The inner loop which iterates on the rows ($k \leq i < n$) of the $k^{th}$ column is parallelized.

d) Update L and U: This is a single for loop that iterates on the columns and can be easily parallelized.

e) Update A: Since there are no loop dependencies, there are two nested for loops inside the main loop which can be parallelized. This optimization has the maximum impact on the total runtime. (I have discussed the order above)

## Performance Metrics

Total Runtime, speedup and efficiency were used to analyse the performance of our parallel programs.

$$S_t = \frac{T_1}{T_t}$$

$$\mathbf{eff}_t = \frac{T_1}{t * T_t}$$

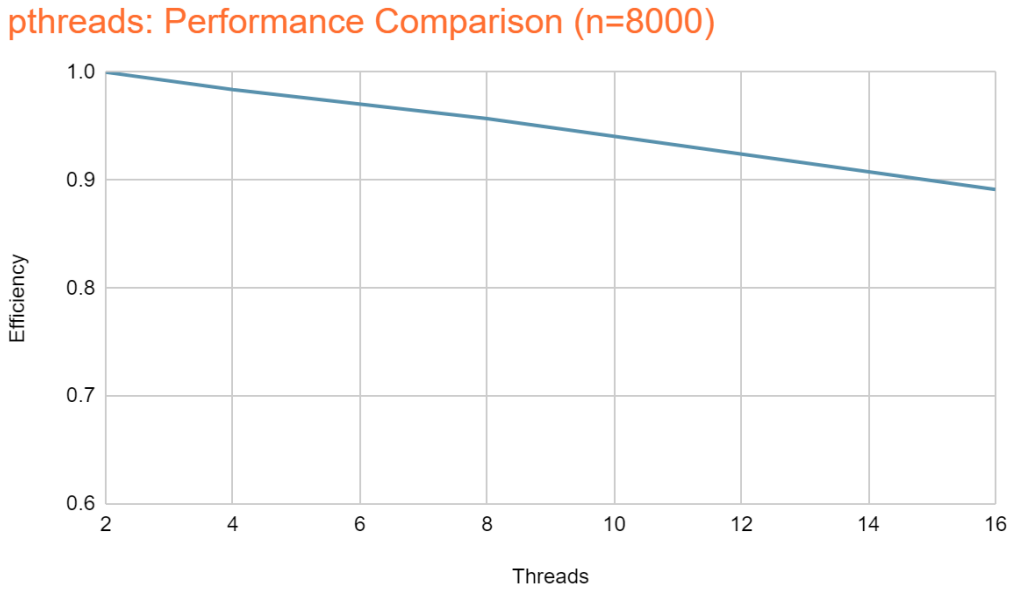
## CPU Information (HPC IITD)

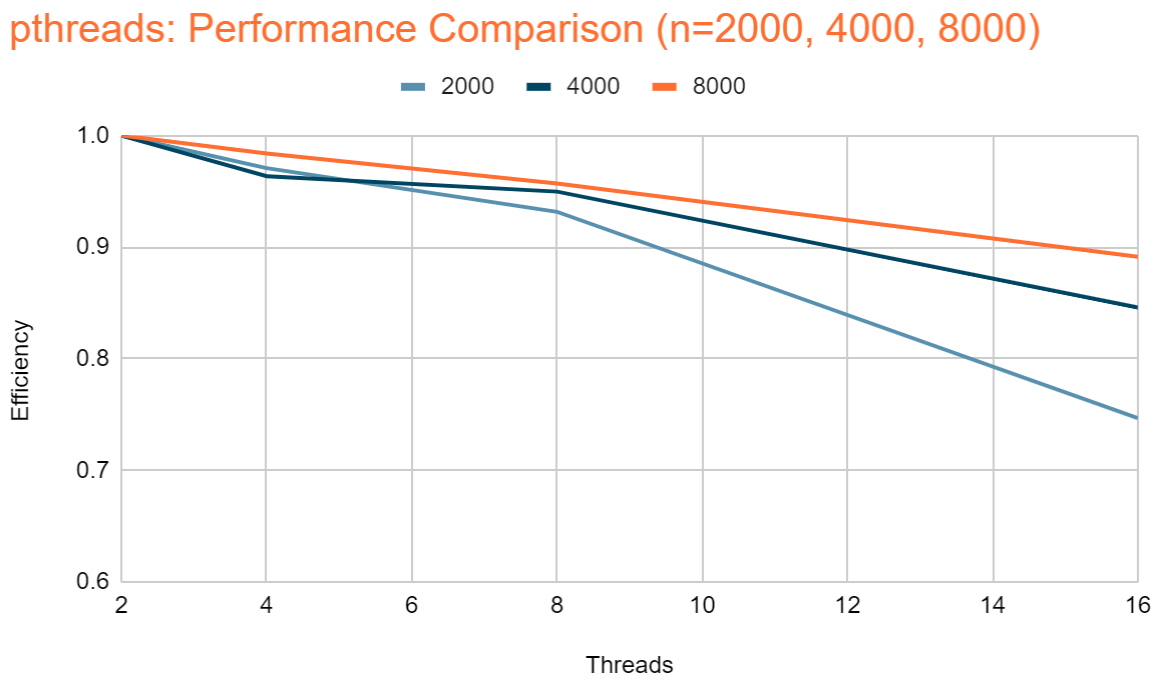
All the experiments were run on Intel Xeon Gold-6148 @ 2.40GHz.

## 4) pthreads Results

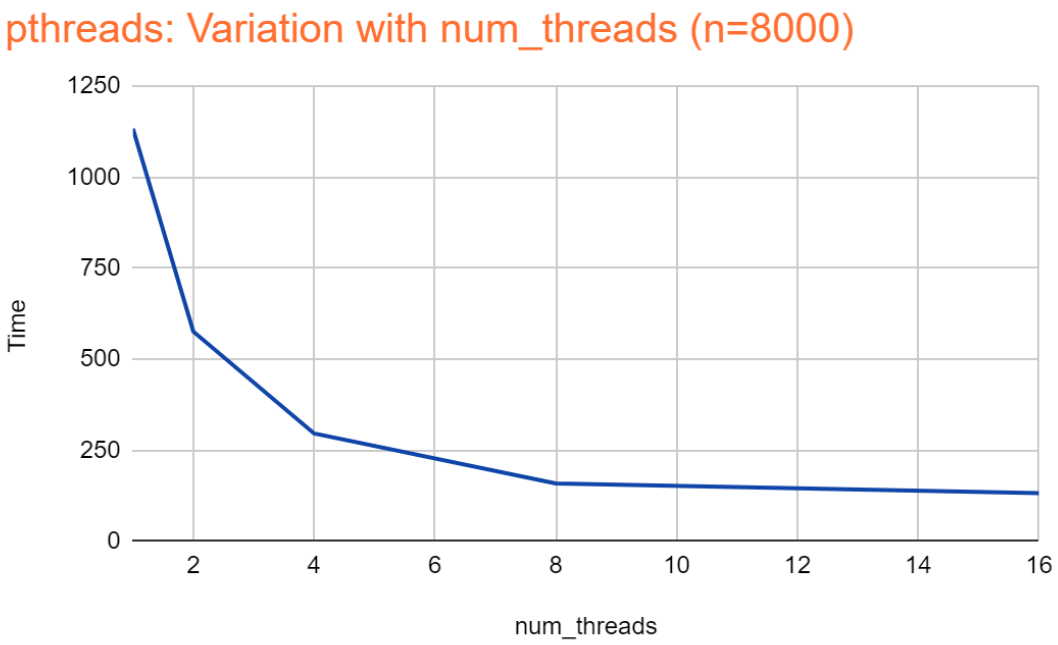
| n \ threads | 1 (Sequential) | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 4 | 0.000385 | 0.000486 | 0.000705 | - | - |
| 10 | 0.000784 | 0.000914 | 0.001355 | 0.002292 | - |
| 100 | 0.01006 | 0.010878 | 0.012822 | 0.021336 | 0.037941 |
| 2000 | 19.13316 | 9.853762 | 5.134571 | 3.202602 | 3.392026 |
| 4000 | 144.1306 | 74.7928 | 37.93881 | 21.2989 | 20.34223 |
| **8000** | **1133.601** | **576.0936** | **296.1369** | **158.9794** | **132.1915** |

**Efficiency vs Number of Threads**





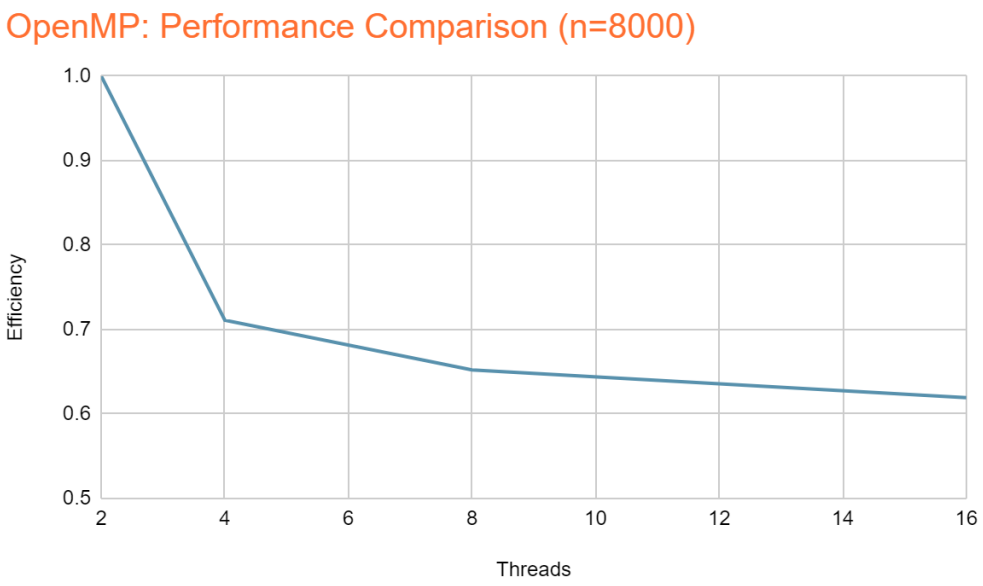

**Running time vs Number of Threads**

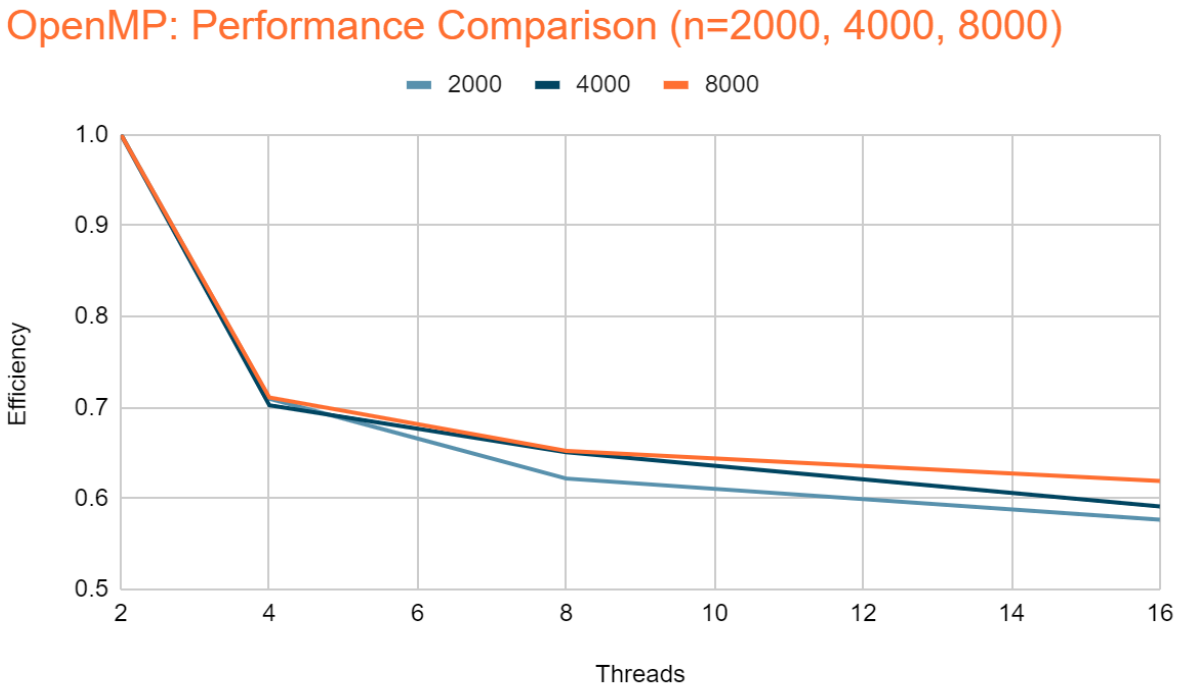


## 5) OpenMP Results

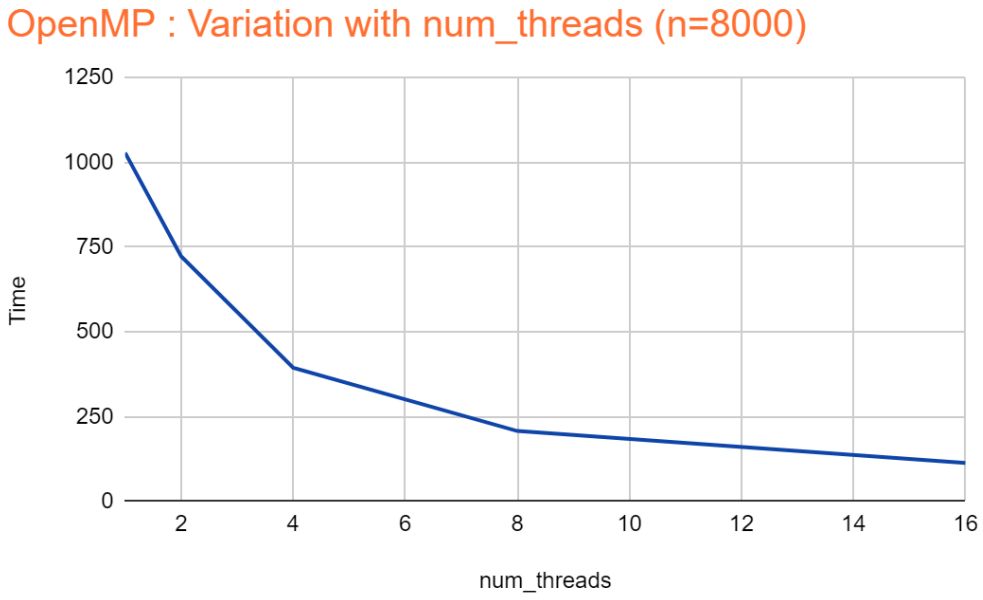
| n \ threads | 1 (Sequential) | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 4 | 0.000127 | 0.000200 | 0.000315 | - | - |
| 10 | 0.000163 | 0.000254 | 0.000266 | 0.000439 | - |
| 100 | 0.002691 | 0.003199 | 0.003072 | 0.003443 | 1.047959 |
| 2000 | 16.44022 | 11.58318 | 6.609498 | 3.563321 | 2.128477 |
| 4000 | 130.3256 | 92.77135 | 50.05883 | 27.56014 | 15.48819 |
| **8000** | **1028.224** | **722.9866** | **394.1345** | **207.5554** | **113.441** |

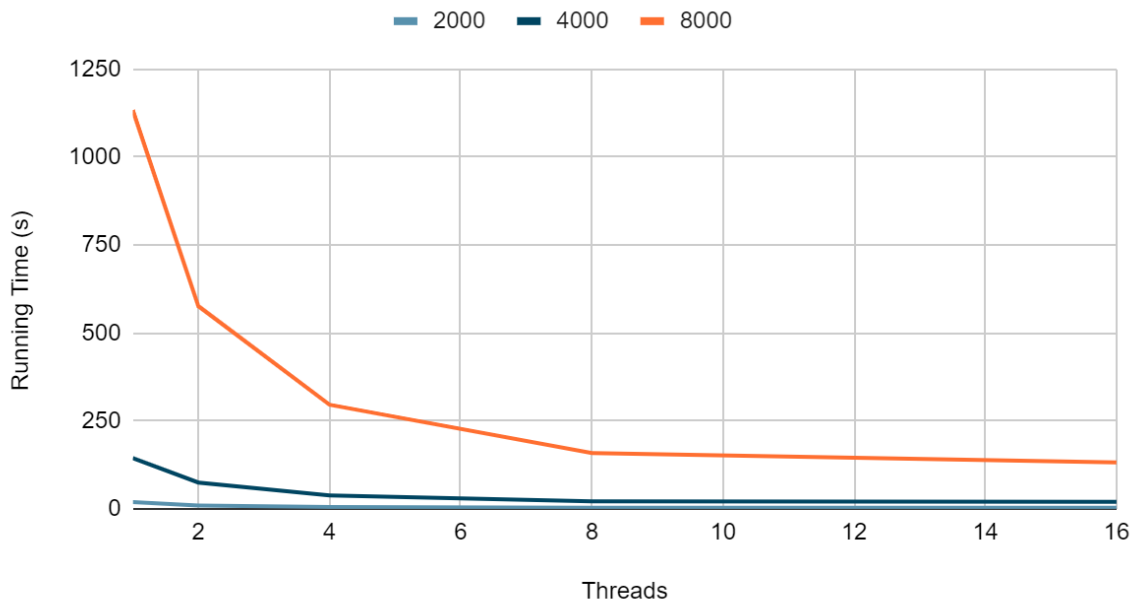
### Efficiency vs Number of Threads






**Running time vs Number of Threads**




OpenMP: Performance Comparison (n=2000, 4000, 8000)
2000
4000
8000
1250
1000
750
500
250
0
Running Time (s)
2
4
6
8
10
12
14
16
Threads