ASSIGNMENT-2

Anannya Mathur 2019TT10953

Pseudo code:

```
inputs: a(n,n)
outputs: π(n), l(n,n), and u(n,n)

initialize π 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
  π[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 π[k] and π[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)

Here, the vector π is a compact representation of a permutation matrix p(n,n),
which is very sparse. For the ith row of p, π(i) stores the column index of
the sole position that contains a 1.
```

Parallelising using OpenMP:

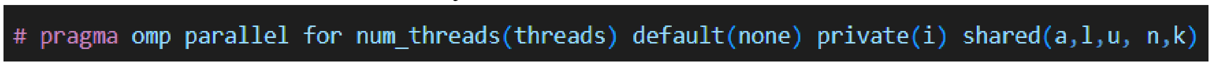
n= size of the matrix, t= number of threads

n	t	Execution time (in s)
7000	1	482
7000	2	269
7000	4	177
7000	8	135
7000	16	123

The error magnitude was 0.0 in all the cases, therefore validating that the parallelisation did not affect the accuracy.

Speedup, S= $T_{serial}/T_{parallel}$
Efficiency, E= S/p, where p= number of processors

As evident from the pseudocode, the last part of the code has three loops, making it of the order $O(n^3)$; therefore, the code attempts to parallelise this part of the code for better efficiency in terms of execution time.



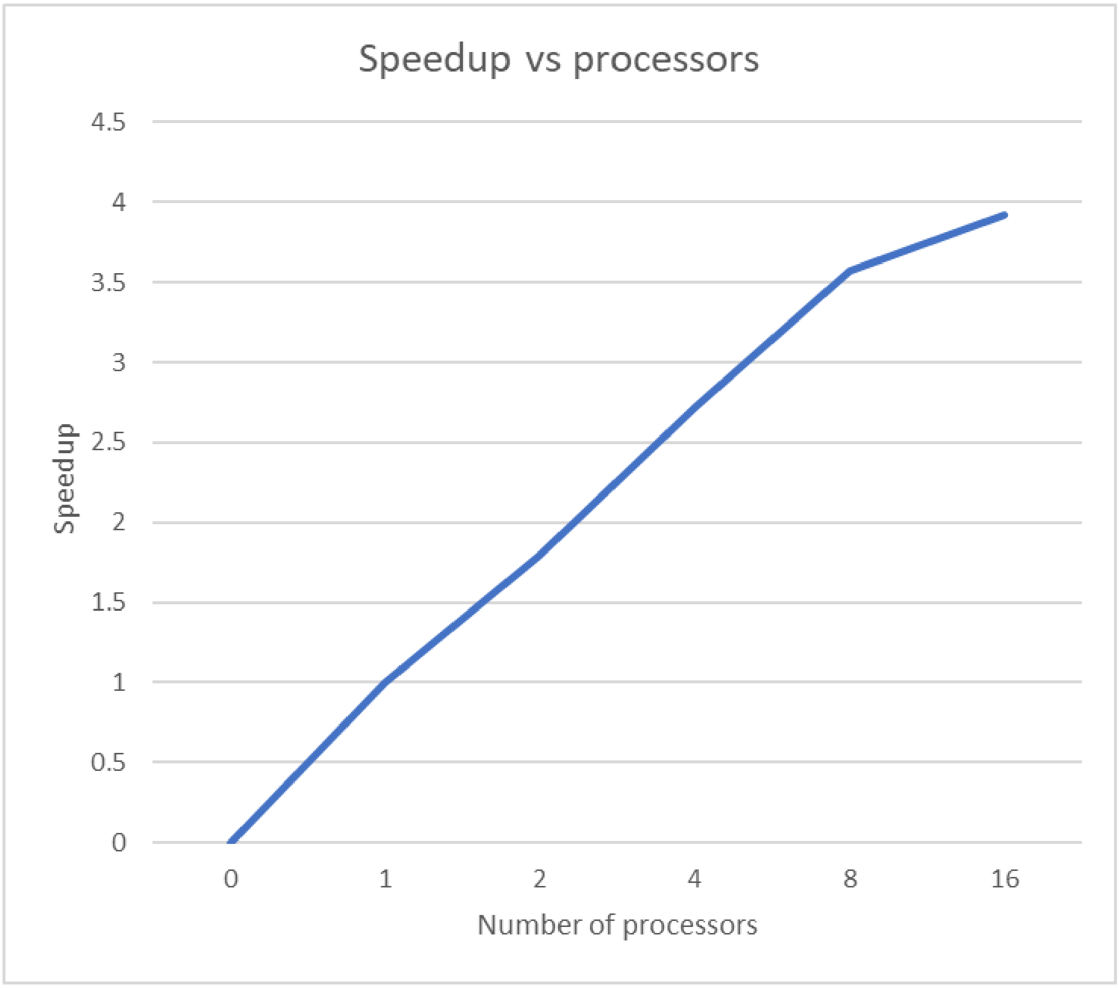
```
# pragma omp parallel for num_threads(threads) default(none) private(i) shared(a,l,u, n,k)
```

private(i) makes sure that the variable i in the loop remains private to each thread, while the matrices a, l, u and variables n, k are shared between the threads. Keeping i private and ensuring the shared variables are contained in a sequential block of code, synchronisation is ensured.

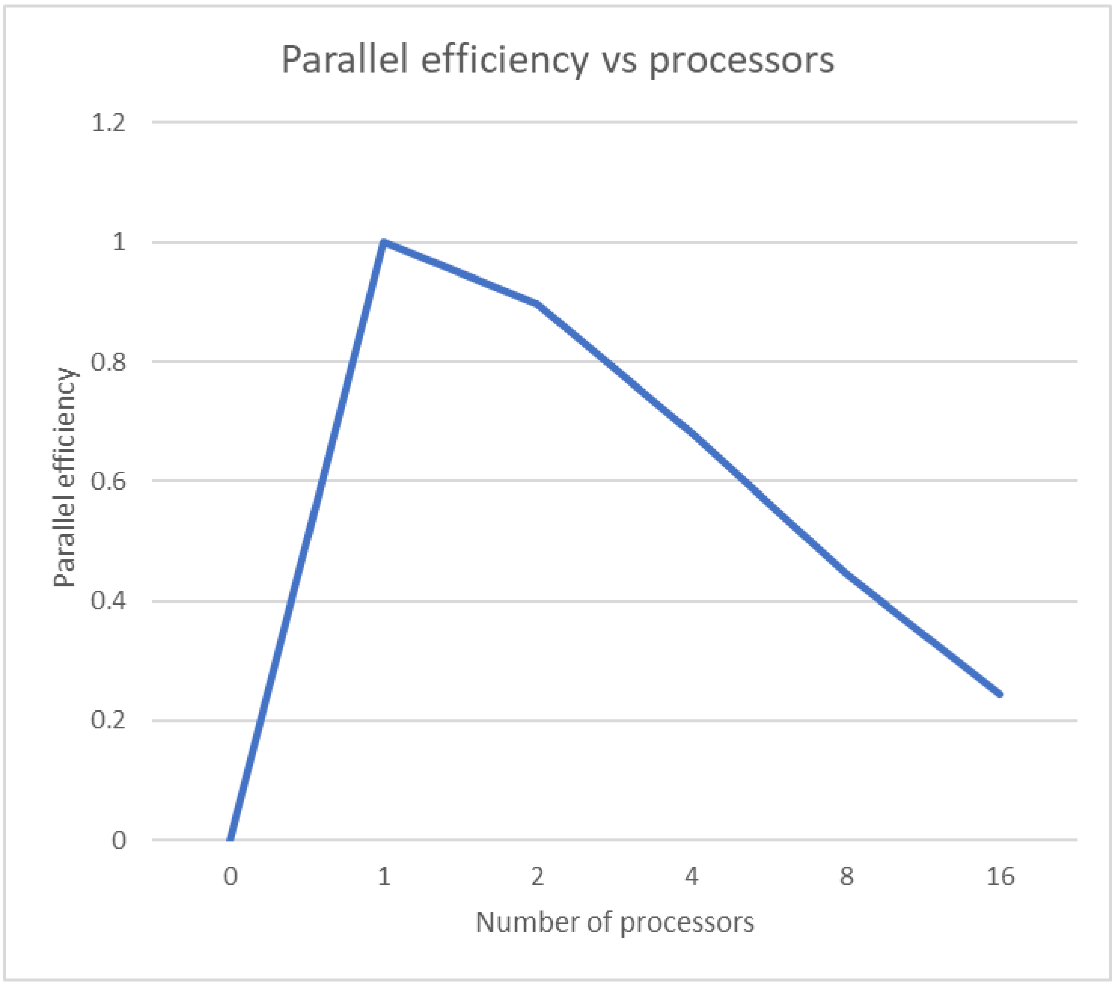
A contiguous layout of the array was implemented to increase the read performance of the memory and the processing speed, but to avoid unnecessary wastage of memory, every memory allocation was freed immediately after its use by calling the free() function.

For n=7000,

p	1	2	4	8	16
S	1	1.79	2.72	3.57	3.92
E=S/p	1	0.8959	0.68	0.446	0.245



Speedup vs processors
Speedup
Number of processors
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
1
2
4
8
16



Parallelising using Pthreads:

n	t	Execution time (in s)
7000	1	400
7000	2	208
7000	4	154
7000	8	146
7000	16	126

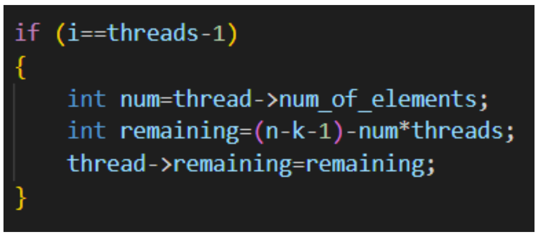
The error magnitude was 0.0 in all the cases, therefore validating that the parallelisation did not affect the accuracy.

Parallelisation strategy:

The last part of the code containing three loops was parallelised as it consumed the maximum component of the execution time, owing to it being O(n^3).

A structure for every thread was created that contained the values- a, l, u, begin, rank, number of elements, where the "begin" variable contained the value where the loop within the thread should begin, "number of elements"= [n-(k+1)]/t, rank=i. Here, "begin" is assigned k+1.

Since t may not always divide n-k-1, to make sure no array element was left behind to be computed, the code additionally maintained a variable "remaining" within the thread structure.



```
if (i==threads-1)
{
    int num=thread->num_of_elements;
    int remaining=(n-k-1)-num*threads;
    thread->remaining=remaining;
}
```

On reaching the last thread, the "remaining" variable was updated, which determined the number of extra elements to be included for the computation in the last thread. For the rest of the threads, this variable was assigned 0.

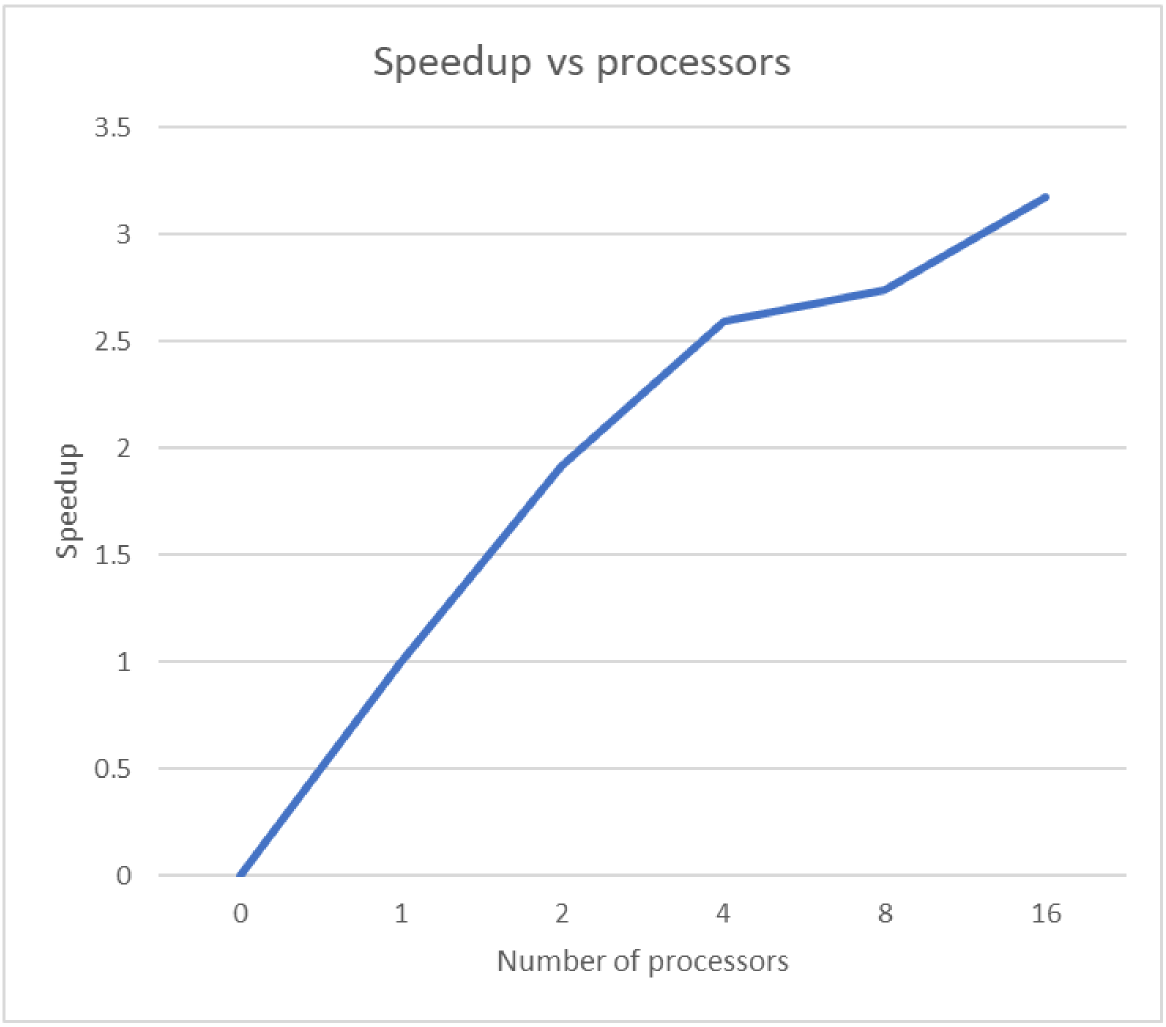
To ensure synchronisation, every thread is assigned equal chunks of shared data that they compute independently of each other, while the updates to the matrix 'a' are contained in a sequential block of code.

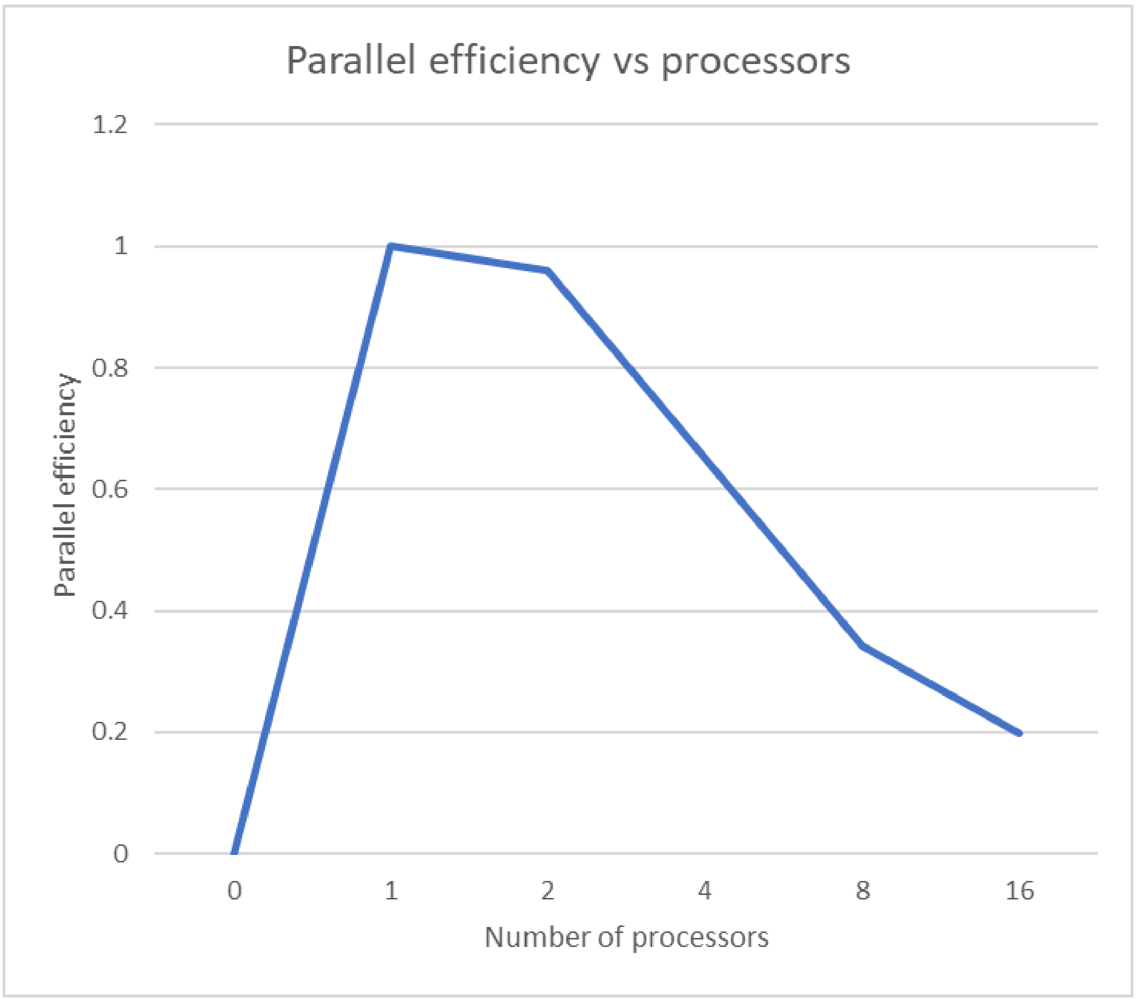
To minimise the occurrences of false sharing, variables private to each thread were maintained to access and update within the loops to avoid interfering with the data shared amongst the threads.

Speedup, S= $T_{serial}/T_{parallel}$
Efficiency, E= S/p, where p= number of processors

For n=7000,

p	1	2	4	8	16
S	1	1.92	2.597	2.7397	3.1746
E=S/p	1	0.96	0.649	0.342	0.198


Speedup vs processors
Speedup
3.5
3
2.5
2
1.5
1
0.5
0
0
1
2
4
8
16
Number of processors


Parallel efficiency vs processors
1.2
1
0.8
0.6
0.4
0.2
0
Parallel efficiency
0
1
2
4
8
16
Number of processors