Abstract

In this project we've been asked to code a program that execute the merge sort using shared memory code in C.

Notes:

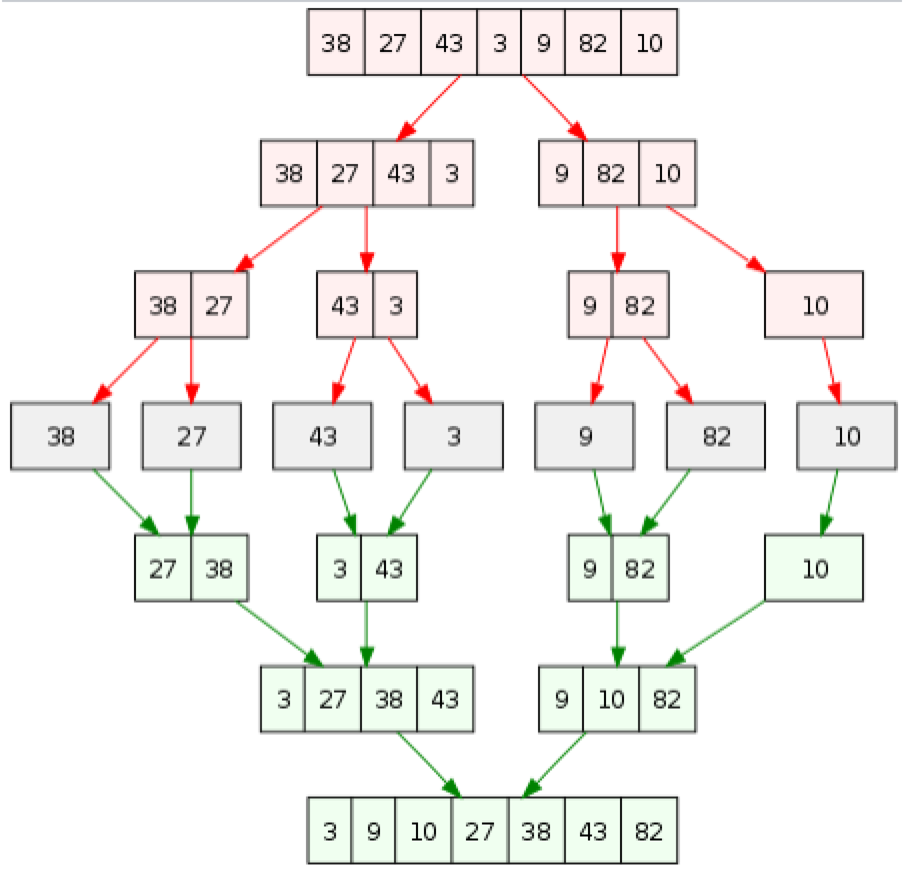
- The full code is attached in the zip file – MergeSort.c .
- The program prints the time it takes to the merge sort algorithm to be done.
- The program asks the user to input the size of the array first.
- After you input the size of the array, the program asks the user to input the values of the array.
- We used the same size for the array as requested: 32, we also stayed we the same size all the time so the comparison will be accurate.
- We used the same values for the array all the time so the array will be accurate:
  Array values: [9, 5, 1, 5, 19, 32, 48, 55, 66, 159, 957, 326, 15, 0, 51, 5, 3, 222, 213, 357, 68, 45, 21, 23, 32, 45, 7, 8, 32, 100, 101, 10].
- To compile the OpenMP code in linux we used: "gcc -fopenmp MergeSort.c".
- To run the OpenMP code we used: "./a.out".
- To check the time it took for the algorithm to work we used: "omp_get_wtime()", as requested.
- To set the number of threads we used everytime we set it as an environment variable, and used: "export OMP_NUM_THREADS=X".

Merge Sort Explanation

Merge Sort is part of divide and conquer algorithm “family”.

The Idea behind merge sort is to divide the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge function is used for merging two halves.

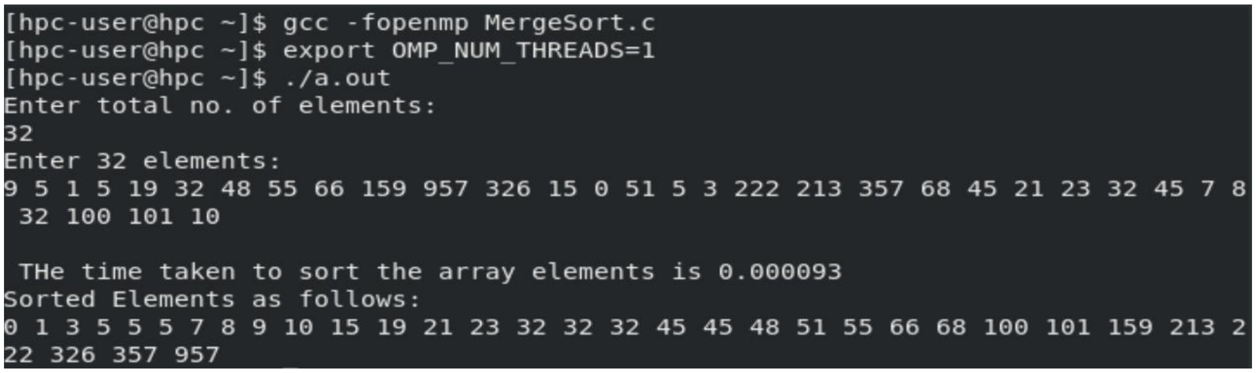
The merge(array, left, middle, right) is a key process that assumes that array[left … middle] and array[middle+1 … right] are sorted and merges the two sorted sub-arrays into one.



Pic 1: a visualization of Merge Sort Performance (from Wikipedia – Merge Sort).

Outputs of different numbers of Threads:

Number of Threads = 1:

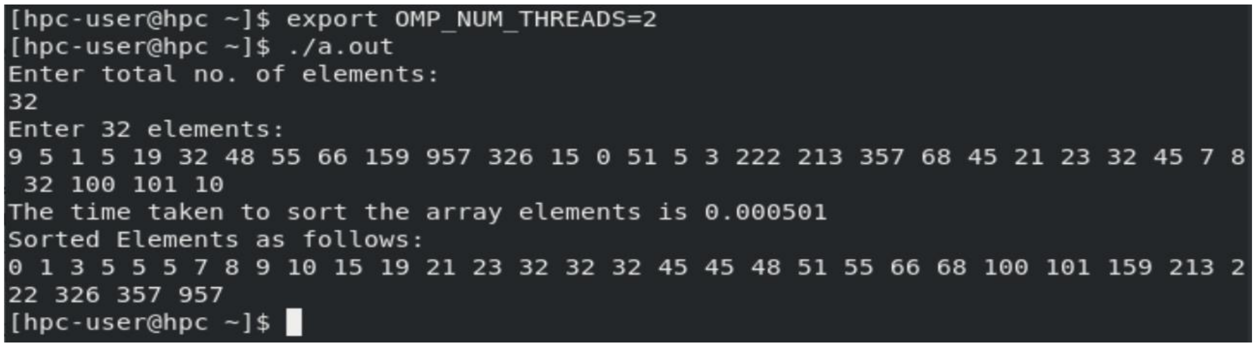


```
[hpc-user@hpc ~]$ gcc -fopenmp MergeSort.c
[hpc-user@hpc ~]$ export OMP_NUM_THREADS=1
[hpc-user@hpc ~]$ ./a.out
Enter total no. of elements:
32
Enter 32 elements:
9 5 1 5 19 32 48 55 66 159 957 326 15 0 51 5 3 222 213 357 68 45 21 23 32 45 7 8
 32 100 101 10

 THe time taken to sort the array elements is 0.000093
Sorted Elements as follows:
0 1 3 5 5 5 7 8 9 10 15 19 21 23 32 32 32 45 45 48 51 55 66 68 100 101 159 213 2
22 326 357 957
```

Pic 2: output of the program for 1 thread.

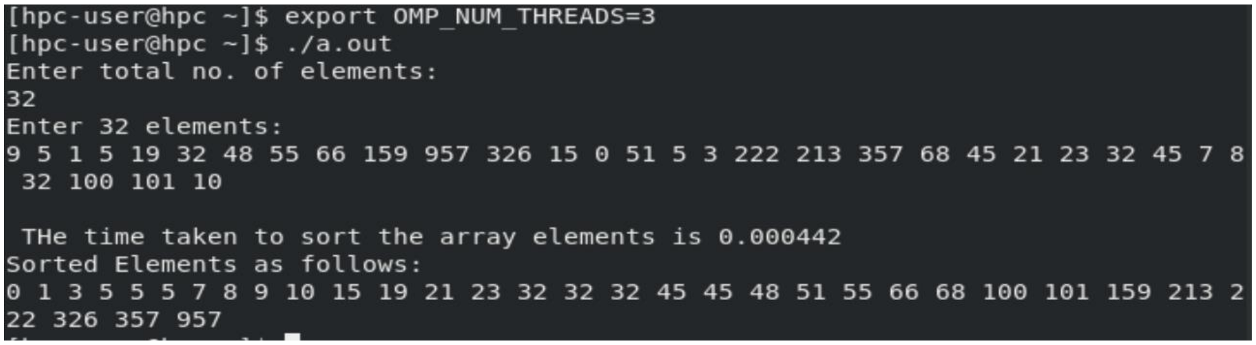
Number of Threads = 2:



```
[hpc-user@hpc ~]$ export OMP_NUM_THREADS=2
[hpc-user@hpc ~]$ ./a.out
Enter total no. of elements:
32
Enter 32 elements:
9 5 1 5 19 32 48 55 66 159 957 326 15 0 51 5 3 222 213 357 68 45 21 23 32 45 7 8
 32 100 101 10
The time taken to sort the array elements is 0.000501
Sorted Elements as follows:
0 1 3 5 5 5 7 8 9 10 15 19 21 23 32 32 32 45 45 48 51 55 66 68 100 101 159 213 2
22 326 357 957
[hpc-user@hpc ~]$
```

Pic 3: output of the program for 2 threads.

Number of Threads = 3:

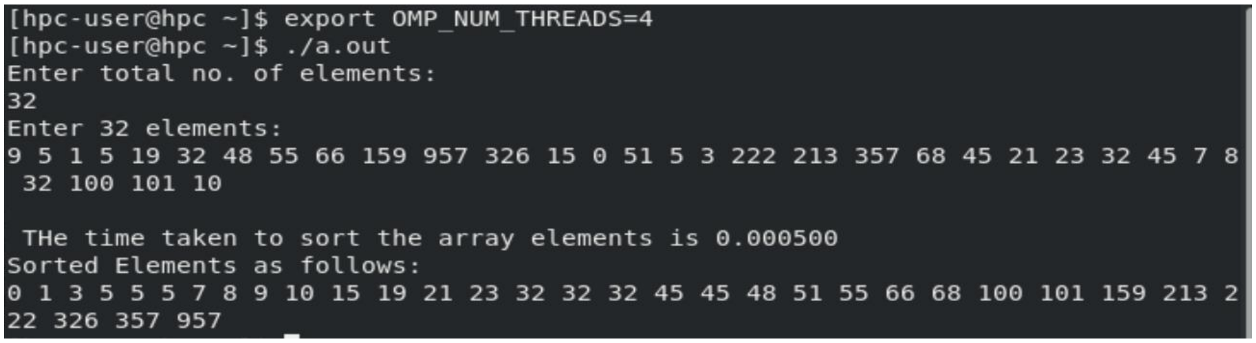


```
[hpc-user@hpc ~]$ export OMP_NUM_THREADS=3
[hpc-user@hpc ~]$ ./a.out
Enter total no. of elements:
32
Enter 32 elements:
9 5 1 5 19 32 48 55 66 159 957 326 15 0 51 5 3 222 213 357 68 45 21 23 32 45 7 8
 32 100 101 10

 THe time taken to sort the array elements is 0.000442
Sorted Elements as follows:
0 1 3 5 5 5 7 8 9 10 15 19 21 23 32 32 32 45 45 48 51 55 66 68 100 101 159 213 2
22 326 357 957
```

Pic 4: output of the program for 3 threads.

Number of Threads = 4:

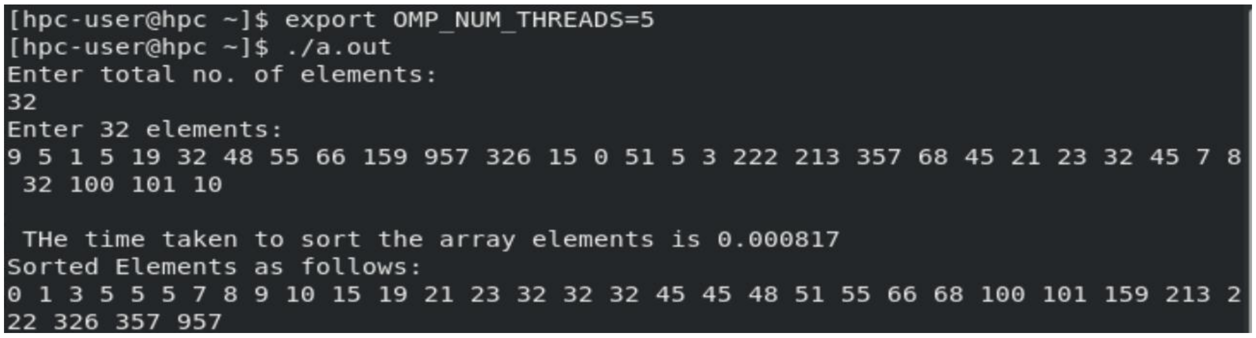


```
[hpc-user@hpc ~]$ export OMP_NUM_THREADS=4
[hpc-user@hpc ~]$ ./a.out
Enter total no. of elements:
32
Enter 32 elements:
9 5 1 5 19 32 48 55 66 159 957 326 15 0 51 5 3 222 213 357 68 45 21 23 32 45 7 8
 32 100 101 10

 THe time taken to sort the array elements is 0.000500
Sorted Elements as follows:
0 1 3 5 5 5 7 8 9 10 15 19 21 23 32 32 32 45 45 48 51 55 66 68 100 101 159 213 2
22 326 357 957
```

Pic 5: output of the program for 4 threads.

Number of Threads = 5:

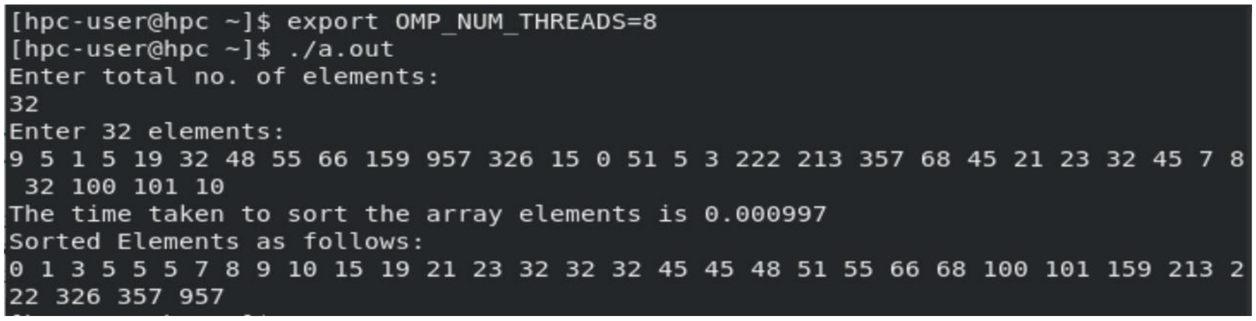


```
[hpc-user@hpc ~]$ export OMP_NUM_THREADS=5
[hpc-user@hpc ~]$ ./a.out
Enter total no. of elements:
32
Enter 32 elements:
9 5 1 5 19 32 48 55 66 159 957 326 15 0 51 5 3 222 213 357 68 45 21 23 32 45 7 8
 32 100 101 10

 THe time taken to sort the array elements is 0.000817
Sorted Elements as follows:
0 1 3 5 5 5 7 8 9 10 15 19 21 23 32 32 32 45 45 48 51 55 66 68 100 101 159 213 2
22 326 357 957
```

Pic 6: output of the program for 5 threads.

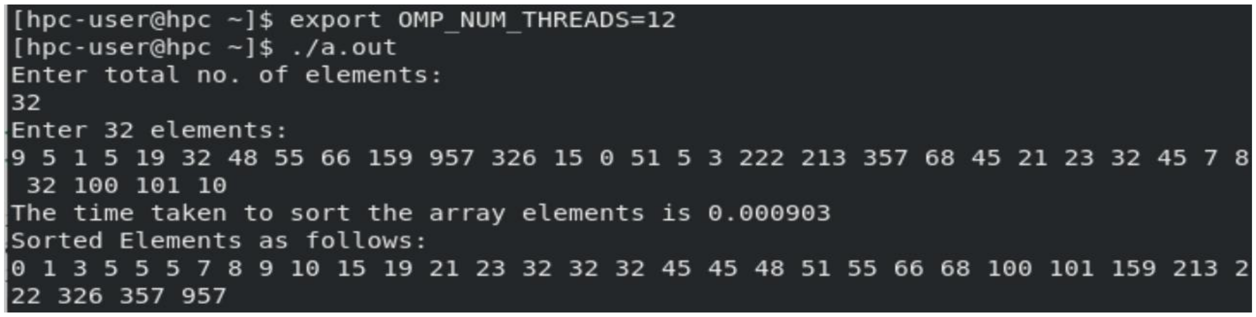
Number of Threads = 8:



```
[hpc-user@hpc ~]$ export OMP_NUM_THREADS=8
[hpc-user@hpc ~]$ ./a.out
Enter total no. of elements:
32
Enter 32 elements:
9 5 1 5 19 32 48 55 66 159 957 326 15 0 51 5 3 222 213 357 68 45 21 23 32 45 7 8
 32 100 101 10
The time taken to sort the array elements is 0.000997
Sorted Elements as follows:
0 1 3 5 5 5 7 8 9 10 15 19 21 23 32 32 32 45 45 48 51 55 66 68 100 101 159 213 2
22 326 357 957
```

Pic 7: output of the program for 8 threads.

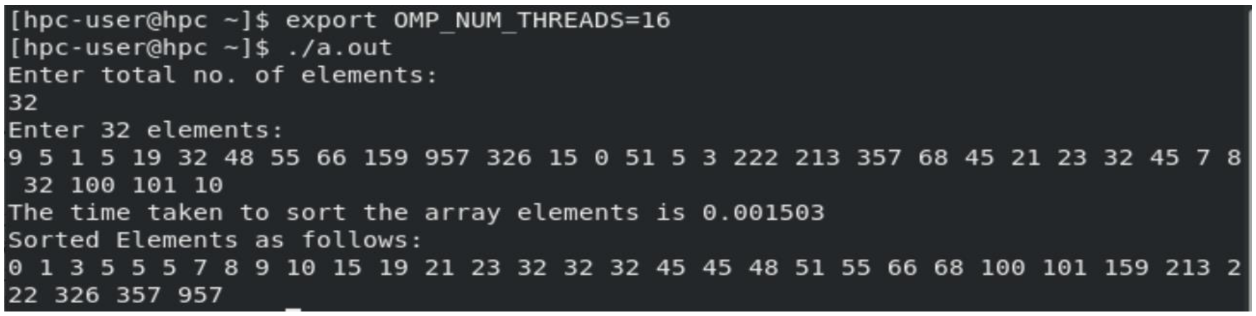
Number of Threads = 12:



```
[hpc-user@hpc ~]$ export OMP_NUM_THREADS=12
[hpc-user@hpc ~]$ ./a.out
Enter total no. of elements:
32
Enter 32 elements:
9 5 1 5 19 32 48 55 66 159 957 326 15 0 51 5 3 222 213 357 68 45 21 23 32 45 7 8
 32 100 101 10
The time taken to sort the array elements is 0.000903
Sorted Elements as follows:
0 1 3 5 5 5 7 8 9 10 15 19 21 23 32 32 32 45 45 48 51 55 66 68 100 101 159 213 2
22 326 357 957
```

Pic 8: output of the program for 12 threads.

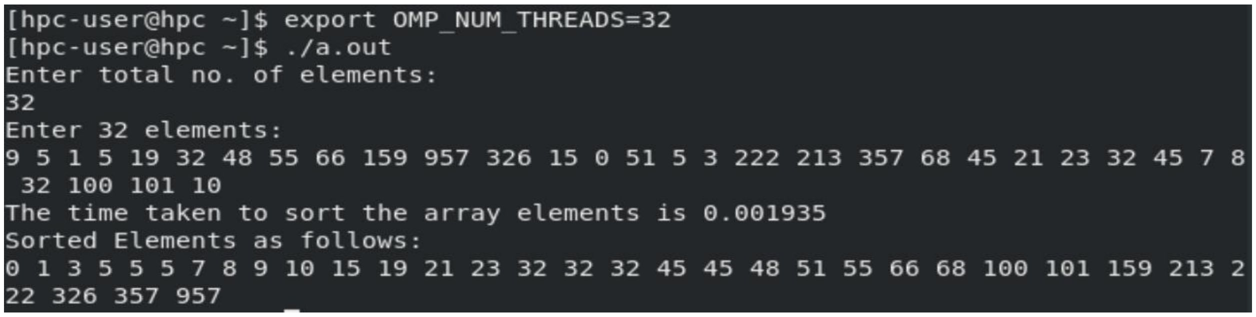
Number of Threads = 16:



```
[hpc-user@hpc ~]$ export OMP_NUM_THREADS=16
[hpc-user@hpc ~]$ ./a.out
Enter total no. of elements:
32
Enter 32 elements:
9 5 1 5 19 32 48 55 66 159 957 326 15 0 51 5 3 222 213 357 68 45 21 23 32 45 7 8
 32 100 101 10
The time taken to sort the array elements is 0.001503
Sorted Elements as follows:
0 1 3 5 5 5 7 8 9 10 15 19 21 23 32 32 32 45 45 48 51 55 66 68 100 101 159 213 2
22 326 357 957
```

Pic 9: output of the program for 16 threads.

Number of Threads = 32:



```
[hpc-user@hpc ~]$ export OMP_NUM_THREADS=32
[hpc-user@hpc ~]$ ./a.out
Enter total no. of elements:
32
Enter 32 elements:
9 5 1 5 19 32 48 55 66 159 957 326 15 0 51 5 3 222 213 357 68 45 21 23 32 45 7 8
 32 100 101 10
The time taken to sort the array elements is 0.001935
Sorted Elements as follows:
0 1 3 5 5 5 7 8 9 10 15 19 21 23 32 32 32 45 45 48 51 55 66 68 100 101 159 213 2
22 326 357 957
```

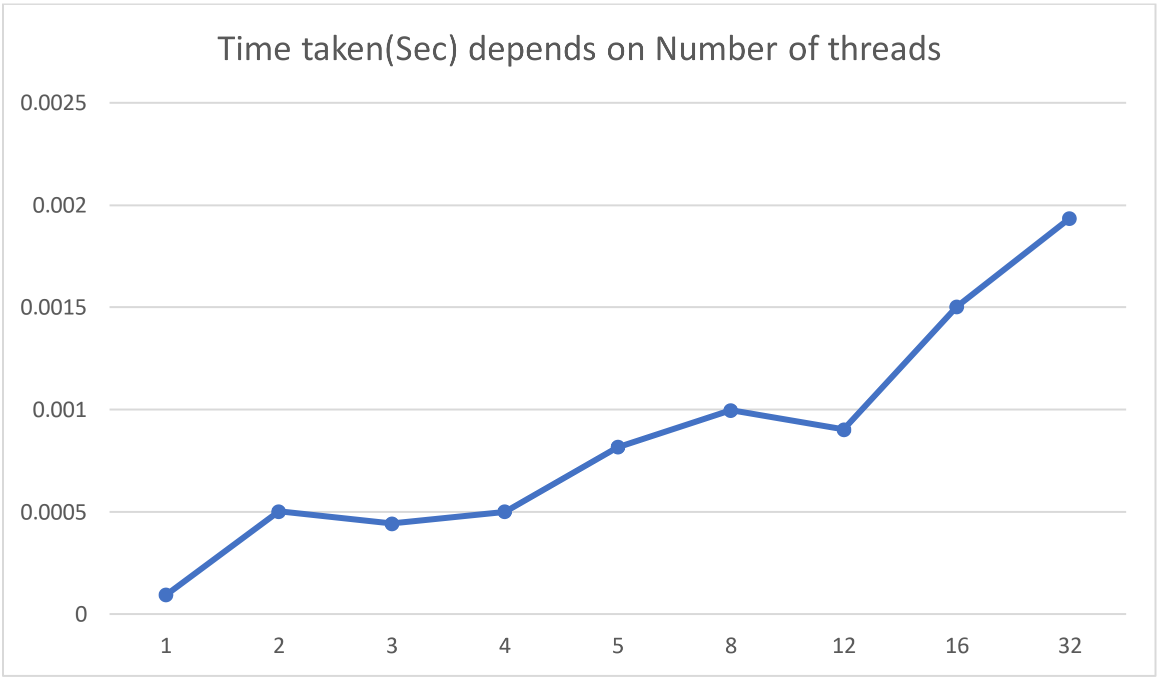
Pic 10: output of the program for 32 threads.

Efficiency Table

We put all the results in a table so we can see clear the time difference:

Number of Threads:	Time Taken(Sec)
1	0.000093
2	0.000501
3	0.000442
4	0.0005
5	0.000817
8	0.000997
12	0.000903
16	0.001503
32	0.001935

Efficiency Graph



We can see that the more threads we add the longer it takes to the program to run,

But we can clearly understand that it shouldn’t happen and it happened only because we ran it on a low size of array, and if we would have run it on a much bigger size – we will see that the more threads, the less time it would take to the program to run.

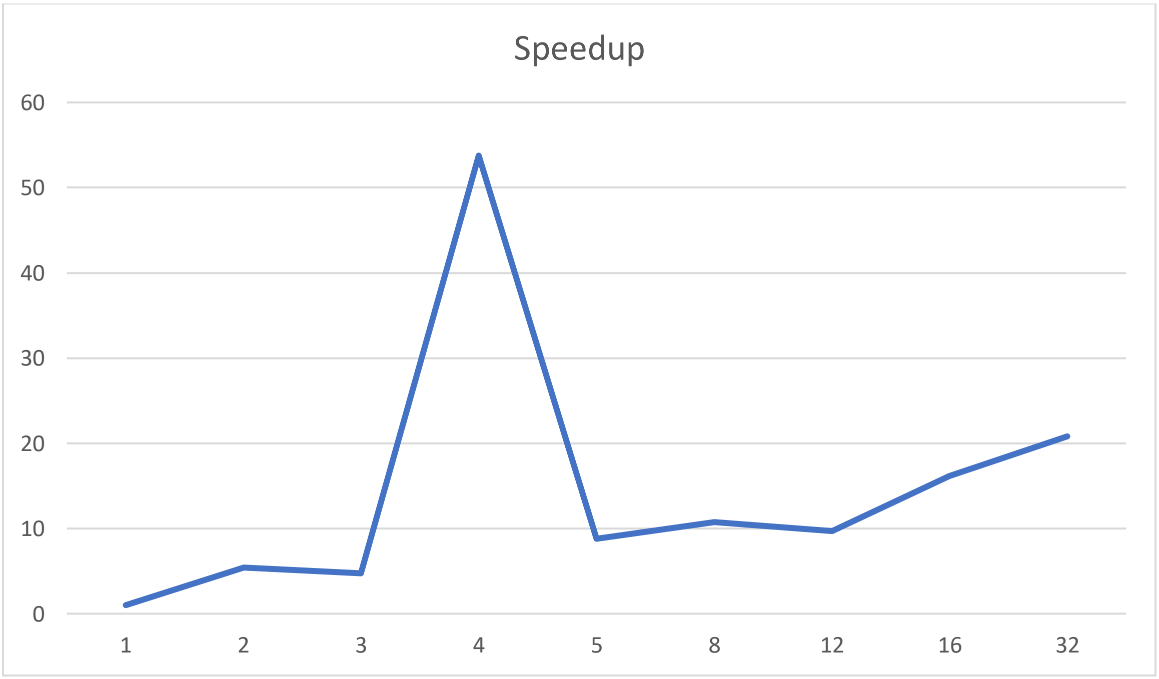
Speedup Table

We calculated the speedup to each number of threads with the speedup equation we've learned on class:

- Speedup = $t_s \ / \ t_n,$ 0=<speedup=<n

Number of Threads:	**Speedup**
1	1
2	5.387
3	4.752
4	53.763
5	8.784
8	10.720
12	9.709
16	16.161
32	20.806

Speedup Graph

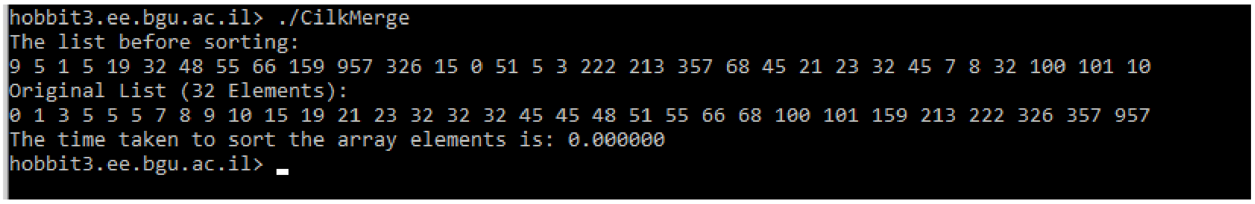


Cilk Code

- We ran the Cilk code on the hobbits.
- The name of the file is CilkMergeSort.c
- We have compiled it with “cilk++ -o CilkMerge ./ CilkMergeSort.c
- And ran it with: ./CilkMerge.
- We used the same array to have a good comparison: [9, 5, 1, 5, 19, 32, 48, 55, 66, 159, 957, 326, 15, 0, 51, 5, 3, 222, 213, 357, 68, 45, 21, 23, 32, 45, 7, 8, 32, 100, 101, 10].
- We changed the number of threads with: “setenv CLIK_NWORKERS X”.

Outputs of Cilk Code

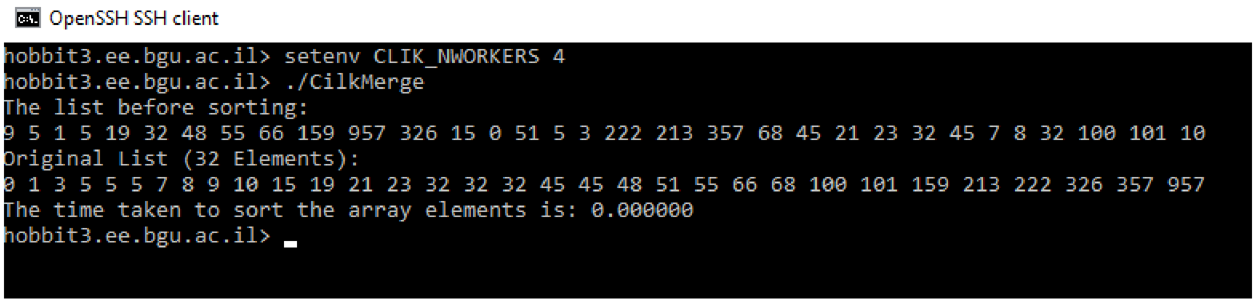
Number of Threads = 2:



```
hobbit3.ee.bgu.ac.il> ./CilkMerge
The list before sorting:
9 5 1 5 19 32 48 55 66 159 957 326 15 0 51 5 3 222 213 357 68 45 21 23 32 45 7 8 32 100 101 10
Original List (32 Elements):
0 1 3 5 5 5 7 8 9 10 15 19 21 23 32 32 32 45 45 48 51 55 66 68 100 101 159 213 222 326 357 957
The time taken to sort the array elements is: 0.000000
hobbit3.ee.bgu.ac.il> _
```

Pic 11: output of the cilk program with 2 threads.

Number of Threads = 4:



OpenSSH SSH client

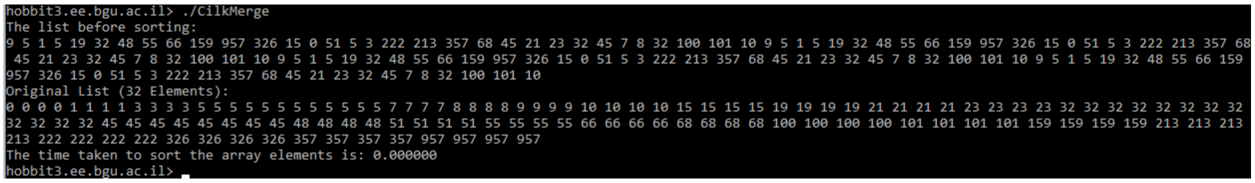
```
hobbit3.ee.bgu.ac.il> setenv CLIK_NWORKERS 4
hobbit3.ee.bgu.ac.il> ./CilkMerge
The list before sorting:
9 5 1 5 19 32 48 55 66 159 957 326 15 0 51 5 3 222 213 357 68 45 21 23 32 45 7 8 32 100 101 10
Original List (32 Elements):
0 1 3 5 5 5 7 8 9 10 15 19 21 23 32 32 32 45 45 48 51 55 66 68 100 101 159 213 222 326 357 957
The time taken to sort the array elements is: 0.000000
hobbit3.ee.bgu.ac.il> _
```

Pic 12: output of the cilk program with 4 threads.

Conclusions for Cilk Code

No matter for what number of threads we kept running the program, the execution time was always -> 0.

So, we’ve tried to run it with different number of threads on a bigger size of array (128 elements) – and we still got the same execution time:



```
hobbit3.ee.bgu.ac.il> ./CilkMerge
The list before sorting:
9 5 1 5 19 32 48 55 66 159 957 326 15 0 51 5 3 222 213 357 68 45 21 23 32 45 7 8 32 100 101 10 9 5 1 5 19 32 48 55 66 159 957 326 15 0 51 5 3 222 213 357 68
 45 21 23 32 45 7 8 32 100 101 10 9 5 1 5 19 32 48 55 66 159 957 326 15 0 51 5 3 222 213 357 68 45 21 23 32 45 7 8 32 100 101 10 9 5 1 5 19 32 48 55 66 159
957 326 15 0 51 5 3 222 213 357 68 45 21 23 32 45 7 8 32 100 101 10
Original List (32 Elements):
0 0 0 0 1 1 1 1 3 3 3 3 5 5 5 5 5 5 5 5 5 5 5 5 7 7 7 7 8 8 8 8 9 9 9 9 10 10 10 10 15 15 15 15 19 19 19 19 21 21 21 21 23 23 23 23 32 32 32 32 32 32 32 32
32 32 32 32 45 45 45 45 45 45 45 45 48 48 48 48 51 51 51 51 55 55 55 55 66 66 66 66 68 68 68 68 100 100 100 100 101 101 101 101 159 159 159 159 213 213 213
213 222 222 222 222 326 326 326 326 357 357 357 357 957 957 957 957
The time taken to sort the array elements is: 0.000000
hobbit3.ee.bgu.ac.il>
```

Pic 13: output of the cilk program with 32 threads & 128 elements.

We assume that for a much bigger size of array we will start to see the run time getting higher, for example million+ elements.

The speedup and efficiency graphs will be a straight line, and due to all the results are the same there is no need to see the graph to understand we stay on the same spot all the time.

2 Open Question on what we've learned so far on OpenMP:

***Right answers will be marked in YELLOW**

Question number 1:

Which of the following about OpenMP is incorrect?

1. OpenMP is an API that enables explicit multi-threaded parallelism.
2. The primary components of OpenMP are compiler directives, runtime library, and environment variables.
3. OpenMP implementations exist for the Microsoft Windows platform.
4. OpenMP is designed for distributed memory parallel systems and guarantees efficient use of memory.

Question number 2:

Which of these parallel programming errors is impossible in the given OpenMP construct?

1. Data dependency in #pragma omp for.
2. Data conflict in #pragma omp critical.
3. Data race in #pragma omp parallel.
4. Deadlock in #pragma omp parallel.