THE ARAB AMERICAN UNIVERSITY

FACULTY OF ENGINEERING

Parallel and Distributed Computing

Parallel and Distributed Computing PROJECT I

ID:	**202111358**
NAME:	**Hala Jabareen**

Introduction:

Sorting algorithms are fundamental in computer science, with applications across various domains such as databases, graphics, and scientific computing. One of these algorithms, Bitonic Sort is best used in parallel processing because its pattern is very straightforward and regular.

I decided to use Bitonic Sort and the POSIX Threads (Pthreads) library in C++ to parallelize our algorithm in this project. The algorithm is chosen for its ability to sort bitonic sequences using recursion and for letting each subarray be sorted before all the subarrays are merged. Due to this, it is highly parallelizable and serves as an excellent example of using multithreading.

The purpose of this project is to investigate the results of both the sequential and parallel versions of the algorithm and to assess the degree of speedup using diverse thread numbers and input sizes. We intend to see how Bitonic Sort can be parallelized and the impact of multithreading when it is actually used.

Important Note:

To ensure that Bitonic Sort works properly, the array needs to have a size power of 2 (such as 8, 16, 32...). The algorithm needs this condition to be true by design.
Although the number of threads can change, for best results and smooth merging, the recommended number should be chosen. It ought to cut the full array in half (e.g., 16 elements and 4 threads → 4 disjoint sections, each of 4). It's best to select thread counts that are equal to 2, 4 or 8, to match the way bitonic merging is performed.

Sequential Implementation:

Code Explanation:

```
void bitonic_sort_seq(int low, int cnt, bool dir)
{
    if (cnt > 1)
    {
        int k = cnt / 2;
        bitonic_sort_seq(low, k, true);
        bitonic_sort_seq(low + k, k, false);
        bitonic_merge(low, cnt, dir);
    }
}
```

low: starting index of the subarray
cnt: number of element to sort
dir: sorting direction (ascending = true , descending = false)
bitonic_sort_seq function works recursively. It splits the array into two subarrays. Sorts the first half in ascending order and the other in descending.
Then , it calls **bitonic_merege** function to combine them into sorted sequence.

```
void bitonic_merge(int low, int cnt, bool dir)
{
    if (cnt > 1)
    {
        int k = cnt / 2;
        for (int i = low; i < low + k; i++)
        {
            if ((dir && arr[i] > arr[i + k]) || (!dir && arr[i] < arr[i + k]))
            {
                swap(arr[i], arr[i + k]);
            }
        }
        bitonic_merge(low, k, dir);
        bitonic_merge(low + k, k, dir);
    }
}
```

Bitonic_merege function compares pairs of elements and swaps them based on the direction, then applies the same logic recursively to the two halves.

Time Measurement:

getimeofday() function used to measure the time.

```
double get_time()
{
    timeval tv;
    gettimeofday(&tv, nullptr);
    return tv.tv_sec + tv.tv_usec * 1e-6;
}
```

Time recorded at start and end of sorting function.
Then we calculate the duration.

```
double start_seq = get_time();
bitonic_sort_seq(0, size_, true);
double end_seq = get_time();
double duration_seq = end_seq - start_seq;
cout << "Sequential time: " << duration_seq << " s" << endl;
```

Parallelization Strategy:

How work is divided among threads:

Each thread is assigned a chunk of the array to sort using **bitonic_sort_seq( )** function.
The sorted parts are merged hierarchically using bitonic_**merege()** function.
The array is divided into equal chunks on threads.

```
int chunk = size_ / num_threads;
```

```
for (int i = 0; i < num_threads; i++)
{
    bool dir = ((i % 2) == 0);
    data[i] = {i * chunk, chunk, dir};
    pthread_create(&threads[i], NULL, bitonic_parallel_sort, &data[i]);
}
```

Pthread Functions & Structs Used:

ThreadData structure defined to pass multiple arguments to each thread

```
struct ThreadData
{
    int low;
    int count;
    bool dir;
};
```

low: start index
count: number of elements in chunk
dir: sorting direction

Each thread calls this structure (on creating) to sort its own part of the array.

```
void *bitonic_parallel_sort(void *arg)
{
    ThreadData *data = (ThreadData *)arg;
    bitonic_sort_seq(data->low, data->count, data->dir);
    return NULL;
}
```

```
for (int i = 0; i < num_threads; i++)
{
    bool dir = ((i % 2) == 0);
    data[i] = {i * chunk, chunk, dir};
    pthread_create(&threads[i], NULL, bitonic_parallel_sort, &data[i]);
}

for (int i = 0; i < num_threads; i++)
{
    pthread_join(threads[i], NULL);
}

for (int size = 2 * chunk; size <= size_; size *= 2)
{
    for (int i = 0; i < size_; i += size)
    {
        bool dir = ((i / size) % 2 == 0);
        bitonic_merge(i, size, dir);
    }
}
```

1. First, we divided the array into equal chunks.
2. Creates threads, each one calls bitonic_ sort_seq( )on its chunk.
3. Wait for all threads to finish (pthread_join())
4. Apply hierarchical bitonic_merge() step by step to the whole array.

Experiments:

Hardware Specifications:

- **CPU**: Intel(R) Core(TM) i5-7300U @ 2.60GHz
- **Cores**: 2 physical cores, 4 logical processors (hyper-threaded)
- **RAM**: 16 GB
- **Operating System**: Windows 11

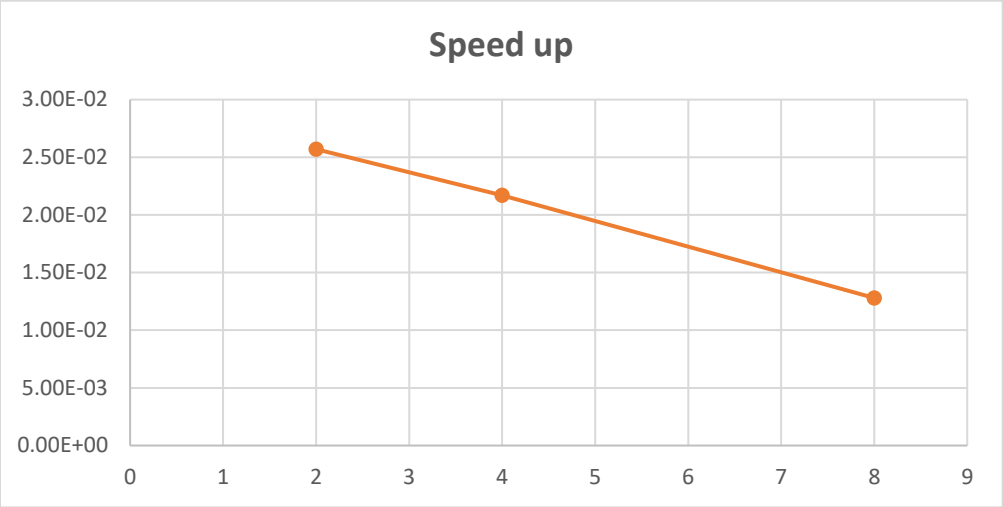
Input sizes:

Tested array sizes = 32, 512, 1024, 4096, 8192, 16384, 32768.
Threads count = 2, 4, 8.

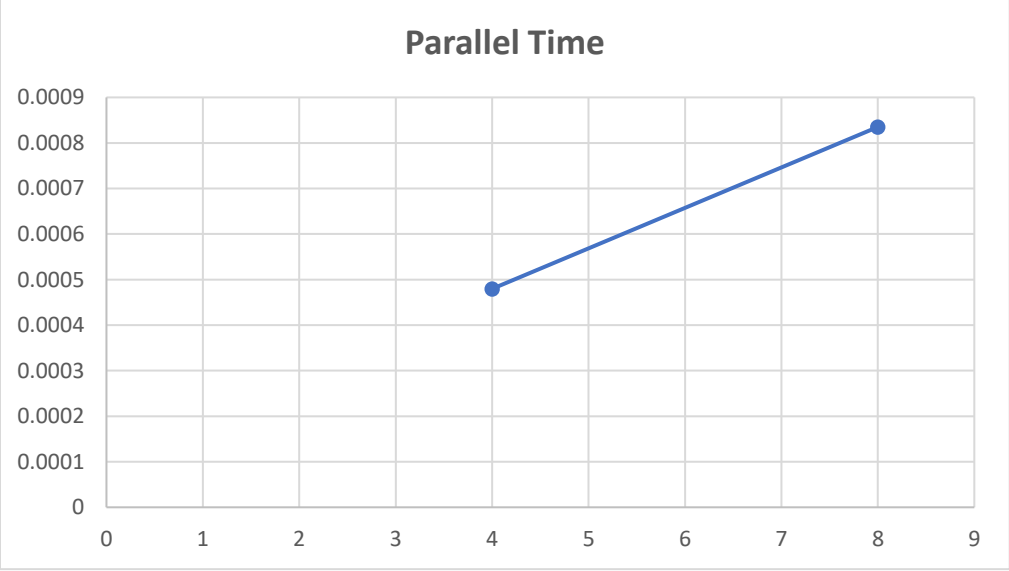
Results:

Array size =32.

Num. of threads	Seq. time	Par. time	Speed. up
2	1.00E-05	0.000389576	2.57E-02
4	1.04E-05	0.000479678	2.17E-02
8	1.06E-05	0.000835102	1.28E-02



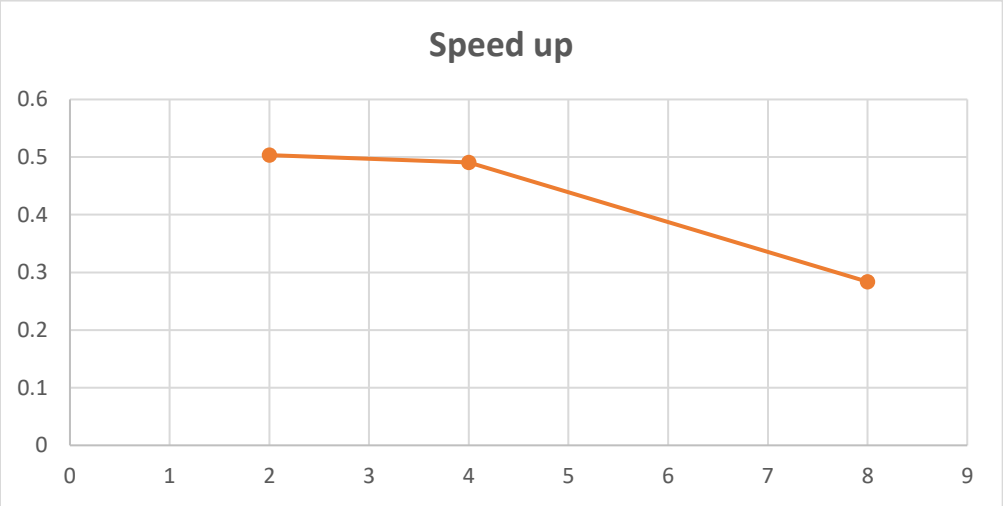
Speed up vs Num. of threads



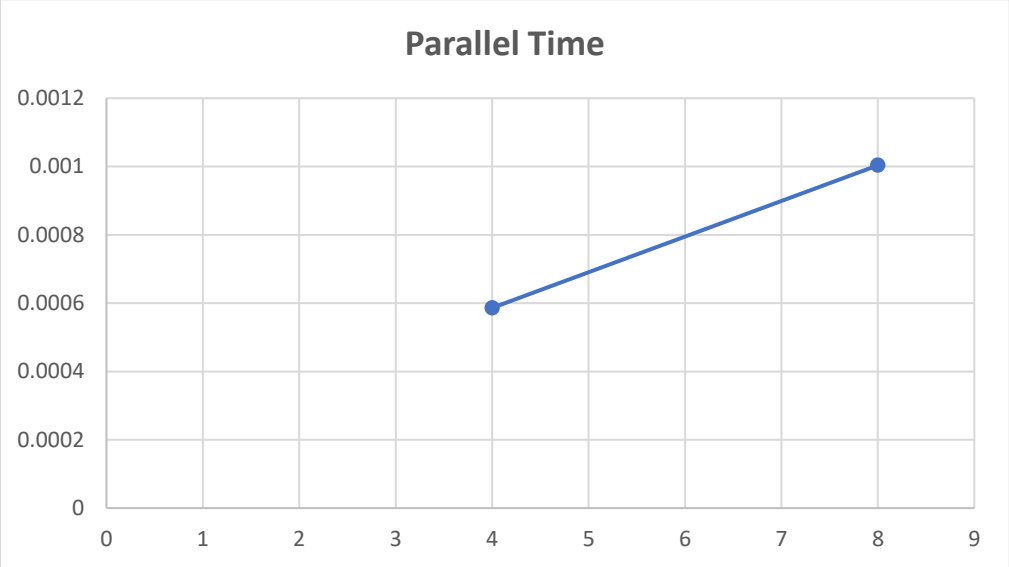
Parallel Time vs Num. of threads

Array size =512.

Num. of threads	Seq. time	Par. time	Speed. up
2	0.000265281	0.000526985	0.50339352
4	0.000287692	0.000586112	0.49084732
8	0.000284751	0.001003506	0.28375658



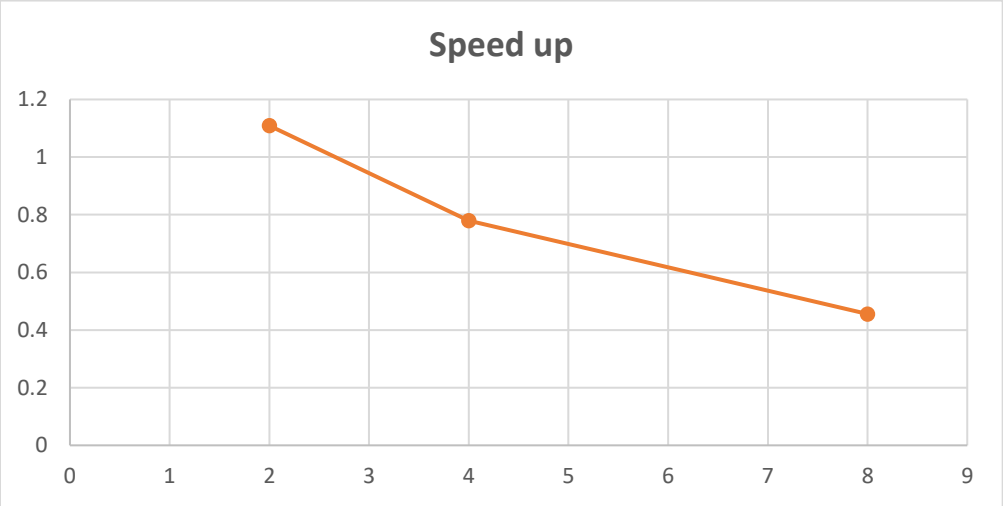
Speed up vs Num. of threads



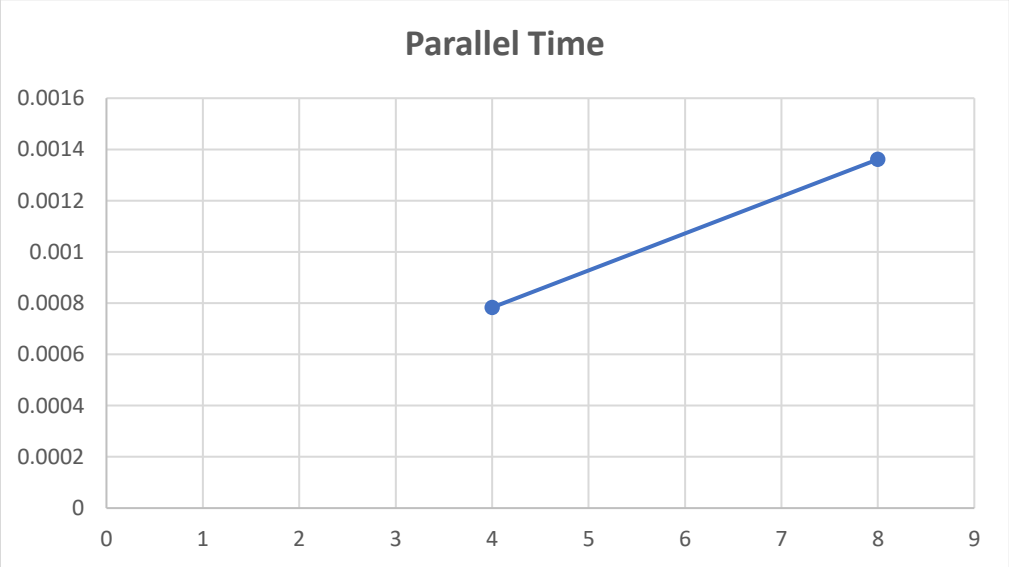
Parallel Time vs Num. of threads

Array size =1024.

Num. of threads	Seq. time	Par. time	Speed. up
2	0.003028949	0.002731085	1.1090646
4	0.000610272	0.000783046	0.77935694
8	0.000620286	0.00136129	0.45566019



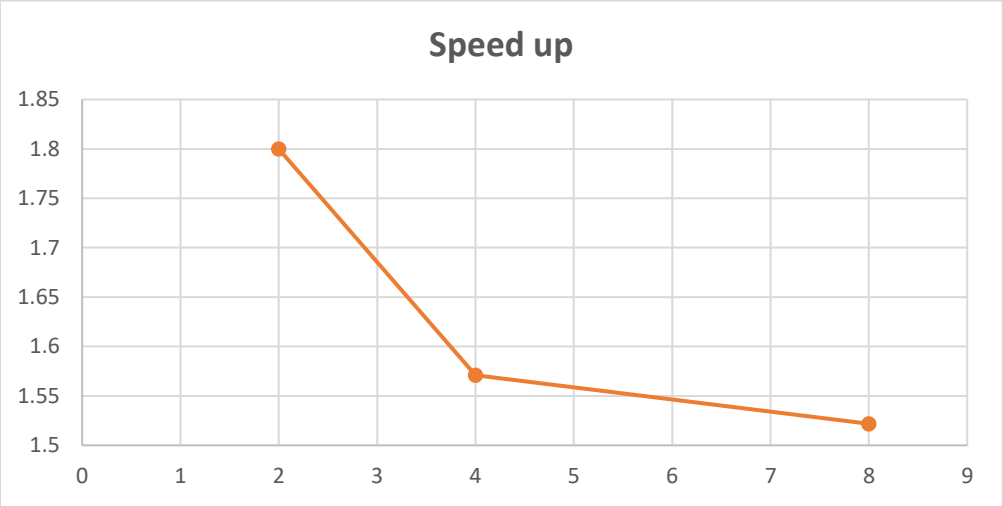
Speed up vs Num. of threads



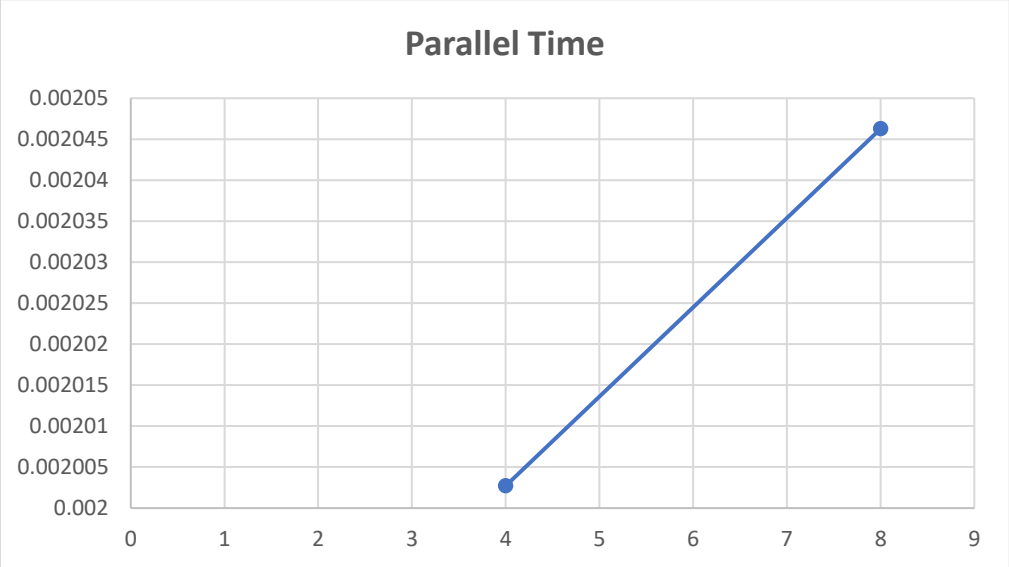
Parallel Time vs Num. of threads

Array size =4096.

Num. of threads	Seq. time	Par. time	Speed. up
2	0.003475347	0.001930953	1.79980873
4	0.00314633	0.002002713	1.57103363
8	0.00311406	0.002046267	1.52182511



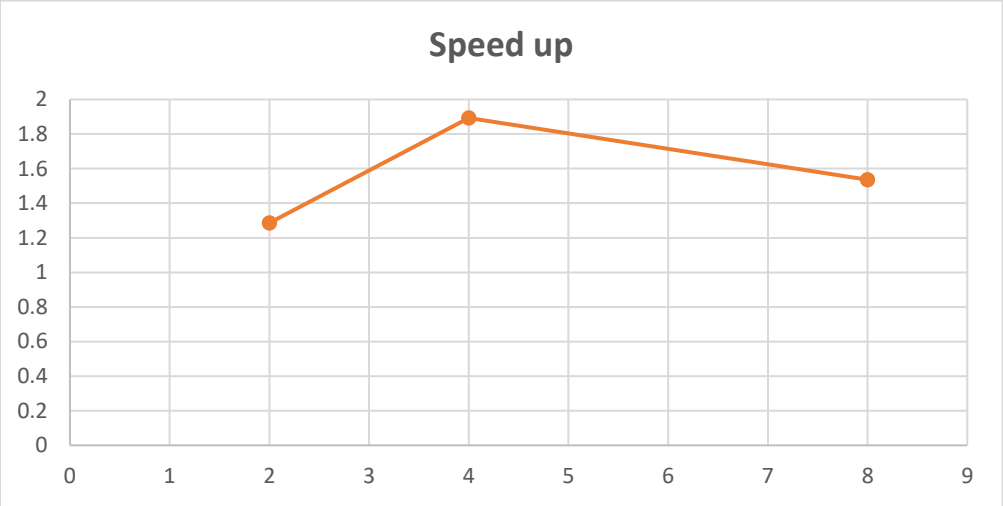
Speed up vs Num. of threads



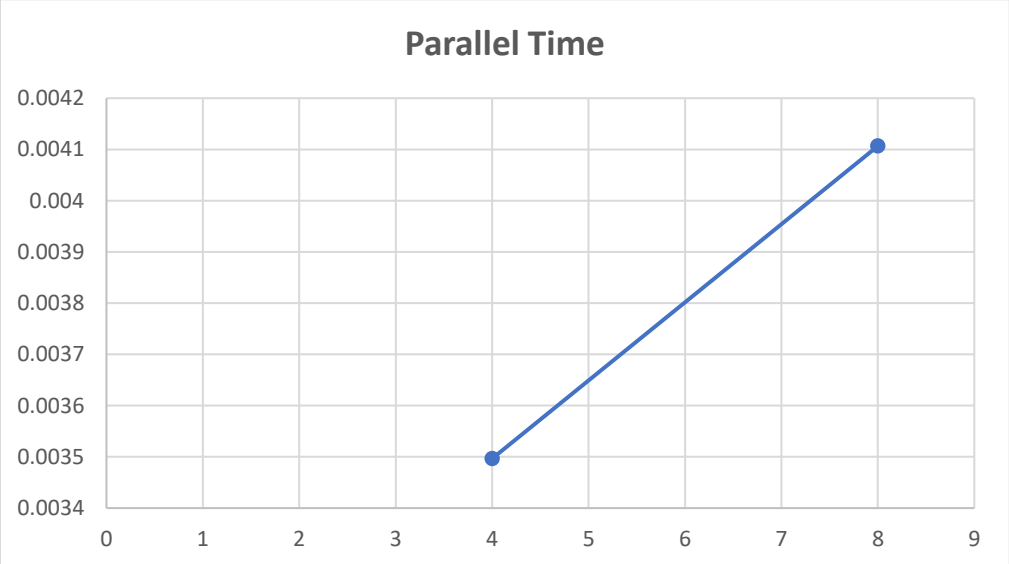
Parallel Time vs Num. of threads

Array size =8192.

Num. of threads	Seq. time	Par. time	Speed. up
2	0.005972623	0.004644713	1.28589708
4	0.006618023	0.003496647	1.89267717
8	0.006306647	0.00410668	1.53570443



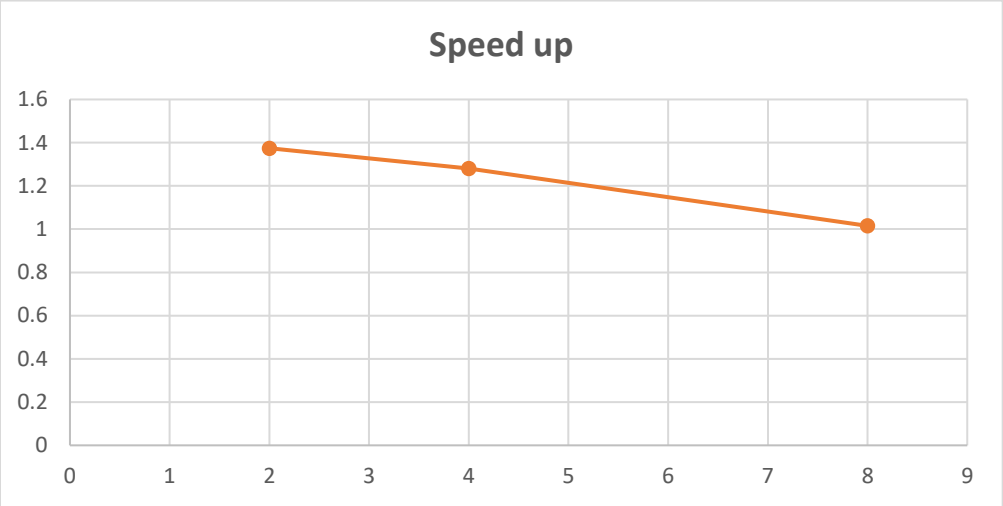
Speed up vs Num. of threads



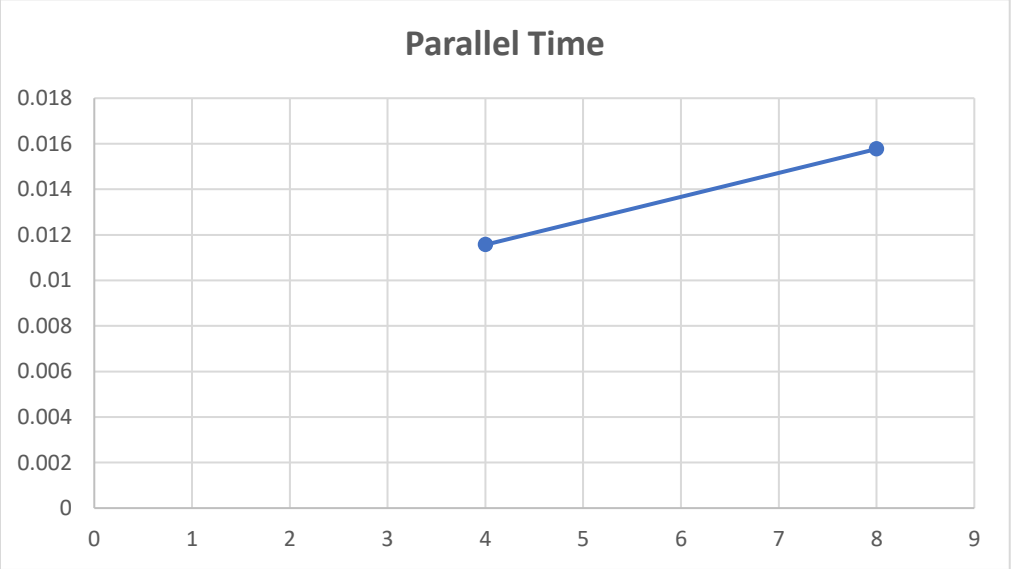
Parallel Time vs Num. of threads

Array size =16384.

Num. of threads	Seq. time	Par. time	Speed. up
2	0.012404633	0.009025667	1.37437308
4	0.014811367	0.011566343	1.28055741
8	0.0160094	0.015769	1.0152451



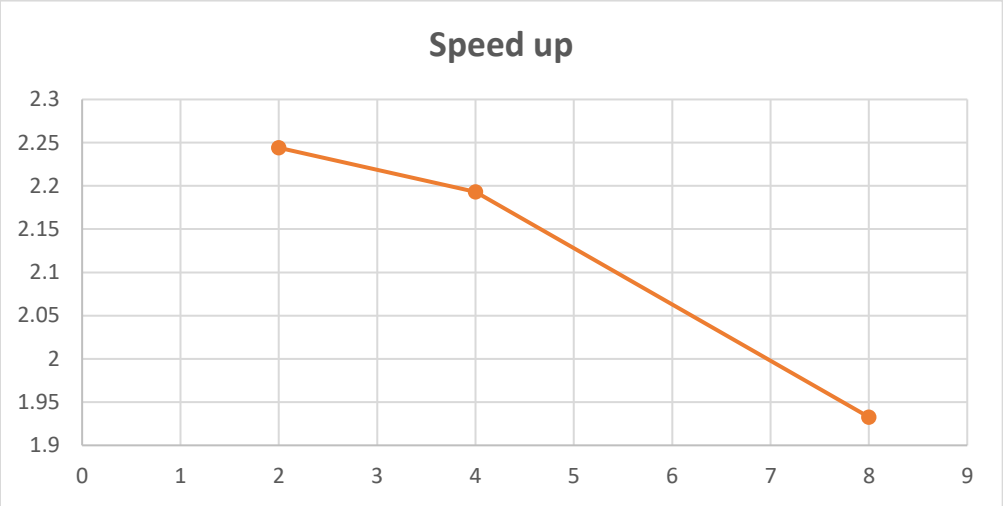
Speed up vs Num. of threads



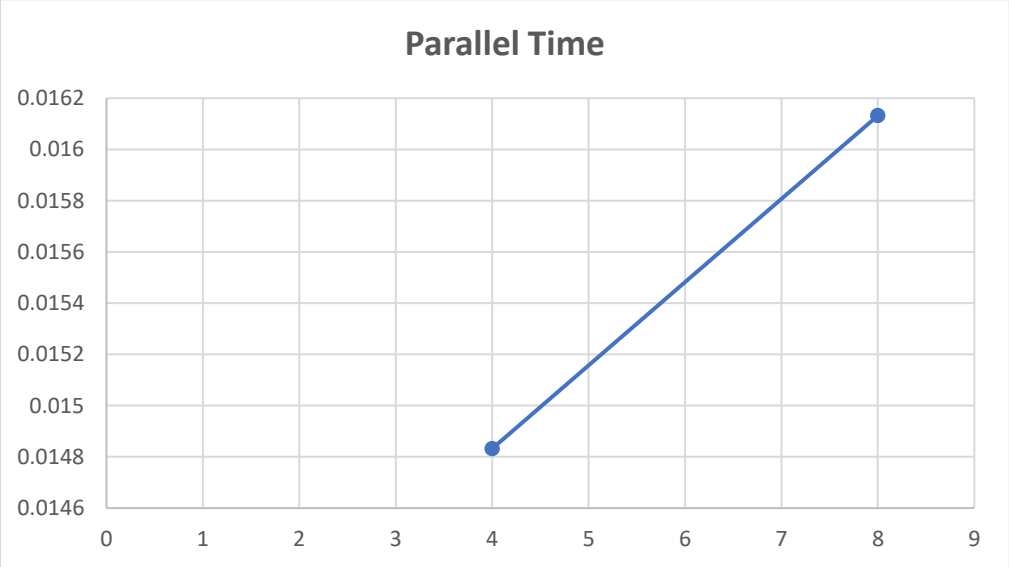
Parallel Time vs Num. of threads

Array size =32768.

Num. of threads	Seq. time	Par. time	Speed. up
2	0.031805033	0.014172667	2.24411073
4	0.032530067	0.0148321	2.19322056
8	0.0311786	0.016132033	1.93271359



Speed up vs Num. of threads



Parallel Time vs Num. of threads

Discussion:

Why speedup is sublinear?

The increase in threads in our Bitonic Sort with Pthreads version did not always reduce the time taken by half. People behaving this way is perfectly understandable because of several main reasons:

Costs Involved in Starting Threads and Synchronizing Them:
Every time pthread_create and pthread_join are called, there is some extra processing cost. If you have a short input or are able to work with many threads, most of your time may be spent managing the threads, rather than the tasks themselves.

Load Imbalance:
It is possible that when threads are mapped according to the way you choose (direction), some threads will complete early and wait for others to finish.

Limos are limited by their CPU cores:
The system includes 2 cores with 4 logical processors which means running more than 4 threads leads to switching between tasks instead of parallel execution. As a result of this behavior, adding more threads does not improve performance much.

Amdahl's Law:
The largest improvement in speedup that Amdahl's Law allows is held back by the part of the algorithm that needs to be done in order. Although part of the merging process in Bitonic Sort can be done at the same time, the need for specific order in many comparisons and merges lowers how much of the sorting can be done in parallel.

Conclusion:

We designed this project by changing the Bitonic Sort algorithm into a parallel version using Pthreads. We figured out how to handle dividing the work among threads and how to use pthread_create and pthread_join.

With large arrays, the parallel version ran more quickly than the sequential one. However, the program ran faster, but not perfectly because of the thread overhead and the small number of CPU cores. Amdahl's Law states the same thing.

Passing data to threads and making sure there were no race conditions were issues, but we resolved them piece by piece.

We used this project to see how multithreading functions and how it can boost the speed of our programs

Resources:

- ChatGPT, we corrected some errors in the parallel code, learned about multithreading and organized and wrote parts of this report. All the code and analysis were studied and evaluated by the student.
- GeeksforGeeks, to understand the sequential implementation for bitonic sort.
- Bitonic Sort - Sorting Algorithms Mini-Series, a video which explain bitonic sorting algorithm.