Rusty Sort

Sorting Algorithms implemented in Rust, in both serial & parallel implementations.

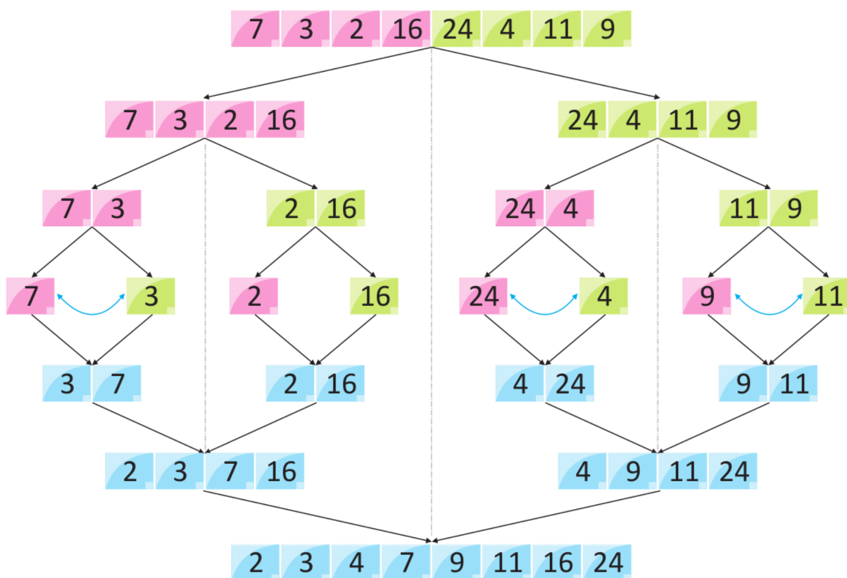
- Mohamed Metwalli Noureldin - 18011587
- AbdelRahman Adel AbdelFattah - 17012296

Problem Statement

Rust is a multi-paradigm, high-level, general-purpose programming language that emphasizes performance, type safety, and concurrency. It enforces memory safety—ensuring that all references point to valid memory—without requiring the use of a garbage collector or reference counting present in other memory-safe languages. To simultaneously enforce memory safety and prevent concurrent data races, its "borrow checker" tracks the object lifetime of all references in a program during compilation. Rust is popularized for systems programming but also has high-level features including some functional programming constructs.
With Rust native support for concurrency and parallelism, and performance, it was decided to implement a few algorithms to show and analyze the performance of Rust in a serial and a parallel Implementation.

Design of Algorithm

Merge Sort



Serial

Steps

1. You split the array into two.
2. You keep splitting until each half has a one element
3. You merge each two inorder.
4. Keep merging until the array is sorted.

Pseudocode

```
fun mergeSort(array, left, right){
  if left >= right{
    return;
  }
  let mid = (left + right)/2;
  mergeSort(array, left, mid);
  mergeSort(array, mid+1, right);
  merge(array, left, mid, right);
}
```

```
fun merge(array, start, mid, end){
  let len1 = mid-start+1;
  let len2 = end - mid;
  let left = array[start:mid+1];
  let right = array[mid:end];
  let i = 0, j = 0, k = start;
  while i < len1 && j < len2{
    if left[i] <= right[j]{
      array[k] = left[i];
      i++;
    }else{
      array[k] = right[j];
      j++;
    }
    k++;
  }
  while i < len1 {
    array[k] = left[i];
    i++;
    k++;
  }
  while j < len2 {
    array[k] = right[j];
    j++;
    k++;
  }
}
```

Parallel

Steps

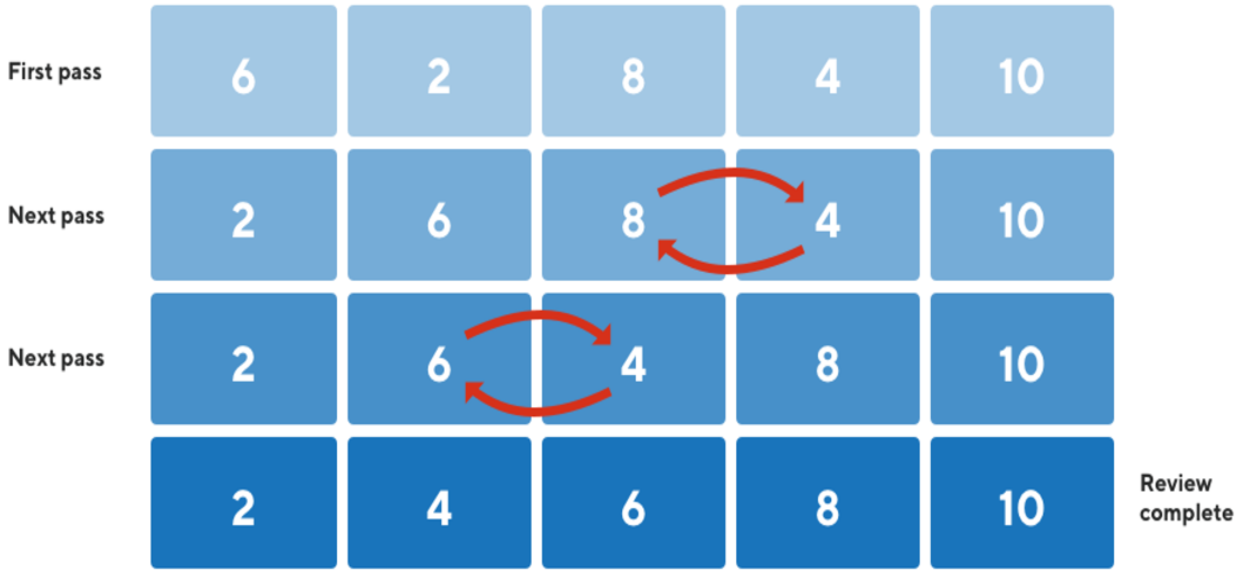
1. Calculate “chunk size” = array.size / number of processors.
2. Divide the array into chunks.
3. For each chunk, sort the chunk in parallel.
4. For each two chunks merge them together in parallel.
5. Repeat Step 4 until sorted and only have one chunk.

Pseudocode

```
fun parallelMergeSort(array, chunkSize, threads){
  for i = 0 to threads do in parallel{
    let chunk = array[i*chunksize: (i+1)*chunksize];
    chunk.sort();
  }
  parallelMerge(array, chunkSize, threads)
}
```

```
fun parallelMerge(array, chunkSize, threads){
  for i = 0 to threads do in parallel{
    let left = array[i*chunkSize:i*(chunkSize+1)];
    let right = array[i*(chunkSize+1):i*(chunkSize+1)];
    merge(right, left);
  }
  chunk *= 2;
  parallelMerge(array, chunkSize, threads);
}
```

Bubble Sort



Serial

Steps

1. Compare the first two elements in the list.
2. If the first element is greater than the second element, swap them.
3. Move to the next pair of elements and repeat step 2, continuing until end of the list.
4. At this point, the largest element should be at the end of the list.
5. Repeat steps 1-4 for all elements except the last one.
6. After completing all iterations, the list will be sorted in ascending order.

Pseudocode

```
fun bubbleSort(array) {
    let n = arr.len();
    for i = 0 to n-1 {
        let swapped = false;
        for j = 0 to n-i-1 {
            if arr[j] > arr[j+1] {
                arr.swap(j, j+1);
                swapped = true;
            }
        }
        if !swapped {
            break;
        }
    }
}
```

Parallel

Steps

1. Initialize a variable “n” with the length of the input array and create an “AtomicBool” flag called “swapped”.

2. Start a loop that continues until swapped is false.

3. Inside the loop :

    - Set “swapped” to false and calculate the “chunk_size”.
    - Split the input array into chunks of size “chunk_size” and process each chunk in parallel using Rayon's “par_chunks_mut()” method.
    - For each chunk, perform a standard bubble sort by iterating through the elements and swapping adjacent elements if they are not in the correct order.
    - If any swaps are made during the bubble sort step, set a local swapped flag to true.
    - After processing a chunk, if its local swapped flag is true, set the global swapped flag to true as well.
    - Call a “parallel_merge()” function to merge the sorted chunks back together into one fully sorted array.

4. The loop repeats until no swaps are made during an iteration, at which point the sorted array is returned.

Pseudocode

```
fun parallelBubbleSort(array, threads_num){
    n = arr.len();
    swapped = true;
    while swapped {
        swapped = false;
        chunk_size = ceil(n/threads_num);
        chunks = split_array_into_chunks(arr,
chunk_size);
        parallel_for each chunk in chunks {
            local_swapped = false;
            for j = 0 to length(chunk)-2 {
                if chunk[j] > chunk[j+1] {
                    swap(chunk[j], chunk[j+1]);
                    local_swapped = true
                }
            }
            if local_swapped {
                swapped = true;
            }
        }
        sorted_arr = parallel_merge(chunks);
    }
}
```

Results

Merge Sort

An Unsigned 64-bit array with a size of n, results are in seconds

Size of Array	std::sort	serial merge sort	parallel merge sort (2 threads)	parallel merge sort (4 threads)	parallel merge sort (8 threads)	parallel merge sort (16 threads)	parallel merge sort (32 threads)
250	0.0000258	0.000091	0.000461	0.000571	0.000656	0.00000108	0.0021
500	0.0000425	0.000247	0.00039	0.000378	0.000577	0.00000161	0.00213
1,000	0.000112	0.00034	0.000453	0.000499	0.000781	0.000936	0.00164
2,500	0.000219	0.000675	0.000473	0.00053	0.0011	0.00128	0.00191
5,000	0.000307	0.00107	0.000526	0.000667	0.000943	0.00123	0.00328
10,000	0.001	0.00336	0.00114	0.00086	0.00104	0.00131	0.00211
25,000	0.00274	0.00799	0.0022	0.00168	0.00146	0.00139	0.00205
50,000	0.00566	0.0145	0.00336	0.00243	0.00206	0.00215	0.00278
100,000	0.0122	0.0248	0.00524	0.00367	0.00293	0.00347	0.00326
250,000	0.0305	0.0413	0.00891	0.00586	0.00604	0.00565	0.00528
500,000	0.049	0.0624	0.0169	0.0151	0.0112	0.0115	0.011
1,000,000	0.0754	0.118	0.0384	0.0247	0.0255	0.0228	0.028
2,500,000	0.153	0.322	0.108	0.0652	0.0615	0.0632	0.0629
5,000,000	0.313	0.658	0.195	0.128	0.127	0.148	0.129
10,000,000	0.64	1.41	0.43	0.27	0.261	0.287	0.331
25,000,000	1.78	3.61	1.14	0.772	0.654	0.704	0.764
50,000,000	3.71	7.79	2.36	2.3	1.93	1.47	1.53
100,000,000	8.52	18.3	7.5	9.09	6	6.25	9.84
250,000,000	30.3	58.1	79.4	80	109	106	133
500,000,000	152	196	255	306	295	356	428

Bubble Sort

An Unsigned 64-bit array with a size of n, results are in seconds

Size of Array	std::sort	serial bubble sort	parallel bubble sort (2 threads)	parallel bubble sort (4 threads)	parallel bubble sort (8 threads)	parallel bubble sort (16 threads)	parallel bubble sort (32 threads)
250	.0001088	.0006593	.1260038	.0899496	.091289	.0546099	.0634431
500	.0002027	.0037859	.1552458	.1754761	.1735109	.1305979	.1156926
1,000	.0004171	.0149691	.4670216	.5748259	.6074637	.4417281	.2504121
2,500	.0010463	.0602642	1.2591349	1.5920907	1.3070621	.9132278	.952541
5,000	.0012014	.2166915	3.7793507	4.3713507	4.2571988	2.4507192	2.8392523
10,000	.002542	.8476321	9.5321968	9.9590651	8.4030982	8.4893748	5.6166441
25,000	.0070504	5.7971178	40.769419	47.659442	40.601143	27.2573557	22.443515
50,000	.0148609	22.814875	143.54574	168.21296	130.11420	108.199399	52.674673
100,000	.031086	91.858939	621.32166	550.94239	448.14573	347.771392	593.71581

Speed up

Merge Sort

Serial merge sort speedup compared to parallel with different threads.
Speedup = old/new;

Size of Array	parallel merge sort (2 threads)	parallel merge sort (4 threads)	parallel merge sort (8 threads)	parallel merge sort (16 threads)	parallel merge sort (32 threads)
250	0.1974	0.1594	0.1387	84.2593	0.0433
500	0.6333	0.6534	0.4281	153.4161	0.116
1000	0.7506	0.6814	0.4353	0.3632	0.2073
2,500	1.4271	1.2736	0.6136	0.5273	0.3534
5,000	2.0342	1.6042	1.1347	0.8699	0.3262
10,000	2.9474	3.907	3.2308	2.5649	1.5924
25,000	3.6318	4.756	5.4726	5.7482	3.8976
50,000	4.3155	5.9671	7.0388	6.7442	5.2158
100,000	4.7328	6.7575	8.4642	7.147	7.6074
250,000	4.6352	7.0478	6.8377	7.3097	7.822
500,000	3.6923	4.1325	5.5714	5.4261	5.6727
1000,000	3.0729	4.7773	4.6275	5.1754	4.2143
2,500,000	2.9815	4.9387	5.2358	5.0949	5.1192

5,000,000	3.3744	5.1406	5.1811	4.4459	5.1008
10,000,000	3.2791	5.2222	5.4023	4.9129	4.2598
25,000,000	3.1667	4.6762	5.5199	5.1278	4.7251
50,000,000	3.3008	3.387	4.0363	5.2993	5.0915
100,000,000	2.44	2.0132	3.05	2.928	1.8598
250,000,000	0.7317	0.7263	0.533	0.5481	0.4368
500,000,000	0.7686	0.6405	0.6644	0.5506	0.4579

Comments

These runs were from an average of 5 different runs on a x86-64 8-core, 8 GBs RAM, Ubuntu Machine.
Some runs have less speed up than expected. Which means communication is taking more time than it is required to sort the array sequentially.

Bubble Sort

Serial bubble sort speedup compared to parallel with different threads.
Speedup = old/new;

Size of Array	parallel bubble sort (2 threads)	parallel bubble sort (4 threads)	parallel bubble sort (8 threads)	parallel bubble sort (16 threads)	parallel bubble sort (32 threads)
250	0.0052	0.0073	0.0072	0.012	0.0103
500	0.0244	0.0215	0.0218	0.0289	0.0327
1000	0.03205	0.02604	0.02464	0.03388	0.05977
2,500	0.0479	0.0379	0.0461	0.0659	0.0633
5,000	0.0573	0.0495	0.0509	0.0884	0.0763
10,000	0.0889	0.0851	0.1008	0.0998	0.1509
25,000	0.1421	0.1216	0.1428	0.2127	0.2583
50,000	0.1589	0.1356	0.1753	0.2108	0.4331
100,000	0.1478	0.1667	0.2049	0.2641	0.1547

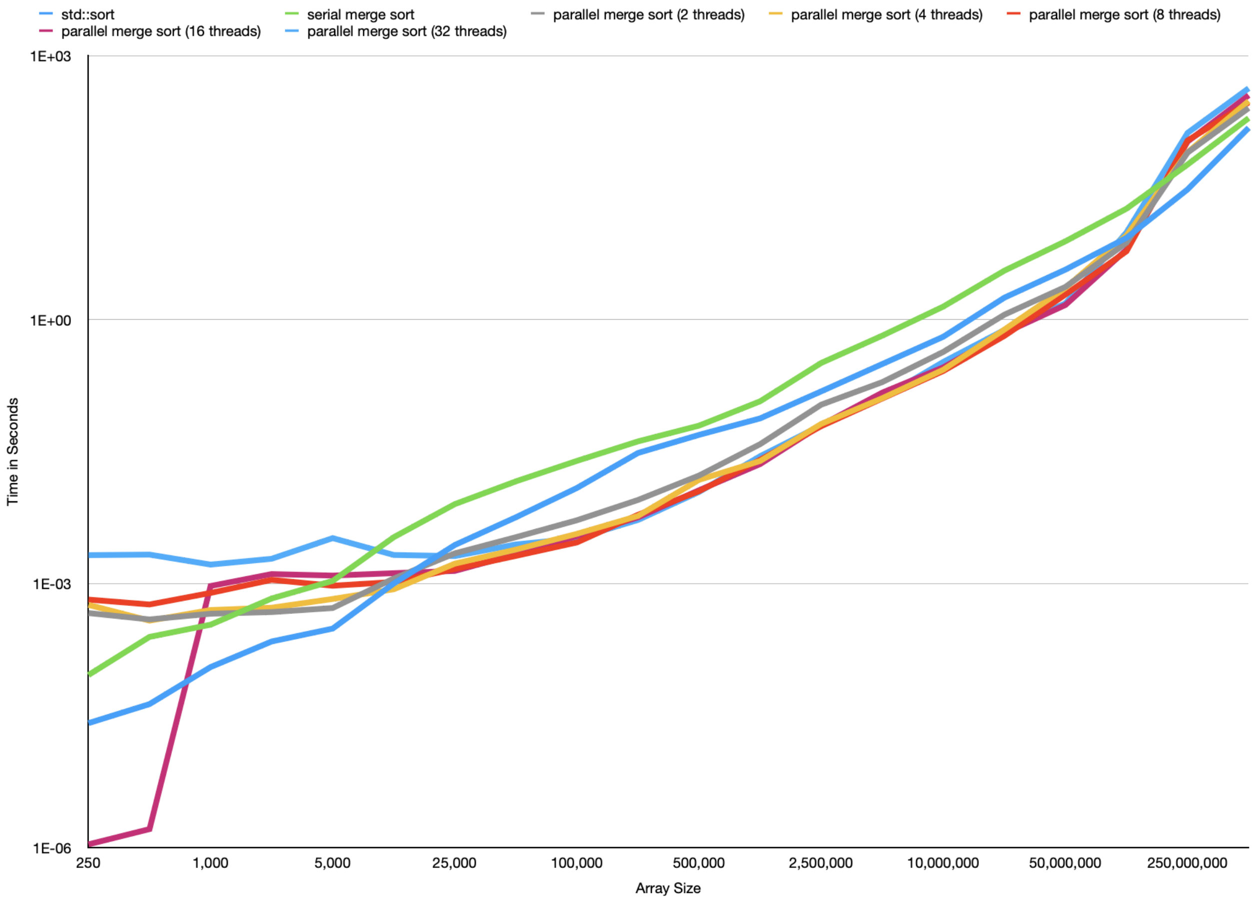
Comments

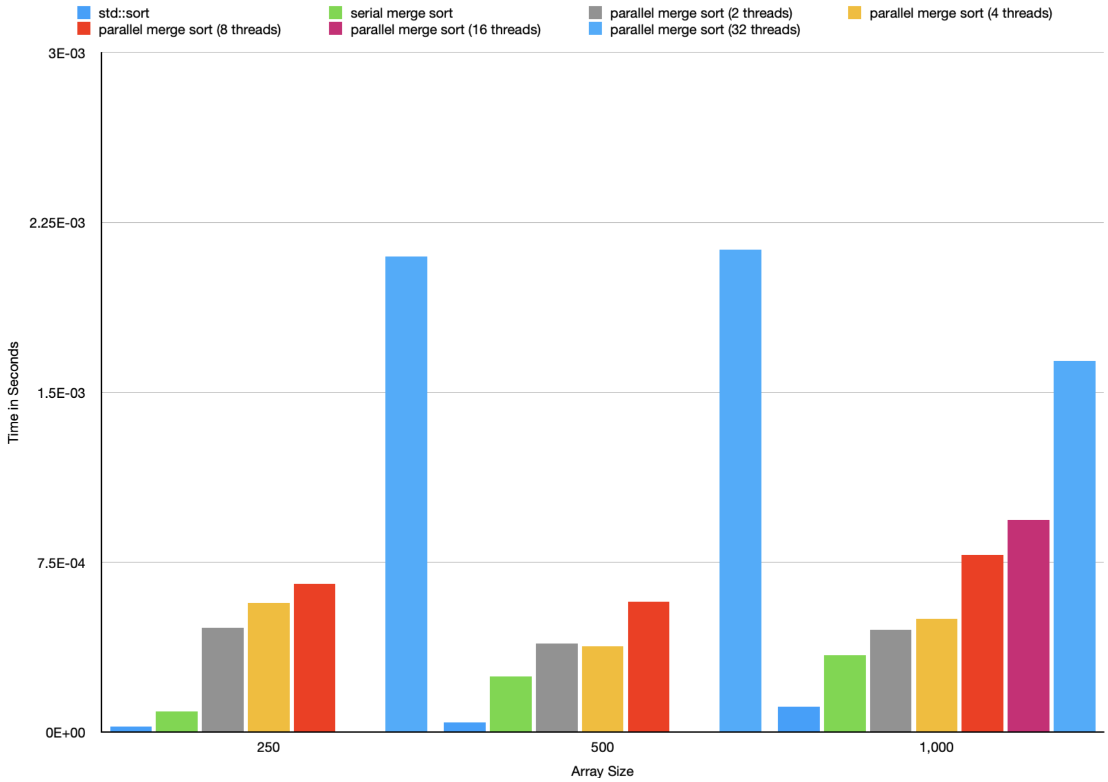
These runs were from an average of 5 different runs on a x86-64 8-core, 8 GBs RAM, Windows Machine.
Some runs have less speed up than expected. Which means communication is taking more time than it is required to sort the array sequentially.

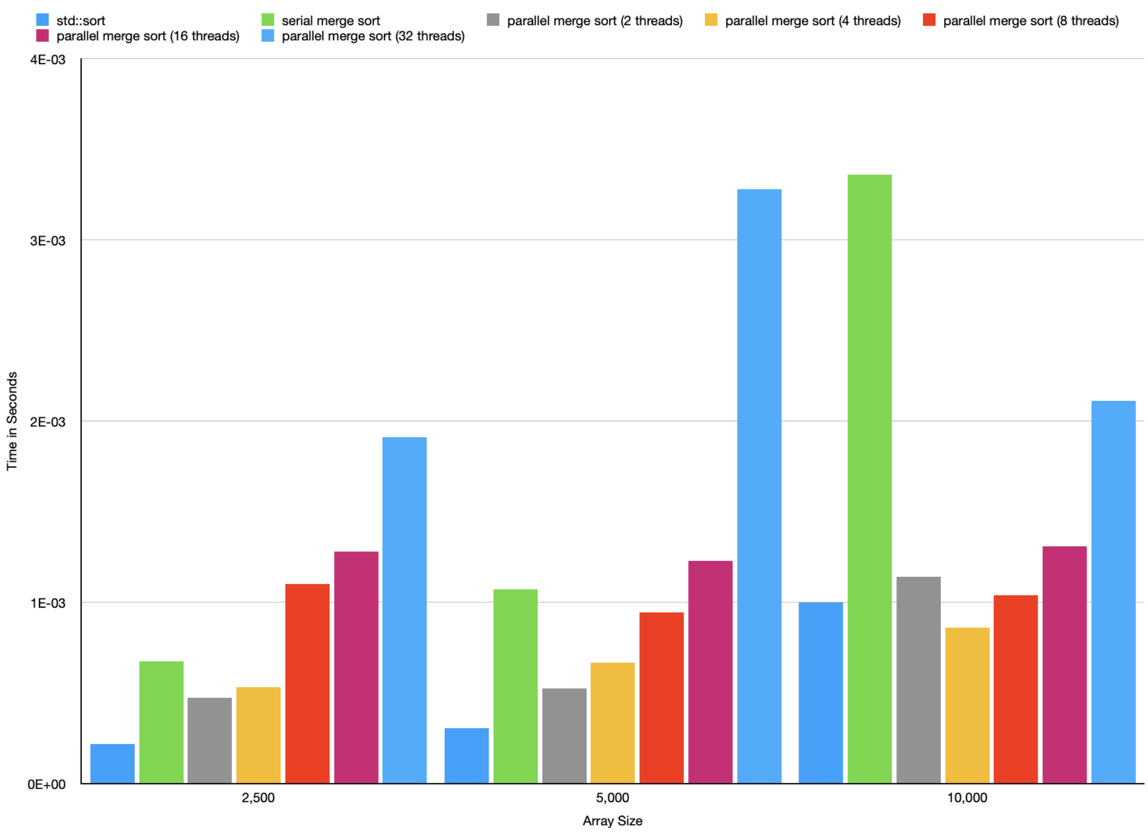
Graphs

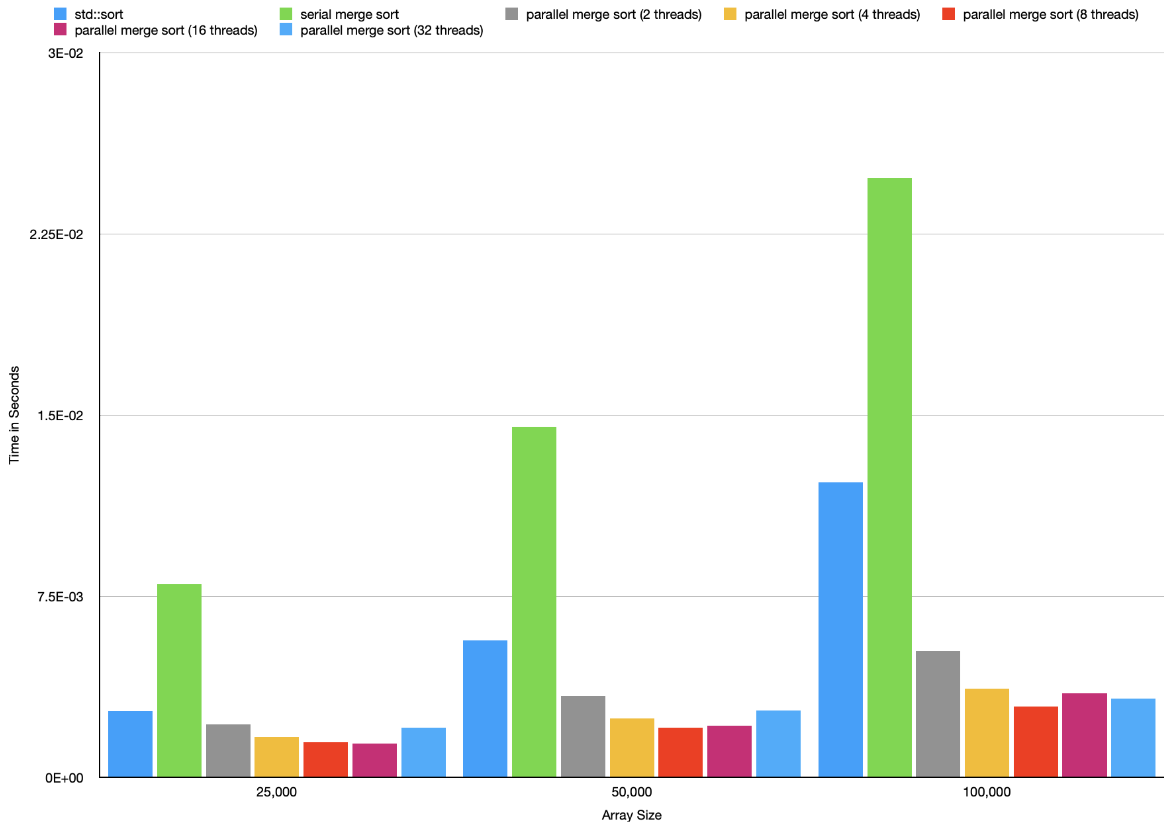
Merge Sort

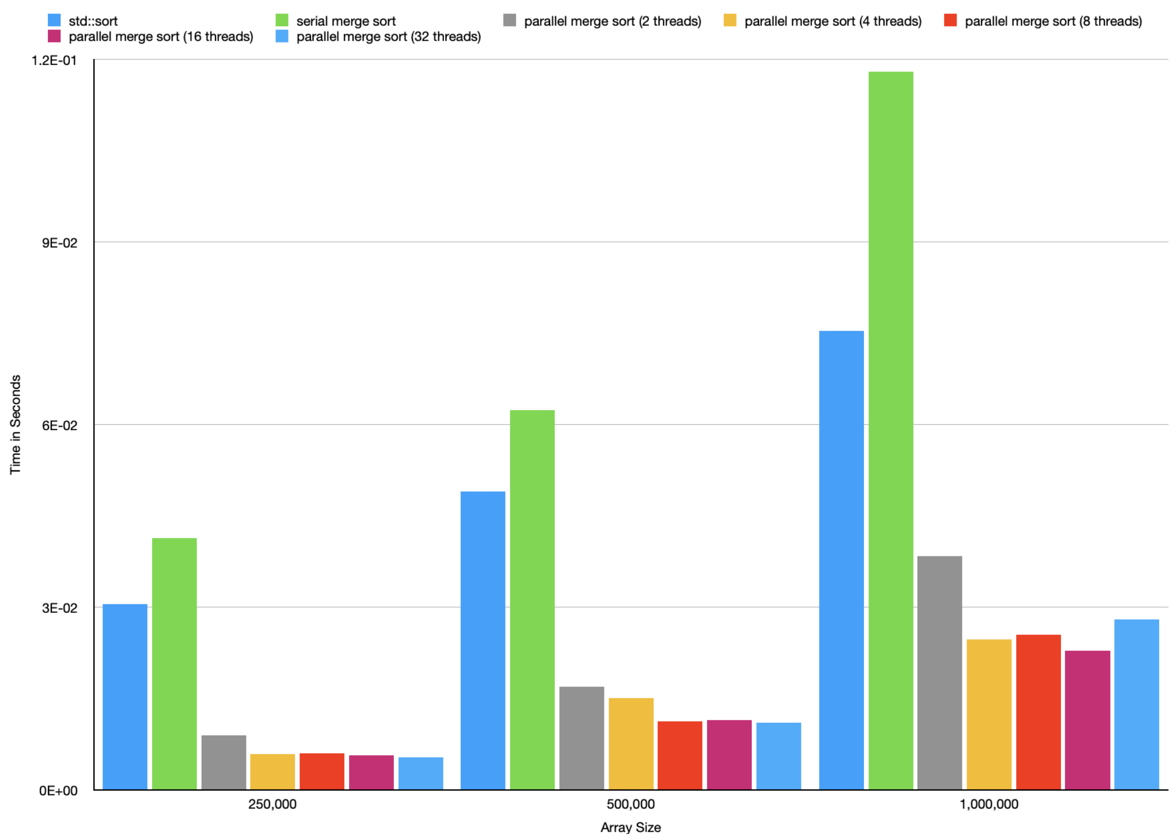
Time To Size Graph

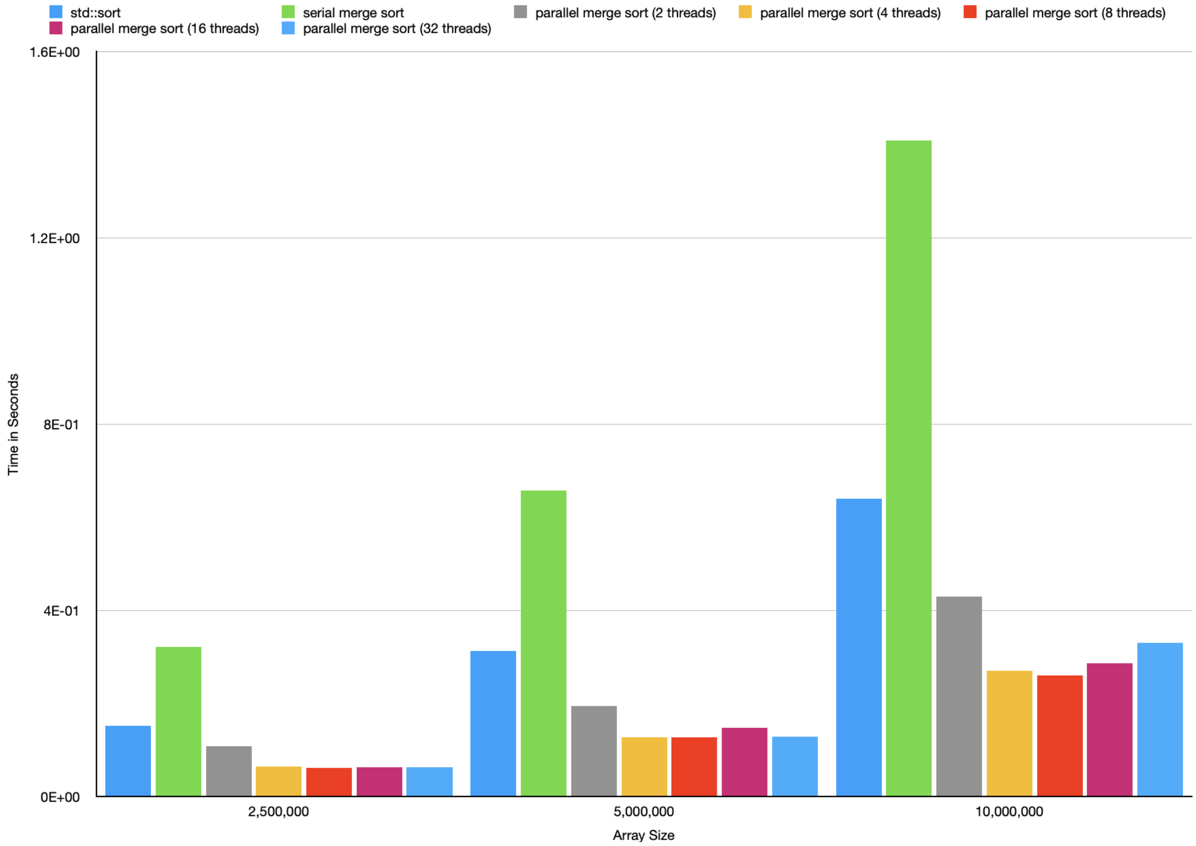



std::sort
serial merge sort
parallel merge sort (2 threads)
parallel merge sort (4 threads)
parallel merge sort (8 threads)
parallel merge sort (16 threads)
parallel merge sort (32 threads)
Time in Seconds
3E-03
2.25E-03
1.5E-03
7.5E-04
0E+00
250
500
1,000
Array Size

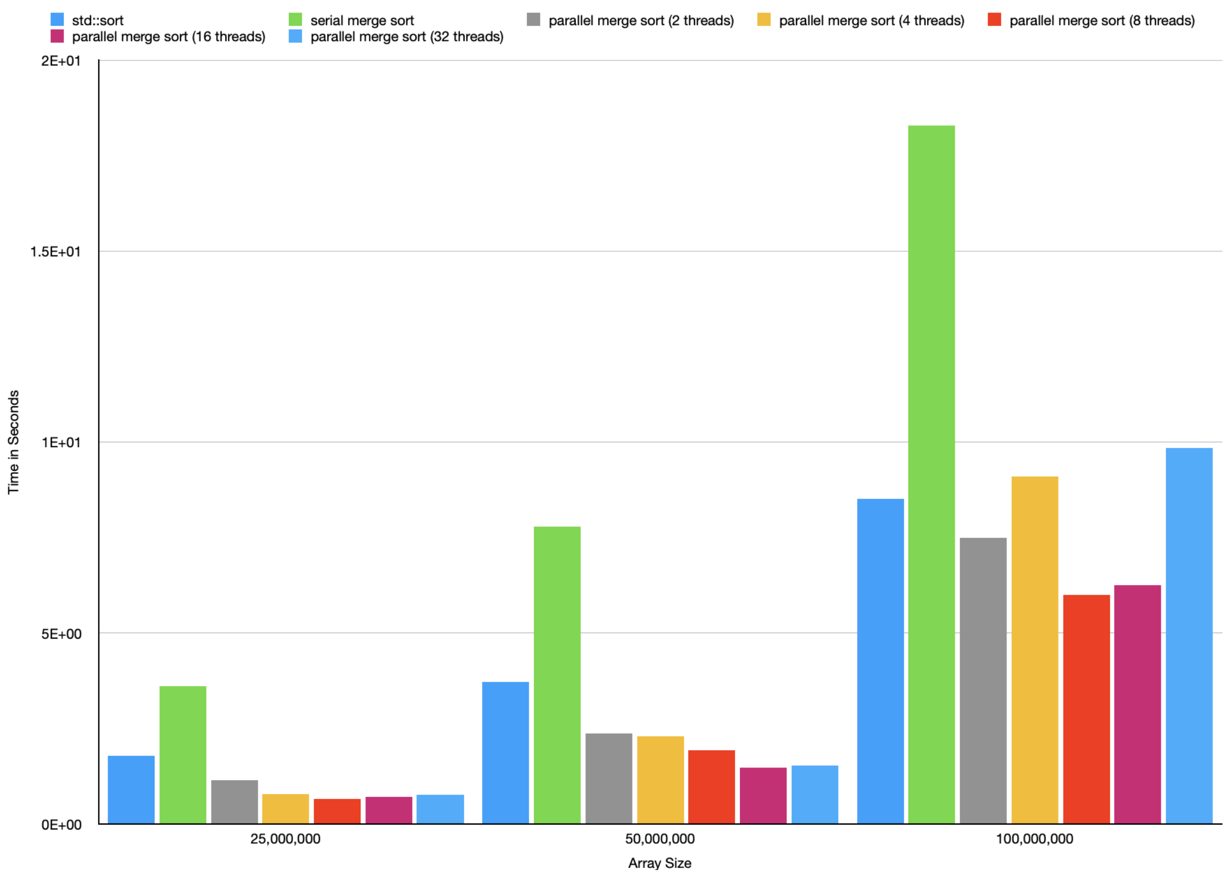

std::sort
serial merge sort
parallel merge sort (2 threads)
parallel merge sort (4 threads)
parallel merge sort (8 threads)
parallel merge sort (16 threads)
parallel merge sort (32 threads)
Time in Seconds
4E-03
3E-03
2E-03
1E-03
0E+00
2,500
5,000
10,000
Array Size


std::sort
serial merge sort
parallel merge sort (2 threads)
parallel merge sort (4 threads)
parallel merge sort (8 threads)
parallel merge sort (16 threads)
parallel merge sort (32 threads)
3E-02
2.25E-02
1.5E-02
7.5E-03
0E+00
Time in Seconds
25,000
50,000
100,000
Array Size

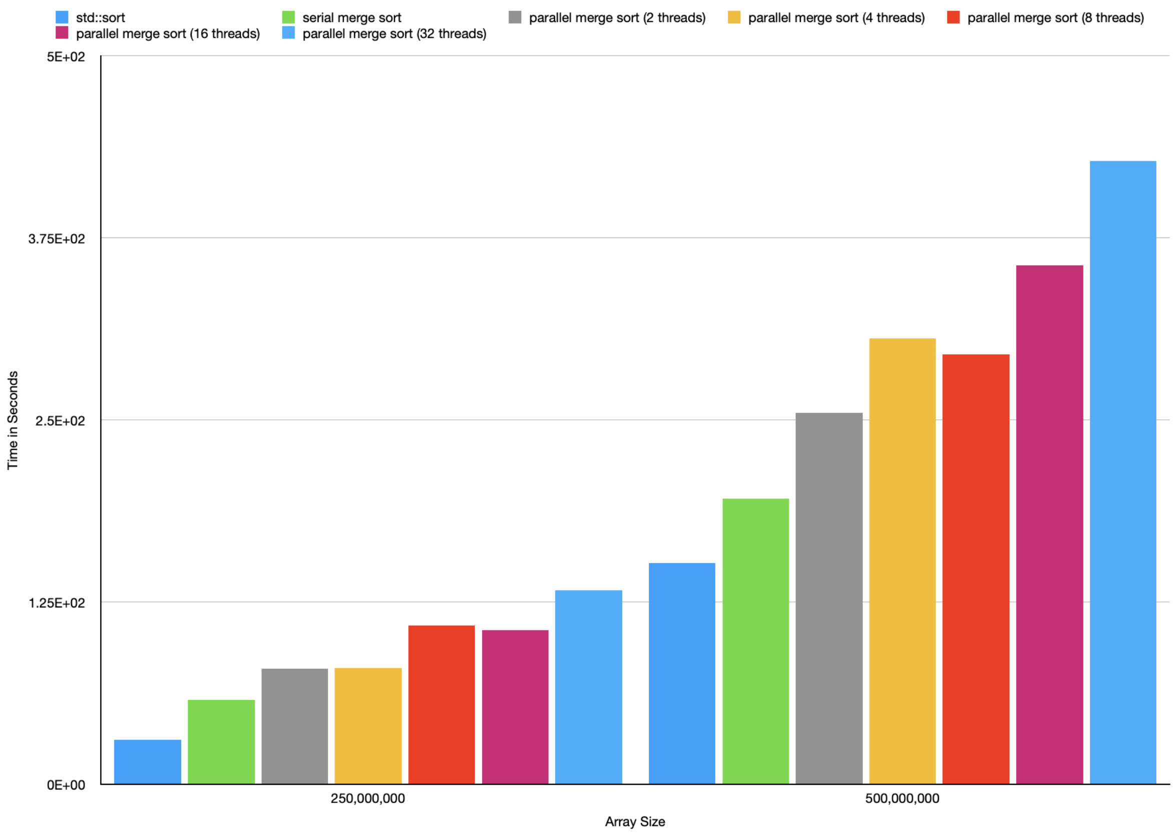

std::sort
serial merge sort
parallel merge sort (2 threads)
parallel merge sort (4 threads)
parallel merge sort (8 threads)
parallel merge sort (16 threads)
parallel merge sort (32 threads)
1.2E-01
9E-02
6E-02
3E-02
0E+00
Time in Seconds
250,000
500,000
1,000,000
Array Size



std::sort
serial merge sort
parallel merge sort (2 threads)
parallel merge sort (4 threads)
parallel merge sort (8 threads)
parallel merge sort (16 threads)
parallel merge sort (32 threads)
1.6E+00
1.2E+00
8E-01
4E-01
0E+00
Time in Seconds
2,500,000
5,000,000
10,000,000
Array Size

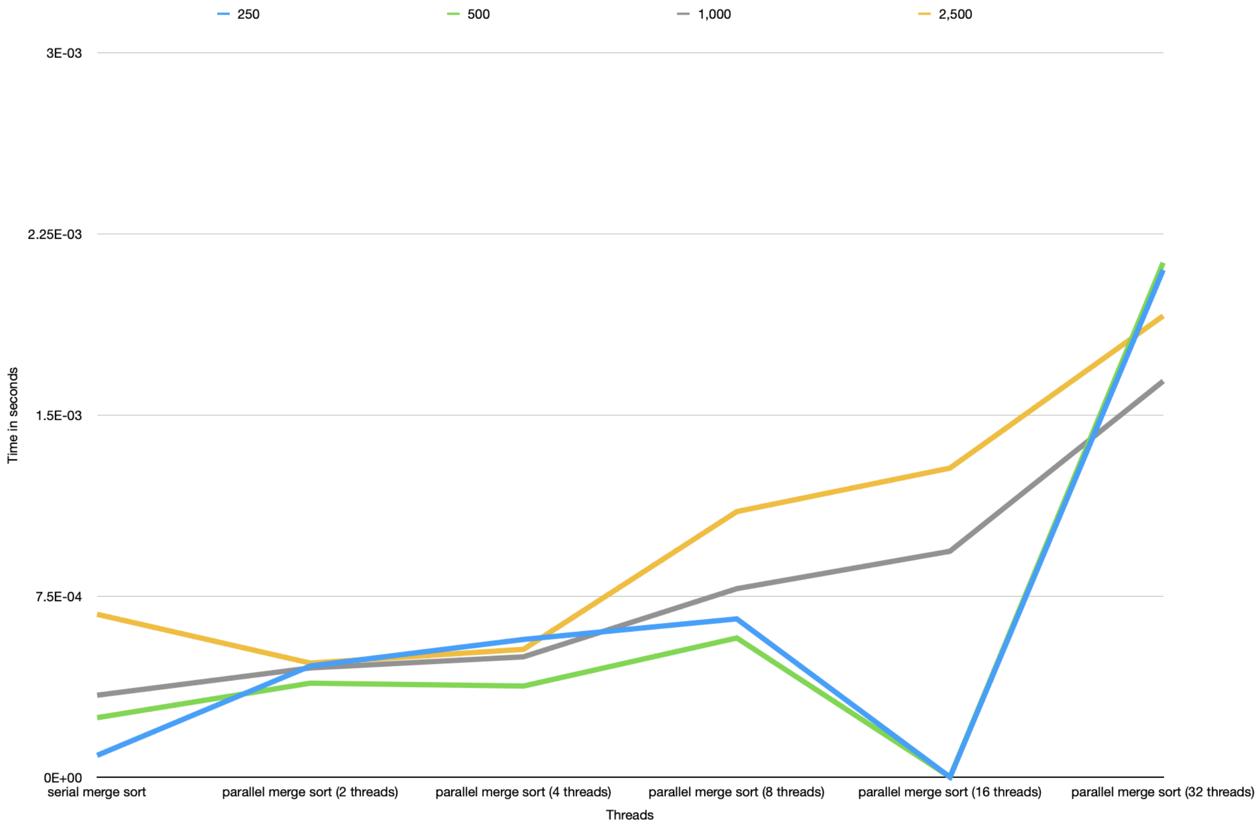


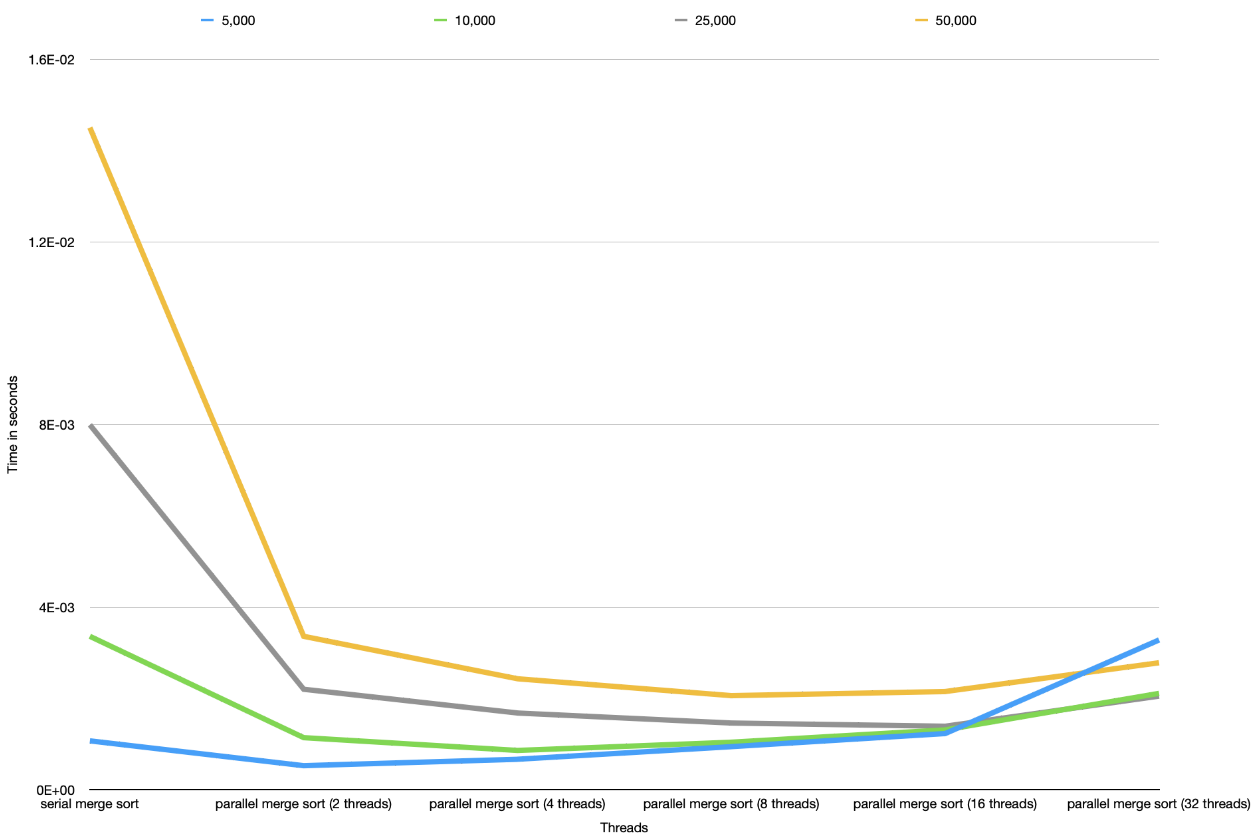
std::sort
serial merge sort
parallel merge sort (2 threads)
parallel merge sort (4 threads)
parallel merge sort (8 threads)
parallel merge sort (16 threads)
parallel merge sort (32 threads)
2E+01
1.5E+01
1E+01
5E+00
0E+00
Time in Seconds
25,000,000
50,000,000
100,000,000
Array Size

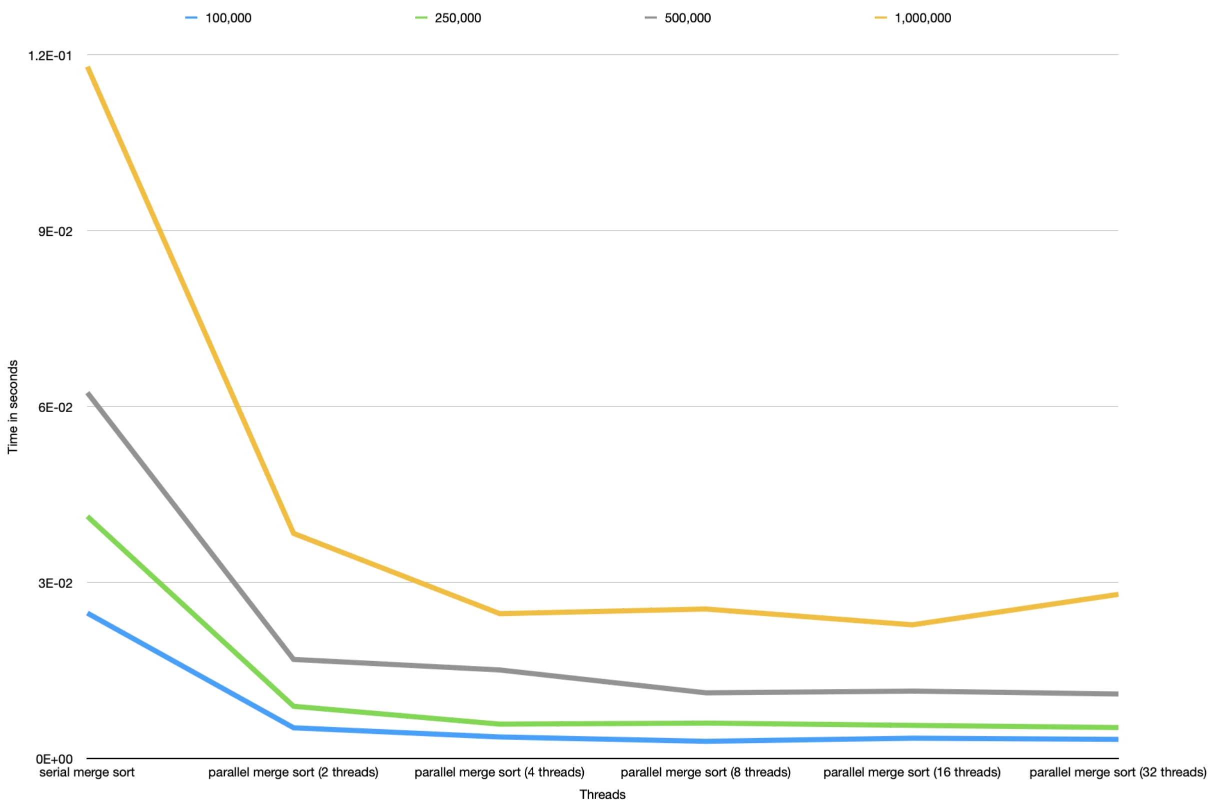


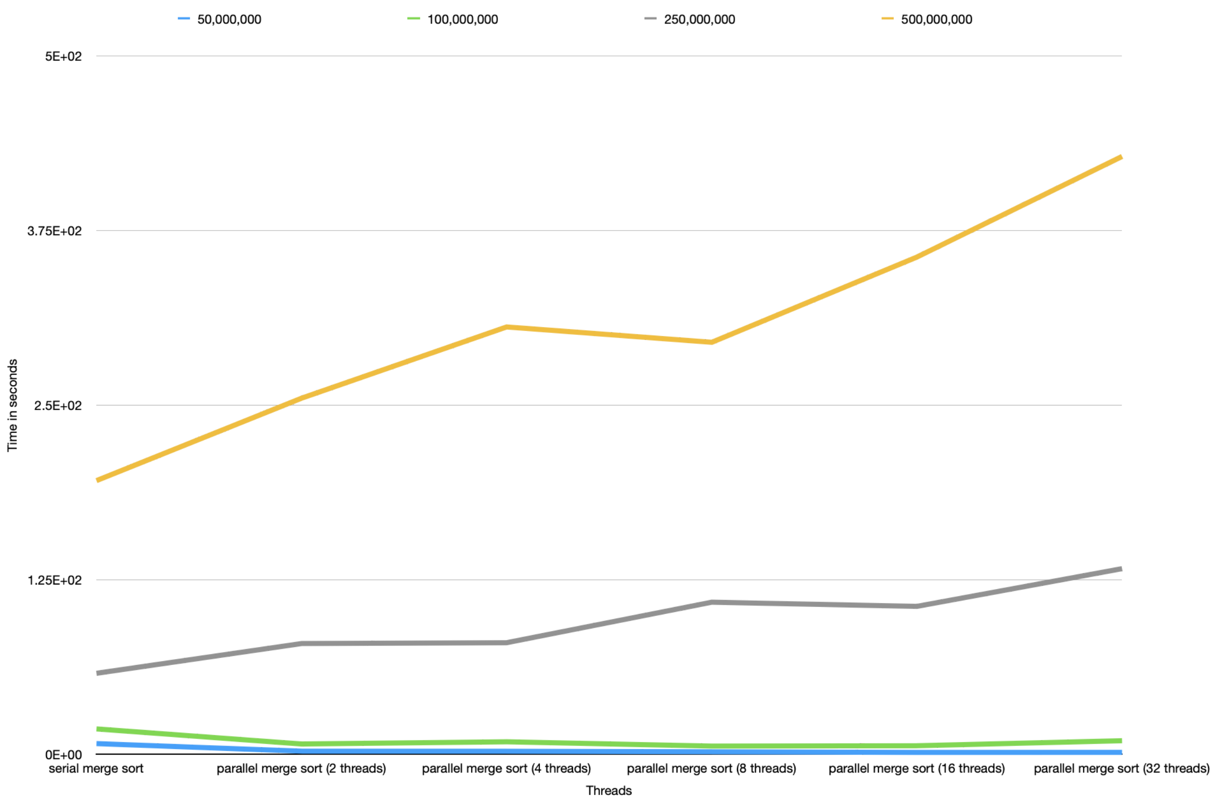
std::sort
serial merge sort
parallel merge sort (2 threads)
parallel merge sort (4 threads)
parallel merge sort (8 threads)
parallel merge sort (16 threads)
parallel merge sort (32 threads)
5E+02
3.75E+02
2.5E+02
1.25E+02
0E+00
Time in Seconds
250,000,000
500,000,000
Array Size

Time vs Thread Count





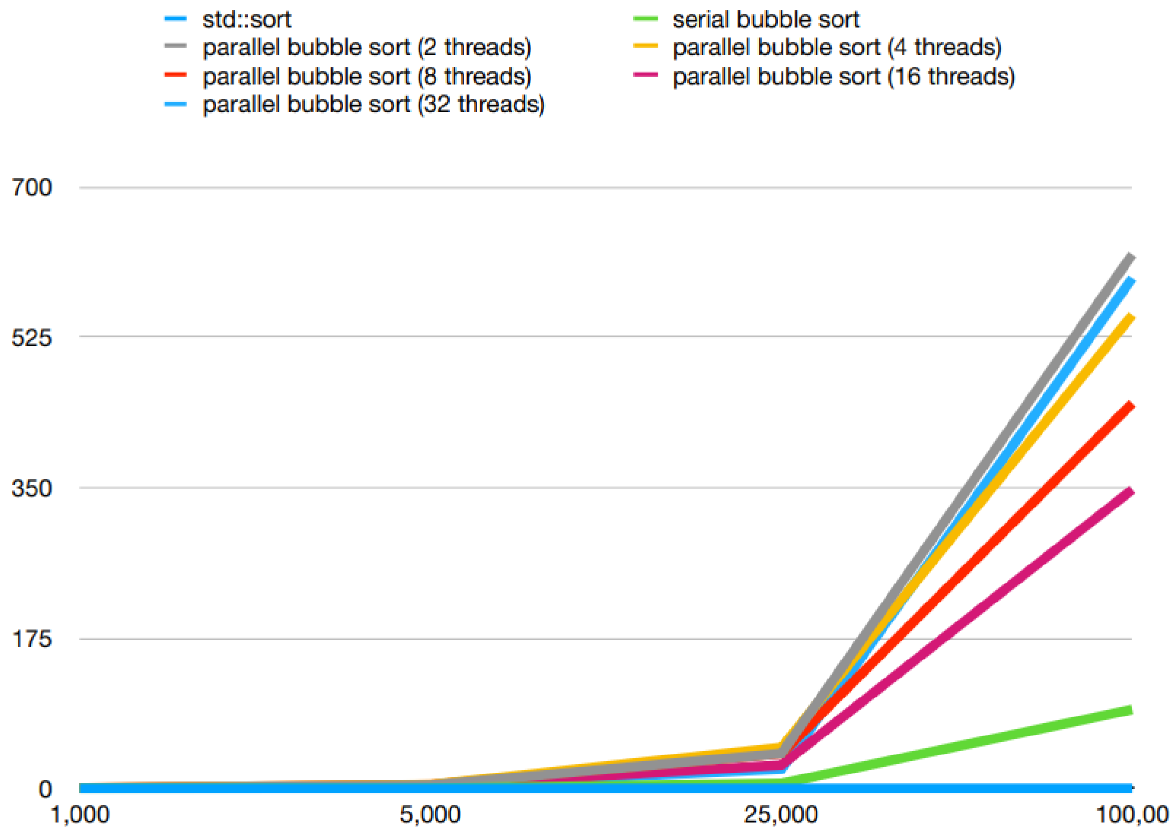




The graphs are divided to represent data in a better view. Each line corresponds to the size of the array.

Bubble Sort

Time To Size Graph



Conclusion

With the divide and conquer strategy that was used on the previous algorithms.
We can devise a standard algorithm to make any sorting algorithm parallel.

1. Divide the array into Chunks, each chunk being = array size / processors.
2. Use the algorithm to sort each chunk in parallel.
3. Take each two chunks and merge them in parallel.
4. Continue until one chunk is left
5. Return the array sorted.