Appendix A

Tables and Figures

	1	**2**	**4**	**8**
Parallel Merge Sort	4.272	2.684	2.913	1.045
Parallel Quick Sort	1.708	.932	.553	.480
Parallel Odd-Even Sort	1213.520	2089.720	3122.871	3320.862
Parallel Bitonic Sort	30917.209	16173.054	8307.189	4631.250

Table A.1: Effect of Thread Count on Sorting Runtime (ms) of 20,000 Element Array

	10	**100**	**1000**	**10000**	**100000**
Parallel Merge Sort	.247	.411	.101	.555	5.571
Parallel Quick Sort	.025	.033	.067	.266	2.164
Parallel Odd-Even Sort	4.525	4.527	24.115	1135.639	77398.873
Parallel Bitonic Sort	2.486	13.797	16.501	1161.760	70026.639

Table A.2: Effect of Array Size on Sorting Runtime (ms) Utilizing 8 Threads

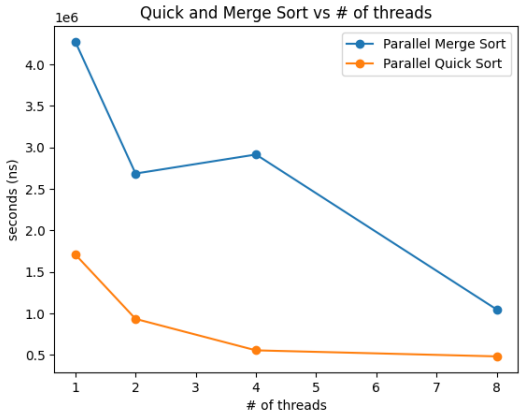

Figure A.1

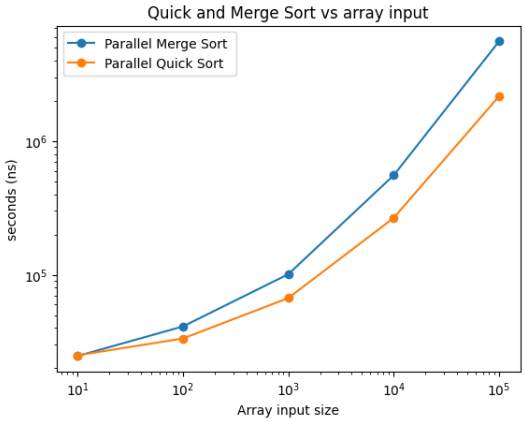

Figure A.2

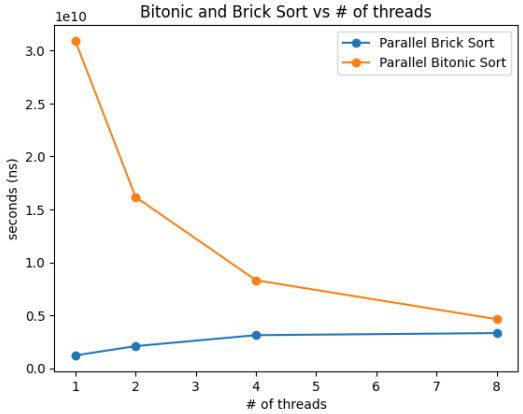

Figure A.3

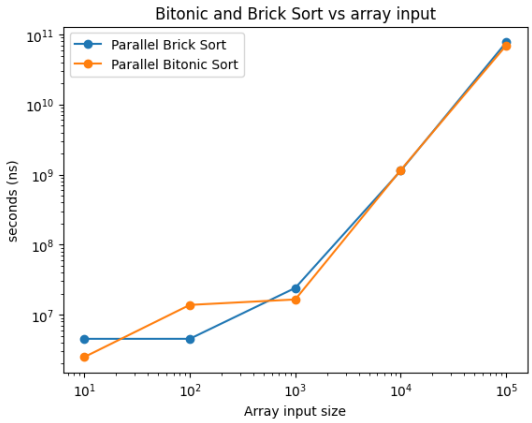

Figure A.4

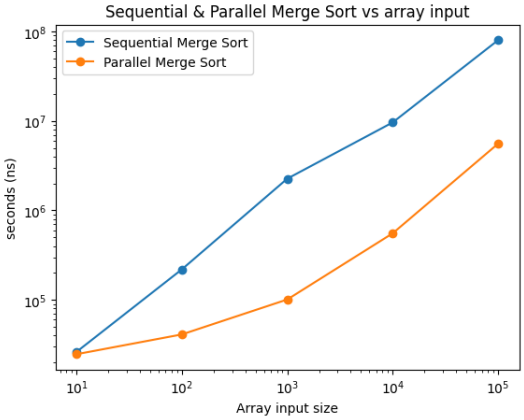

Figure A.5

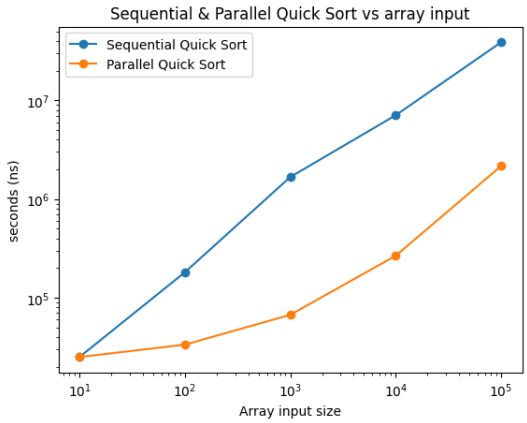

Figure A.6

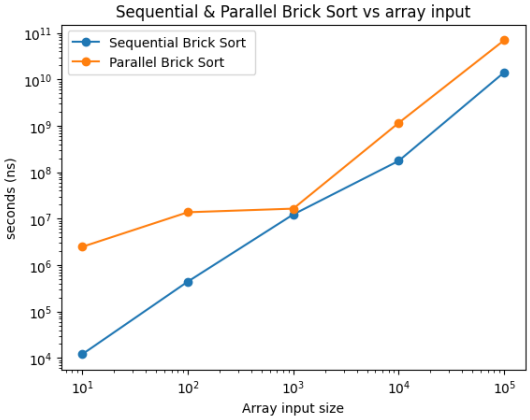

Figure A.7

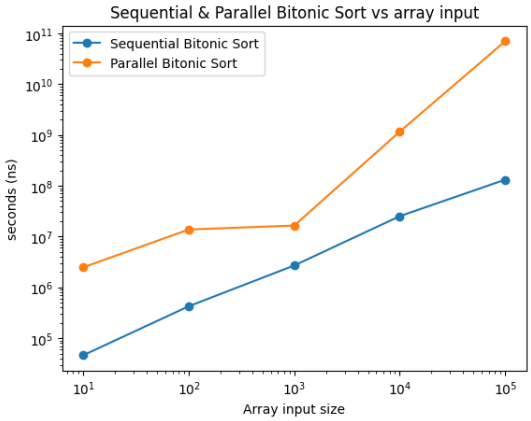
Sequential & Parallel Bitonic Sort vs array input
Sequential Bitonic Sort
Parallel Bitonic Sort
seconds (ns)
Array input size
10^{11}
10^{10}
10^{9}
10^{8}
10^{7}
10^{6}
10^{5}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}

Figure A.8

Appendix B

Pseudocode

Algorithm 1 Merge Sort [1]

Require: Input: Array of length n
1: **function** MERGESORT(A)
2: **if** $n == 1$ **then**
3: **return** A
4: **end if**
5: $l \leftarrow \text{MergeSort}(A[0, ..., n/2])$
6: $r \leftarrow \text{MergeSort}(A[n/2 + 1, ..., n])$
7: **return** MERGE(l, r)
8: **end function**

Algorithm 2 Parallel Merge Sort [1]

1: **function** PARALLELMERGESORT(A, p, r)
2: **if** $p < r$ **then**
3: $q \leftarrow \lfloor (p + r)/2 \rfloor$
4: **Spawn** ParallelMergeSort(A, p, q)
5: **Spawn** ParallelMergeSort(A, $q + 1$, r)
6: **Sync**
7: Merge(A, p, q, r)
8: **end if**
9: **end function**

Algorithm 3 Parallel Quick Sort [1]

```
1: function ParallelQuickSort(A, low, high)
2:     if low < high then
3:         pivot ← Partition(A, low, high)
4:         Spawn ParallelQuickSort(A, low, pivot − 1)
5:         Spawn ParallelQuickSort(A, pivot + 1, high)
6:         Sync
7:     end if
8: end function
9: function Partition(A, low, high)
10:    pivot ← A[high]
11:    i ← low − 1
12:    for j ← low to high − 1 do
13:        if A[j] ≤ pivot then
14:            i ← i + 1
15:            Swap(A[i], A[j])
16:        end if
17:    end for
18:    Swap(A[i + 1], A[high])
19:    return i + 1
20: end function
```

Algorithm 4 Parallel Odd-Even Sort [1]

```
1: var A : array[1..n] of int;
2: repeat
3:     found ← false;
4:     for all odd(j), j ∈ [1..n − 1] in parallel do
5:         if (A[j] > A[j + 1]) then
6:             found ← true;
7:             Swap(A[j], A[j + 1]);
8:         end if
9:     end for
10:    for all even(j), j ∈ [1..n − 1] in parallel do
11:        if (A[j] > A[j + 1]) then
12:            found ← true;
13:            Swap(A[j], A[j + 1]);
14:        end if
15:    end for
16: until ¬found;
```

Algorithm 5 Parallel Bitonic Sort [1]

```
1:  procedure BitonicSort(low, n, isAscending)
2:      if n > 1 then
3:          m = n/2
4:          parallel do:
5:              BitonicSort(low, m, true)
6:              BitonicSort(low + m, m, false)
7:          BitonicMerge(low, n, isAscending)
8:      end if
9:  end procedure
10: procedure BitonicMerge(low, n, isAscending)
11:     if n > 1 then
12:         m = n/2
13:         for i = low to low + m − 1 do
14:             parallel do:
15:                 j = i + m
16:             if (A[i] > A[j]) and isAscending then
17:                 Swap(A[i], A[j])
18:             else if (A[i] < A[j]) and ¬isAscending then
19:                 Swap(A[i], A[j])
20:             end if
21:         end for
22:         parallel do:
23:         BitonicMerge(low, m, isAscending)
24:         BitonicMerge(low + m, m, isAscending)
25:     end if
26: end procedure
```