SF2568 Project Report

Bitonic Merge Sort

Anirudh Seth (aniseth@kth.se) Magnus Pierrau (mpierrau@kth.se)

Spring 2020

Abstract

Sorting is often used as a basic building block when designing algorithms. It is an important component for many applications like searching, closest pair, element uniqueness, frequency distribution, selection, convex hulls etc. Batchers Bitonic sort is one such algorithm which is represented by a sorting network consisting of multiple butterfly stages.

In this project we study and implement a sequential and a parallel version of the Bitonic sorting algorithm, evaluate its performance using multiple metrics and compare it to some other state of the art sorting algorithms.

Contents

1 Introduction

Batchers Bitonic Sorting algorithm was developed by Ken Batcher in 1968 [5]. It is a comparison-based, internal sorting algorithm with a sequential time complexity of $\Theta(n\log(n)^2)$, which is close to the optimal complexity of $\Theta(n\log(n))$ for sequential comparison-based sorting methods. There is a sorting algorithm (ϵ-nearsort [2]) which has a complexity of $O(n\log(n))$, but due to its large constant, Bitonic sort is a faster algorithm for all practical sorting purposes.

Bitonic Sort is not a stable sorting method, meaning that if two keys are equal, they still switch position. For our experimental purposes this is not an issue, but it is important to keep in mind if using the algorithm for other purposes.

In this project we implement both the sequential and parallel version of Batchers Bitonic sort in `C`, using the `MPI` library, and measure their performance when run on the PDC Tegner system. The performance is measured on random sequences of $n = 2^d$ signed doubles between 0-1, where d ranges from 17 to 26, and for p processors, where p is a power of 2 and ranges from 1 to 64. We run the algorithm 5 times for each setup and present and analyse the average run time. We analyse the time complexity of the implementations and compare our empirical results to the expected theoretical results as well as compute the parallel speedup, efficiency and cost for the parallel implementation.

1.1 Bitonic Sorting algorithm

The Bitonic Sort algorithm utilizes multiple sorting networks in order to sort an arbitrary sequence of numbers. The sorting networks are made up of comparator networks B_n, each comparing n numbers by using $n/2$ comparators.

By using a divide-and-conquer strategy we can build two networks, *Bitonic Merge* and *Bitonic Sort*, which together make up the Bitonic Sort algorithm.

1.1.1 Bitonic Merge network

Bitonic Merge is a network which sorts a given bitonic sequence, a, into two bitonic subsequences b and c, where $max(b) \leq min(c)$. This is done using the comparator network B_n, in which entries $[0, n/2]$, $[1, n/2+1]$, ..., $[n/2-1, n-1]$, are compared and switched depending on the given predicate (whether the list is to be sorted in an ascending or decreasing order).

More precisely, if we let $a = a_0, a_1, ..., a_k$ be a bitonic 0-1 sequence, then applying the comparator network B_n (some times called a half-cleaner [3]) to a yields $B_n(a) = b_0, ..., b_{n/2-1}, c_0, ...c_{n/2-1}$, where all b_i are smaller than or equal to all c_j. Furthermore $b_0, ..., b_{n/2-1}$ and $c_0, ..., c_{n/2-1}$ are bitonic [1].

1.1.2 Bitonic Sorting network

The Bitonic sorting network gets a sequence of unsorted numbers as input, which is recursively divided into two and sent as input into the same network, but with the subsequence having opposite order of sorting (ascending and descending respectively, indicated by the arrows in figure 2) and produce a bitonic sequence when returned. When reaching the base case $n = 1$ the sequence is returned and passed into *Bitonic Merge*, which then returns a sorted bitonic sequence of length 2, which in turn is returned to the upper recursive layers, eventually producing a sorted list.

Due to the zero-one principle, we can apply the network to any arbitrary input that can be represented by a sequence of 0's and 1's, meaning that we can sort any input, given a valid comparator function.

The networks are illustrated in figure 4 for a 0-1 sequence and an example for a sequence of random integers is given in figure 2.

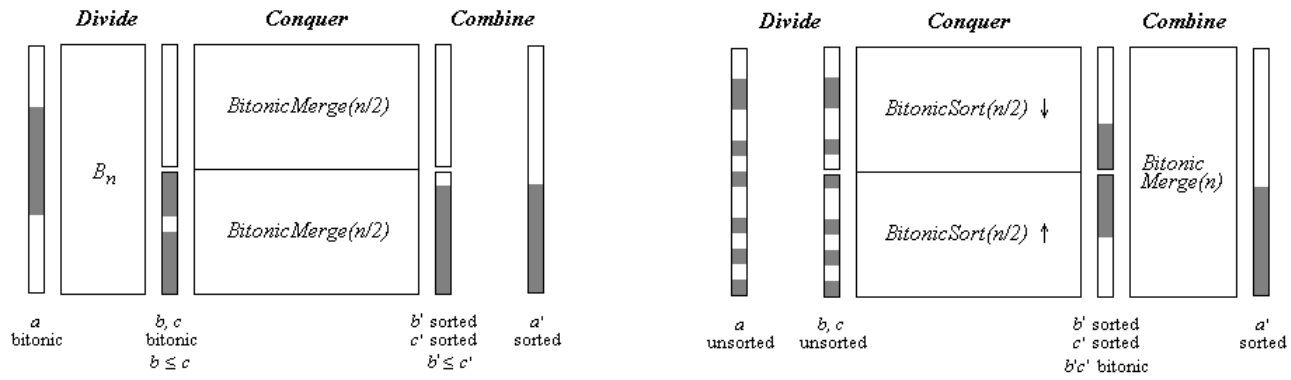

Figure 1: Structures of networks *Bitonic Merge* (left) and *Bitonic Sort* (right) for a 0-1 sequence of length n. Here 0's are drawn white and 1's are drawn gray. Image credit [1].

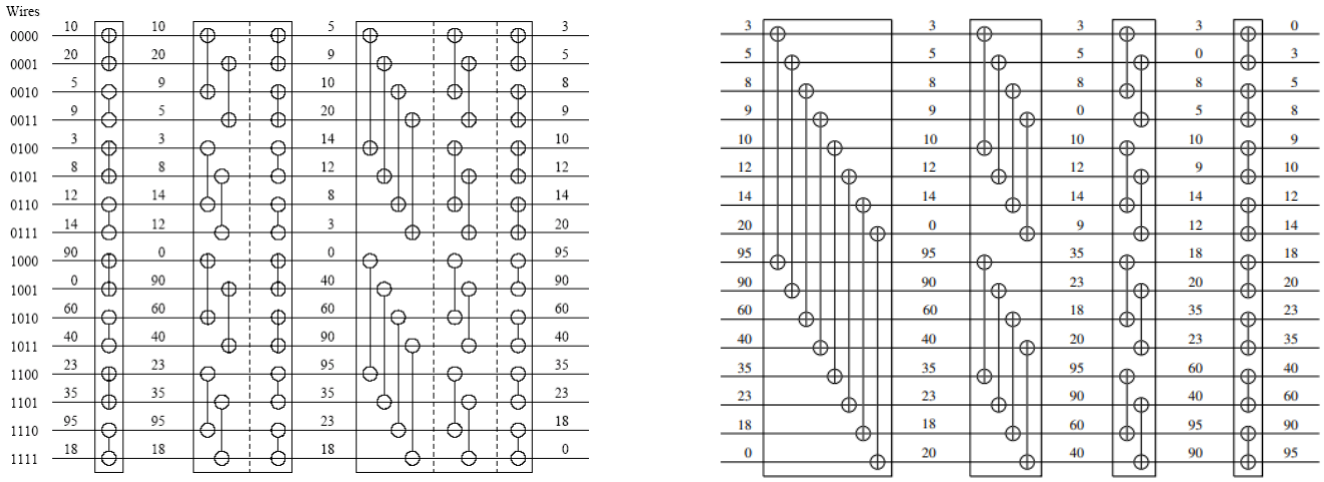

Figure 2: An example of the Bitonic Merge network (left) creating a bitonic sequence from an input sequence of 16 random numbers and the *Bitonic Sort* network (right) merging the bitonic sequences into a sorted sequence. The $\oplus$ indicate an increasing comparator, which switches the smallest value to the top and the Θ vice versa. Image credit [4].

2 Serial Implementation

The serial implementation was written in `C` and executed on a laptop with configuartions as seen in table 1.

Model name	Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz
CPU MHz	1800.000
Cache size	256 KB
CPU cores	4

Table 1: Machine Configuration

2.1 Psuedocode

Algorithm 1 shows the psuedocode for our implementation of the sequential version of the Bitonic sort algorithm, with n being a multiple of 2.

Algorithm 1 Recursive algorithm for Sequential Bitonic Sort

Require: *arr* (Sequence), *low* (0), *count* (Number of elements),*direction* (Ascending or Descending)

```
1: function BitonicMerge(arr, low, count, direction)
2:     if (count>1) then
3:         index=count/2
```

```
    for i ← low to  low + index do
        compareAndSwap(arr,i,i+index,direction)
    end for
    BitonicMerge(arr, low, index, direction)
    BitonicMerge(arr, low + index, index, direction)
  end if
end function

function BitonicSort(arr, low, count, direction)
  if (count>1) then
    index=count/2
    BitonicSort(arr, low, index, True)
    BitonicSort(arr, low + index, index, False)
    BitonicMerge(arr, low, count, direction)
  end if
end function

function compareAndSwap(arr, a, b, direction)
  if (direction==(arr(a)>arr(b)) then
    swap(arr(a),arr(b))
  end if
end function
```

Figure 3 shows the output of algorithm on a random sequence of length $n = 16$ (run on tegner). The figure shows the correctness of our implementation of the algorithm, which executes in less than one microsecond for this specific setup.

```
[aniset@tegner-login-1 project]$ mpirun -np 1 ./a.out
Orignal Array: 0.840188 0.394383 0.783099 0.798440 0.911647 0.197551 0.335223 0.768230 0.277775 0.553970 0.477397 0.628871 0.364784 0.513401 0.952230 0.916195
Sorted Array: 0.197551 0.277775 0.335223 0.364784 0.394383 0.477397 0.513401 0.553970 0.628871 0.768230 0.783099 0.798440 0.840188 0.911647 0.916195 0.952230
Number of elements: 16
Time taken: 0.000000 seconds.
[aniset@tegner-login-1 project]$
```

Figure 3: Example of sequential implementation of Bitonic sort algorithm sorting a sequence of length $n = 16$ of doubles in <1e-4 ms.

2.2 Time Analysis

To form a sorted sequence of length n from two sequences of length $n/2$, we need to perform $\log(n)$ comparisons, i.e. the comparator network has depth $\log(n)$. Since at each depth of the bitonic merge network we compare $\log(k)$ (for some $k \leq n$) numbers and then call for a merge of $k/2$ numbers, the total number of comparator stages $T(n)$ of the entire sorting network can be given as:

$$T(n) = \log(n) + T(n/2).$$

The base case is $n = 1$, which returns 0, and the solution to this recursive formula is thus given by (by arithmetic sum):

$$T(n) = \log(n) + \log(n) - 1 + \log(n) - 2 + \ldots + 1 = \frac{\log(n) \cdot (\log(n) + 1)}{2}.$$

The entire sorting procedure consists of $n/2$ comparators, which finally gives

$$\frac{n}{2} \frac{(\log(n) \cdot (\log(n) + 1))}{2} = \frac{n \log(n)^2}{4} + \frac{n \log(n)}{4} = \Theta\left(n \log(n)^2\right).$$

2.3 Results

To see if there is any difference in performance depending on the arrangement of the input we have run the algorithm once each for three different cases; when the input sequence is completely random (yellow) and when it is already sorted, in descending (red) and ascending (blue) order respectively.

The exact running time in seconds for the sequential recursive implementation of the Bitonic Sorting algorithm can be found in table 2 in the appendix.

The results are visualized in figure 4.

By plotting a $Cn\log(n)^2$ curve with a fitted proportionality constant $C = 4.17\text{E+}08$ we can confirm that the computational complexity agrees with the theoretical result. The constant was computed by normalizing $y = n\log(n)^2$ with the mean of the results from the random sequence.

As we can see from figure 5, there is no difference between giving ascending or descending sorted sequences as input. When giving a random sequence we do note an increase in running time. Although only visible for larger n ($n > 2^{24}$), the running time is on average 21% larger for the random sequence than the sorted samples. As the Bitonic Sort algorithm uses multiple comparator networks it will perform equally many comparisons for all sequences of equal length and should thus not create a difference in run time. The increase might instead be explained by the additional operations required to switch the two entries in the comparisons that arise in the random sequence but not in the sorted lists.

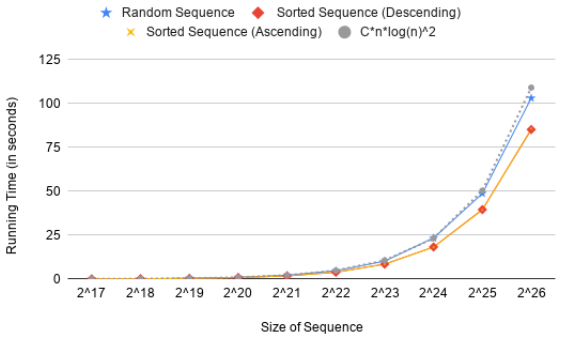

Figure 4: Running time for three different cases of input sequence together with a $Cn\log(n)^2$ curve for comparison, where C =4.17E+08 is a proportionality constant.

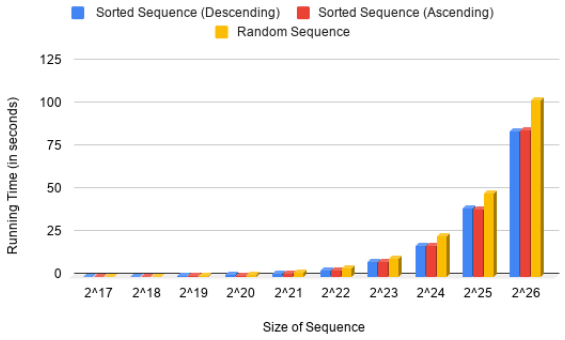

Figure 5: Bar plot of running time for three different cases of input sequence.

3 Parallel Implementation

Since there is a one-to-one correspondence between two numbers being compared at any point in time by a comparator, this means that data is non-dependent, allowing for very effective parallelisation of this algorithm.

The parallel implementation was written in `C` using the `MPI` library and was run on the PDC Tegner system on multiple numbers of processors, using the `-o3` optimisation flag.

3.1 Processor topology

For the parallel implementation, we assume a hypercube topology for the processes. The compare and exchange/split operations take place between neighbouring processors whose binary label differ by only one bit. Consider how processes are paired for their compare-exchange steps in a d-dimensional hypercube (that is, p = 2^d). During the first step of this stage, processes that differ only in the d^{th} bit of the binary representation of their labels (that is, the most significant bit) compare-exchange their elements. Thus, the compare-exchange operation takes place between processes along the d^{th} dimension. Similarly, during the second step of the algorithm, the compare-exchange operation takes place among the processes along the $(d-1)^{th}$ dimension. In general, during the i^{th} step of the final stage, processes communicate along the $(d-(i-1))^{th}$ dimension. Figure 6 shows the processes communicating at each time step for the same example discussed above.

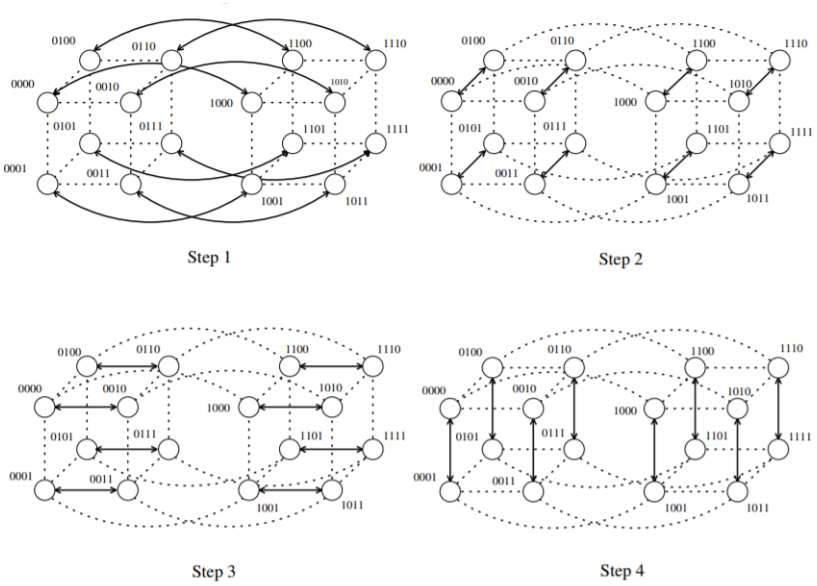

Figure 6: Communication during one stage of bitonic sort implemented on a hypercube. Each wire is mapped to a hypercube process; each connection represents a compare-exchange between processes. Image credit [6].

The communication for the entire algorithm can be broken down as :

- Each process is assigned a block of n/p elements.
- The first step is a local sort of the local block. We use quick sort for this purpose.
- Each subsequent compare-exchange operation is replaced by a compare-split operation.
- We can effectively view the bitonic network as having $(1 + \log(p))(\log(p))/2$ steps.

3.2 Psuedocode

Algorithm 2 shows the psuedocode for our implementation of the parallel version of Bitonic Sort on p processors, where p is a multiple of 2.

Algorithm 2 Parallel implementation for Bitonic Sort

Require: *arr* (Sequence), *low* (0), *count* (Number of elements),*direction* (Ascending or Descending)

```
1:  function BitonicSort_Increasing(list, listSize, processorSize)
2:      processorDimension = log2(processorSize)
3:      eor_bit=1 << (processorDimension - 1)
4:      for stage ← 0 to processorDimension do
5:          partner = myRank^eor_bit
6:          if (myRank < partner) then
7:              mergeSplit(listSize,LOW,list,partner)
8:          else
9:              mergeSplit(listSize,HIGH,list,partner)
10:         end if
11:     end for
12: end function
13:
14: function BitonicSort_Decreasing(list, listSize, processorSize)
15:     processorDimension = log2(processorSize)
16:     eor_bit=1 << (processorDimension - 1)
17:     for stage ← 0 to processorDimension do
18:         partner = myRank^eor_bit
19:         if (myRank > partner) then
20:             mergeSplit(listSize,LOW,list,partner)
21:         else
22:             mergeSplit(listSize,HIGH,list,partner)
```

```
        end if
    end for
end function

function MERGE_LOW(list1,list2,listSize)
    index1 = index2 = 0
    for i ← 0 to listSize do
        if (list1[index1] ≤ list2[index2]) then
            mergeList[i] = list1[index1++]
        else
            mergeList[i] = list2[index2++]
        end if
    end for
    list1 = mergeList
end function

function MERGE_HIGH(list1,list2,listSize)
    index1 = index2 = listSize
    for i ← listSize to 1 do
        if (list1[index1] ≥ list2[index2]) then
            mergeList[i] = list1[index1–]
        else
            mergeList[i] = list2[index2–]
        end if
    end for
    list1 = mergeList
end function

function MERGE_SPLIT(listSize,list,partner,keys)
    partnerList = sendAndReceiveList(partner)
    if (keys=HIGH) then
        Merge_High(listSize,list,partnerList)
    else
        Merge_Low(listSize,list,partnerList)
    end if
    list1 = mergeList
end function
```

Algorithm 3 Parallel formulation of bitonic sort on a hypercube with n = 2^d processes

```
andBit = 2
for processorSize ← 2, to p do
    if (myRank&andBit = 0) then
        BitonicSort_Increasing(listSize,list,processorSize)
    else
        BitonicSort_Decreasing(listSize,list,processorSize)
    end if
    processorSize = 2 * processorSize
    andBit << 1
end for
```

```
[aniset@tegner-login-1 project]$ mpicc bitonicParallel.c -lm
[aniset@tegner-login-1 project]$ mpirun -np 2 ./a.out 16
Processor 0:0.230870 0.059107 0.668104 0.606553 0.785917 0.559260 0.475998 0.044352
Processor 1:0.588435 0.473691 0.472162 0.425704 0.721515 0.281971 0.835934 0.840965
N: 16
P: 2
Time(sec): 0.000029
Final Sorted List: 0.04435 0.05911 0.23087 0.28197 0.42570 0.47216 0.47369 0.47600 0.55926 0.58843 0.60655 0.66810 0.72151 0.78592 0.83593 0.84097
--------------------------------------------------------------------------
```

Figure 7: Example of result from sorting a sequence of length $n = 16$ on $p = 2$ processors in 3e-2 ms.

Figure 7 shows the result of our implementation for a sequence of $n = 16$ random doubles on two processors on the Tegner system. The figure shows the correctness of our implementation, which for this setup executes in ca 30 μs.

3.3 Time Analysis

Given n numbers and p processes (2^d), each process is assigned a block of n/p elements. The first step is to locally sort this block. This can be performed in $\Theta((n/p)\log(n/p))$ time. Every process then performs a compare-exchange operation. This has the complexity of $\Theta(\log^2(p))$. The total time complexity for our implementation (which assumes a hypercube topology) is thus given by:

$$T_p = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(\frac{n}{p}\log^2 p\right)}^{\text{comparisons}} + \overbrace{\Theta\left(\frac{n}{p}\log^2 p\right)}^{\text{communication}}.$$

We note here that if we keep the number of processors, p, constant, and increase n, the parallel run time increases with $\Theta(n\log(n))$. If we keep the problem size n fixed and instead increase the number of processors, p, we will find that for large n and small p, the local sort part will dominate, as it is $\Theta(\frac{1}{p}\log\frac{1}{p})$, and run time should decrease with p. However, as p increases, the comparison and communication terms will begin to dominate, and the run time will again increase with $\Theta\left(\frac{1}{p}\log^2(p)\right)$.

There seem to exist a balance between problem size n and number of processors p in order to achieve an optimal run time.

3.4 Performance Metrics

To compare the performance between sequential and parallel implementation we consider the speedup, efficiency and cost for various values of n and p.

3.4.1 Speedup

The speedup is defined as the quotient between the serial run time of the fastest sequential algorithm for solving the sorting problem, T_S^*, and the run time for the parallel implementation, T_P.

$$S_p = \frac{T_S^*}{T_p}$$

In our case we have used the standard `C` implementation of the *quicksort* algorithm, `qsort`, which is considered to be one of, if not the, fastest sequential sorting algorithms [7]. Quicksort has an average time complexity of $\Theta(n\log n)$.

The time complexity for the parallel implementation of Bitonic sort is

$$\begin{aligned} T_p &= \Theta\left(\frac{n}{p}\log\frac{n}{p}\right) + \Theta\left(\frac{n}{p}\log^2 p\right) + \Theta\left(\frac{n}{p}\log^2 p\right) \\ &= \Theta\left(\frac{n}{p}\log\frac{n}{p}\right) + \Theta\left(\frac{n}{p}\log^2 p\right), \end{aligned}$$

which gives a theoretical complexity of

$$S_p = \frac{\Theta(n\log n)}{\Theta((n/p)\log(n/p)) + \Theta\left((n/p)\log^2 p\right)} \tag{1}$$

for the speedup.

3.4.2 Efficiency

The efficiency, E, of a parallel implementation is given as the fraction between the speedup S_P and the number of processors, p:

$$E = \frac{S_p}{p}.$$

The efficiency measures the fraction of time for which a processor is usefully utilized.

Given the speedup S_p in (1) we get a complexity of

$$
\begin{aligned}
E &= \frac{\Theta(n \log n)}{p \cdot \left(\Theta((n/p)\log(n/p)) + \Theta\left((n/p)\log^2 p\right) + \Theta\left((n/p)\log^2 p\right)\right)} \\
&= \frac{\Theta(n \log n)}{\Theta(n \log(n/p)) + \Theta\left(n \log^2 p\right)} \\
&= \frac{1}{1 - \Theta\left(\frac{\log p}{\log n}\right) + \Theta\left(\frac{\log^2 p}{\log n}\right)}.
\end{aligned}
$$

We note that as we keep p fixed and increase n, the denominator will decrease, causing the efficiency to increase. This is quite intuitive as increasing the workload will cause more processors to be useful for a larger fraction of the total run time. If we keep n fixed and increase p we will observe the opposite - the denominator will increase in size and the efficiency will consequently decrease. Here we can apply the same intuitive reasoning, as more processors leads to a smaller individual workload for each processor and thus increasing the risk of overhang.

3.4.3 Cost

The cost of solving a problem on a parallel system is given as the number of processors times the run time for solving the task:

$$
\begin{aligned}
\text{Cost} =& p \cdot T_{\text{p}} \\
=& p \cdot \left(\Theta((n/p)\log(n/p)) + \Theta\left((n/p)\log^2 p\right)\right) \\
=& \Theta(n \log(n/p)) + \Theta\left(n \log^2 p\right)
\end{aligned}
$$

3.5 Results

Table 3, found in the appendix, shows the exact running time in seconds for the parallel implementation of Bitonic sort for various lengths of the input sequence, n, and the number of processors, p.

By comparing tables 2 and 3 we find that the parallel implementation executes the sorting task in between 9 - 25% of the total sequential time for the various setups. However, from figures 8 and 9 we note that as we increase the number of processors above 8, the running time again increases.

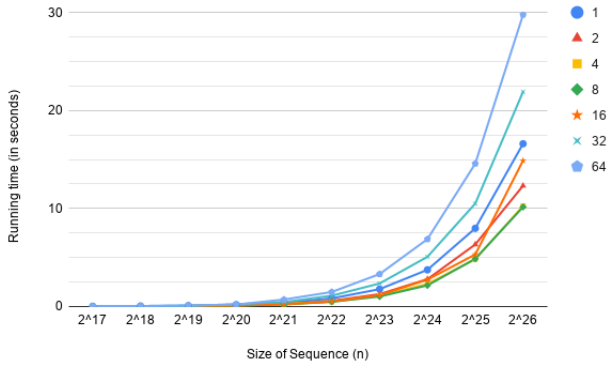

Figure 8: Running time for various numbers of processors p given length n of random input sequence.

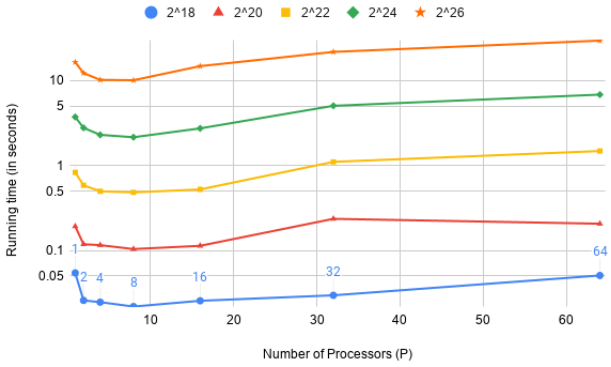

Figure 9: Running time as a function of the numbers of processors, p, for five fixed values of n. Note that the vertical axis is logarithmic.

A more reliable performance measure is parallel speedup, S, and efficiency, E, shown in figures 10 and 11 (exact values are found in table 4 and 5 in the appendix).

By regarding the speedup in figure 10 we find that the speedup first increases with the number of processors, up to $p = 8$, with a maximum value of 2.1 when $n = 2^{19}$, before decreasing again. For $p = 1$ and $p \geq 32$ the speedup is around or below 1 for most values of n, indicating inefficient usage of the processors. This might be due to excessive communication and overhang.

Figure 11 shows the computed efficiency (S/p) as a function of n (plots of efficiency as a function of p is found in the appendix). We note here that the efficiency decreases with p, and is only optimal for $p = 1$.

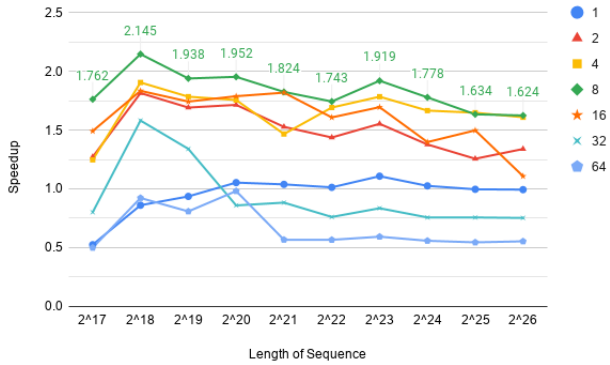

Figure 10: Parallel speedup as a function of the length of sequence, n.

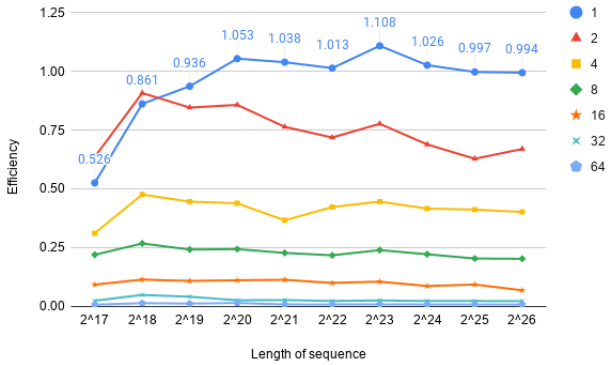

Figure 11: Efficiency as a function of the length of sequence, n.

Figure 12 shows the cost as a function of n for various values of p. Full table of results is shown in table 6, found in the appendix. We note that the cost increases for each addition of processors and with n. The increase looks to be linearly increasing, but both axes are logarithmic, the horisontal with base 2 and the vertical with base 10, so the increase is more likely exponential.

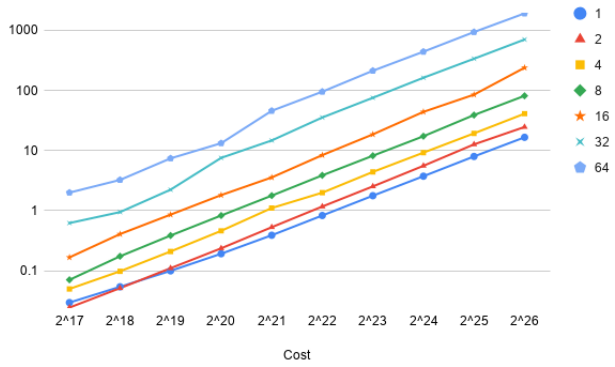

Figure 12: Cost as a function of the length of sequence, n.

4 Discussion

Comparing Bitonic sort to a cost optimal sequential algorithm with $\Theta(n \log n)$ gives that for the parallel implementation to be cost optimal ($E = 1$) we need $\frac{log^2 p}{\log n} = O(1)$. This in turn yields that the algorithm can optimally use $p = \Theta\left(2^{\sqrt{\log n}}\right)$ processors [18]. In our experiment, n ranges between 2^{17} and 2^{26}, yielding $p \approx 17$ for the former and $p \approx 34$ for the latter. This states that, theoretically, we would expect a higher efficiency than what we have observed, for higher values of p as n increases.

However, in the time complexity analysis we have assumed a hypercube topology, but Tegner (Beskow) uses a 3-level Dragonfly topology. The scalability of the bitonic sort algorithm on such a topology is likely different than on a hypercube one, but computing the ISO-efficiency function for such a topology is outside the scope of this project.

We can state that since the efficiency decreases slightly with the increasing number of processors, our parallel algorithm doesn't scale as expected.

And indeed, as stated in [18], Bitonic sort is neither very efficient nor scalable on a mesh, hypercube or ring topology, primarily due to the sequential algorithm being suboptimal. Good speedup is possible on a large number of processors, but for very large values of n.

4.1 Comparison with other Parallel sorting algorithms

We compare our algorithm with a parallel implementation of merge sort [19] and sample sort [20]. The results are summarized in table 7, found in the appendix.

According to the results in figure 13 the bitonic sort algorithm outperforms the Sample sort and Merge sort algorithms for all number of processors. The sequential complexity for bitonic sort $\Theta(n \log(n)^2)$ compared to

the optimal sorting complexity of $\Theta(n \log(n))$ is not optimal but it outperforms in a parallel implementation because we always compare elements in a predefined sequence and the sequence of comparison doesn't depend on data. The cost of execution is the least for bitonic sort as can be seen in figure 15. The efficiency however performs a little worse than sample sort for lower number of processors as seen in figure 16 but outperforms all algorithms with increasing number of processors.

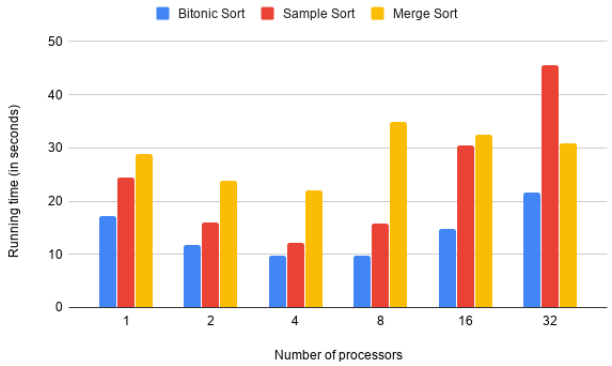

Figure 13: Comparison of running time as a function of the numbers of processors for n=2^{26}, between Bitonic sort, Sample sort and Merge sort

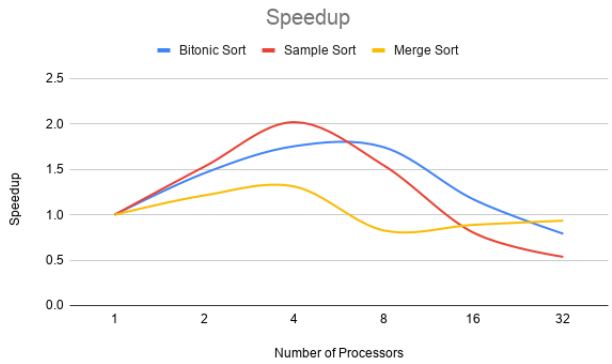

Figure 14: Comparison of speedup between Bitonic sort, Sample sort and Merge sort algorithms.

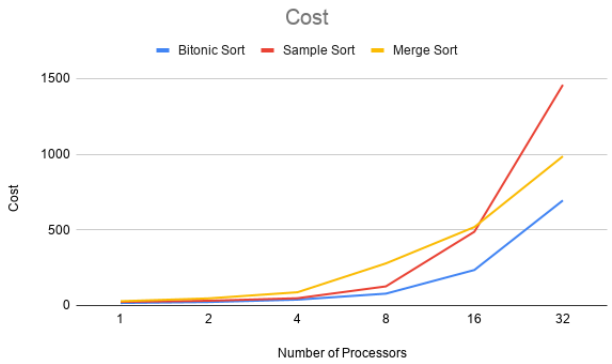

Figure 15: Comparison of Cost between Bitonic sort, Sample sort and Merge sort algorithms.

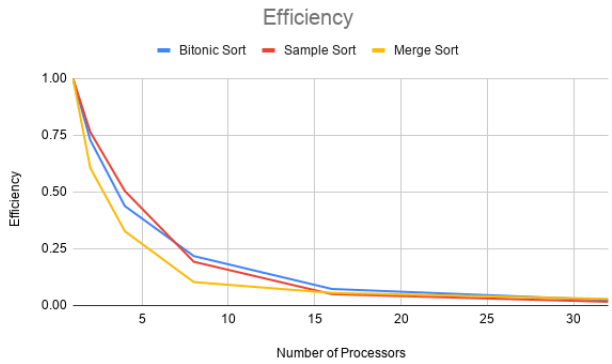

Figure 16: Comparison of efficiency between Bitonic sort, Sample sort and Merge sort algorithms.

5 Future Work

5.1 Sequence of arbitrary length

Bitonic sort as introduced by Batcher and implemented for this project can only sort sequences with length of sequence being a power of 2. A trivial way to sort sequences of arbitrary length in ascending (analog for descending) order is to pad the sequence with max-values to make the sequence of size equal to the next greater power of 2. Since Batcher's Bitonic Sort sorts subsequences in alternating directions these elements would be moved while sorting the sequence and therefore will have to exist physically. Thus an input sequence of length $2^k + 1$ would result in sorting a sequence of length 2^{k+1} [8]. Alternatively several variants of Bitonic sort have been proposed in the literature that are capable of sorting sequences of arbitrary length [9][10][11].

5.2 Alternate Topologies

One of the key aspects of the bitonic algorithm is that it is communication intensive, and a proper mapping of the algorithm must take into account the topology of the underlying interconnection network. In the project, we implemented a hypercube process topology for Bitonic split/merge operations. Alternatively several other topolgies like 2D Mesh , ring and migratory thread architecture have been studied for bitonic sort [12][14][13]. The connectivity of a mesh topology is lower than that of a hypercube. Authors in [18] explain how; (a) row

major mapping; (b) row-major snakelike mapping, and; (c) row-major shuffled mapping can be implemented to overcome these issues.

5.3 Variants of Bitonic Sort

Bitonic sort becomes slower when sorting very long sequences, because of the limitation of shared memory. In order to merge long bitonic sequences, global memory has to be used instead of shared memory. Furthermore, global memory has to be accessed during every step of bionic merge. In order to increase the speed of sort, multiple steps of bitonic merge have to be executed with a single kernel invocation. This can be achieved with *multistep bitonic sort* [15].
Another variant, *Interval Based Rearrangement (IBR) bitonic sort* [16] is based on adaptive bitonic sort implemented by Bilardi et. al. [17]. Adaptive bitonic sort operates on the idea, that every bitonic sequence can be merged, if first half of a bitonic sequence and second half of a bitonic sequence are ring-shifted by a value, q, which can be found with a variation of a binary search. In order to exchange elements in a bitonic merge step in time $\Theta(\log(n))$, a variation of a binary tree is used [17][16].

6 Conclusion

In this project we have studied the bitonic sort algorithm, discussed and implemented some parallelization strategies. We present a quantitative comparison of a sequential and a parallel algorithm for data ranging up to 67 million 32-bit keys. Parallel programming goes beyond the limits imposed by sequential computing, this can be validated from our results. A significant improvement in execution time can be observed especially with the increasing amount of data (n).We also compare our implementation with other parallel sorting algorithms and found the results to be satisfactory and in accordance with the literature presented in the seminars. However, we also observe that our algorithm doesn't scale to its true potential utilizing the computational capacity of a supercomputer like Tegner. Bitonic Sort in general is a communication intensive algorithm, the correct topology and mapping of the interconnection network plays a significant role in its performance. We also present some alternate topolgies and strategies reviewed by researchers in the field.

7 Appendix

Size of Sequence	Random Sequence	Sorted Sequence (Descending)	Sorted Sequence (Ascending)
2^{17}	0.0938	0.0938	0.0781
2^{18}	0.219	0.172	0.172
2^{19}	0.469	0.406	0.391
2^{20}	1.016	0.891	0.844
2^{21}	2.234	1.781	1.828
2^{22}	4.769	3.891	3.969
2^{23}	10.266	8.516	8.484
2^{24}	23.375	18.266	18.141
2^{25}	48.703	39.563	39.297
2^{26}	102.953	85.031	85.563

Table 2: Running time (in seconds) for the sequential recursive implementation of Bitonic Sort.

Size of Sequence	Number of Processors (p)						
	1	2	4	8	16	32	64
2^{17}	0.030	0.012	0.013	0.009	0.010	0.019	0.031
2^{18}	0.054	0.026	0.025	0.022	0.026	0.030	0.051
2^{19}	0.100	0.055	0.053	0.048	0.054	0.070	0.116
2^{20}	0.193	0.119	0.116	0.104	0.114	0.236	0.207
2^{21}	0.391	0.266	0.277	0.223	0.223	0.460	0.716
2^{22}	0.833	0.587	0.499	0.484	0.525	1.108	1.488
2^{23}	1.777	1.268	1.104	1.026	1.162	2.356	3.315
2^{24}	3.747	2.790	2.308	2.162	2.750	5.073	6.875
2^{25}	7.979	6.332	4.828	4.868	5.311	10.502	14.574
2^{26}	16.605	12.321	10.255	10.159	14.891	21.904	29.759

Table 3: Running Time (in seconds) for parallel implementation of the Bitonic Sorting algorithm for p number of processors and input number sequence of length n.

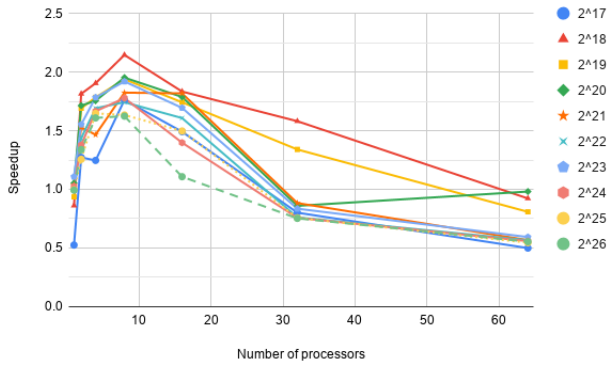

Figure 17: Speedup as a function of the numbers of processors, p, for all investigated values of n.

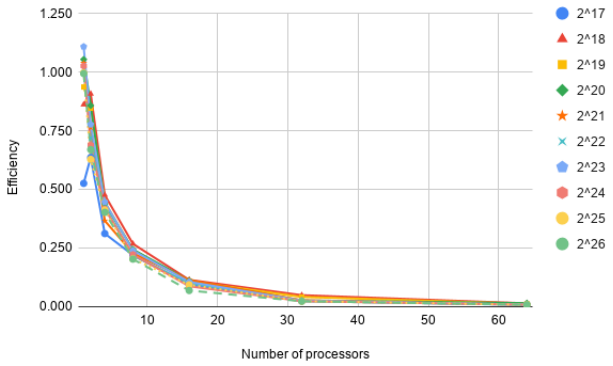

Figure 18: Efficiency as a function of the numbers of processors, p, for all investigated values of n.

Size of Sequence	Number of Processors						
	1	2	4	8	16	32	64
2^{17}	0.030	0.012	0.013	0.009	0.010	0.019	0.031
2^{18}	0.054	0.026	0.025	0.022	0.026	0.030	0.051
2^{19}	0.100	0.055	0.053	0.048	0.054	0.070	0.116
2^{20}	0.193	0.119	0.116	0.104	0.114	0.236	0.207
2^{21}	0.391	0.266	0.277	0.223	0.223	0.460	0.716
2^{22}	0.833	0.587	0.499	0.484	0.525	1.108	1.488
2^{23}	1.777	1.268	1.104	1.026	1.162	2.356	3.315
2^{24}	3.747	2.790	2.308	2.162	2.750	5.073	6.875
2^{25}	7.979	6.332	4.828	4.868	5.311	10.502	14.574
2^{26}	16.605	12.321	10.255	10.159	14.891	21.904	29.759

Table 4: Speedup for Parallel Bitonic Sorting algorithm.

Size of Sequence	Number of processors						
	1	2	4	8	16	32	64
2^{17}	0.526	0.636	0.312	0.220	0.093	0.025	0.008
2^{18}	0.861	0.907	0.476	0.268	0.115	0.049	0.014
2^{19}	0.936	0.846	0.446	0.242	0.109	0.042	0.013
2^{20}	1.053	0.857	0.439	0.244	0.112	0.027	0.015
2^{21}	1.038	0.764	0.367	0.228	0.114	0.028	0.009
2^{22}	1.013	0.718	0.423	0.218	0.100	0.024	0.009
2^{23}	1.108	0.777	0.446	0.240	0.106	0.026	0.009
2^{24}	1.026	0.689	0.416	0.222	0.087	0.024	0.009
2^{25}	0.997	0.628	0.412	0.204	0.094	0.024	0.009
2^{26}	0.994	0.670	0.402	0.203	0.069	0.024	0.009

Table 5: Efficiency for Parallel Bitonic Sorting algorithm.

Size of Sequence	Number of processors						
	1	2	4	8	16	32	64
2^{17}	0.030	0.025	0.050	0.071	0.168	0.624	2.000
2^{18}	0.054	0.052	0.098	0.175	0.409	0.949	3.251
2^{19}	0.100	0.111	0.210	0.387	0.861	2.239	7.419
2^{20}	0.193	0.237	0.463	0.832	1.818	7.561	13.244
2^{21}	0.391	0.532	1.108	1.782	3.575	14.718	45.802
2^{22}	0.833	1.174	1.995	3.873	8.398	35.458	95.207
2^{23}	1.777	2.535	4.415	8.206	18.588	75.400	212.174
2^{24}	3.747	5.579	9.230	17.295	43.994	162.336	439.993
2^{25}	7.979	12.663	19.310	38.943	84.977	336.050	932.719
2^{26}	16.605	24.642	41.022	81.274	238.250	700.913	1904.606

Table 6: Cost for Parallel Bitonic Sorting algorithm.

Number of Processors	Bitonic Sort	Sample Sort	Merge Sort
1	16.474	24.467	28.812
2	9.529	15.994	23.745
4	7.754	12.113	21.967
8	7.714	15.853	34.827
16	10.930	30.492	32.422
32	16.485	45.568	30.824

Table 7: Running Time (in seconds) for parallel implementation of the Bitonic Sort,Merge Sort, Sample Sort for an input sequence of length $n = 2^{26}$.

References

[1] Lang, H.W. Bitonic Sort, Hochschule Flensburg: https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm (1997, updated 2018)

[2] M. Ajtai, J. Komlos, E. Szemeredi: *An O(n log n) Sorting Network*. Proceedings of the 25th ACM Symposium on Theory of Computing, 1-9 (1983)

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: *Introduction to Algorithms*. 2nd edition, The MIT Press (2001)

[4] J. Mellor-Crummey. *COMP 322: Fundamentals of Parallel Programming, Lecture 28: Bitonic sort*, Rice University, Houston, Texas. https://wiki.rice.edu/confluence/download/attachments/4435861/comp322-s12-lec28-slides-JMC.pdf?version=1&modificationDate=1333163955158.

[5] S. Venugopal, V. Cohen, B. Hillman, K. Suarez. Byrne Students Summer Research 2012 Rutgers School of Arts and Sciences, Dpt. CS. https://www.cs.rutgers.edu/~venugopa/parallel_summer2012/bitonic_overview.html

[6] M. Cole, Lecture Slides *Design and Analysis of Parallel Algorithms*, School of Informatics, University of Edinborough. http://www.inf.ed.ac.uk/teaching/courses/dapa/overheads.pdf.

[7] S. Skiena. *The Algorithm Design Manual*, 2nd edition. Springer Publishing Company Inc. 2008.

[8] Lang, H.W. Bitonic sorting network for n not a power of 2, Hochschule Flensburg: https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm (1997, updated 2018)

[9] K. J. Liszka and K. E. Batcher, "A Generalized Bitonic Sorting Network," 1993 International Conference on Parallel Processing - ICPP'93, Syracuse, NY, USA, 1993, pp. 105-108.

[10] T. Nakatani, S. -. Huang, B. W. Arden and S. K. Tripathi, "K-way bitonic sort," in IEEE Transactions on Computers, vol. 38, no. 2, pp. 283-288, Feb. 1989.

[11] Wang, Biing-Feng et al. "Bitonic Sort with an Arbitrary Number of Keys." ICPP (1991).

[12] Nassimi and Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," in IEEE Transactions on Computers, vol. C-28, no. 1, pp. 2-7, Jan. 1979.

[13] Velusamy, Kaushik Rolinger, Thomas Mcmahon, Janice Simon, Tyler. (2018). Exploring Parallel Bitonic Sort on a Migratory Thread Architecture. 10.1109/HPEC.2018.8547568.

[14] D. Nassimi and S. Sahni. 1979. Bitonic Sort on a Mesh-Connected Parallel Computer. IEEE Trans. Comput. 28, 1 (January 1979), 2–7. DOI:https://doi.org/10.1109/TC.1979.1675216

[15] Peters H, Schulz-Hildebrandt O, Luttenberger n. Fast in-place, comparison-based sorting with CUDA: A study with bitonic sort. Concurr. Comput. : Pract. Exper. May 2011; 23(7):681–693, doi:10.1002/cpe.1686. URL http://dx.doi.org/10.1002/cpe.1686.

[16] Peters H, Schulz-Hildebrandt O, Luttenberger N. A novel sorting algorithm for many-core architectures based on adaptive bitonic sort. 26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012, 2012; 227–237, doi:10.1109/IPDPS.2012.30. URL http://dx.doi.org/ 10.1109/IPDPS.2012.30

[17] Bilardi G, Nicolau A. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. Technical Report, Ithaca, NY, USA 1986

[18] Vipin K. *Introduction to Parallel Computing*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc. USA 2002.

[19] Roberto A. C. Parallel Merge Sort with MPI: https://github.com/racorretjer/Parallel-Merge-Sort-with-MPI/blob/master/merge-mpi.c

[20] Dheeraj B. Parallel Sample Sort with MPI: http://www.cse.iitd.ernet.in/~dheerajb/MPI/codes/day-3/c/samplesort.c