Parallel Bitonic Sort

Gary Steelman,

Missouri University of Science and Technology

Computer Science 387, Dr. Ercal

Contents

[Background 3](#_Toc288759079)

[Method 4](#_Toc288759080)

[Implementation Specifics 4](#_Toc288759081)

[Swapping Algorithm 4](#_Toc288759082)

[Local Sort Algorithm 5](#_Toc288759083)

[Data Collection and Reports 5](#_Toc288759084)

[Calculation of Average Run-Times 5](#_Toc288759085)

[Calculation of Speedup and Efficiency 6](#_Toc288759086)

[Data 7](#_Toc288759087)

[Average Run-Times 7](#_Toc288759088)

[Speedup Ratios 8](#_Toc288759089)

[Approximate Speedup Formula 8](#_Toc288759090)

[Efficiency Ratios 8](#_Toc288759091)

[Approximate Efficiency Formula 8](#_Toc288759092)

[First 10 Sorted Numbers after Index 100,000 for N=400K 8](#_Toc288759093)

[First 10 Sorted Numbers after Index 200,000 for N=400K 8](#_Toc288759094)

[Best Run-Time for P=16 and N=128M 8](#_Toc288759095)

[Analysis of Data 9](#_Toc288759096)

[Appendix 10](#_Toc288759097)

[Algorithm 1 10](#_Toc288759098)

[Algorithm 2 10](#_Toc288759099)

[Algorithm 3 10](#_Toc288759100)

# Background

Sorting is one of the most studied areas in computer science. The amount of data stored has rapidly increased and quick access to data is desired. One of the ways to speed up access to data is to utilize known properties of the data set. It is for this reason that sorting is useful. If a data set can be assumed to be sorted, we can write algorithms based on that assumption and utilize the special properties.

There are many sorting algorithms: each one varying in the assumptions it makes about the data set it is sorting, each one varying in its strategy for iterating over the set.

The parallel bitonic sort is an extremely fast sorting algorithm as long as the communication overhead across the network can be minimized. This sort utilizes the fact that a set of data is bitonic, that is

This ordering of data in the set *S* allows a repeated binary swap and merge of subsets of the data with the guarantee that the resulting subsets will also be bitonic. This property is further utilized to guarantee that the resulting subsets of the input set *S* are stored in ascending order in ascending processors. Thus the set is considered sorted.

# Method

## Implementation Specifics

The bitonic sort was implemented in C++ using the MPI library. A C++ Standard Library vector<bool> was used to hold the numbers for each process. The number of processes, P, was varied from execution to execution, and the number of numbers sorted, N, was also varied.

Each process was given a uniformly sized number of numbers to sort:

Each process received numbers in the range .

The data set of size N was either read in from a file or had values generated randomly at run-time using the rand() function from the C++ STL.

### Swapping Algorithm

This implementation of the parallel bitonic sort is the result of implementation of three separate algorithms and the quickest one being used for the final reporting of times. A short description of the algorithms follows.

1. The first algorithm attempted to send a percentage of the compareLow call’s data to compareHigh. compareHigh would replace numbers in its set if the numbers sent were greater than the numbers it held. compareHigh would then send the replaced numbers back to compareLow. This process was repeated until no replacements were made or the entire sets were swapped. This algorithm was not used in final implementation because it did not scale well. Sorting of small sets (<100K) was very quick, but larger sets became very slow to sort. The communication overhead was much larger than expected.
2. The second algorithm has compareLow send its maximum value of it set to compareHigh. compareHigh sends compareLow all values in its set less than the maximum received. compareLow concatenates that set with its own and performs a local sort. compareLow then sends as many numbers as it received back to compareHigh from the high end of its set and compareHigh overwrites the numbers it sent with those received. This algorithm was not used in final implementation because, as in Algorithm 1, the communication overhead caused it to be slow.
3. The third, and final, algorithm is perhaps at first glance the most naïve algorithm. compareLow sends its entire data set to compareHigh. compareHigh concatenates its data set with the received set and performs a local sort. compareHigh sends the lower half of the resulting set to compareLow, who overwrites its local set. This is the algorithm used in final implementation as it avoids as much communication overhead as possible as a trade for a much more complex local sort.

### Local Sort Algorithm

The local sort algorithm was tested as two different algorithms and the quicker of the two was used in final implementation.

1. The C++ Standard Library sort() function was initially used as a place-holder until the entire algorithm was written. The intention was to replace this with merge sort, but my implementation of merge sort resulted in massively slower (three times) run-times.
2. Merge sort was implemented and tested versus the C++ Standard Library sort(). My implementation of merge sort was much slower than the STL sort() function, and thus the sort() function was used instead.

## Data Collection and Reports

### Calculation of Average Run-Times

Each (N,P) pair was executed three times and the average of the results were used as the reported figures in the Data section. The average was calculated as follows:

N varied as follows:

1. 400 thousand
2. 16 million
3. 32 million
4. 64 million
5. 128 million

P varied as follows:

1. 1
2. 2
3. 4
4. 8
5. 16
6. 32

### Calculation of Speedup and Efficiency

The Speedup of a parallel program is the ratio of the time the program would take to complete in a serial implementation versus the time taken to complete in its parallel implementation. All figures reported as “Speedup” figures are calculated as such:

Where normally, is the amount of time in seconds that the fastest sequential implementation of the algorithm can produce. For this project, is equal to the time with P=1. is then the amount of time taken for each run with P > 1 (ie, P=2, 4, 8, or 16).

The efficiency of a parallel program indicates the percentage of time every process is active. Ideally this figure would be 1, but this is almost never the case. All figures reported as “Efficiency” figures are calculated as such:

Where is the same as described in the calculation of Speedup and P is the number of processes used.

# Data

## Average Run-Times

Required for Part 1

The data in the following table show the average amount of time, in seconds, taken to complete the sort for each (N,P) pair.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Run-Times | | | | | | | |
|  |  | P | | | | | |
|  |  | **1** | **2** | **4** | **8** | **16** | **32** |
| N | **4.00E+05** |  |  |  | 0.47422 |  |  |
| **1.60E+07** | 12.22503 | 36.99703 | 35.26747 | 27.15473 | 19.51063 | 18.34813 |
| **3.20E+07** | 21.4264 | 59.1808 | 56.8567 | 36.97627 | 36.03343 | 36.96513 |
| **6.40E+07** | 35.26327 | 111.8257 | 118.9863 | 77.2699 | 64.39577 | 77.99423 |
| **1.28E+08** | 102.2943 | 265.459 | 264.646 | 162.062 | 135.0333 | 157.8923 |

## Speedup Ratios

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Speedup | | | | | | |  |
|  |  | P | | | | |  |
|  |  | **1** | **2** | **4** | **8** | **16** | **32** |
| N | **4.00E+05** |  |  |  |  |  |  |
| **1.60E+07** | 1 | 0.330433 | 0.346638 | 0.450199 | 0.626583 | 0.666282 |
| **3.20E+07** | 1 | 0.36205 | 0.376849 | 0.579464 | 0.594626 | 0.579638 |
| **6.40E+07** | 1 | 0.315341 | 0.296364 | 0.456365 | 0.547602 | 0.452127 |
| **1.28E+08** | 1 | 0.385349 | 0.386533 | 0.631205 | 0.757549 | 0.647874 |

### Approximate Speedup Formula

Required for Part 3

The approximate speedup for this implementation of the bitonic sort assumes that communication of a message of size 1 takes 1 operation to be sent and received and thus a message of size N takes N steps.

Asymptotic speedup behavior when the algorithm has

* N=P is extremely inefficient. The speedup is a very small ratio.
* N>>P has an O(1) complexity and is almost constant.

## Efficiency Ratios

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Efficiency | | | | | | |  |
|  |  | P | | | | |  |
|  |  | **1** | **2** | **4** | **8** | **16** | **32** |
| N | **4.00E+05** |  |  |  |  |  |  |
| **1.60E+07** | 1 | 0.165216 | 0.086659 | 0.056275 | 0.039161 | 0.020821 |
| **3.20E+07** | 1 | 0.181025 | 0.094212 | 0.072433 | 0.037164 | 0.018114 |
| **6.40E+07** | 1 | 0.157671 | 0.074091 | 0.057046 | 0.034225 | 0.014129 |
| **1.28E+08** | 1 | 0.192674 | 0.096633 | 0.078901 | 0.047347 | 0.020246 |

### Approximate Efficiency Formula

Required for Part 3

Asymptotic efficiency behavior when then algorithm has

* N=P is extremely inefficient. The efficiency is a very small ratio.
* N>>P has almost an O(1) complexity because the Ps drop out.

## First 10 Sorted Numbers after Index 100,000 for N=400K

Required for Part 2.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 222352 | 222354 | 222355 | 222356 | 222358 |
| 222359 | 222361 | 222362 | 222365 | 222367 |

## First 10 Sorted Numbers after Index 200,000 for N=400K

Required for Part 2.2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 479822 | 479824 | 479835 | 479837 | 479841 |
| 479845 | 479847 | 479848 | 479852 | 479853 |

## Best Run-Time for P=16 and N=128M

Required for Bonus

The best run-time achieved by this implementation of the parallel bitonic sort for P=16 and N=128M was

155.071 seconds

# Analysis of Data

Required for Part 4

1. For a fixed P, increasing N by a factor of two results in a relatively stable speedup ratio for all tested P and N. This is consistent with the equation derived for the approximate speedup. Because N>>P for all tested cases, the term drops out rather quickly.
2. For a fixed P, increasing N by a factor of two results in a relatively stable efficiency ratio for all tested P and N. This is consistent with the equation derived for the approximate efficiency. The reason for is similar to question (1). Because N>>P the term and the multiplication by 1/P drop out.
3. For a fixed N, increasing P by a factor of two results in a decrease in efficiency by a factor of two for all tested P and N. This is consistent with the equation derived for the approximate efficiency. The reason for this can be seen easily. Since N>>P the term drops out readily, and the multiplication by 1/2 results in a decrease in efficiency by a factor of two.