CS 6230: Homework 4

Christopher Mertin

Due Date: March 13, 2016

Question 1: Bitonic Sorting: Multiple Input Sizes . . . .

How would you modify the bitonic sort algorithm if the number of input keys are arbitrary. For example, assume that as input you get $n = rank$ keys per process. While you are not required to maintain the same number of keys per process for all stages of the bitonic sort, you should not explicitly balance the distribution of the sort before sorting. In other words, any load balancing must happen as part of the bitonic sort.

Hint: While merging two bitonic sequences of unequal lenths, what needs to be done to ensure that the resulting sequences are bitonic and satisfy the $low \leq high$ requirement.

Solution: A bitonic sequence is built such as a "rise then falls" or a "falls than rises" sequence. This point in the middle is known as the *pivot point* of the array. You can perform it on an arbitrary sequence on p nodes in the following way.

First, you would have to sort the invididual arrays on each node. Then, after sorting, you would have to compare the values on the nodes. For example, p_1 would compare its values for the pivot point of p_2, and vice-versa.

When a swap is supposed to take place, each node has no idea how many items the other node is going to send/receive since they are uneven. Therefore, before sending and receiving, p_1 would have to tell p_2 how many elements it was going to send, and p_2 would have to tell p_1.

From here, you would have to figure out how to deal with the different number of elements. The easiest approach would be to resize the arrays to how much more memory you're going to need before initiating the send/receive between the nodes.

To achieve proper load balancing and make it more efficient, you would resize the array to be $\left\lceil \frac{a+b}{2} \right\rceil$ where a and b are the size of the elements on p_1 and p_2, respectively. Then, when you send the data you would send it to this new size such that each of the two nodes would have the same number of elements. In sorting all the data, each node would wind up having *roughly* the same number of elements.

Question 2: Bitonic Sort . . . .

Implement an MPI parallel bitonic sort algorithm. You can assume that the number of input keys per process will be the same. The number of processes can be arbitrary. Write a driver routine called `sort` that takes one argument, the size of the array to be sorted (initialized to random integers). On the write-up, give pseudocode for your algorithm, and report wall clock time/core/n results for $n/p = 1, 10, 100, 1000, 1M$ elements using 16 MPI tasks per node for as large a number of nodes as possible. At least try for $1, 2, 3, 4$ nodes. Report weak and strong scaling results.

Solution: The implemented solution to this problem first performs a *local* bitonic sort on the data for each node, which can be seen in Algorithm 1. After which, it calls another funciton which sorts it via the bitonic sort algorithm between each node with Algorithm 3. The pseudocode for each was taken from a source and converted into `C++` code. The respective sources are labeled next to each algorithm.

The results for the timings can be found in Table 1. To obtain these timings, each task was run 10 times and the average timings were taken. The strong scaling results can be seen in Figure 1 while the results for the weak scaling can be found in Figure 2.

Algorithm 1 $BitonicSort(low, high, up, A)$ [1]

```
Input: lowest variable to sort, highest variable to sort, $A \in \mathbb{R}^{n\times 1}$, boolean up for ascending order
Output: $A \in \mathbb{R}^{n\times 1}$ sorted
  if $high > 1$ then
    $midPoint \leftarrow high/2$
    $BitonicSort(low, midPoint, A,$ not $up)$
    $BitonicSort(low + midPoint, high - midPoint, A, up)$
    $BitonicMerge(low, high, A, up)$ [see Algorithm 2]
  end if
```

Algorithm 2 $BitonicMerge(low, high, up, A)$ [1]

```
Input: lowest variable to sort, highest variable to sort, $A \in \mathbb{R}^{n\times 1}$, boolean up for ascending order
  if $high > 1$ then
    $m \leftarrow$Greatest Powrer of 2 less than $high$
    for $i = low$ to $low + high - m - 1$ do
      if $(A_{low} > A_{high}) = up$ then
        $swap(A_{low}, A_{high})$
      end if
    end for
    $BitonicMerge(low, m, A, up)$
    $BitonicMerge(low + m, high - m, A, up)$
  end if
```

Algorithm 3 Parallel Bitonic Sorting [2]

```
Input: $A \in \mathbb{R}^{n\times 1}$ (sorted), $numNodes$, $nodeID$
Output: $A \in \mathbb{R}^{n\times 1}$ sorted between the nodes
  for $i = 1$ to $\log_2(numNodes)$ do
    $nodeBit \leftarrow$ Most significant $(d-i)$ bits of $nodeID$
    for $j = i-1$ to 0 do
      if ($nodeBit \in$ even and $j^{th}$ bit of $nodeID = 0$) or ($nodeID \in$ odd and $j^{th}$ bit of $nodeID = 1$)
      then
        $CompareLow(j)$ {Starts merging from the high end between nodes}
      else
        $CompareHigh(j)$ {Starts merging from the low end between nodes}
      end if
      $BitonicSort(A)$ {sequential}
    end for
  end for
```

Table 1: Average Bitonic Sort Timings

Nodes	10^0	10^1	10^2	10^3	10^4	10^5	10^6	10^7
1	0.0001	0.0003	0.0007	0.0050	0.0521	0.4597	3.1554	37.0015
2	0.0002	0.0005	0.0011	0.0074	0.0932	1.0341	4.7064	53.4787
4	0.0004	0.0007	0.0026	0.0170	0.1507	1.1991	6.6821	73.5433
8	0.0005	0.0022	0.0040	0.0242	0.1940	1.2787	8.8620	96.9317
16	0.0006	0.0032	0.0288	0.0378	0.2421	1.5918	10.9514	125.2008
32	0.0005	0.0037	0.0230	0.4341	0.3226	1.7681	14.0641	157.7905

† Each node was assigned 16 MPI Processes

‡ Given values are in seconds

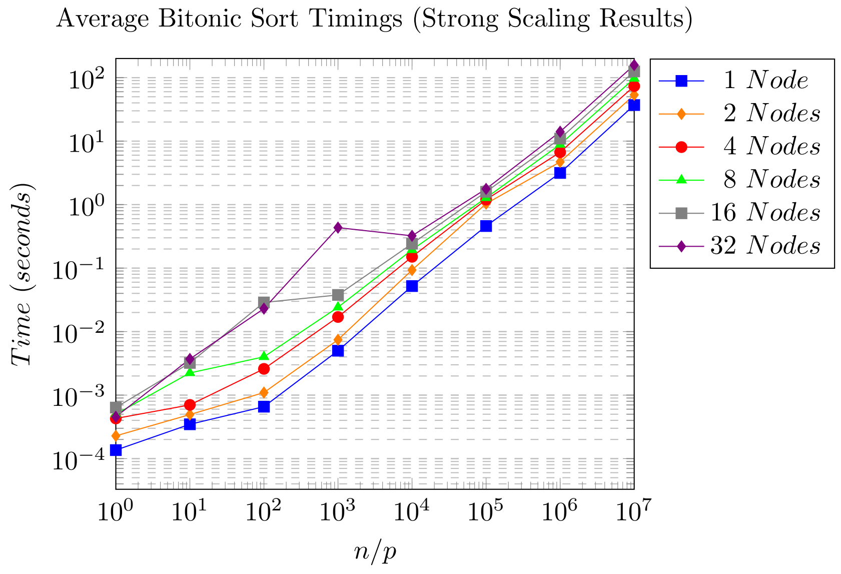

Figure 1: Each node was assigned 16 MPI Processes

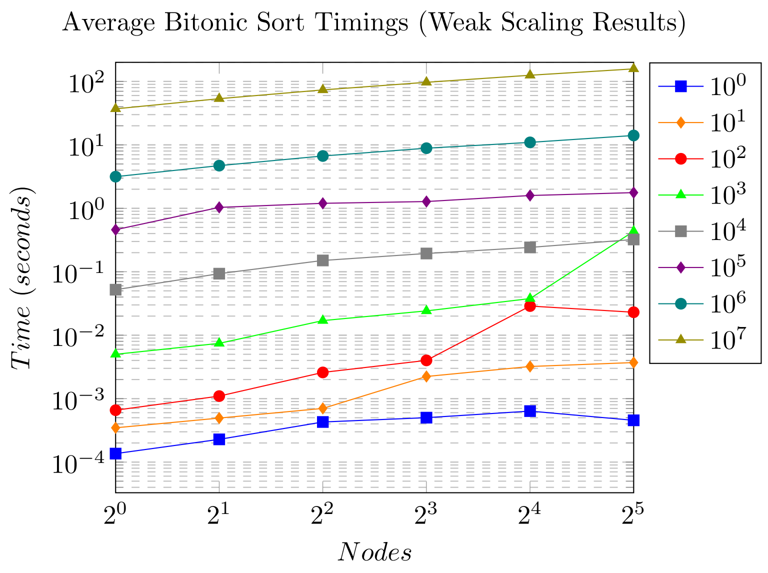

Figure 2: Each node was assigned 16 MPI Processes

References

[1] H.W. Lang, FH Flensburg, `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm`

[2] Fikret Ercal, Missouri University of S&T, `http://web.mst.edu/~ercal/387/P3/pr-proj-3.pdf`