Parallel Bitonic Sort Implementation

CSE 702 Programming Massively Parallel Systems
Course Instructor – Dr. Russ Miller
Prepared by – Sajid Khan(UB person no: 50248743)
Agenda

• Introduction to Bitonic Sort
• MPI Implementation
• Example Comparison
• Results and Analysis
• Challenges
• Outcome
• References
What is Bitonic Sort?

• To understand Bitonic Sort, we must first understand what is Bitonic Sequence and how to make a given sequence Bitonic.

• A sequence is called Bitonic if it is first increasing, then decreasing. In other words, an array arr[0..n-i] is Bitonic if there exists an index i where 0<=i<=n-1 such that

\[ x_0 \leq x_1 \ldots \leq x_i \text{ and } x_i \geq x_{i+1} \ldots \geq x_{n-1} \]

• A sequence, sorted in increasing order is considered Bitonic with the decreasing part as empty.
How to make a sequence Bitonic?

Time complexity = (Logn)^2 comparisons
MPI Implementation
<table>
<thead>
<tr>
<th>Step No.</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>2</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>3</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>4</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>5</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>6</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>H</td>
</tr>
</tbody>
</table>
How to find the right pair?

for (i = 0; i < dimensions; i++) { // dimensions = log(n), iterates on stages
    for (j = i; j >= 0; j--) { // combinations in each stage
        if (((process_rank >> (i + 1)) % 2 == 0 && (process_rank >> j) % 2 == 0)
            || ((process_rank >> (i + 1)) % 2 != 0 && (process_rank >> j) % 2 != 0)) {
            CompareLow(j);
        } else {
            CompareHigh(j);
        }
    }
}
Example comparison

• Consider the first comparison, where process rank in binary is 000
• It finds the partner using bitwise EXOR operation.
  
  \[
  \text{partner’s rank} = \text{process\_rank} \ ^ \ (1\ll j)
  \]

• Where j is comparison bit varies from [0, logn) and n is the number of processors.
What happens in a Comparison?

- $\text{sends} \{2,3,6,7\} > \text{min}$
- $\text{max} = 7$
- $\text{min} = 1$
- $\text{sends} \{1,4,5\} < \text{max} = 7$

**3, 6, 2, 7**
*2,3,6,7 Sorted*

**4, 5, 1, 8**
*1,4,5,8 Sorted*

- Collects all the elements and keeps the lowest half
  1,2,3,4
- Collects all the elements and keeps the highest half
  5,6,7,8
Few Results – 4 million keys

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Time in sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.753486</td>
</tr>
<tr>
<td>4</td>
<td>0.558978</td>
</tr>
<tr>
<td>8</td>
<td>0.484986</td>
</tr>
<tr>
<td>16</td>
<td>0.383336</td>
</tr>
<tr>
<td>32</td>
<td>0.232034</td>
</tr>
<tr>
<td>64</td>
<td>0.156750</td>
</tr>
<tr>
<td>128</td>
<td>0.102977</td>
</tr>
</tbody>
</table>
## 10,000 keys

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Time in sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.00272</td>
</tr>
<tr>
<td>4</td>
<td>0.00199</td>
</tr>
<tr>
<td>8</td>
<td>0.00149</td>
</tr>
<tr>
<td>16</td>
<td>0.00242</td>
</tr>
<tr>
<td>32</td>
<td>0.00381</td>
</tr>
</tbody>
</table>

### Graph: Run time in sec vs. Number of Nodes

- **X Axis:** Number of Nodes
- **Y Axis:** Run time in sec

The graph shows the relationship between the number of nodes and the run time for 10,000 keys.
40,000 keys

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Time in sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.007921</td>
</tr>
<tr>
<td>4</td>
<td>0.007094</td>
</tr>
<tr>
<td>8</td>
<td>0.004884</td>
</tr>
<tr>
<td>16</td>
<td>0.005504</td>
</tr>
<tr>
<td>32</td>
<td>0.006473</td>
</tr>
</tbody>
</table>
Speed up factor

### Speed up for 10,000 keys

- 2 nodes: 0.76
- 4 nodes: 1.04
- 8 nodes: 1.39
- 16 nodes: 0.85
- 32 nodes: 0.54

### Speed up for 40,000 keys

- 2 nodes: 1.04
- 4 nodes: 1.16
- 8 nodes: 1.69
- 16 nodes: 1.5
- 32 nodes: 1.28
Challenges

• Allocation of higher order nodes 128, 256.
• Difficulty in debugging the Algorithmic flaws.
Outcome

• Found out how parallel implementation can reduce runtime by significant amount compared to sequential runs.
• How runtime behaves as number of cores is increased.
• Observe speedup in latency with Amdahl’s law.
• Knowledge of MPI, Open MPI.
References

• https://ubccr.freshdesk.com/support/solutions/articles/13000026245-tutorials-and-training-documents
• https://ubccr.freshdesk.com/support/solutions/articles/5000688140-submitting-a-slurm-job-script
• https://cse.buffalo.edu/faculty/miller/teaching.shtml
• https://www.geeksforgeeks.org/bitonic-sort/
• Find my code on /github.com/sajid912/MPI-Bitonic-Sort
Thank you!!