COMP 322: Fundamentals of Parallel Programming

Lecture 28: Bitonic Sort

John Mellor-Crummey
Department of Computer Science, Rice University
johnmc@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Introduction

• Why study sorting?
  — one of the most common operations performed on computers

• Sorting algorithm attributes
  — internal vs. external
    – internal: data fits in memory
    – external: uses tape or disk
  — comparison-based or not
    – comparison sort
      basic operation: compare elements and exchange as necessary
      $\Theta(n \log n)$ comparisons to sort $n$ numbers
    – non-comparison-based sort
      e.g. radix sort based on the binary representation of data
      $\Theta(n)$ operations to sort $n$ numbers
  — parallel vs. sequential

Today’s focus
Bitonic sort: internal, comparison-based, parallel sort
Sorting Network

- Network of comparators designed for sorting
- Comparator: two inputs $x$ and $y$; two outputs $x'$ and $y'$
  - Types
    - Increasing (denoted $\oplus$): $x' = \min(x,y)$ and $y' = \max(x,y)$
      
      \[
      \begin{array}{c}
      x \\
      \oplus
      \end{array}
      \quad \min(x,y)
      \]
      
      \[
      \begin{array}{c}
      y \\
      \oplus
      \end{array}
      \quad \max(x,y)
      \]
    - Decreasing (denoted $\ominus$): $x' = \max(x,y)$ and $y' = \min(x,y)$
      
      \[
      \begin{array}{c}
      x \\
      \ominus
      \end{array}
      \quad \max(x,y)
      \]
      
      \[
      \begin{array}{c}
      y \\
      \ominus
      \end{array}
      \quad \min(x,y)
      \]
  
- Sorting network speed is proportional to its depth
Sorting Networks

- Network structure: a series of columns
- Each column consists of a vector of comparators (in parallel)
- Sorting network organization:
Example: Bitonic Sorting Network

- **Bitonic sequence**
  - two parts: increasing and decreasing
    - \(1,2,4,7,6,0\): first increases and then decreases (or vice versa)
  - cyclic rotation of a bitonic sequence is also considered bitonic
    - \(8,9,2,1,0,4\): cyclic rotation of \(0,4,8,9,2,1\)

- **Bitonic sorting network**
  - sorts \(n\) elements in \(\Theta(\log^2 n)\) time
  - network kernel: rearranges a bitonic sequence into a sorted one
Bitonic Split (Batcher, 1968)

Let $s = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ be a bitonic sequence such that

- $a_0 \leq a_1 \leq \cdots \leq a_{n/2-1}$, and
- $a_{n/2} \geq a_{n/2+1} \geq \cdots \geq a_{n-1}$

Consider the following subsequences of $s$

$s_1 = \langle \min(a_0,a_{n/2}), \min(a_1,a_{n/2+1}), \ldots, \min(a_{n/2-1},a_{n-1}) \rangle$

$s_2 = \langle \max(a_0,a_{n/2}), \max(a_1,a_{n/2+1}), \ldots, \max(a_{n/2-1},a_{n-1}) \rangle$

Sequence properties

- $s_1$ and $s_2$ are both bitonic
- $\forall x \forall y \ x \in s_1, y \in s_2 , \ x < y$

Apply recursively on $s_1$ and $s_2$ to produce a sorted sequence

Works for any bitonic sequence, even if $|s_1| \neq |s_2|$
Splitting Bitonic Sequences - I

Sequence properties

$s_1$ and $s_2$ are both bitonic

$\forall x \forall y \ x \in s_1, \ y \in s_2, \ x < y$
Sequence properties

$s_1$ and $s_2$ are both bitonic

$\forall x \forall y \ x \in s_1, y \in s_2, x < y$
Splitting Bitonic Sequences - I

Sequence properties

\(s_1\) and \(s_2\) are both bitonic

\[\forall x \ \forall y \ x \in s_1, y \in s_2, x < y\]
Splitting Bitonic Sequences - I

Sequence properties

\( s_1 \) and \( s_2 \) are both bitonic

\[ \forall x \forall y \ x \in s_1, y \in s_2, x < y \]
Sequence properties

$s_1$ and $s_2$ are both bitonic

$\forall x \forall y \ x \in s_1, y \in s_2, x < y$
Sequence properties

$s_1$ and $s_2$ are both bitonic

$\forall x \ \forall y \ x \in s_1, y \in s_2, x < y$
### Bitonic Merge

Sort a bitonic sequence through a series of bitonic splits

**Example:** use bitonic merge to sort 16-element bitonic sequence

**How:** perform a series of \( \log_2 16 = 4 \) bitonic splits

<table>
<thead>
<tr>
<th>Original sequence</th>
<th>3</th>
<th>5</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>20</th>
<th>95</th>
<th>90</th>
<th>60</th>
<th>40</th>
<th>35</th>
<th>23</th>
<th>18</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Split</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>0</td>
<td>95</td>
<td>90</td>
<td>60</td>
<td>40</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>2nd Split</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>0</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>9</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>20</td>
<td>95</td>
<td>90</td>
<td>60</td>
<td>40</td>
</tr>
<tr>
<td>3rd Split</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>10</td>
<td>9</td>
<td>14</td>
<td>12</td>
<td>18</td>
<td>20</td>
<td>35</td>
<td>23</td>
<td>60</td>
<td>40</td>
<td>95</td>
<td>90</td>
</tr>
<tr>
<td>4th Split</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>18</td>
<td>20</td>
<td>23</td>
<td>35</td>
<td>40</td>
<td>60</td>
<td>90</td>
<td>95</td>
</tr>
</tbody>
</table>
Sorting via Bitonic Merging Network

- Sorting network can implement bitonic merge algorithm
  - bitonic merging network

- Network structure
  - \( \log_2 n \) columns
  - each column
    - \( n/2 \) comparators
    - performs one step of the bitonic merge

- Bitonic merging network with \( n \) inputs: \( \oplus \text{BM}[n] \)
  - produces an increasing sequence

- Replacing \( \oplus \) comparators by \( \Theta \) comparators: \( \Theta \text{BM}[n] \)
  - produces a decreasing sequence
Bitonic Merging Network, $⊕$ BM[16]

- Input: bitonic sequence
  - input wires are numbered $0, 1, \ldots, n - 1$ (shown in binary)
- Output: sequence in sorted order
- Each column of comparators is drawn separately
Bitonic Sort

How do we sort an unsorted sequence using a bitonic merge?

Two steps

- Build a bitonic sequence
- Sort it using a bitonic merging network
Building a Bitonic Sequence

• Build a single bitonic sequence from the given sequence
  — any sequence of length 2 is a bitonic sequence.
  — build bitonic sequence of length 4
    - sort first two elements using $\oplus BM[2]$
    - sort next two using $\Theta BM[2]$

• Repeatedly merge to generate larger bitonic sequences
  — $\oplus BM[k]$ & $\Theta BM[k]$: bitonic merging networks of size $k$
# Building a Bitonic Sequence

**Input:** sequence of 16 unordered numbers  
**Output:** a bitonic sequence of 16 numbers
Bitonic Sort, n = 16

- First 3 stages create bitonic sequence input to stage 4
- Last stage ($\oplus BM[16]$) yields sorted sequence
Complexity of Bitonic Sorting Networks

- Depth of the network is $\Theta(\log^2 n)$
  - $\log_2 n$ merge stages
  - $j^{th}$ merge stage depth is $\log_2 2^j = j$

  $$\text{depth} = \sum_{j=1}^{\log_2 n} \log_2 2^j = \sum_{j=1}^{\log_2 n} j = (\log_2 n + 1)(\log_2 n)/2 = \theta(\log^2 n)$$

- Each stage of the network contains $n/2$ comparators
- Complexity of serial implementation = $\Theta(n \log^2 n)$
Batcher’s Bitonic Sort

- **bmerge(s)**: recursively sort a bitonic sequence as follows
  1. compute $s_1$ and $s_2$ as shown earlier for ascending sort of $s$
     - both will be bitonic by Batcher’s Lemma
     - note: for a descending sort, just swap min & max
  2. recursively call bmerge on $s_1$ and $s_2$
  3. return $s = \text{concat}(\text{bmerge}(s_1), \text{bmerge}(s_2))$

- **bsort(s)**: create a bitonic sequence then sort it
  1. convert an arbitrary sequence $s$ into a bitonic sequence
     - sort $s[1 \ldots n/2]$ in place in ascending order (recursive call to sort)
     - sort $s[n/2+1 \ldots n]$ in place in descending order (recursive call to sort)
  2. after step 1, the sequence will be bitonic; sort it using bmerge($s$)
void bmerge(final int[] A, final int low, final int high, boolean asc) {
    if (high-low > 1) {
        final int mid = (low + high)/2;
        final int size = high - low + 1;
        forall (point[i]:[low:mid]) orderElementPair(A, i, i+size/2, asc);
        finish {
            async bmerge(A, low, mid, asc);
            async bmerge(A, mid+1, high, asc);
        } // finish
    } // if
} // bmerge

void bsort (final int[] A, final int low, final int high, boolean asc) {
    if (high-low > 1) {
        finish {
            final int mid = (low + high)/2;
            async bsort(A, low, mid, asc);
            async bsort(A, mid+1, high, !asc);
        } // finish
        bmerge(A, low, high, asc); // asc = true is for ascending order
    } // if
} // sort
Batcher's Bitonic Sort in NESL

```nesl
function bitonic_merge(a) =
if (#a == 1) then a
else
  let
    halves = bottop(a)
    mins = {min(x, y) : x in halves[0]; y in halves[1]};
    maxs = {max(x, y) : x in halves[0]; y in halves[1]};
  in flatten({bitonic_merge(x) : x in [mins,maxs]});

function bitonic_sort(a) =
if (#a == 1) then a
else
  let b = {bitonic_sort(x) : x in bottop(a)};
  in bitonic_merge(b[0]++reverse(b[1]));
```

Run this code at http://www.cs.rice.edu/~johnmc/nesl.html
References

- Adapted from slides “Sorting” by Ananth Grama
- Based on Chapter 9 of “Introduction to Parallel Computing” by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003
- http://www.cs.cmu.edu/~scandal/nesl/algorithms.html#sort