A Top-Down Parallel Semisort

Yan Gu
Julian Shun
Yihan Sun
Guy Blelloch

Carnegie Mellon University
What is semisort?

- **Input:**
  - An array of records with associated keys
  - Assume keys can be hashed to the range \([n^k]\)

- **Goal:**
  - All records with equal keys should be adjacent
What is semisort?

- **Input:**
  - An array of records with associated keys
  - Assume keys can be hashed to the range $[n^k]$

- **Goal:**
  - All records with equal keys should be adjacent
What is semisort?

- **Input:**
  - An array of records with associated keys
  - Assume keys can be hashed to the range $[n^k]$

- **Goal:**
  - All records with equal keys should be adjacent
  - Different keys are not necessarily sorted
  - Records with equal keys do not need to be sorted by their values
What is semisort?

<table>
<thead>
<tr>
<th>key</th>
<th>45</th>
<th>45</th>
<th>45</th>
<th>45</th>
<th>12</th>
<th>61</th>
<th>61</th>
<th>61</th>
<th>28</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

- **Input:**
  - An array of records with associated keys
  - Assume keys can be hashed to the range \([n^k]\)

- **Goal:**
  - All records with equal keys should be adjacent
  - Different keys are not necessarily sorted
  - Records with equal keys do not need to be sorted by their values
Why is parallel semisort important?

- The simulation of PRAM model – concurrent write
  [Valiant 1990]
  - Key: memory addresses
  - Value: operations

<table>
<thead>
<tr>
<th>Thread</th>
<th>Concurrent writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a[3]=71</td>
</tr>
<tr>
<td>2</td>
<td>a[1]=99</td>
</tr>
<tr>
<td>3</td>
<td>a[2]=19</td>
</tr>
<tr>
<td>4</td>
<td>a[3]=10</td>
</tr>
<tr>
<td>5</td>
<td>a[5]=50</td>
</tr>
<tr>
<td>6</td>
<td>a[3]=12</td>
</tr>
<tr>
<td>7</td>
<td>a[1]=16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread</th>
<th>Sorted operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a[3]=71</td>
</tr>
<tr>
<td>6</td>
<td>a[3]=12</td>
</tr>
<tr>
<td>5</td>
<td>a[5]=50</td>
</tr>
<tr>
<td>7</td>
<td>a[1]=16</td>
</tr>
<tr>
<td>2</td>
<td>a[1]=99</td>
</tr>
<tr>
<td>3</td>
<td>a[2]=19</td>
</tr>
</tbody>
</table>

Why is parallel semisort important?

- The map-(semisort-)-reduce paradigm
Why is parallel semisort important?

- The map-(semisort-)reduce paradigm
- Generate adjacency array for a graph

<table>
<thead>
<tr>
<th>Edge list</th>
<th>Sorted edge list</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3,5)</td>
<td>(3,5)</td>
</tr>
<tr>
<td>(1,7)</td>
<td>(3,7)</td>
</tr>
<tr>
<td>(2,3)</td>
<td>(3,6)</td>
</tr>
<tr>
<td>(3,6)</td>
<td>(5,4)</td>
</tr>
<tr>
<td>(5,4)</td>
<td>(1,6)</td>
</tr>
<tr>
<td>(3,7)</td>
<td>(1,7)</td>
</tr>
<tr>
<td>(1,6)</td>
<td>(2,3)</td>
</tr>
</tbody>
</table>

![Graph diagram]
Why is parallel semisort important?

- The map-(semisort-)reduce paradigm
- Generate adjacency array for a graph

Other applications:
- In database, the relational join operation
- Gather words that differ by a deletion in edit-distance application
- Collect shared edges based on endpoints in Delaunay triangulation
- Etc.
Attempts – Sequentially Hash Table With Open Addressing

Problem:
- Maintaining linked lists in parallel can be hard
Attempts – Sequentially
Pre-allocated array

key value

Arrays of values

key

value

12

9

52

11

19

8

44

31

56
Attempts - Parallelized Pre-allocated array

Problem

- Need to pre-count the number of each key
Attempts – In parallel

- Comparison-based sort
  - $O(n \log n)$ work
  - Not work-efficient 😞

- Radix-sort (probably the best work-efficient option previously)
  - $O(n^\epsilon)$ depth
  - Not highly-parallelized 😞
Attempts – In parallel

- R&R integer sort [Rajasekaran and Reif 1989]: sort $n$ records with keys in the range $[n]$ in $O(n)$ work and $O(\log n)$ depth
  - Linear work and logarithmic depth
  - Should map keys to range $[n]$  
  - Too much global data movement – practically inefficient
    - Hashing and packing – 1 time
    - Random radix sort – 1 time
    - Deterministic radix sort – 2 times

☹
How to design an efficient semisort?

- Theoretically efficient:
  - Linear work
  - Logarithmic depth

- Practically efficient:
  - Less data communication
  - Cache-friendly

- Space efficient:
  - Linear space
Our Top-Down Parallel Semisort Algorithm
Key insight: estimate key count from samples

- Once the count of each key is known, we can pre-allocate an array for each key.

- The exact number is hard to compute - estimate the upper bound by **sampling**
  - Those appearing many times: we could make reasonable estimations from the sample.
  - Those with few samples: hard to estimate precisely.

- Solution: Treat “heavy” keys and “light” keys differently.
Our parallel semisort algorithm

1. Select a sample $S$ of keys and sort it
   - Sample rate $\Theta(1/\log n)$

2. Partition $S$ into heavy keys and light keys
   - **Heavy**: appears $= \Omega(\log n)$ times; will be assigned an individual bucket
   - **Light**: appears $= O(\log n)$ times. We evenly partition the hash range to $n/\log^2 n$ buckets for them

3. Scatter each record into its associated bucket
   - The only global data communication

4. Semisort light key buckets
   - Performed locally

5. Pack and output
Heavy vs. Light...Why?

[Rajasekaran and Reif 1989] If the records are sampled with probability \( p = 1/\log n \), and for a key \( i \) which appears \( a_i \) times in the original array, and \( c_i \) times in the sample:

- \( c_i = \Omega(\log n) \), then \( a_i = \Theta(c_i \log n) \) w.h.p.
- \( c_i = O(\log n) \), then \( a_i = O(\log^2 n) \) w.h.p.

(Can be proved using Chernoff bounds)
Estimate upper bounds for the counts \( a_i \)

- Key insight: if the records are sampled with probability \( p = 1/ \log n \), and key \( i \) has:
  - \( c_i = \Omega(\log n) \) samples, then \( a_i = \Theta(c_i \log n) \) w.h.p.
  - \( c_i = O(\log n) \) samples, then \( a_i = O(\log^2 n) \) w.h.p.

- \( u_i = c' \max(\log^2 n, c_i \log n) \)
  - \( c' \) is a sufficiently large constant to provide the high probability bound
Estimate upper bounds for the counts $a_i$

- Key insight: if the records are sampled with probability $p = 1/\log n$, and key $i$ has:
  - $c_i = \Omega(\log n)$ samples, then $a_i = \Theta(c_i \log n)$ w.h.p.
  - $c_i = O(\log n)$ samples, then $a_i = O(\log^2 n)$ w.h.p.

- Extreme case: all samples are of the same key
  - $c_i = \frac{n}{\log n} \Rightarrow u_i = O(n)$
  - $c_i = 0 \Rightarrow u_i = O(\log^2 n)$
  - Require keys to be in range $[n/\log^2 n]$

- Solution: combine light keys
  - evenly partition the hash range to $n/\log^2 n$ intervals as buckets
Phase 1: Sampling and sorting

1. Select a sample $S$ of keys with probability $p = \Theta(1/\log n)$
2. Sort $S$
Phase 2: Array Construction

Sorted samples:

5 5 5 8 8 8 8 8 8 8 11 17 17 17 17 ......

Heavy keys

| keys | 8 20 65 ... |

Light keys

<table>
<thead>
<tr>
<th>Range</th>
<th>0-15</th>
<th>16-31</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys</td>
<td>5 11 17 21 26 31 ...</td>
<td></td>
</tr>
</tbody>
</table>

Counting & Filtering
# Phase 2: Array Construction

## Heavy Keys

<table>
<thead>
<tr>
<th>keys</th>
<th>$k_1$</th>
<th>$k_2$</th>
<th>$k_3$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td># samples</td>
<td>$c_1$</td>
<td>$c_2$</td>
<td>$c_3$</td>
<td>...</td>
</tr>
</tbody>
</table>

Array length: $f(c_1)$, $f(c_2)$, $f(c_3)$, ...

## Light Keys

<table>
<thead>
<tr>
<th>keys</th>
<th>$k'_1$</th>
<th>$k'_2$</th>
<th>$k'_3$</th>
<th>$k'_4$</th>
<th>$k'_5$</th>
<th>$k'_6$</th>
<th>$k'_7$</th>
<th>$k'_8$</th>
<th>$k'_9$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td># samples</td>
<td>$c'_1$</td>
<td>$c'_2$</td>
<td>$c'_3$</td>
<td>$c'_4$</td>
<td>$c'_5$</td>
<td>$c'_6$</td>
<td>$c'_7$</td>
<td>$c'_8$</td>
<td>$c'_9$</td>
<td>...</td>
</tr>
</tbody>
</table>

Array length: $f(c'_1 + c'_2)$, $f(c'_3 + ... + c'_6)$, $f(c'_7 + c'_8 + c'_9)$, ...
Phase 3: Scattering

Light keys

Conflict!

Heavy keys
Phase 4: Local sort

Phase 5: Packing
Size Estimation for Arrays
- High Probability

Now consider an array that has $s$ samples. We define the following size-estimation function:

$$f(s) = \left( s + c \ln n + \sqrt{c^2 \ln^2 n + 2sc \ln n} \right)/p$$

where $p = \Theta \left( \frac{1}{\log n} \right)$ is the sampling probability and $c$ is a constant, to be an upper bound of the size of the array.

Lemma 1: If there are $s$ samples of an array, the probability that number of records is more than $f(s)$ is at most $n^{-c}$.
Size estimation for arrays
- Linear Space in Expectation

\[ f(s) = \left( s + c \ln n + \sqrt{c^2 \ln^2 n + 2sc \ln n} \right) / p \]

- Lemma 1: If there are \( s \) samples of an array, the probability that number of records is more than \( f(s) \) is at most \( n^{-c} \)

- Corollary 1: The probability that \( f \) gives an upper bound on all buckets is at least \( 1 - n^{-c+1} / \log^2 n \)

- Lemma 2: \( \sum_i f(s_i) = \Theta(n) \) holds in expectation
Comparison with R&R integer sort

- **R&R algorithm:**
  - Preprocessing: hashing and packing – global data movement
  - Three times bottom-up radix sort – global data movement

- **Our parallel semisort:**
  - Sample and sort – on a small set
  - Bucket construction – more about calculations
  - **Scatter: the only global data communication**
  - Local sort: performed locally
  - Pack: performed locally
Experiments
Experimental setup

- Experiments are run on a 40-core (with 2-way HT, 40h) machine with 2.4GHz Intel 10-core E7-8879 Xeon processors, with a 1066MHz bus and 30MB L3 cache.

- Our code are compiled with `g++ 4.8.0` with `-O2` flag, and parallelized with `Cilk+`, which is supported by `g++`.

- We use parallel hash table with linear probing [Shun and Blelloch 2014].

- We compare to the parallel STL sort [Singler et al. 2007], parallel radix sort and sample sort from Problem Based Benchmark Suite [Shun et al. 2012].
## The parallel semisort algorithm

<table>
<thead>
<tr>
<th>Notation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array length</td>
<td>$n$</td>
</tr>
<tr>
<td>Hashed key range</td>
<td>$n^k$</td>
</tr>
<tr>
<td>Sample rate</td>
<td>$p = \Theta\left(\frac{1}{\log n}\right)$</td>
</tr>
<tr>
<td>Threshold to distinguish heavy keys from light keys</td>
<td>$\Omega(\log n)$</td>
</tr>
<tr>
<td># buckets for light key</td>
<td>$\Theta\left(\frac{n}{\log^2 n}\right)$</td>
</tr>
</tbody>
</table>
Input distribution

- Uniform distribution (parameter: $m$. range of integers are from $[m]$)
- Exponential distribution (parameter: $\lambda$. mean $1/\lambda$, variance $1/\lambda^2$)
Input distribution

- The different distributions and parameters are used to control the ratio of heavy keys.

- Uniform distribution (parameter: $m$. range of integers are from $[m]$)

- Exponential distribution (parameter: $\lambda$. mean $1/\lambda$, variance $1/\lambda^2$)

- Two representative distributions:
  - Uniform distribution with $m = n$ (0% heavy keys)
  - Exponential distribution with $\lambda = n/1000$ (70-80% heavy keys)
Efficiency & Scalability
Our parallel semisort outperforms STL sort, sample sort and radix sort.

- Number of threads: 40 cores with hyperthreading
- Array length: $10^8$
- Distribution: exponential
Efficiency & Scalability with input size
Our parallel semisort outperforms STL sort, sample sort and radix sort.

- Number of threads: 40 cores with hyperthreading
- Array length: $10^8$
- Distribution: uniform

![Graphs showing performance metrics](image)
Parallel Performance

Linear speedup

- PBBS radix sort [Shun et al 2012]

Radix sort proposed in [Polychroniou and Ross 2014]
  - Crashed on exponential distribution
Parallel performance
Linear speedup

- We show the running time of our algorithm and the radix sort with varying number of threads
- The input contains $10^8$ records
Breakdown of running time

- Exponential
- Uniform

- Sample and sort
- Array construction
- Scatter
- Local sort
- Pack

Percentage of running time
Other experiments - The stabability

- We also have more experiments on testing the stability with different distributions
  - Three different distributions
  - 17 cases in total

- We refer you to our paper to see the details.
Conclusion
Conclusion

- We introduced a parallel algorithm for semisorting that is:

  - **Theoretically efficient**: requires linear work and space, and logarithmic depth.

  - **Practically efficient**: achieves good parallel speedup on various input distributions and input size, and outperforms a similarly-optimized radix sort and other commonly-used sorts.
Thank you.