Chapter 14

Sorting
Outline

- Sorting problem
- Sequential quicksort
- Parallel quicksort
- Hyperquicksort
- Parallel sorting by regular sampling
Sorting Problem

- Permute: unordered sequence $\Rightarrow$ ordered sequence
- Typically key (value being sorted) is part of record with additional values (satellite data)
- Most parallel sorts designed for theoretical parallel models: not practical
- Our focus: internal sorts based on comparison of keys
Sequential Quicksort

Unordered list of values

17  14  65  4  22  63  11
Sequential Quicksort

Choose pivot value
Sequential Quicksort

Low list \( \leq 17 \) 
High list \( > 17 \)
Sequential Quicksort

Recursively apply quicksort to low list
Sequential Quicksort

Recursively apply quicksort to high list

4 11 14 17 22 63 65
Sequential Quicksort

Sorted list of values

| 4 | 11 | 14 | 17 | 22 | 63 | 65 |
Attributes of Sequential Quicksort

- Average-case time complexity: $\Theta(n \log n)$
- Worst-case time complexity: $\Theta(n^2)$
  - Occurs when low, high lists maximally unbalanced at every partitioning step
- Can make worst-case less probable by using sampling to choose pivot value
  - Example: “Median of 3” technique
Quicksort: Starting Point for Parallel Algorithm

- **Speed**
  - Generally recognized as fastest sort in average case
  - Preferable to base parallel algorithm on fastest sequential algorithm

- **Natural concurrency**
  - Recursive sorts of low, high lists can be done in parallel
Definitions of “Sorted”

- **Definition 1**: Sorted list held in memory of a single processor

- **Definition 2**:
  - Portion of list in every processor’s memory is sorted
  - Value of last element on $P_i$’s list is less than or equal to value of first element on $P_{i+1}$’s list

- We adopt Definition 2: Allows problem size to scale with number of processors
Parallel Quicksort

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>75, 91, 15, 64, 21, 8, 88, 54</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>50, 12, 47, 72, 65, 54, 66, 22</td>
</tr>
<tr>
<td>$P_2$</td>
<td>83, 66, 67, 0, 70, 98, 99, 82</td>
</tr>
<tr>
<td>$P_3$</td>
<td>20, 40, 89, 47, 19, 61, 86, 85</td>
</tr>
</tbody>
</table>
Parallel Quicksort

Process $P_0$ chooses and broadcasts randomly chosen pivot value
Parallel Quicksort

75, 91, 15, 64, 21, 8, 88, 54
50, 12, 47, 72, 65, 54, 66, 22
83, 66, 67, 0, 70, 98, 99, 82
20, 40, 89, 47, 19, 61, 86, 85

Exchange “lower half” and “upper half” values
Parallel Quicksort

After exchange step

Lower “half”

75, 15, 64, 21, 8, 54, 66, 67, 0, 70

50, 12, 47, 72, 65, 54, 66, 22, 20, 40, 47, 19, 61

Upper “half”

83, 98, 99, 82, 91, 88

89, 86, 85

$P_0$

$P_1$

$P_2$

$P_3$
Parallel Quicksort

Processes $P_0$ and $P_2$ choose and broadcast randomly chosen pivots.
Parallel Quicksort

- Lower "half":
  - 75, 15, 64, 21, 8, 54, 66, 67, 0, 70
  - 50, 12, 47, 72, 65, 54, 66, 22, 20, 40, 47, 19, 61

- Upper "half":
  - 83, 98, 99, 82, 91, 88
  - 89, 86, 85

Exchange values
Parallel Quicksort

Exchange values
Parallel Quicksort

Each processor sorts values it controls
Analysis of Parallel Quicksort

- Execution time dictated by when last process completes
- Algorithm likely to do a poor job balancing number of elements sorted by each process
- Cannot expect pivot value to be true median
- Can choose a better pivot value
Hyperquicksort

- Start where parallel quicksort ends: each process sorts its sublist
- First "sortedness" condition is met
- To meet second, processes must still exchange values
- Process can use median of its sorted list as the pivot value
- This is much more likely to be close to the true median
Hyperquicksort

Number of processors is a power of 2

$P_0$

75, 91, 15, 64, 21, 8, 88, 54

$P_1$

50, 12, 47, 72, 65, 54, 66, 22

$P_2$

83, 66, 67, 0, 70, 98, 99, 82

$P_3$

20, 40, 89, 47, 19, 61, 86, 85
Hyperquicksort

$P_0$

$8, 15, 21, 54, 64, 75, 88, 91$

$P_1$

$12, 22, 47, 50, 54, 65, 66, 72$

$P_2$

$0, 66, 67, 70, 82, 83, 98, 99$

$P_3$

$19, 20, 40, 47, 61, 85, 86, 89$

Each process sorts values it controls
Hyperquicksort

Process $P_0$ broadcasts its median value

$P_0$: 8, 15, 21, 54, 64, 75, 91, 88

$P_1$: 12, 22, 47, 50, 54, 65, 66, 72

$P_2$: 0, 66, 67, 70, 82, 83, 98, 99

$P_3$: 19, 20, 40, 47, 61, 85, 86, 89
Hyperquicksort

Processes will exchange “low”, “high” lists.
Hyperquicksort

Processes merge kept and received values.
Hyperquicksort

Processes $P_0$ and $P_2$ broadcast median values.
Hyperquicksort

Communication pattern for second exchange
Hyperquicksort

- \( P_0: \{0, 8, 12, 15\} \)
- \( P_1: \{19, 20, 21, 22, 40, 47, 50, 54, 54\} \)
- \( P_2: \{61, 64, 65, 66, 66, 67, 70, 72, 75, 82\} \)
- \( P_3: \{83, 85, 86, 88, 89, 91, 98, 99\} \)

After exchange-and-merge step
Complexity Analysis
Assumptions

- Average-case analysis
- Lists stay reasonably balanced
- Communication time dominated by message transmission time, rather than message latency
Complexity Analysis

- Initial quicksort step has time complexity $\Theta((n/p) \log (n/p))$
- Total comparisons needed for log $p$ merge steps: $\Theta((n/p) \log p)$
- Total communication time for log $p$ exchange steps: $\Theta((n/p) \log p)$
Still a Scalability Concern

- Analysis assumes lists remain balanced
- As $p$ increases, each processor’s share of list decreases
- Hence as $p$ increases, likelihood of lists becoming unbalanced increases
- Unbalanced lists lower efficiency
- Would be better to get sample values from all processes before choosing median
Parallel Sorting by Regular Sampling (PSRS Algorithm)

- Each process sorts its share of elements
- Each process selects regular sample of sorted list
- One process gathers and sorts samples, chooses pivot values from sorted sample list, and broadcasts these pivot values
- Each process partitions its list into $p$ pieces, using pivot values
- Each process sends partitions to other processes
- Each process merges its partitions
PSRS Algorithm

Number of processors does not have to be a power of 2.

$P_0$: 75, 91, 15, 64, 21, 8, 88, 54

$P_1$: 50, 12, 47, 72, 65, 54, 66, 22

$P_2$: 83, 66, 67, 0, 70, 98, 99, 82
PSRS Algorithm

Each process sorts its list using quicksort.

- $P_0$: 8, 15, 21, 54, 64, 75, 88, 91
- $P_1$: 12, 22, 47, 50, 54, 65, 66, 72
- $P_2$: 0, 66, 67, 70, 82, 83, 98, 99
PSRS Algorithm

Each process chooses \( p \) regular samples.
PSRS Algorithm

One process collects $p^2$ regular samples.
PSRS Algorithm

One process sorts $p^2$ regular samples.

$P_0 = \{8, 15, 21, 54, 64, 75, 88, 91\}$

$P_1 = \{12, 22, 47, 50, 54, 65, 66, 72\}$

$P_2 = \{0, 66, 67, 70, 82, 83, 98, 99\}$

$P_3 = \{15, 22, 50, 54, 65, 66, 70, 75, 83\}$
PSRS Algorithm

8, 15, 21, 54, 64, 75, 88, 91

12, 22, 47, 50, 54, 65, 66, 72

0, 66, 67, 70, 82, 83, 98, 99

15, 22, 50, 54, 65, 66, 70, 75, 83

One process chooses $p$-1 pivot values.
PSRS Algorithm

One process broadcasts \( p-1 \) pivot values.
PSRS Algorithm

Each process divides list, based on pivot values.
PSRS Algorithm

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>8, 15, 21</td>
<td>12, 22, 47, 50</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>54, 64</td>
<td>54, 65, 66</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>75, 88, 91</td>
<td>72</td>
<td>67, 70, 82, 83, 98, 99</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each process sends partitions to correct destination process.
PSRS Algorithm

Each process merges $p$ partitions.
Summary

- Three parallel algorithms based on quicksort
- Keeping list sizes balanced
  - Parallel quicksort: poor
  - Hyperquicksort: better
  - PSRS algorithm: good