Chapter 14

Sorting

Outline

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

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

Sequential Quicksort

Unordered list of values
Sequential Quicksort

Choose pivot value

Low list \((\leq 17)\) \hspace{1cm} High list \((> 17)\)
Sequential Quicksort

Recursively apply quicksort to low list

Sequential Quicksort

Recursively apply quicksort to high list
Sequential Quicksort

Sorted list of values

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 Good 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
Communication Pattern

Parallel Quicksort

- \( P_0 \): 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
Parallel Quicksort

Process $P_0$ chooses and broadcasts a randomly chosen pivot value

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

After exchange step

Processes P0 and P2 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

Lower “half” of lower “half”

15, 21, 8, 0, 12, 20, 19

50, 47, 72, 65, 54, 66, 22, 40, 47, 61, 75, 64, 54, 66, 67, 70

Upper “half” of lower “half”

83, 82, 91, 88, 89, 86, 85

98, 99

Exchange values
Parallel Quicksort

- Lower “half” of lower “half”: $0, 8, 12, 15, 19, 20, 21$ (P0)
- Upper “half” of lower “half”: $22, 40, 47, 47, 50, 54, 54, 61, 64, 65, 66, 66, 67, 70, 72, 75$ (P1)
- Lower “half” of upper “half”: $82, 83, 85, 86, 88, 89, 91$ (P2)
- Upper “half” of upper “half”: $98, 99$ (P3)

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
- Communication is set up like a hypercube

Hypercube
Hyperquicksort

Number of processors is a power of 2

Each process sorts values it controls
### Hyperquicksort

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

Process $P_0$ broadcasts its median value.

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

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

Communication pattern for second exchange

Hyperquicksort

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: $\Theta((n/p) \log p)$
Isoefficiency Analysis

- Sequential time complexity: $\Theta(n \log n)$
- Parallel overhead: $\Theta(n \log p)$
- Isoefficiency relation:
  
  \[
  n \log n \geq C \log n \Rightarrow \log n \geq C \log p \Rightarrow n \geq p^C
  \]

  \[
  M(p^C) / p = p^C / p = p^{C-1}
  \]

- The value of $C$ determines the scalability.
  Scalability depends on ratio of communication speed to computation speed.

Another Scalability Concern

- Our 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

\[
\begin{align*}
P_0 & \quad = \{75, 91, 15, 64, 21, 8, 88, 54\} \\
P_1 & \quad = \{50, 12, 47, 72, 65, 54, 66, 22\} \\
P_2 & \quad = \{83, 66, 67, 0, 70, 98, 99, 82\}
\end{align*}
\]

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

Each process sorts its list using quicksort.

PSRS Algorithm

Each process chooses $p$ regular samples.
PSRS Algorithm

One process collects $p^2$ regular samples.

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

\[ P_0 = 8, 15, 21, 54, 64, 75, 88, 91 \]
\[ P_1 = 12, 22, 47, 50, 65, 66, 72 \]
\[ P_2 = 0, 66, 67, 70, 82, 83, 98, 99 \]

One process chooses \( p-1 \) pivot values.

PSRS Algorithm

\[ P_0 = 8, 15, 21, 54, 64, 75, 88, 91 \]
\[ P_1 = 12, 22, 47, 50, 65, 66, 72 \]
\[ P_2 = 0, 66, 67, 70, 82, 83, 98, 99 \]

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

Each process divides list, based on pivot values.

PSRS Algorithm

Each process sends partitions to correct destination process.
PSRS Algorithm

Each process merges $p$ partitions.

Assumptions

- Each process ends up merging close to $n/p$ elements
- Experimental results show this is a valid assumption
- Processor interconnection network supports $p$ simultaneous message transmissions at full speed
- 4-ary hypertree is an example of such a network
Time Complexity Analysis

- Computations
  - Initial quicksort: $\Theta((n/p)\log(n/p))$
  - Sorting regular samples: $\Theta(p^2 \log p)$
  - Merging sorted sublists: $\Theta((n/p)\log p)$
  - Overall: $\Theta((n/p)(\log n + \log p) + p^2\log p)$

- Communications
  - Gather samples, broadcast pivots: $\Theta(\log p)$
  - All-to-all exchange: $\Theta(n/p)$
  - Overall: $\Theta(n/p)$

Isoefficiency Analysis

- Sequential time complexity: $\Theta(n \log n)$
- Parallel overhead: $\Theta(n \log p)$
- Isoefficiency relation:
  
  \[ n \log n \geq Cn \log p \Rightarrow \log n \geq C \log p \]

- Scalability function same as for hyperquicksort
- Scalability depends on ratio of communication to computation speeds
Summary

- Three parallel algorithms based on quicksort
- Keeping list sizes balanced
  - Parallel quicksort: poor
  - Hyperquicksort: better
  - PSRS algorithm: excellent
- Average number of times each key moved:
  - Parallel quicksort and hyperquicksort: log \( p / 2 \)
  - PSRS algorithm: \( (p-1)/p \)