Sorting

Parallel Sorting
Sorting Data

- Sorting is a very common problem that needs to be solved in computer science; a good sorting algorithm can make or break a program, especially with large data sets.
- Note that in order for a data set to be sorted, it needs to be comparable. So, “generic” collections cannot be sorted.
- Let’s examine a few candidates for sorting methods, and their relative strengths and weaknesses.
Sorting Problem

• Permute: unordered sequence ⇒ 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
Bubble Sort

- Bubble sort is the simplest of the sort methods, and runs in $O(n^2)$ time.
- It can be accomplished with a remarkably small amount of memory, but it is also the least efficient sort that we will look at.
- The basic idea is to look at each pair of nodes in turn, and swap them if they are out of order. You do this comparison until no swaps are made, which can be up to $n$ times, if there are $n$ elements in the list.
Selection Sort

• The idea behind selection sort is to look through the list for the smallest item, and put it in front.
• Then look for the next smallest, and put it second, then for the third smallest, and put it third, and so on, up to the nth smallest (which is the largest).
• This also takes $O(n^2)$ to run, but in practice is faster than bubble sort.
Insertion Sort

• The idea behind insertion sort is to take the unsorted elements one at a time in sequence, and insert them into their appropriate spot in the list.
• This also takes $O(n^2)$ to run.
• One thing to remember about insertion sort is that it works faster the closer the list is to already being sorted.
Shell Sort

• Shell sort is a modified insertion sort that doesn't seem like it should improve the running time of insertion sort all that much, but in practice, it does.

• It still takes $O(n^2)$ to run, but in practice it never actually takes that long, and compares favorably with many of the $n \log n$ sorts.

• The idea behind shell sort is to start by sorting regularly spaced sub-arrays before sorting the whole list with insertion sort.
Shell Sort, Continued

Thus, when the insertion sort is done, the list will be overall "more sorted" than it was before the sub-sorts. We start with a spacing of $n/2$, and halve the spacing each time, doing an insertion sort on the subarrays, until the spacing reduces to 1, at which point it becomes a regular insertion sort.
Merge Sort

- Merge sort is the first recursive sort that we examine, and also the first one that runs in less than $O(n^2)$ time. In fact, it runs in $O(n \log n)$ time.
- The sort is accomplished by breaking the array into two halves, sorting the halves, and merging them back together (Divide and Conquer). Done recursively, this produces a sorted array.
- This is a very fast sort in practice, but it has the disadvantage of requiring extra memory for a temporary array to "merge" the sorted halves back into.
- This is a “Divide and Conquer” sort.
Quick Sort

• Quick sort is also a recursive sort, whose basic algorithm is to
  1. pick a pivot point in the array.
  2. put all values less than that point before it, and all values greater than that after it
  3. quick sort the smaller values, then the larger values.

• This is a “Divide and Conquer” sort.
Quick Sort, Continued

- This is also an $O(n \log n)$ sort, but a great deal of difference is made in the run time depending on the pivot that you pick.
- You want to find one that is the median value for the list, so that it is split into even halves. This will limit the depth of the recursion necessary.
- This is expensive, so one shortcut is just to pick three numbers randomly, and use the median one to be the pivot.
- Quick sort runs worst when the list is already mostly sorted.
Sequential Quicksort

Unordered list of values
Sequential Quicksort

Choose pivot value
Sequential Quicksort

| 14 | 4 | 11 | 17 | 65 | 22 | 63 |

Low list \( \leq 17 \)  
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
Radix Sort is a strange animal, and when you hear about it, it seems like it shouldn’t work. It does, and works well, but it is restricted as to the types of items that it can sort. Radix Sort is a type of distribution, or “bucket” sort, and it works only on comparable items of a fixed size, where the comparison can be done on each digit (or character) of the value. So, it works well on positive numbers, or on strings of limited length.
Radix Sort, Continued

The idea behind Radix Sort is to sort the values by their least significant digit (or character), then by their next least significant, then third least significant, and so on. Each time we sort, we keep the list in the relative sort order from the previous iteration. Radix sort is the fastest sort we have examined, asymptotically. It runs in $O(n)$ time.
Heap Sort

- Heap sort uses our good friend, the Heap ADT.
- The basic process is to use a min heap, build the heap from the unsorted array, and then pull the elements out one at a time.
- This is an $O(n \log n)$ sort.
- It generally runs slower than Quick Sort or Merge Sort, but does not require recursion or extra arrays, which means that for very large data sets, it may be a more attractive option.
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
Parallel Quicksort

\begin{align*}
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 \\
\end{align*}
Parallel Quicksort

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

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

P₀
P₁
P₂
P₃
Parallel Quicksort

Processes $P_0$ and $P_2$ choose and broadcast randomly chosen pivots

**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

Exchange values
Parallel Quicksort

Lower “half” of lower “half”
15, 21, 8, 0, 12, 20, 19

Upper “half” of lower “half”
50, 47, 72, 65, 54, 66, 22, 40,
47, 61, 75, 64, 54, 66, 67, 70

Lower “half” of upper “half”
83, 82, 91, 88, 89, 86, 85

Upper “half” of upper “half”
98, 99

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

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

Number of processors is a power of 2
Hyperquicksort

Each process sorts values it controls
Hyperquicksort

Process $P_0$ broadcasts its median value
Hyperquicksort

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

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

\[ P_0 \]
\[ P_1 \]
\[ P_2 \]
\[ P_3 \]

Processes merge kept and received values.
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:
  
  $n \log n \geq C n \log p \Rightarrow \log n \geq C \log p \Rightarrow n \geq p^C$

  $M(p^C) / p = p^C / p = p^{C-1}$

- 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

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

One process sorts $p^2$ regular samples.
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

Each process sends partitions to correct destination process.

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

Each process merges $p$ partitions.

$P_0$: 0, 8, 12, 15, 21, 22, 47, 50

$P_1$: 54, 54, 64, 65, 66, 66

$P_2$: 67, 70, 72, 75, 82, 83, 88, 91, 98, 99
Assumptions

• Each process ends up merging close to \( \frac{n}{p} \) 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: $(p-1)/p$