Parallel Sorting of Roughly-Sorted Sequences

Anthony Pfaff, Jason Treadwell

CSCI 5172 | CU Denver | Fall '16

12.10.2016

Introduction

Sorting a sequence according to some ordering is a foundational problem of computer science.

Sorting has crucial and non-obvious applications. There exist a great many such algorithms.

The optimal runtime of sorting a sequence of length n by *comparison* is $\Theta(n \lg n)$.

Sort Algorithms

The comparison sorts include Mergesort, Heapsort, Quicksort, etc. and have diverse strengths.

The former two are asymptotically optimal; Quicksort frequently outruns them, but is $O(n^2)$.

We can beat the linearithmic runtime bounds through additional analysis of the input array.

Roughly-Sorted Sequences

The sequence $A = \{a_0, a_1, \cdots, a_{n-1}\}$ is k-sorted if it satisfies

$$a_i \leq a_j \quad \forall\, i < j - k\,, \quad 0 \leq i \leq j < n\,.$$

The radius of A is the smallest k such that A is k-sorted:

$$\text{radius}(A) = \max\{j - i \mid j > i,\, a_i > a_j\}\,.$$

Conventions

We'll only consider sequences of 32-bit integers backed by *random-access arrays*.

The array $A = \{a_0, a_1, \cdots, a_{n-1}\}$ is *sorted* if its elements are all ordered in nondecreasing order.

When we say A is k-sorted, we imply that k is minimal (i.e., the radius of A is k).

Roughly-Sorted Sequences (cont.)

Some applications involve sorted arrays that become slightly perturbed.

We might be able to exploit their partial sortedness to beat the $\Omega(n \lg n)$ lower runtime bound.

Comparison-based sorts are prone to divide-and-conquer and, therefore, opportunities for parallel speedup.

Roughsort

Roughsort exploits the presortedness of the array A by applying a radius *halving* algorithm.

The algorithm sorts A by halving its radius $\lg k$ times in runtime $\Theta(n \lg k) = O(n \lg n)$.

Roughsort does invite parallel speedup, yet must determine k to effectively sort A.

Determining the Radius

Using min/max prefix scans, we compute the *characteristic sequences* of A:

$$LR(A) = \{b_i\}\,, \quad RL(A) = \{c_i\}\,, \quad 0 \leq i < n\,,$$

$$b_i = \max\{a_0, a_1, \cdots, a_i\}\,, \quad c_i = \min\{a_i, a_{i+1}, \cdots, a_{n-1}\}\,.$$

The radius k of A is the maximum element from the *disorder measure* of A:

$$DM(A) = \{d_i\}\,, \quad d_i = \max\left\{\{i - j \mid c_i < b_j\} \cup \{0\}\right\}\,, \quad k = \max DM(A)\,.$$

Finding k thus takes linear time and space, preserving the $\Theta(n \lg k)$ complexity of Roughsort.

Halving the Radius

1. Partition each consecutive run of k elements in A about the mean of the run.
2. Starting at element $a_{\lfloor k/2 \rfloor}$ to stagger the partitions, repeat step 1 and go to 3.
3. Repeat step 1 and halt.

Each partition is performed using STL's `nth_element()` and takes linear time.

Our sequential implementation thus sorts A in place by halving its radius $\lg k$ times, taking linear space and $\Theta(3(n/k \cdot k) \lg k) = \Theta(n \lg k)$ time.

Parallel Roughsort: Failed Approach

Our first implementation simply parallelized the halving algorithm of our sequential Roughsort.

We replaced `nth_element()` with Thrust's `sort()`, an optimized radix sort.

We tried to launch every partitioning in parallel using *streams* and got abysmal results.

Parallel Roughsort: Better Approach

1. In parallel, **sort** all consecutive runs of $2k$ elements in A.
2. Repeat step 1, but begin the first run at a_k; halt.

This process sorts each segment and needs only a single iteration to fully sort A.

We implemented it as a CUDA kernel where the array segments were divided up among many parallel threads, but each thread sequentially sorted its designated array segment.

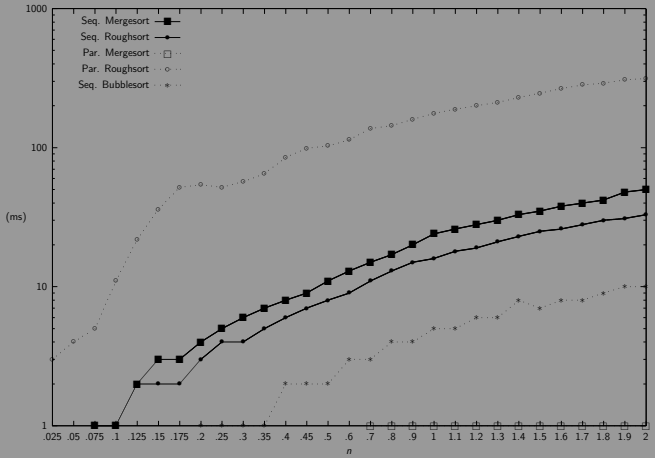
Testing the Implementation

We tested our Roughsort implementations against sequential and parallel Mergesort as well as against Bubblesort.

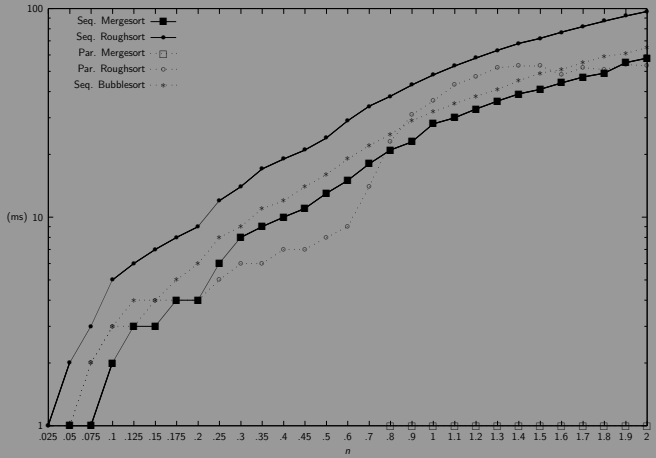
Each test generated a random k-sorted array for some given k and length n, then fed it to all five algorithms, averaging their results over eight runs. Each array was sourced by a hardware RNG and sorted before being k-perturbed and shuffled.

The tests were run on a workstation with a high-end, quad-core Xeon CPU as well as a high-end, consumer-grade Nvidia GeForce 1080 GTX GPU suitable for workloads over 32-bit array elements.

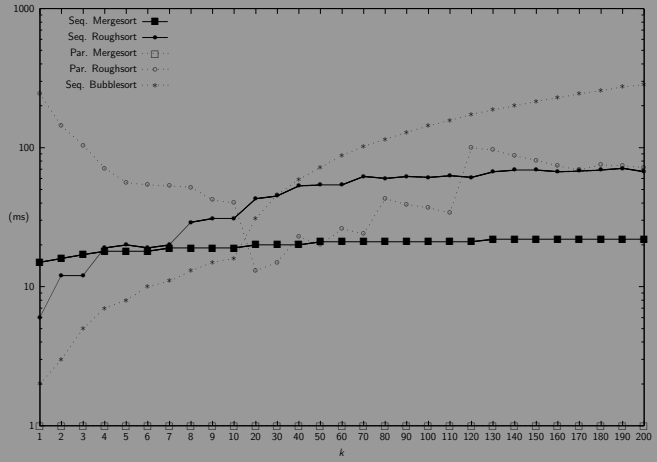
Sort Runtimes ($k = 2$)



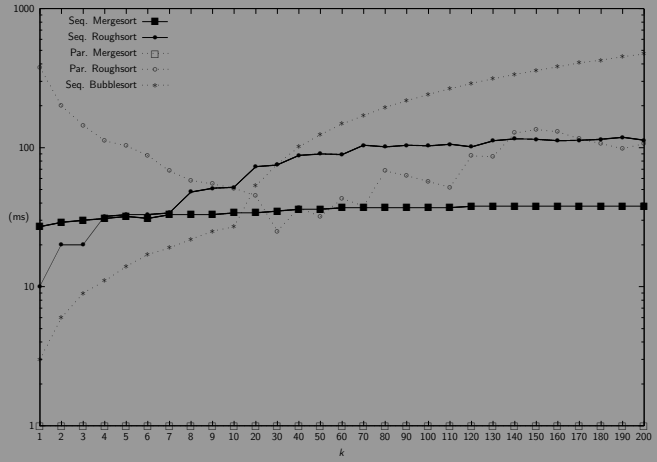
Sort Runtimes ($k = 15$)



Sort Runtimes ($n = 0.75 \cdot 10^6$)



Sort Runtimes ($n = 1.25 \cdot 10^6$)



Conclusion and Further Research