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.

Parallelization

Roughsort may be parallelized, but assumes underlying CRCW PRAM to achieve this.

CRCW PRAM:

1. Model for RAM which essentially says multiple slots in memory may be updated or concurrently
2. This update is done by a fixed number of processors
3. Several other restrictions to the model

CUDA: Our Answer for CRCW PRAM

However, CRCW PRAM is just a useful abstraction. The real world is just that: real

Must find a proxy for CRCW PRAM to develop an implementation.

CUDA: Our Answer for CRCW PRAM (2)

Enter CUDA. Similar characteristics:

1. Multiple slots in global memory may be updated simultaneously
2. Multiple slots may also be read simultaneously
3. Thousands more processor cores than standard CPUs, providing a mechanism to update all this parallel memory

CUDA: Background

Nvidia provides APIs and an SDK to allow developers the use of GPU functions.

Model:

1. CPU-homed (host) program is specifically programmed to talk to GPU
2. Host program is compiled ahead of time with CUDA hooks and calls embedded in its C++ source
3. Payload data is explicitly shipped off to the GPU
4. Host ships the compiled GPU code (kernel) to the GPU and launches it asynchronously
5. GPU executes payload, places data in defined locations, host program picks up result data

CUDA: Background(2)

Organization

1. Core component and lowest-level work unit of CUDA execution context is a thread
2. Up to thousands of threads may be organized in a group called a block
3. Multiple blocks make up a grid
4. Threads per block and other parameters are specified by launching program
5. CUDA cheaply switches threads in and out of processing on cores

CUDA: Background(3)

Warps

1. Groups of 32 threads which execute together are called a warp - SIMT
2. This is a concept that is abstracted away from programmer during development/kernel launch (32 is fixed)
3. Threads in a warp executing on a group of 32 cores are all executing the same instruction at the same time, optimally
4. In cases of a conditional, some cores may pause to allow threads with a true value to process, then false values process - Warp divergence

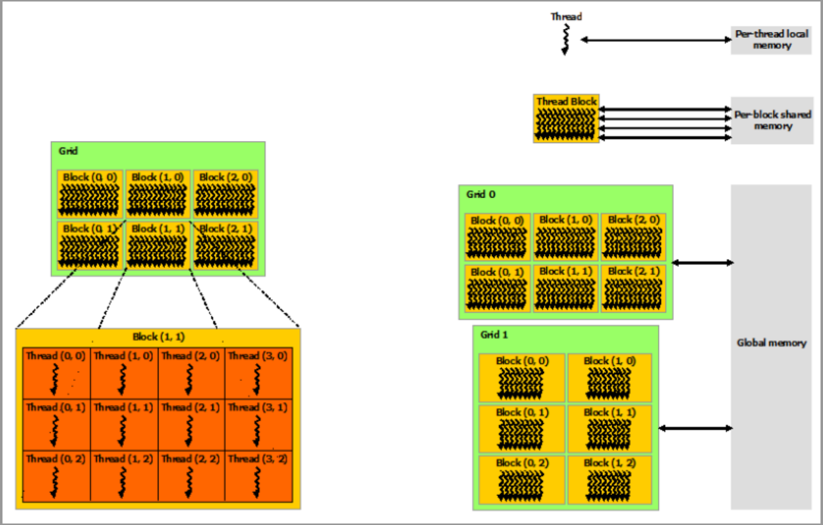
CUDA: Pattern

Threads, blocks, and grid all have identifiers

Developer references these identfiers in code

Using an understanding of how work is to be divided up among threads, these identifiers can inform a thread what work it needs to do

CUDA: Visual


Thread
Per-thread local memory
Thread Block
Per-block shared memory
Grid
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Grid 0
Grid 1
Block (0, 2)
Block (1, 2)
Global memory

Parallel Radius Finding: Translating

Back to parallel Roughsort: we know we can find the radius in parallel. Merely need to convert the pseudocode to CUDA code.

Straightforward for *LR* list, slightly less so for *RL*, least straightforward for *DM*.

Eventually settled on a *DM* list determination based on the sequential, with array indicies fixed to thread ID.

Added an efficient method to check for full array sortness as part of the *RL* and *LR* list components.

Parallel Radius Finding: Scaling Up

Testing:

1. Worked fine for a low values
2. Did not function at all for large values of n (widely inaccurate k values)

Parallel Radius Finding: Re-Visiting

Cause/Fix:

1. Asked individual threads to allocate their slot in the global memory *RL* and *LR* lists upon startup
2. Not all threads are immediately scheduled for large thread counts–bug
3. Subsequently started copying all values before kernel launch

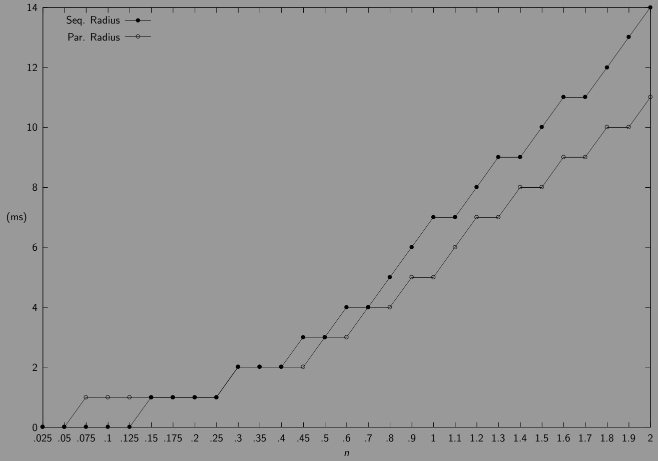
Parallel Radius Finding: Scaling Up (2)

After fixing, *DM* list was correct to an extent, but it was slow.

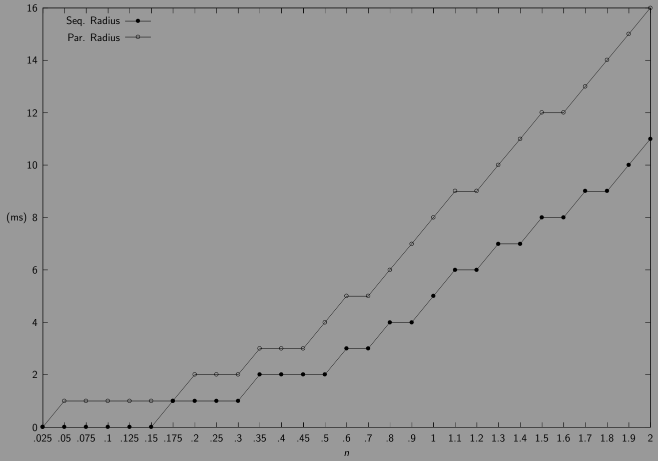
Resolved this by realizing only need our "radius" to approximate the real k value, in that we just need the next power of two greater

Added `Thrust max_element` code to find the greatst value in the *DM* list, thus yielding our radius

Parallel Radius Finding: Results ($k = 2$)



Parallel Radius Finding: Results($k = 100$)



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.

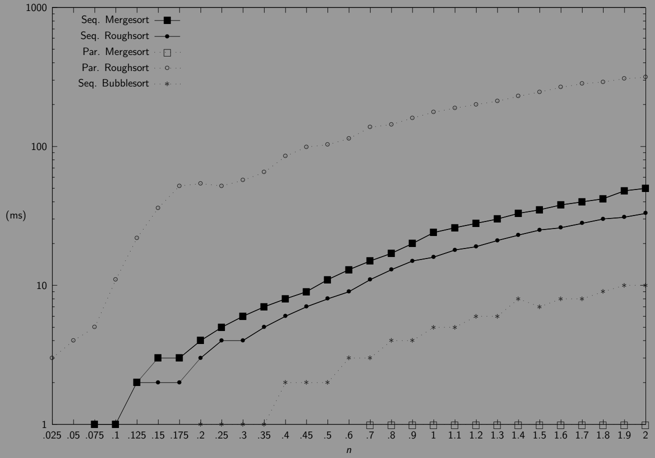
Testing the Implementation

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

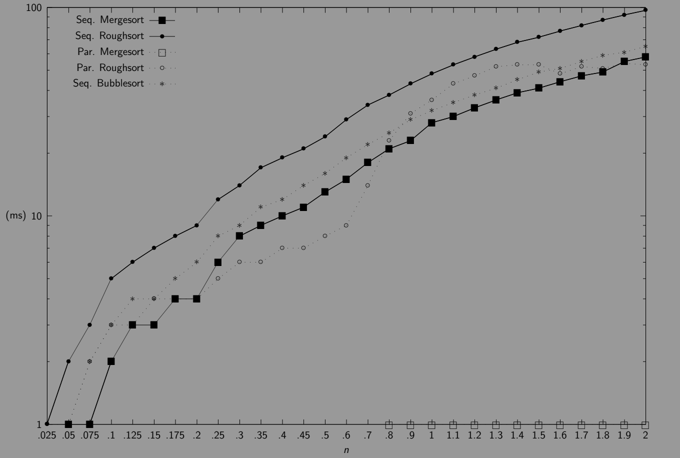
Each test generated a random *k*-sorted array for some given *k* and length *n*, then fed it to all five algorithms, averaging their results over eight runs. Each array was sourced by a hardware RNG and sorted before being *k*-perturbed and shuffled.

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.

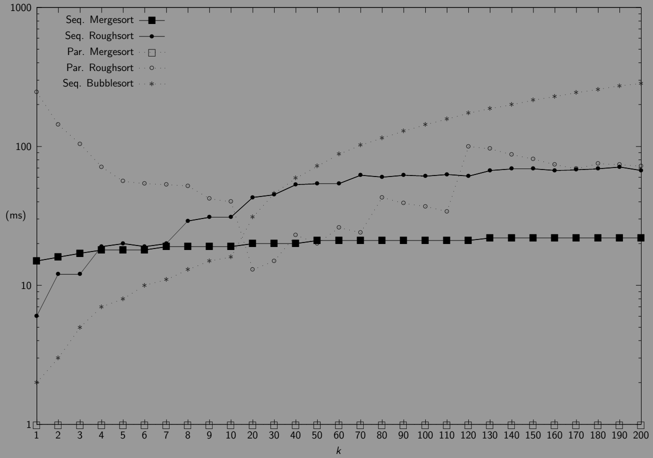
Sort Runtimes ($k = 2$)



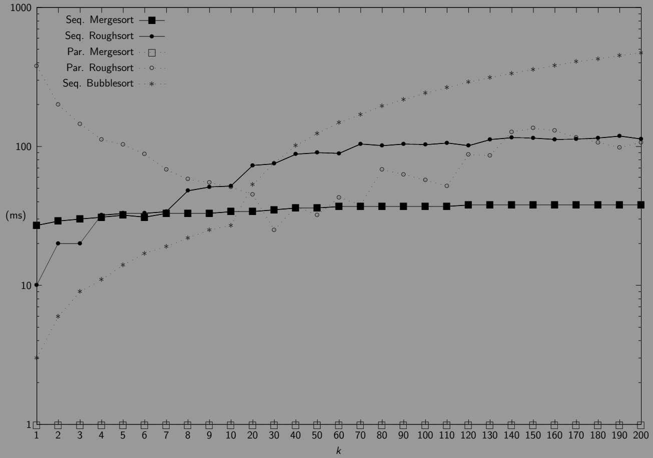
Sort Runtimes ($k = 15$)



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



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



Parallel Performance

Below expectations:

1. Parallel radius runs in linear time
2. Parallel Roughsort by Mergesort consistently

Why: CUDA is not CRCW PRAM

1. Warp divergence
2. Memory coalescence

Conclusion and Further Research

Demonstrated a good method for evaluating sorting algorithms, by being able to generate k-sorted sequences

Developed a first implementation of Roughsort—to be continued by others

Bottlenecks in implementation should be researched so they may be resolved