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 rife with 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 identifiers 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