Parallel Sorting of Roughly-Sorted Sequences

CSCI 5172 Fall '16 Project

Anthony Pfaff, Jason Treadwell

December 6, 2016

Sorting a collection according to some ordering among its items is among the most classic problems of computer science. A well-established result is the linearithmic (i.e. $O(n \lg n)$) optimal upper bound for sorting sequences of length n by comparison. Commonly-used sorting algorithms such as Mergesort and Quicksort have $\Theta(n \lg n)$ and $\Omega(n \lg n)$ runtimes, respectively.

We can exploit the ordering of *roughly-sorted* sequences to sort them in $O(n \lg k)$ time, where k is the *radius* of a sequence S or the smallest k such that S is *k-sorted.*[2] A k-sorted sequence $\{a_0, a_1, \cdots, a_n\}$ satisfies $a_i \leq a_j \; \forall \; 1 \leq i \leq j \leq n, i \leq j - k$. Since an unsorted sequence can be at most n-sorted, the worst-case runtime of this algorithm has complexity $O(n \lg n)$.

We intend to study and implement sorting of roughly-sorted sequences using both sequential and parallel[1] versions of the algorithm. In particular, I shall investigate the applicability of the method over very large sequences, accelerated using graphics hardware. A GPU is dedicated graphics hardware that also excels at performing frequent, basic computations over large amounts of data. The use of GPUs for numerical applications is a recent and rapidly-growing development in computer science and software engineering. Suitable hardware is now commonly available in consumer devices and may herald a paradigm shift in how we efficiently perform common and indispensable tasks like sorting.

I propose to develop implementations of the method using C and Nvidia's CUDA framework, comparing the results with sequential implementations running on both the GPU and CPU as well as with the classic Mergesort and Quicksort algorithms. If time permits, I would also like to compare the method to parallel versions of the latter two algorithms.

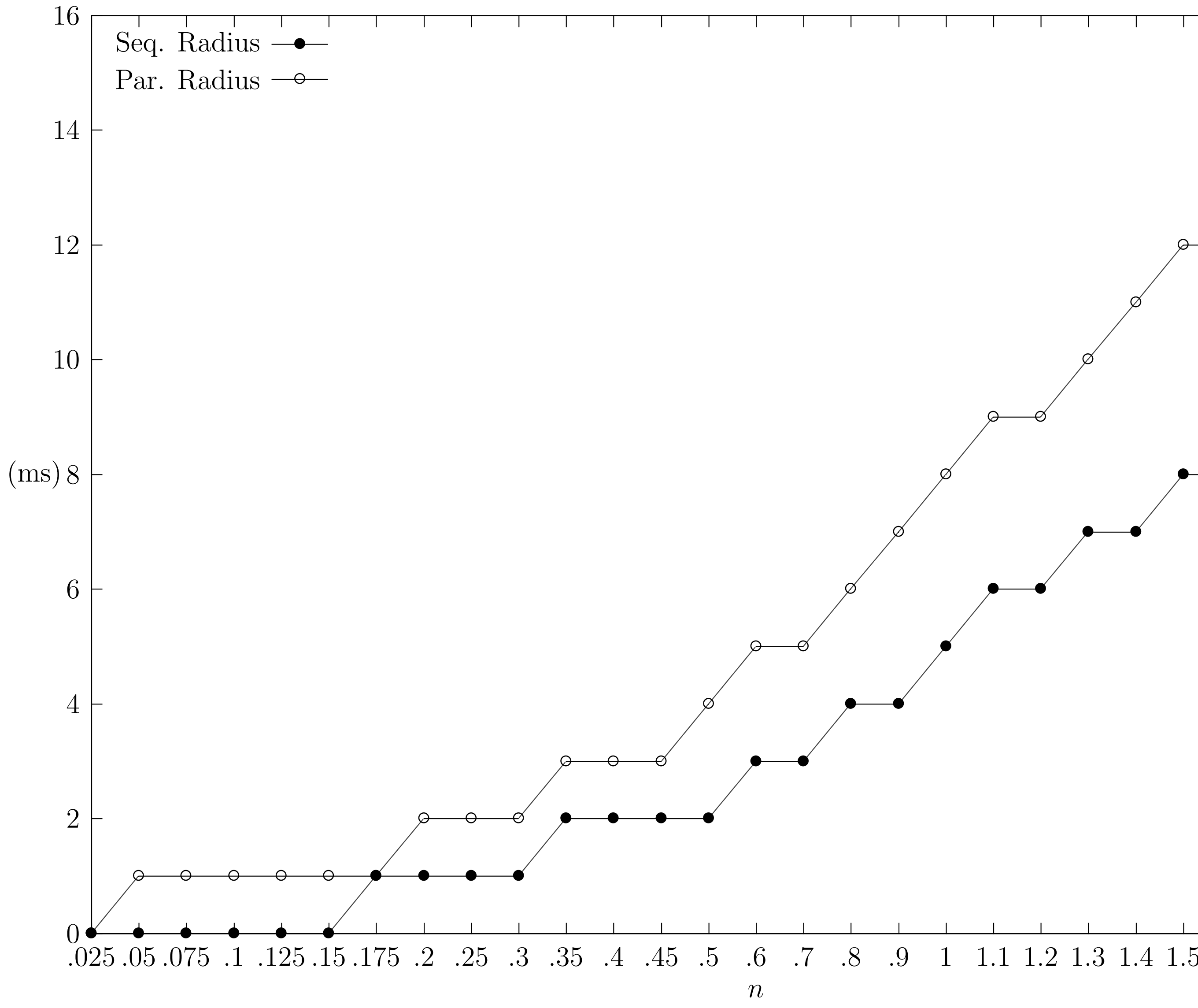
The aforementioned algorithms assume a specific underlying memory in achieving the provided parallel implementation runtimes: the concurrent-read concurrent-write parallel random access model (CRCW PRAM). Shiloach and Vishkin elaborate on this memory model, providing that it is a purely theoretical model which, inter alia, meets the following criteria: a shared, global memory of size m ¿ p is available to p processors, and up to p positions in this memory may be simultaneously read or written in one step[9]. As with roughsort, a number of algorithms take advantage of this models assumptions to provide efficiency improvements over their serial counterparts[10][8].

Conspicuous, however, with these and other CRCW PRAM algorithms is a lack of presented implementations. The model is merely that, and thus implementing such an algorithm requires a suitable proxy which shares the characteristics of CRCW

PRAM. In contemporary computing, there are most notable such proxy is graphics processing unit (GPU) parallelization.

GPUs flip invert core principles of general purpose computing to yield a performance improvement on specific workloads.

Given many GPU processor cores may update multiple memory locations simultaneously, and likewise many memory location may be read in parallel on GPU, the underlying NVIDIA hardware that CUDA provides access to is reminiscent of CRCW PRAM. However, GPUs differ from true CRCW PRAM in fundamental ways. First, GPU memory is finite, while PRAM is of arbitrary size. Furthermore, for GPU memory access times to provide the concurrent read and write characteristics reminiscent of PRAM, these memory operations must be performed on contiguous portions of GPU memory, whereas CRCW PRAM has no such ordering limitation. Foremost in the differences between CRCW PRAM and GPU architectures however is the warp divergence concept addressed above. What this implies for memory operations is that workloads exist which may prevent all but one thread in a warp from performing a memory operation at a particular time. Consequently, there are practical limitations to what a GPU may achieve, and these limitations restrict the workloads which are practical to run on a GPU. Not only must a GPU workload access constrain its memory usage within practical limitations, the accesses to that memory must be orderly, and the underlying algorithm performing those accesses must be amenable to parallelization on GPU if the advantages of GPU processing over CPU processing are to be realized[**?**].



Seq. Radius
Par. Radius
(ms)
16
14
12
10
8
6
4
2
0
.025
.05
.075
.1
.125
.15
.175
.2
.25
.3
.35
.4
.45
.5
.6
.7
.8
.9
1
1.1
1.2
1.3
1.4
1.5
n

References

[1] Tom Altman and Bogdan Chlebus. Sorting roughly sorted sequences in parallel. *Information Processing Letters*, 33(6), February 1990.

[2] Tom Altman and Yoshihide Igarashi. Roughly sorting: Sequential and parallel approach. *Journal of Information Processing*, 12(2), 1989.

[3] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. Wrox, 1st edition, September 2014.

[4] MARTTI FORSELL. On the performance and cost of some pram models on cmp hardware. *International Journal of Foundations of Computer Science*, 21(03):387–404, 2010.

[5] Intel. *Intel Digital Random Generator (DRNG) Software Implementation Guide*, 1.1 edition, August 2012.

[6] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1998.

[7] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14), July 2003.

[8] Sanguthevar Rajasekaran and John H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, 1989.

[9] Yossi Shiloach and Uzi Vishkin. *Finding the maximum, merging and sorting in a parallel computation model*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1981.

[10] Yossi Shiloach and Uzi Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57 – 67, 1982.

[11] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley Professional, 1st edition, June 2013.