Parallel Sorting of Roughly-Sorted Sequences

CSCI 5172 Fall '16 Project

Anthony Pfaff, Jason Treadwell

December 7, 2016

Roughsort is a sorting algorithm that exploits the *radius* of a sequence to beat the linearithmic lower runtime bound of the comparison sort family of algorithms. Its structure invites implementation using parallel execution. Here we present a parallel Roughsort implementation using Nvidia's CUDA platform for heterogeneous GPGPU programming and interpret our results.

1 The Array Sorting Problem

Sorting the elements of an array is among the most classic problems of computer science, often considered to be the most fundamental algorithmic problem [6]. Not merely just for arranging data to make it easier to read, sorting algorithms are frequently employed to render graphics and to improve the performance of other algorithms (e.g., set intersection or finding the unique elements of a list).

The *comparison* sorts are an indispensible family of sorting algorithms containing such classics as Heapsort, Mergesort and Quicksort. True to its name, algorithms in this family determine how to reorder a sequence by comparing its elements against each other according to some natural or explicitly-provided ordering; for the sake of simplicity, we shall only consider the sorting of integral elements in nondecreasing order, backed by random-access array storage[1]. A well-established result is the "linearithmic" $\Omega(n \lg n)$ runtime lower bound for the general problem of sorting an array of n elements by comparison; algorithms such as Mergesort achieve $\Theta(n \lg n)$ runtime (and are thus asymptotically optimal) whereas Quicksort, while usually outperforming Mergesort over arrays that fit in memory, has quadratic-runtime pathological cases.

Comparison-based sorting algorithms, such as Mergesort, often have divide-and-conquer structures that can easily and substantially be sped up by using *nested parallelism* and other means of parallel execution [6].

2 Sorting Roughly-Sorted Sequences

The optimal $\Omega(n \lg n)$ runtime bound of comparison sorting can be beaten when additional analysis is performed on the input sequence [6]. An algorithm by Altman and Igarashi, which we call *Roughsort*, improves upon this lower bound by exploiting the degree to which the unprocessed input array is partially sorted [3]. Specifically, Roughsort sorts an array in $\Theta(n \lg k)$ time, where k is the *radius* of the input sequence.

[1]Sequential-access storage demands distinct and divergent design considerations.

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

$$a_i \leq a_j\,, \quad i < j - k \quad \forall\, 0 \leq i \leq j \leq n-1\,,$$

where a 0-sorted sequence is fully-sorted. The radius of A is the smallest such k for which A is k-sorted. Alternatively, and perhaps more usefully, the radius of A measures how far away its most out-of-order element is from its position in the fully-sorted array:

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

Roughsort exploits the radius of A by using a *halving* algorithm that takes a $2l$-sorted sequence and partially sorts it into an $\lfloor l-1 \rfloor$-sorted sequence. By repeatedly halving the sortedness of A, we fully sort it in $\lg k$ runs, whence the runtime. The radius of A must therefore be given *a priori* or determined from the input in order to effectively perform the algorithm.

Some applications involve sorted arrays that are updated or extended in such a way as to only slightly perturb the ordering (i.e., the radius k of an out-of-order sequence is small) [3]. In such cases, Roughsort could be employed to quickly resort the sequences faster than by using a full $\Theta(n \lg n)$ comparison sort. Since an unsorted sequence of length n can be at most $(n-1)$-sorted, the runtime of Roughsort is $\Theta(n \lg k) = O(n \lg n)$ and is thus asymptotically-optimal among comparison-based sorting algorithms.

Similar to other comparison sorts, Roughsort has divide-and-conquer characteristics that support parallel execution on various multiprocessor models [3, 2].

3 Sequential Implementation

As noted, Roughsort must know the radius k of the input array $A = \{a_0, a_1, \cdots, a_{n-1}\}$ in order to completely sort it. If not known, the radius can be determined in linear time [3], preserving the $\Theta(n \lg k)$ complexity of the algorithm.

To determine the radius, we must first compute the $LR(A) = \{b_i\}$ and $RL(A) = \{c_i\}$ *characteristic sequences*:

$$b_i = \max\{a_0, a_1, \cdots, a_i\}, \quad c_i = \min\{a_i, a_{i+1}, \cdots, a_{n-1}\}, \quad 0 \leq i < n$$

Both sequences are easily computed in linear time and space using simple min/max prefix scans. Note that $c_i \leq b_i \ \forall \ i$. By performing a linear-time scan of these sequences and observing where they "cross" each other, we can compute the *disorder measure* sequence $DM(A) = \{d_i\}$, where each d_i represents how displaced a_i is from its position in the sorted ordering of the elements of A:

$$d_i = \max\big\{\{i - j \mid c_i < b_j\} \cup \{0\}\big\}$$

The radius of A is thus the maximum element from $DM(A)$. We employ a modification of the DM algorithm from [3] where we save memory by simply keeping track of the maximum-encountered d_i instead of storing the entire sequence; our sequential radius determination algorithm thus runs in linear time and requires $2n = \Theta(n)$ space.

The core of the sequential Roughsort implementation is the aforementioned radius halving algorithm from [3]. This algorithm halves the radius k of A in place through the following three steps:

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.

A run of elements is "partitioned" by partially sorting it such that no element before the median of the run exceeds the median and no element thereafter is less than it. We partition each run using the `nth_element` function from the C++ STL, which performs an optimized, linear-time selection algorithm such as Introselect[2] [1].

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

The halving algorithm fails to halve the radius of 1-sorted sequences, since it's impossible to stagger their radial runs; our implementation thus performs a single linear Bubblesort scan to quickly sort such sequences.

For an analysis of the sequential implementation's runtime performance, please see Section 7.

[2]Introselect is based on Quickselect, which resembles a Quicksort that searches for the nth element of the sorted array by only partitioning one side of each pivot.

4 Parallel Acceleration Using CUDA

5 Parallel Radius Determination

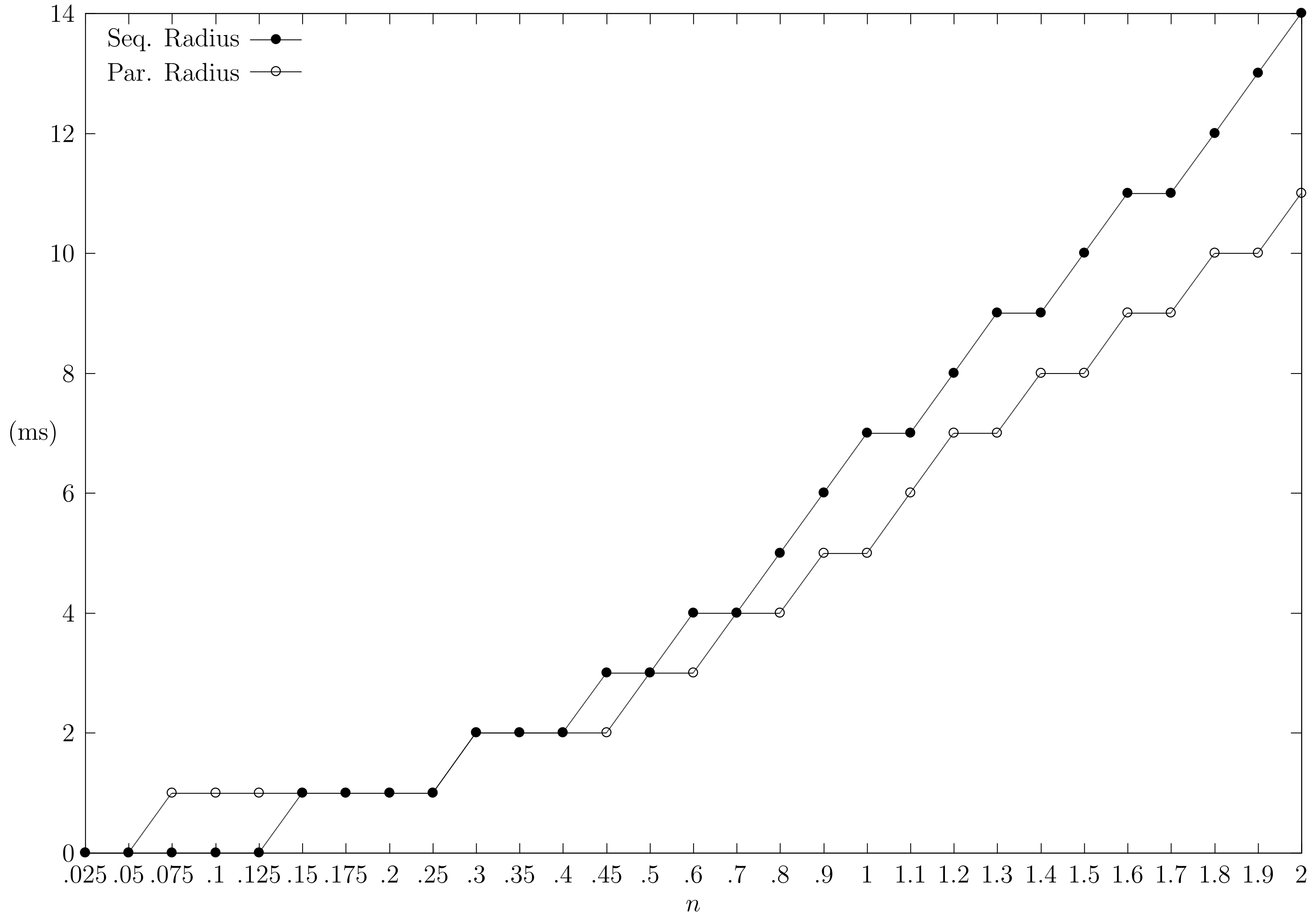

Figure 1: *Radius Determination Runtimes over Arrays of Length* $n \cdot 10^6$, $k = 2$

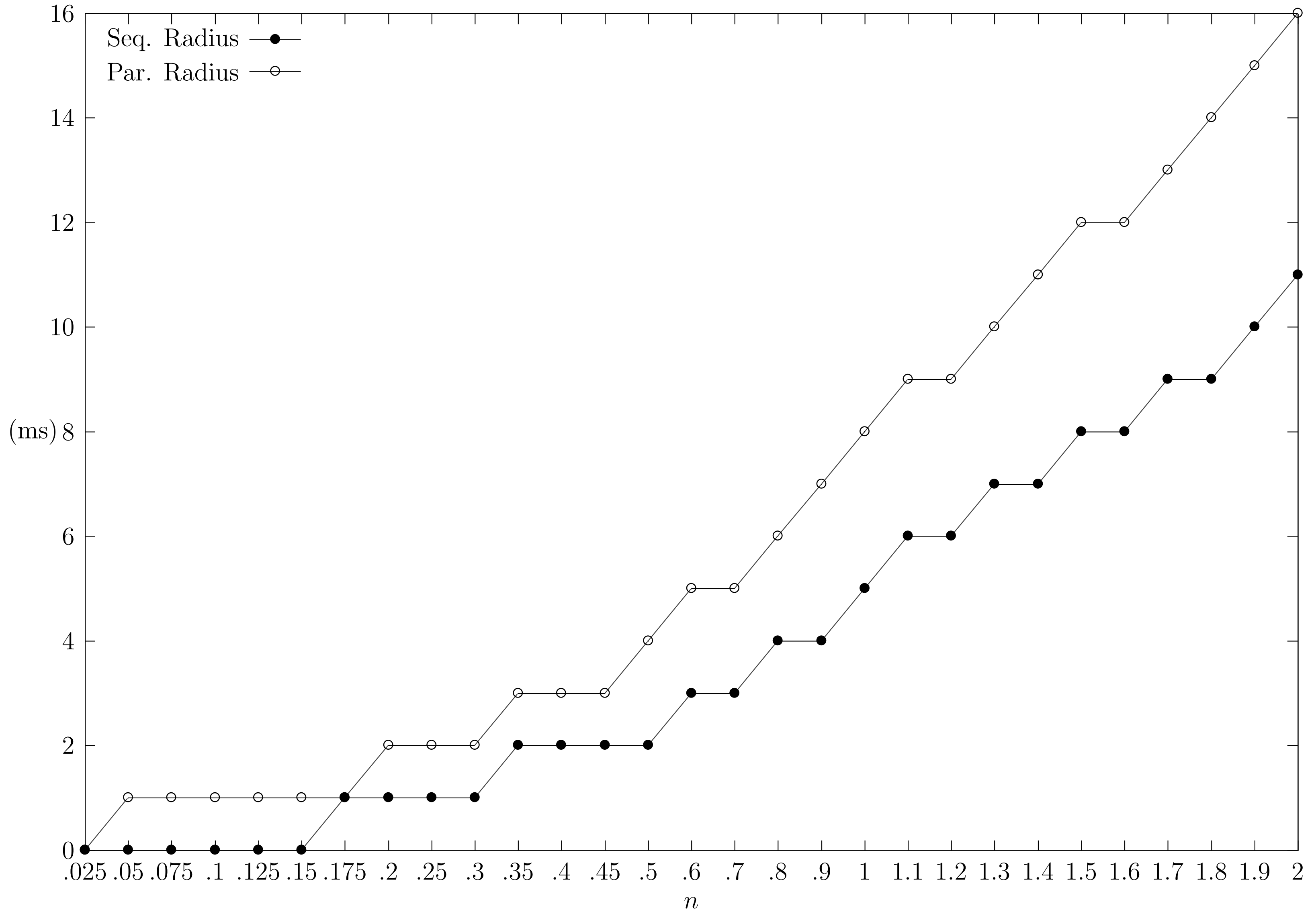

Figure 2: *Radius Determination Runtimes over Arrays of Length* $n \cdot 10^6$, $k = 100$

6 Parallel Roughsort Implementation and Results

rand shuffling

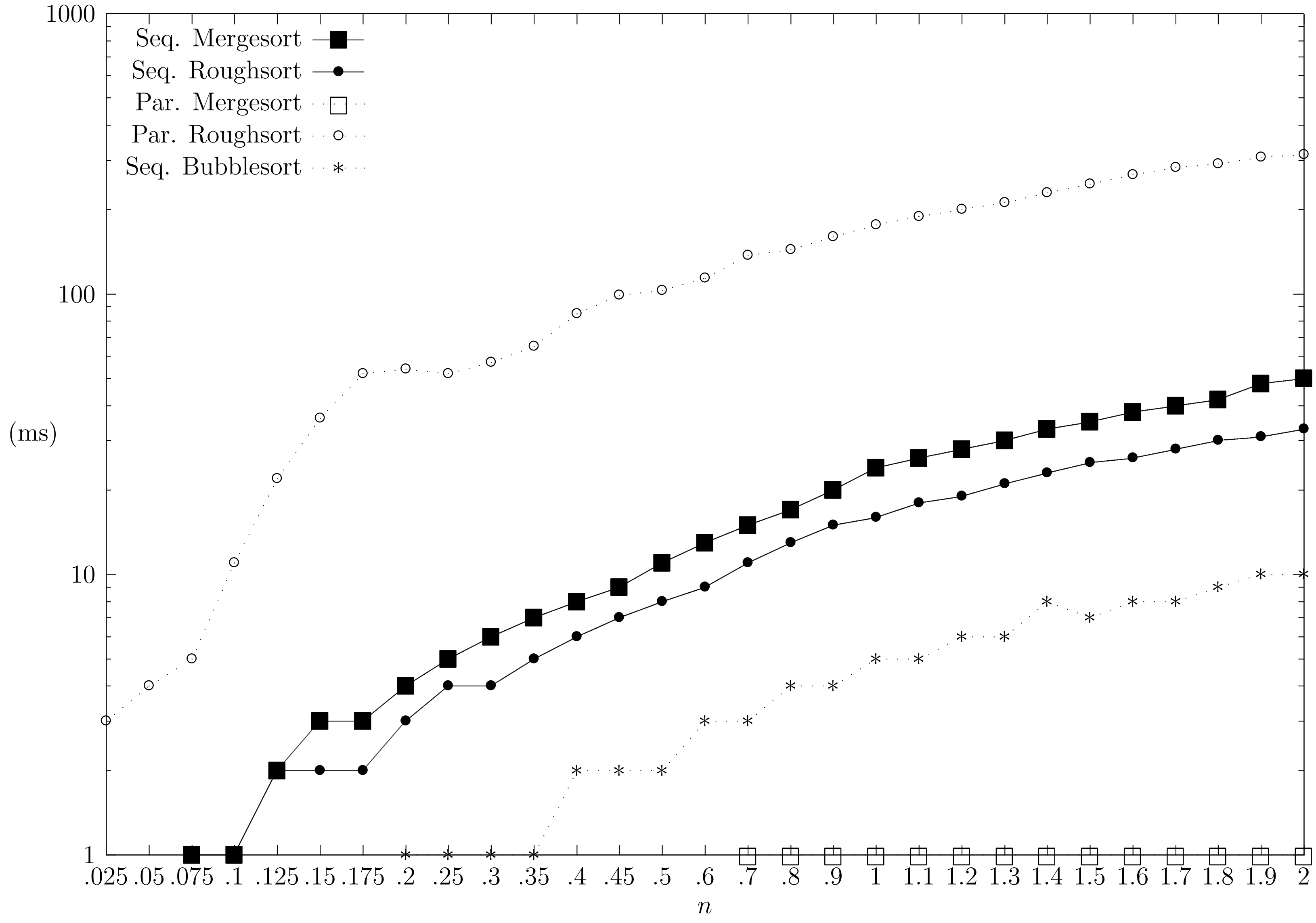

Figure 3: *Sort Runtimes over Arrays of Length* $n \cdot 10^6$, $k = 2$

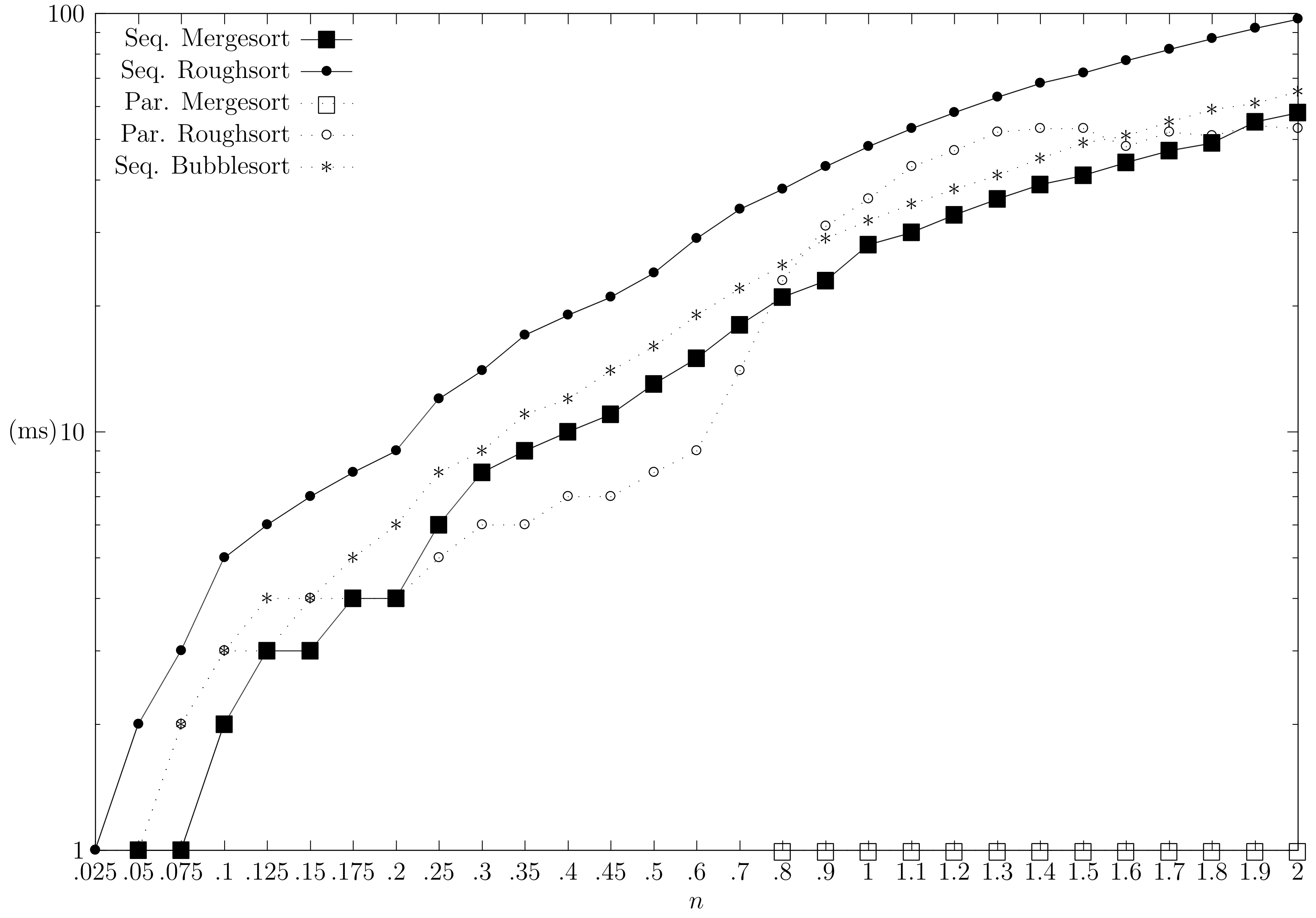

Figure 4: *Sort Runtimes over Arrays of Length* $n \cdot 10^6$, $k = 15$

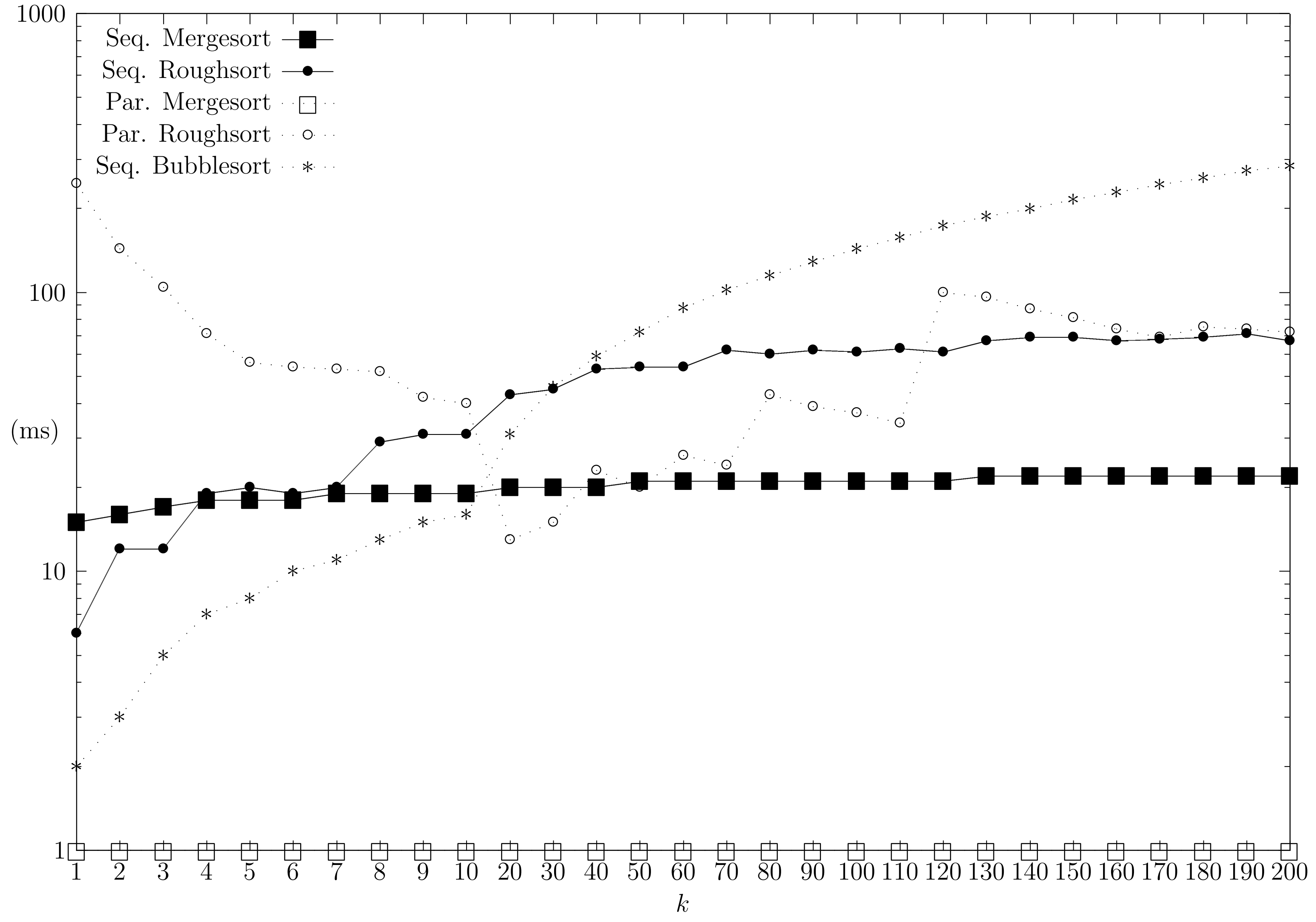

Figure 5: *Sort Runtimes over Arrays of Radius* k, $n = 0.75 \cdot 10^6$

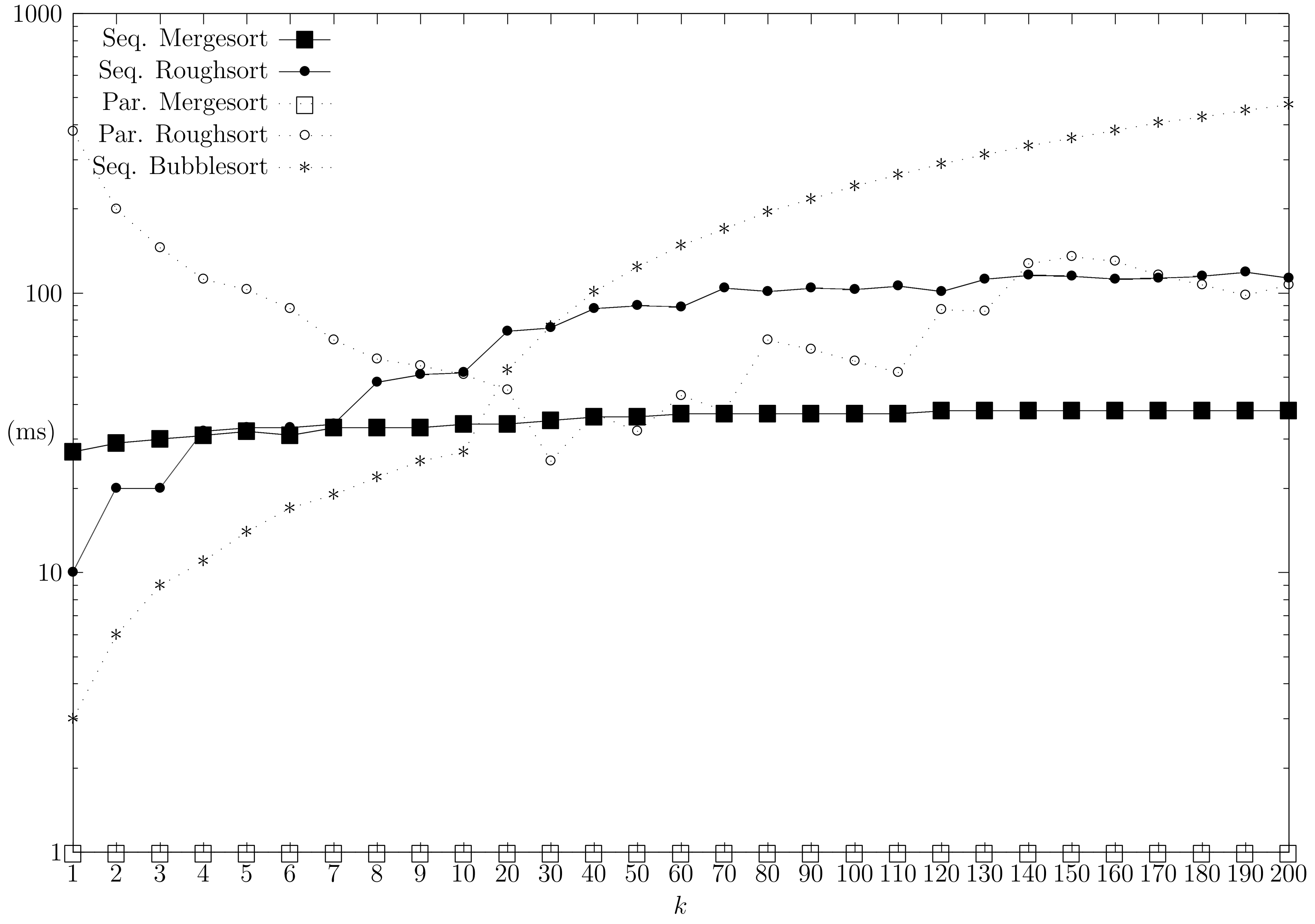

Figure 6: *Sort Runtimes over Arrays of Radius* k, $n = 1.25 \cdot 10^6$

7 Explanation of Results

8 Conclusion and Further Research

References

[1] std::nth_element. `http://en.cppreference.com/w/cpp/algorithm/nth_element`, March 2016. Accessed: 2016-12-07.

[2] T. Altman and B. Chlebus. Sorting roughly sorted sequences in parallel. *Information Processing Letters*, 33(6), February 1990.

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

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

[5] R. Cole and C. Yap. A parallel median algorithm. *Information Processing Letters*, 20(3), April 1985.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

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

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

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

[10] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. September 2009.

[11] NVIDIA Corporation. CUDA C Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide`, September 2016. Accessed: 2016-12-06.

[12] OrangeOwl. Sorting with CUDA libraries. `http://www.orangeowlsolutions.com/archives/900`, May 2014. Accessed: 2016-12-06.

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

[14] R. Senser. *GPU Declarative Framework*. PhD thesis, University of Colorado Denver, November 2014.

[15] Y. Shiloach and U. Vishkin. *Finding the maximum, merging and sorting in a parallel computation model*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1981.

[16] Y. Shiloach and U. Vishkin. An O(log n) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

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