Parallel Sorting of Roughly-Sorted Sequences

CSCI 5172 Fall '16 Project

Anthony Pfaff, Jason Treadwell

December 8, 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 [8]. 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 indispensable 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 [8].

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 [8]. 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 [5]. 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) [5]. 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 character-

istics that support parallel execution on various multiprocessor models [5, 4].

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 [5], preserving the $\Theta(n \lg k)$ complexity of the algorithm.

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

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

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:

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

The radius of A is thus the maximum element from $DM(A)$. We employed a modification of the DM algorithm from [5] 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 [5]. 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 implements an optimized, linear-time selection algorithm [8] such as Introselect[2] [2].

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 9.

[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.

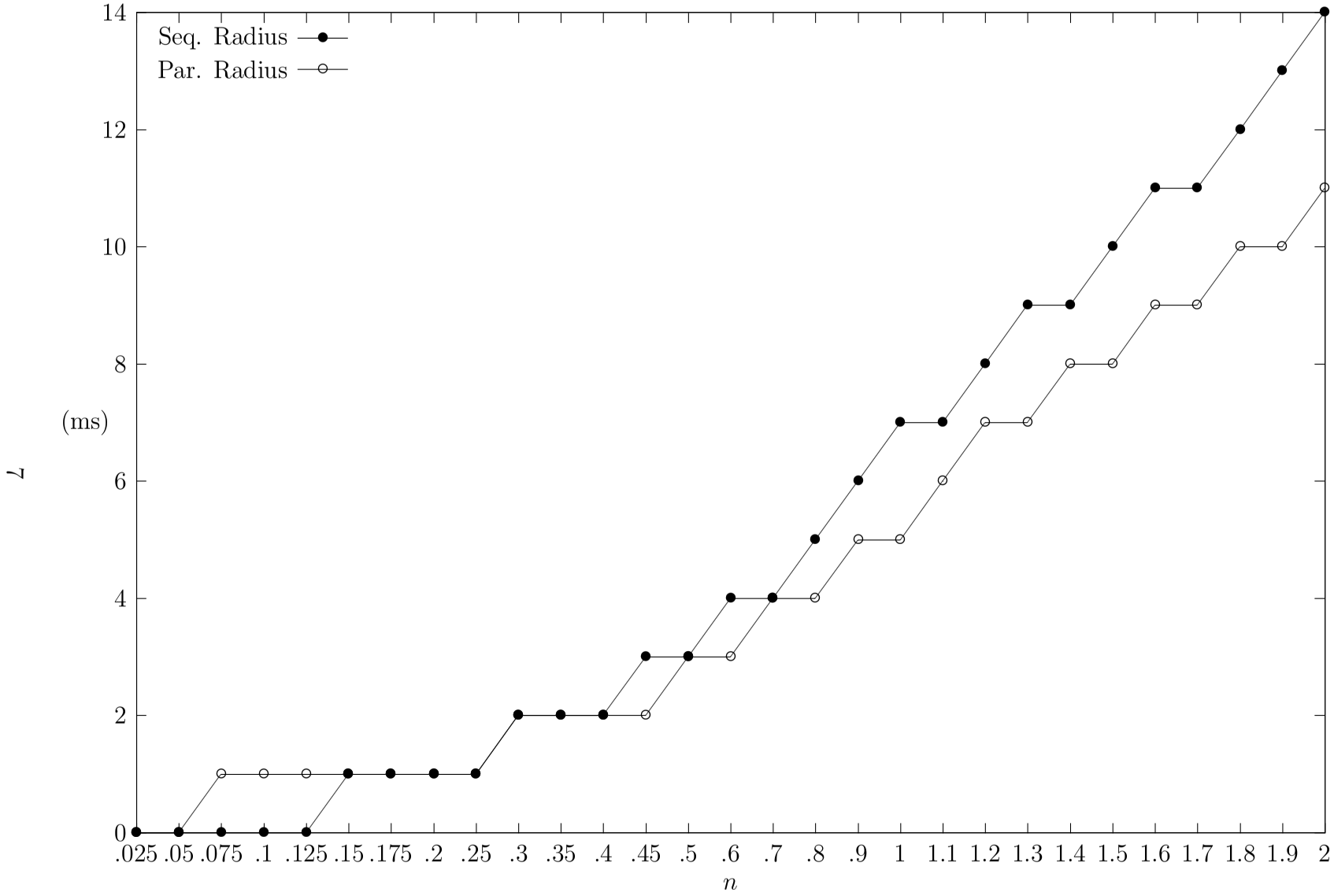

Figure 1: *Radius Determination Runtimes over Arrays of Length* $n \cdot 10^6$, $k = 2$

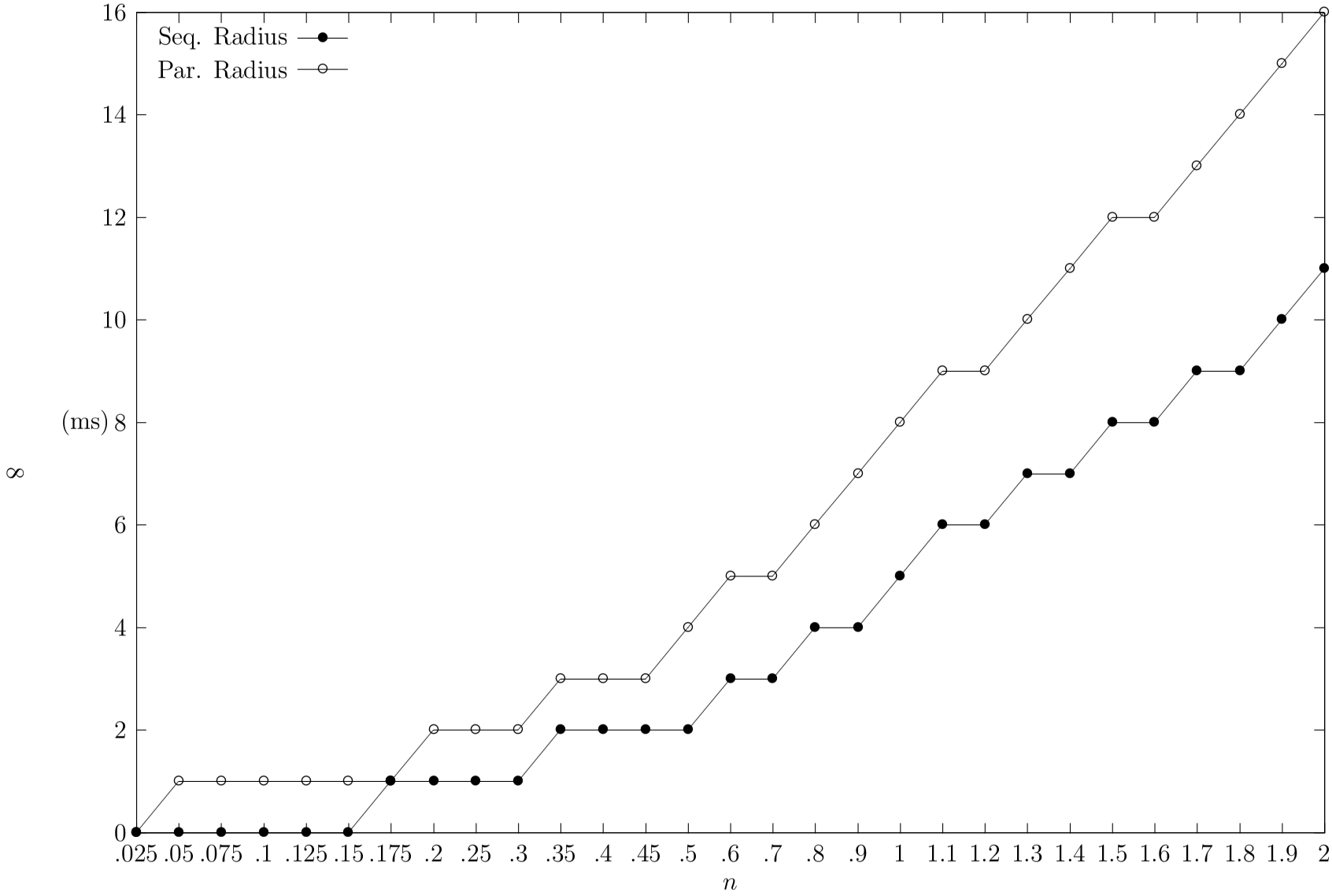

Figure 2: *Radius Determination Runtimes over Arrays of Length* $n \cdot 10^6$, $k = 100$

7 Parallel Roughsort Implementation and Results

We first attempted to implement the parallel Roughsort described in [5]. This approach follows the sequential version from Section 3, except in that the segment partitions are all launched in parallel at each of the 3 steps, synchronizing between steps. We employed Nvidia's *Thrust* library, which provides a parallel CUDA implementation of the C++ STL.

Regrettably, Thrust has yet to provide an `nth_element` selection algorithm implementation, so we replaced it with calls to Thrust's `sort` algorithm, which performs a "state-of-the-art", highly-optimized radix sort implementation when invoked over arrays of primitive integer elements [15, 13]. Radix sort is a generalized counting sort that runs in linear time when operating over fixed-size array elements. While radix sort requires additional storage (unlike Introselect, which was used in the sequential implementation), this additional storage ought to be acceptable as the segments we wish to partition ought to be relatively small.

Each Thrust `sort` call issues at least one kernel launch; we attempted to run radix sorts over the entire array in parallel by establishing a fixed number of CUDA *streams*. Whereas kernels launched through the same stream are executed in order, up to 16 kernels launched through distinct streams can be run concurrently on the GPU, available resources permitting [14]. We established a fixed number of CUDA streams and launched each `sort` call through them in a circular, round-robin fashion.

Unfortunately, while this implementation was very simple to code and correctly sorted its input, it was thousands of times slower than the sequential implementation. Running the implementation through CUDA's `nvvp` profiler showed that the

`sort` calls weren't being launched in parallel as expected, possibly due to some non-obvious synchronization mechanism that made concurrent launches of this function impossible.

We then implemented the parallel Roughsort approach described in [4], which prescribes the parallel Mergesort algorithm by Cole in lieu of weak partitioning and is implemented as follows for sorting the array A with radius k in place:

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.

Unlike the sequential algorithm, this approach fully sorts each array segment and thus needs only two batches of sorts without intra-radial staggering. Furthermore, just a single iteration of this process suffices to fully sort the k-sorted input.

We implemented this approach using a CUDA kernel where each thread in the launch is responsible for sorting exactly one segment from step 1 or 2; the steps are divided into separate kernel launches, each given an offset from which begin segmenting the array. While the threads in each launch run concurrently[3], each thread sorts its designated segment sequentially. In contrast to [4], we used Thrust's radix sort instead of its Mergesort implementation in order to leverage the linear runtime of the former. While this approach to the parallel Roughsort was far more successful than the previous, it still struggled against its competitors (see Section 9).

[3]For large arrays, the GPU can't host every thread simultaneously and must sequentially execute the thread blocks in batches.

8 Random k-Sorted Sequence Generation

We developed a k-sorted random sequence generator to test our implementations. The generator takes k and the length of the desired array n as arguments (specified by the user on the command line) and allocates a contiguous array of n 32-bit random integers. In order to avoid pseudorandom statistical weaknesses, we sourced these random integers from the high-end, cryptographically-secure hardware random number generator included in recent Intel processors [11].

We k-sort the random array A by fully sorting it using the C++ STL before perturbing the sorted elements. This perturbance is achieved by randomly choosing sorted elements a_p, a_q such that $q - p = k$ in order to guarantee the k-sortedness of the sequence. To increase the rigor of the randomized k-sorting, each consecutive, disjoint run of at most $k + 1$ array elements (excluding a_p and a_q) is then randomly shuffled to produce a random subarray that is itself at most k-sorted.

9 Implementation Performance Analysis

We tested our sequential and parallel Roughsort implementations against sequential and parallel Mergesort (as implemented by the C++ STL and by CUDA Thrust, respectively) as well as against our simple, sequential Bubblesort implementation. We chose Mergesort since it's a classic sort algorithm that also scales very well with parallel execution, capable of achieving a speedup of at least $\Theta(n/\lg^2 n)$ when given unlimited processors for a runtime of $\Theta(\lg^3 n)$ [8]. While Bubblesort has a quadratic runtime upper bound, its lightweight, linear lower bound emerges when

given nearly-sorted sequences and provides a means of comparing our implementation's performance for small values of k.

All tests were run on a workstation sporting a high-end, quad-core Intel CPU as well as an Nvidia GeForce GTX 1080—a recent and high-end[4], consumer-grade GPU suitable for heavy computational workloads over 32-bit values.

The time needed to transfer data to the GPU prior to running the parallel sorts isn't counted in the runtime results, since we'd expect such sorting operations to run as part of larger GPU workloads and not in isolation. Since our parallel radius determiner wasn't complete until late in development and ran slower than the sequential implementation over arrays with larger radii, our parallel Roughsort implementation used the sequential implementation to compute k, the runtime of which wasn't figured into the results.

Figures 3 and 4 compare all five sorting algorithms, varying n while fixing k to 2 and 15, respectively. Roughsort ought to have a runtime advantage with the smaller radius that dissipates as we increase k. Figures 5 and 6 invert the axes and vary k while fixing n at $750 \cdot 10^3$ and $1.25 \cdot 10^6$, respectively. For each tested (n, k), all five sorters were run over the same, k-sorted random array of length n; their runtimes were averaged over eight such runs. The runtime axis is logarithmic in every graph.

Figure 3 demonstrates the utility of the sequential Roughsort algorithm, which beats even the highly-optimized Mergesort from the C++ STL at handling the nearly-sorted input arrays. This victory is greatly overshadowed by the sequential Bubblesort, which is operating close to its optimal domain.

[4]At the time of this writing.

The next battery of tests in Figure 4 raises k to 15, causing the sequential Roughsort's advantage over Mergesort to evaporate. The parallel Roughsort generally outperforms Bubblesort here (which fairs better than expected against the sequential Mergesort in a trend that can't last for much longer to the right of the graph) and defeats the sequential Roughsort. Unexpectedly, the parallel implementation exhibits about a factor-of-ten speedup over the $k = 2$ graph; this shows that the overhead of launching many CUDA threads over small array subsequences substantially counterbalances the greater runtime of sorting larger subsequences.

Figures 5 and 6 fix n and vary k, yielding similar results. For arrays of these lengths, the parallel Roughsort's crossover point lies somewhere around the interval $20 \leq k \leq 30$, where it manages to best the three sequential algorithms. An invariant among all four graphs is the utter dominance of the parallel Mergesort implementation provided by Thrust, whose runtimes never depart from the x-axis.

These results suggest that our parallel Roughsort's constant runtime factor and crossover point are relatively high. We were unable to observe a benefit from its superior asymptotic bound over nearly-sorted arrays that could fit within the GPU's memory.

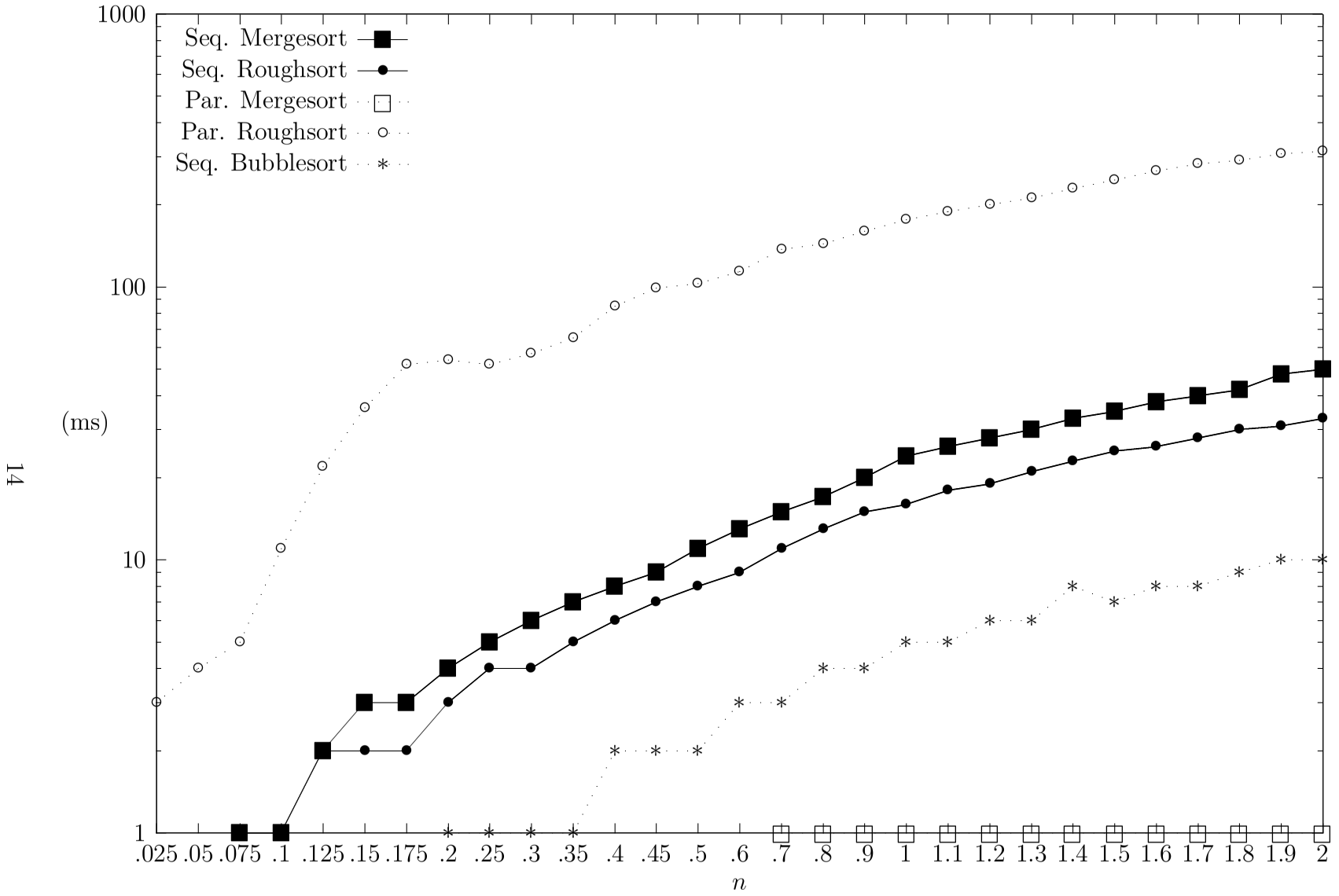

Figure 3: *Sort Runtimes over Arrays of Length* $n \cdot 10^6$, $k = 2$

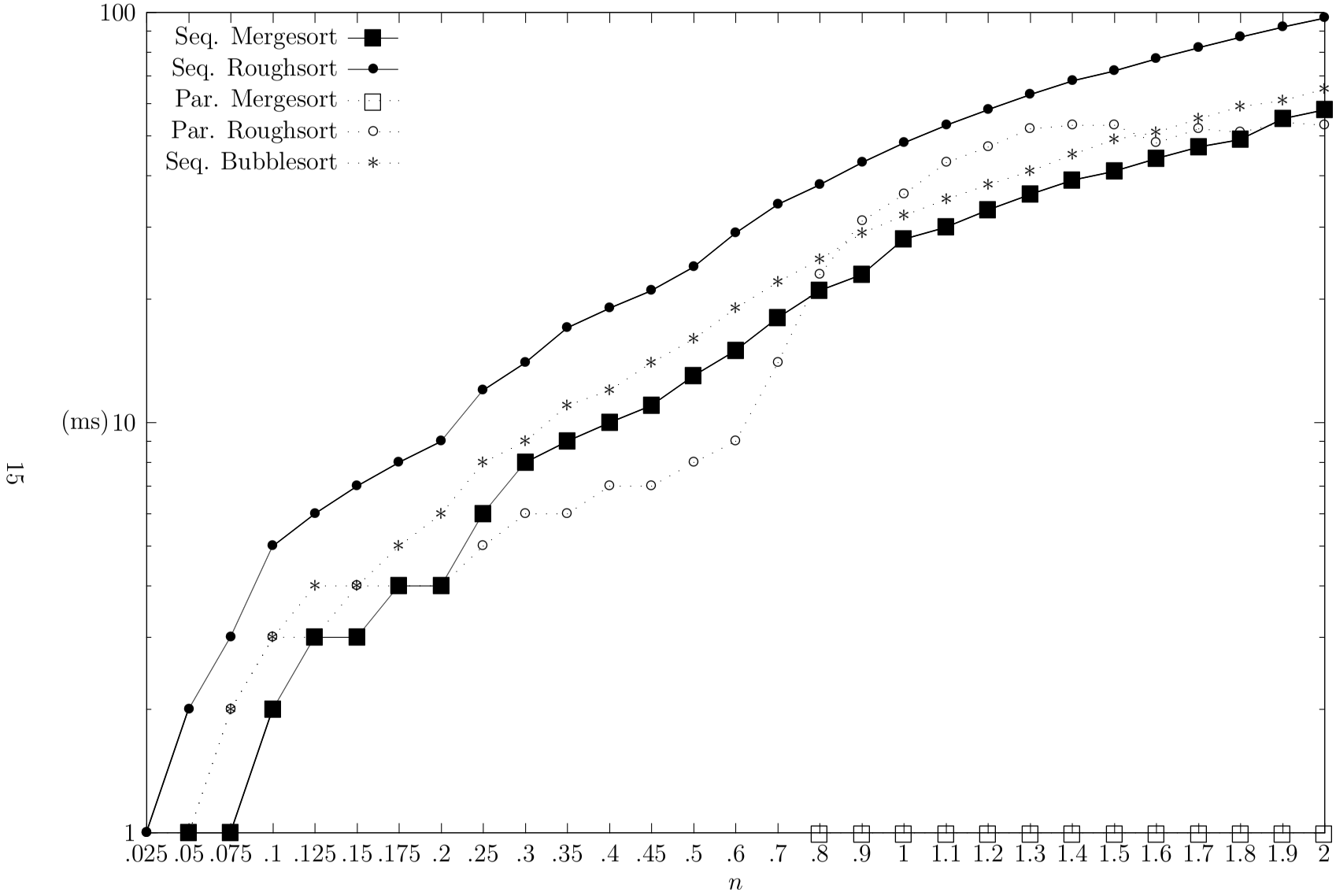

Figure 4: *Sort Runtimes over Arrays of Length* $n \cdot 10^6$, $k = 15$

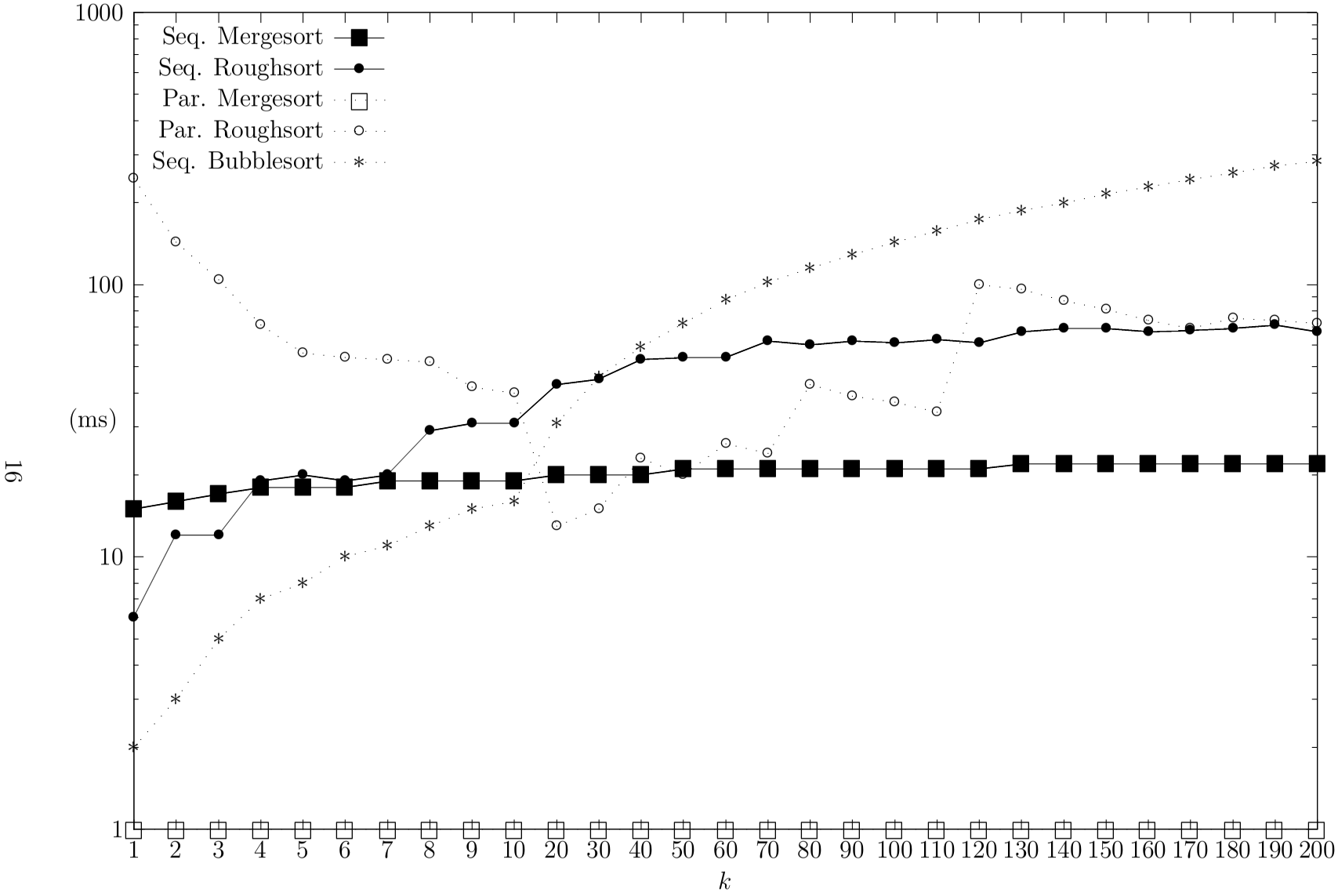

Figure 5: *Sort Runtimes over Arrays of Radius k*, $n = 0.75 \cdot 10^6$

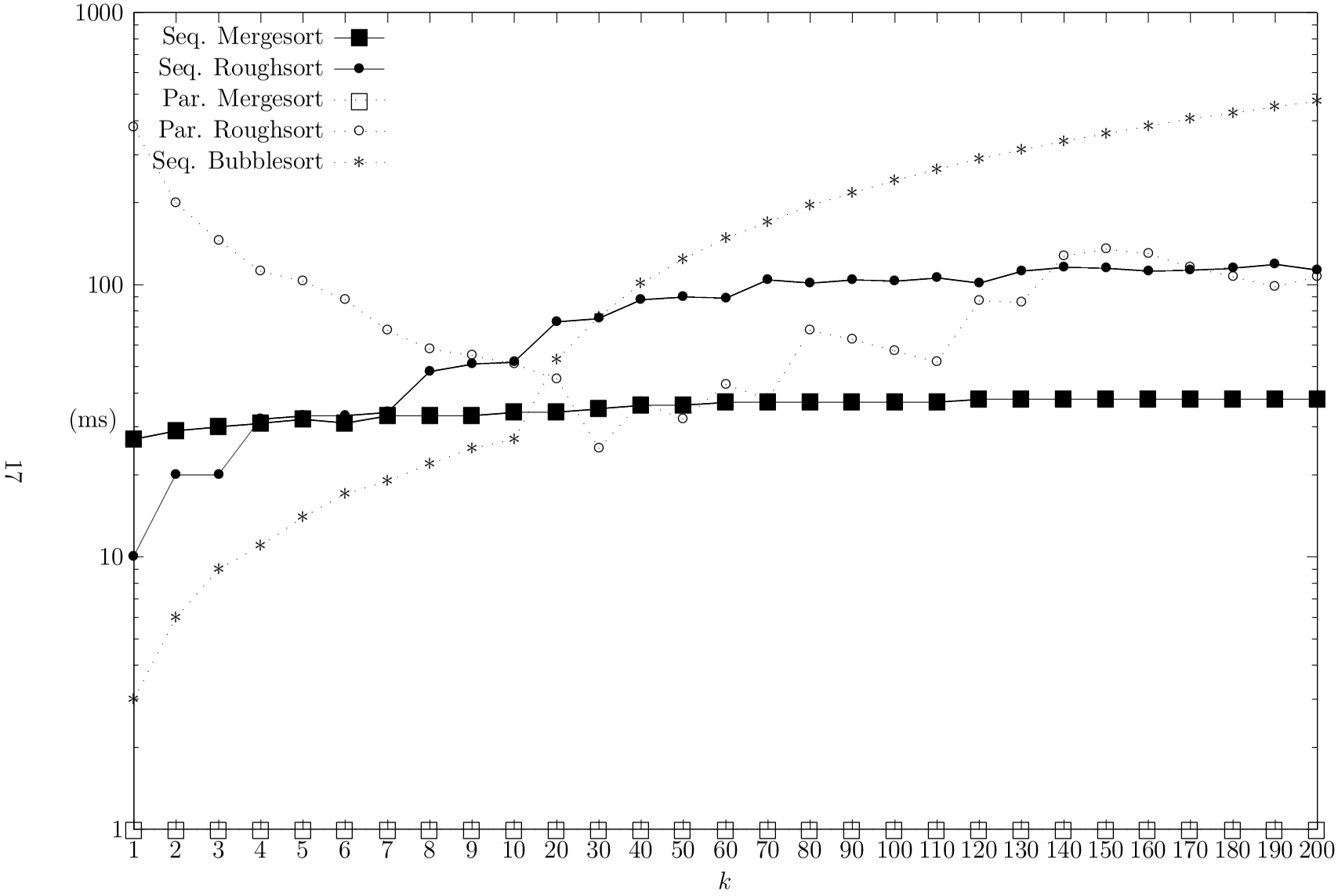

Figure 6: *Sort Runtimes over Arrays of Radius k*, $n = 1.25 \cdot 10^6$

10 Explanation of Results

11 Conclusion and Further Research

An implementation of Cole's parallel median algorithm might enable a more efficient parallel Roughsort [7].

References

[1] thrust: Extrema. `https://thrust.github.io/doc/group__extrema.htm`, March 2015. [Online; accessed 7-December-2016].

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

[3] Thrust :: Cuda toolkit documentation. `http://docs.nvidia.com/cuda/thrust/`, September 2016. [Online; accessed 7-December-2016].

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

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

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

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

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

[9] Frank Dehne and Kumanan Yogaratnam. Exploring the limits of gpus with parallel graph algorithms. *arXiv preprint arXiv:1002.4482*, 2010.

[10] 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.

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

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

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

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

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

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

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

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

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

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