COSC6326 Programming Assignment 1

Michael Yantosca

Contents

1 Introduction

As problem sizes continue to scale, it would behoove algorithmic researchers to explore not only more distributed and decentralized approaches but especially randomized versions of these approaches. One particular information retrieval problem, namely the selection of the ith element among a large set of n inherently ordered elements provides a basic but effective overview of some of the challenges at hand and the benefits that a randomized, distributed approach provides along with some of the insights that implementing such a system may offer, not only in the field of randomized algorithms, but in the field of deterministic algorithms, as well.

2 Analysis

2.1 `nkith`

Algorithm 1 MP-Ith-Select, Distributed Algorithm for Selecting the ith Element of n Numbers

Require: G, a PRNG[4]
Require: n, global population size
Require: r, node rank
Require: k, number of nodes
Require:

$$m \leftarrow n \div k + \begin{cases} 1, & \text{if } n \bmod k > r \\ 0, & \text{otherwise} \end{cases}, \text{ local population size}$$

Require: $p \leftarrow 0$, round-robin pivot selection counter
Require: $b_l \leftarrow 0$, live local population lower bound
Require: $b_u \leftarrow m$, live local population upper bound
Require: $|S_{1,r}| \leftarrow 0$, local cardinality counter for elements below the pivot
Require: $|P_r| \leftarrow 0$, local cardinality counter for elements less than or equal to the pivot
Require: $|S_1| \leftarrow 0$, global cardinality counter for elements below the pivot
Require: $|P| \leftarrow 0$, global cardinality counter for elements less than or equal to the pivot

Seed G with the current epoch time.
for $j = 0$ to $m - 1$ **do**
  $M[j] =$ Generate(G)
Qsort(M)[1]
while $\neg(|S_1| < i$ and $|S_1| + |P| \geq i)$ **do**
  $c_r \leftarrow M[b_l + (b_u - b_l)/2]$
  All nodes send c_r to the root node.
  if $r = 0$ **then**
    $c = c_p$
    $p = (p + 1) \bmod k$
    Broadcast c to all nodes.
  $|S_{1,r}| \leftarrow b_l$
  while $M[|S_{1,r}|] < c$ and $|S_1| < b_u$ **do**
    $|S_{1,r}| = |S_{1,r}| + 1$
  $|P_r| \leftarrow |S_{1,r}|$
  while $M[|P_r|] == c$ and $|P_r| < b_u$ **do**
    $|P_r| = |P_r| + 1$
  All nodes send $|S_{1,r}|$ and $|P_r|$ to the root node.
  if $r = 0$ **then**
    $|S_1| \leftarrow \sum_{r=0}^{k-1} |S_{1,r}|$
    $|P| \leftarrow \sum_{r=0}^{k-1} |P_r|$
    Broadcast $|S_1|$ and $|P|$ to all nodes.
  if $|S_1| > i - 1$ **then**
    $b_u \leftarrow |S_{1,r}|$
  else if $|P| < i$ **then**
    $b_l \leftarrow |P_r|$
The root node announces c as the ith element.

A full analysis of Algorithm 1 is omitted since it does not faithfully reproduce the sequential algorithm. Indeed, a proper analysis is beyond the scope of this assignment since the implementation of QSORT is dependent on the particular POSIX implementation on which the code is run. The pseudocode is included for reference and completeness since this first attempt served as the progenitor for the correct implementation in Algorithm 3.

2.2 `nkithr`

Algorithm 2 RANDOM-PIVOT-SELECT, Distributed Algorithm for Selecting a Pivot Candidate

Require: k, number of nodes
Require: b_l, live local population lower bound
Require: b_u, live local population upper bound
  $G_{cand} \leftarrow$ uniform distribution over $[b_l, b_u)$[6]
  $c_r \leftarrow M[\text{GENERATE}(G_{cand})]$
  $w_r \leftarrow b_u - b_l$
  All nodes send c_r and w_r to the root node.
  if $r = 0$ **then**
    $w \leftarrow \sum_{r=0}^{k-1} w_r$
    $G_{pivot} \leftarrow$ distribution over $[0, k)$ weighted by w_r/w.[3]
    $p = \text{GENERATE}(G_{pivot})$
    $c = c_p$
    Broadcast c to all nodes.

Theorem 1. *Algorithm 3 determines the ith element of a set of n numbers distributed over k nodes with time complexity $O(\log n)$ rounds and message complexity $O(k \log n)$ on average.*

Lemma 1. *Algorithm 3 determines the ith element of a set of n numbers distributed over k nodes with time complexity $O(\log n)$ rounds on average.*

Proof. The proof follows the sketch outlined by the proof for the time complexity of Randomized Quicksort given in the sequential algorithms text by Pandurangan[9, p. 168]. For simplicity, we limit the proof to the case where the ith element is the median, i.e., $\lfloor \frac{n}{2} \rfloor$ among n numbers and the pivot selection is random and independent of the population distribution.

Algorithm 3 MP-Ith-Select-Revised, Distributed Algorithm for Selecting the ith Element of n Numbers

Require: G_{popl}, a population PRNG[4]
Require: n, global population size
Require: r, node rank
Require: k, number of nodes
Require: c, the current pivot
Require:

$$m \leftarrow n \div k + \begin{cases} 1, & \text{if } n \bmod k > r \\ 0, & \text{otherwise} \end{cases}, \text{ local population size}$$

Require: $b_l \leftarrow 0$, live local population lower bound
Require: $b_u \leftarrow m$, live local population upper bound
Require: $|S_1| \leftarrow 0$, global cardinality for elements below the pivot
Require: $|P| \leftarrow 0$, global cardinality for elements less than or equal to the pivot
  Seed G_{popl} with the current epoch time.
  for $j = 0$ to $m - 1$ **do**
    $M[j] =$ Generate(G_{popl})
  while $\neg(|S_1| < i$ and $|S_1| + |P| \geq i)$ **do**
    Random-Pivot-Select(b_l, b_u) {Algorithm 2}
    $s_1 \leftarrow b_l$
    $s_p \leftarrow 0$
    $s_2 \leftarrow b_u$
    $j \leftarrow b_l$
    for $j = b_l$ to $b_u - 1$ **do**
      if $M[j] < c$ **then**
        $Q[s_1] \leftarrow M[j]$
        $s_1 = s_1 + 1$
      else if $M[j] > c$ **then**
        $s_2 = s_2 - 1$
        $Q[s_2] \leftarrow M[j]$
      else
        $s_p \leftarrow s_p + 1$
    $s_p \leftarrow s_p + s_1$
    for $j = s_1$ to s_p **do**
      $Q[j] \leftarrow c$
    if $b_l < b_u$ **then**
      Copy $Q[b_l, b_u - 1]$ to $M[b_l, b_u - 1]$.
    All nodes send s_1 and s_p to the root node.
    if $r = 0$ **then**
      $|S_1| \leftarrow \sum_{r=0}^{k-1} s_{1,r}$
      $|P| \leftarrow \sum_{r=0}^{k-1} s_{p,r}$
      Broadcast $|S_1|$ and $|P|$ to all nodes.
    if $|S_1| > i - 1$ **then**
      $b_u \leftarrow s_1$
    else if $|P| < i$ **then**
      $b_l \leftarrow s_p$
  The root node announces c as the ith element.

The first order of business is to simplify the proof so that we can speak in global terms without having to perform complicated if not intractable calculations on local probabilities.

Lemma 2. *The pivot selected by Algorithm 2 chooses any member of the live (i.e., considered) population n_j in the jth iteration with uniform probability $\frac{1}{n_j}$.*

Proof. The local candidate is selected from the local live population of size $m_j = b_{u,j} - b_{l,j}$ with $\frac{1}{m_j}$ probability. The corresponding weight sent to the root node with the candidate is m_j. The global live population is faithfully recorded at the root as $n_j = \sum_{h=0}^{k-1} m_{h,j}$. The weights are normalized against n_j so that each becomes respectively $\frac{m_{h,j}}{n_j}$. The node identifier is selected according to a discrete distribution weighted with the aforementioned weights. The probability that any given element will be selected as the pivot is given as follows:

$$\begin{aligned} P(\text{element selected globally}) &= P(\text{node selected globally}|\text{element selected locally}) \\ &\quad \cdot P(\text{element selected locally}) \\ &= \frac{m_{h,j}}{n_j} \cdot \frac{1}{m_{h,j}} \\ &= \frac{1}{n_j} \end{aligned}$$

Thus, Algorithm 2 selects each pivot in each iteration from among the live element population with uniform probability. □

By Lemma 2, we can consider the partitioning of the population in a global sense so long as the implementation faithfully reports the cardinalities of the subsets of S_1 (less than the pivot) and P (equal to the pivot). We know that this is the case since the element-wise comparisons done locally appropriately bin the live local population into the respective subsets, and the cardinalities of the subsets are summed at the root and rebroadcast to all nodes for the purpose of updating the live population boundaries.

As with the proof for Theorem 8.8[9, pp. 168-169], let us consider then the division of outcomes into good and bad sets. Let the good outcome be that where the pivot is chosen in the middle third of the live global population, i.e., that neither $|S_1|$ nor $|S_2|$ (greater than the pivot) exceeds $\frac{2n_j}{3}$, and let the bad outcome be the converse.

We cannot have more than $\log_{3/2} n$ good outcomes, or else the population will have been exhausted by the constant reduction by $\frac{2}{3}$. , the expectation for execution time is $3c \log n$, where c is a fixed constant independent of the problem size such that $\log_{3/2} n < c \log n$.

If we consider longer paths of execution that have at most $c \log n$ good outcomes but are longer than that, i.e., paths of length $ac \log n$ for some $a > 1$, we can use the Chernoff Lower Tail Bound[9, p. 323] to provide a probabilistic limit on the execution time in terms of number of iterations. Since good outcomes occur with probability $\frac{1}{3}$, the expectation of good outcomes on a path of length $ac \log n$ would be $\frac{1}{3}ac \log n$. By choosing $\delta = 3/a$, we get a probability $P(X_{good} < c \log n) \leq \frac{1}{n^2}$. Consequently, with n elements in play at the start, the single path probability becomes union bounded as $1/n$[9, p. 169].

Thus, the number of iterations is on the order of $O(\log n)$ rounds with high probability since the greatest possible probability to exceed $ac \log n$ rounds is $1/n$. □

Lemma 3. *Algorithm 3 determines the ith element of a set of n numbers distributed over k nodes with message complexity $O(k \log n)$ on average.*

Proof. By Lemma 1, we know that Algorithm 3 takes $O(\log n)$ iterations on average. Within each iteration, the following message exchanges occur:

1. $k - 1$ nodes send pivot candidates to the root.
2. $k - 1$ nodes send pivot weights to the root.
3. The root sends the pivot-elect to $k - 1$ nodes.
4. $k - 1$ nodes send local cardinalities s_1 and s_p to the root.
5. The root sends global cardinalities $|S_1|$ and $|S_1| + |P|$ to $k - 1$ nodes.

With 5 exchanges of $k - 1$ messages, each iteration produces $5k - 5$. Even if we are nitpicky and account the 2 element local cardinality messages as 2 exchanges, we still only have $6k - 6$, which in either case is $O(k)$ messages per iteration. Multiplying by $O(\log n)$ iterations, we arrive at total message complexity of $O(k \log n)$, which was to be shown. □

Proof. By Lemma 1 and Lemma 3, we see that the two complexities are satisfied. Hence, Theorem 1 holds. □

3 Results

3.1 Test Procedures

Tests to validate correctness were performed locally on a dual-core laptop [1] running Pop!OS. [2] Tests for which results were collected systematically and graphed were done on the UH `crill` cluster with the following parameters:

- $n \in 2^{[10,28]}$, the total population size
- $k \in \{1, 2, 4, 8, 16, 32\}$, the number of distributed nodes
- $i \in \{1, n/2, n\}$, the sought element index

Each parameter combination was executed at least 10 times.

Several executables or versions of executables were used over the course of experimentation. Here are the primary variants.

`nkmax` A specialized implementation that used a simple local variable and stream processing to reduce local candidacy to one element during the generation phase.

`nkith` An ill-considered attempt to try and pre-sort local populations so that the exchange phases were simple cursor movements.

`nkithr` A revised version of `nkith` that properly implemented a distributed version of the given sequential algorithm. The first set of test runs used `std::default_random_engine`[2]. Out of a (largely unfounded) concern that the C++ default random engine was generating skewed pivot choices, the second set of test runs used `std::knuthb`[5]. A third set of test runs was executed with a deterministic pivot selection mechanism similar to that used in `nkith` but with additional logic to skip candidates supplied by nodes whose search spaces were actually $\emptyset$.

`popldump` In order to empirically verify correctness, each of the above programs was instrumented with MPIO[8] commands so that the entire population of a given run could be saved and read later. The program takes the native byte representations of the unsigned 32-bit integers comprising the population and prints their decimal form with one number per line.

[1] 2 x Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz.
[2] An Ubuntu 18.04 variant.

The reader is directed to the accompanying `README.md` for explicit usage instructions on the various programs as well as for a deeper explanation of implementation considerations, engineering tradeoffs, and known issues with the programs (mostly concerning the absence of defensive logic against bad command-line arguments). A command sequence for verifying the output correctness is given in the section on `popldump`. It should be noted that the MPI 2.1 standard[7] and g++-7.2 was used for development since they were available through the development laptop's package system, but the program compiled and ran on the `crill` cluster with g++-5.3.0 and MPI 3.0. The code requires the C++-11 standard.

3.2 Finding the Maximum

The empirical complexity of finding the maximum is shown in Figure 1. The bespoke `nkmax` implementation performs best in terms of time and message complexity. This is to be expected since finding the max requires a simple counter at each node and only one round of exchange. The rejection occurs during generation. Consequently, the total time required for `nkmax` corresponds to the total time minus the exchange time for `nkithr`.

Interestingly, the span of the time complexity in terms of rounds corresponds almost exactly to the base-2 exponent of the population size for `nkithr`. The original `nkith` implementation with its naive deterministic pivot selection appears to have double the span of the revised version whether the pivot was chosen deterministically or at random. This is reasonable since the revised deterministic pivot selection depends on the calculated pivot weights to void "non-participating" nodes and avoid wasting iterations.

The total time complexity of `nkith` is clearly overshadowed by the local pre-sort time since the doubling of exchange time is not enough to explain the difference of *half an order of magnitude* with respect to `nkithr`. It should be noted that early experiments with trying to do an insertion sort during the generation phase performed even more absymally, as might be expected.

3.3 Finding the Minimum

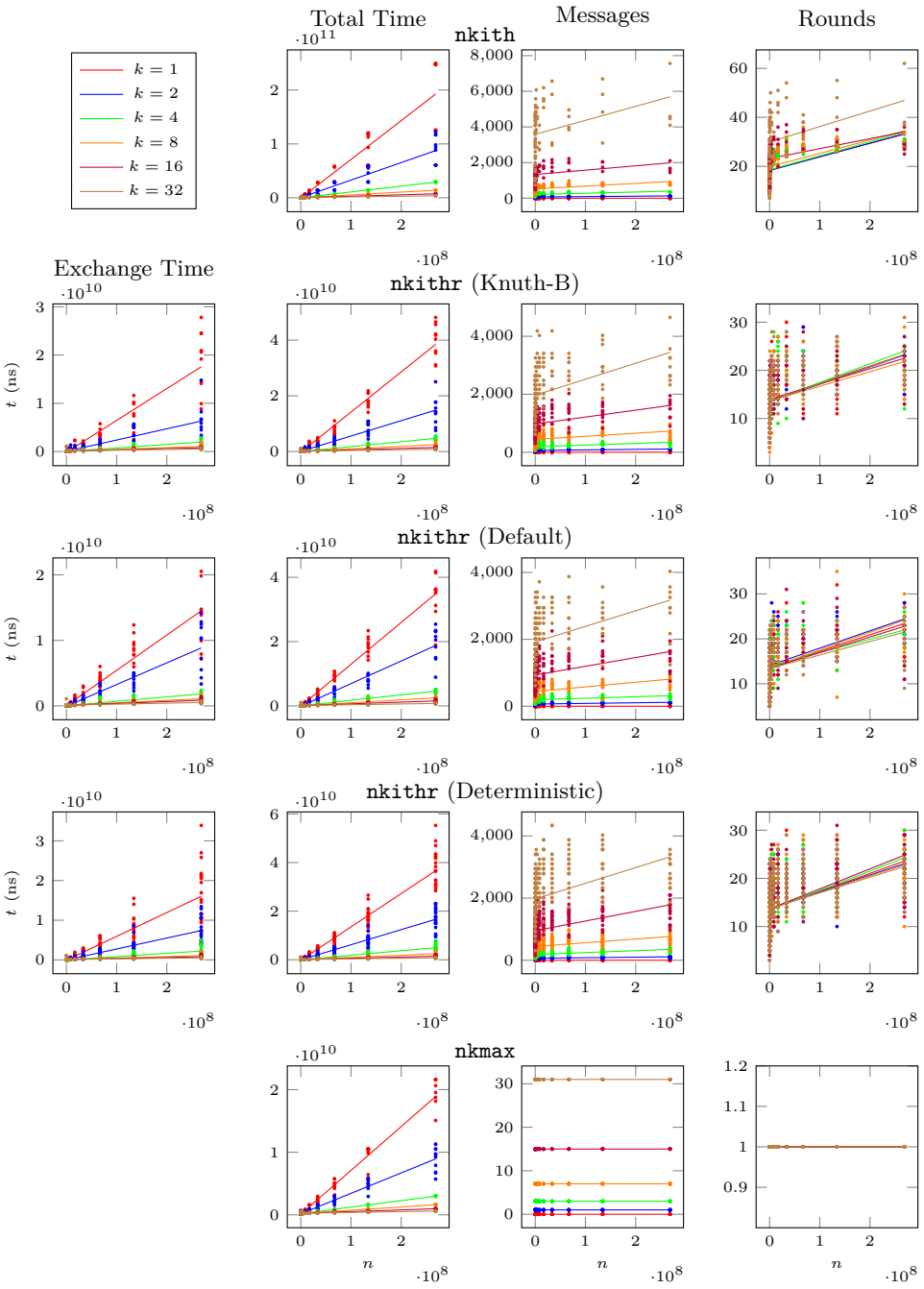

Figure 1: Empirical Complexity of Finding the Maximum

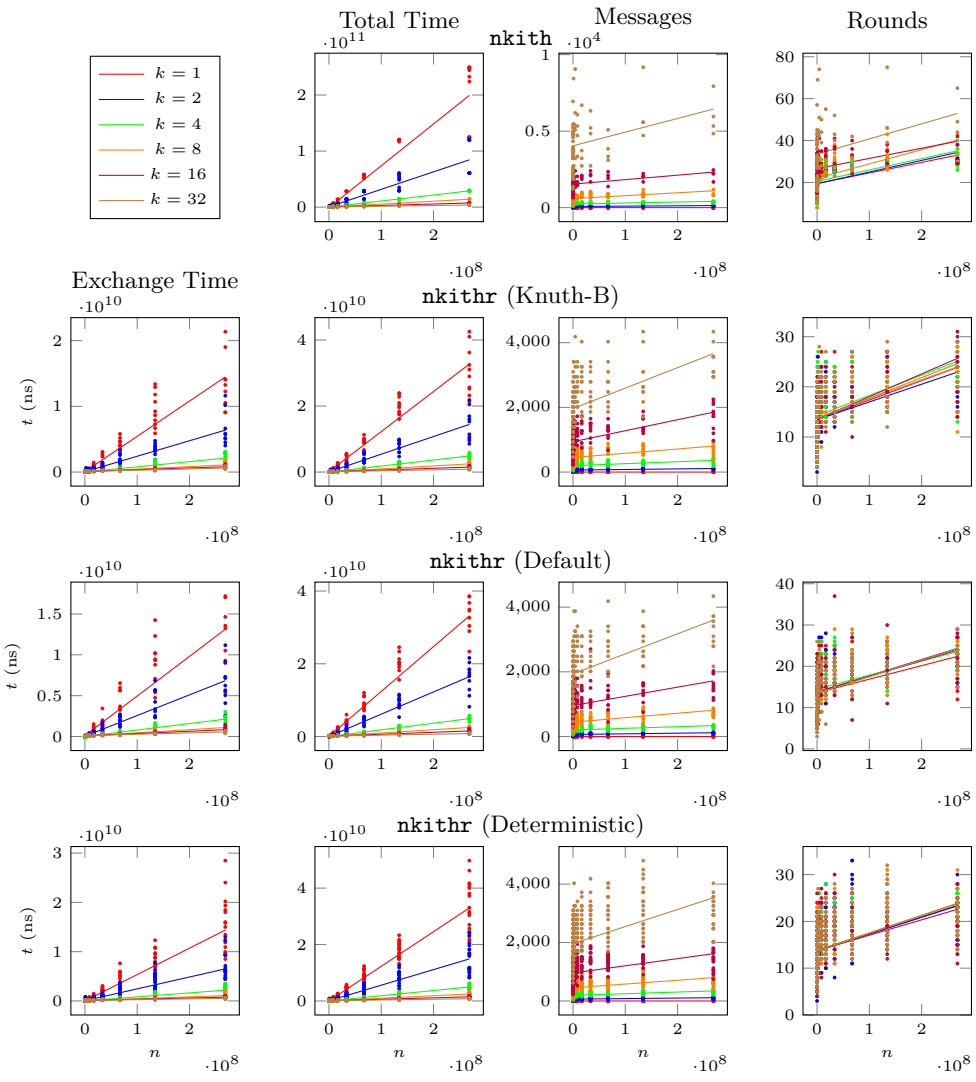

Figure 2: Empirical Complexity of Finding the Minimum

Though not strictly part of the assignment, the time and message complexity for finding the minimum was tested as a sanity check on the implementation consistency. The graphs (see Figure 2) are similar to those charted for finding the max with the exception of a bespoke `nkmin` program that was not created.

In future implementations of `nkithr`, it might be worth the added code complexity to add case logic for the minimum and maximum cases that employ the more specialized algorithms since finding the extrema can be a frequent use case of the more general ith selection in practice.

3.4 Finding the Median

The complexities observed in finding the median (see Figure 3) are again similar to those seen for the maximum and minimum, but with some striking differences. The time complexity for `nkithr` to find the median was longer by about half in terms of wall-clock time and rounds. The pivot selection was notably worse in the deterministic `nkithr` case, at least in terms of variance, if not mean, over the rounds required. In retrospect, a better deterministic selection would have been made by simply choosing the candidate with the maximum pre-normalization weight instead of the first non-zero weight.

Some of this behavior may be explained by the fact that random selection leading to the extrema heavily favors reduction from one side of the equation across all nodes, and so larger fractions of the population are taken out of consideration in each iteration. With the median, the side on which the reduction occurs may oscillate slowly until convergence.

4 Conclusions

The inherent power of algorithm randomization is aptly demonstrated in the trial results. Choosing a deterministic algorithm is not difficult, but in almost all cases, the random versions outperformed the implementations which used deterministic methods. It would appear that even the potentially

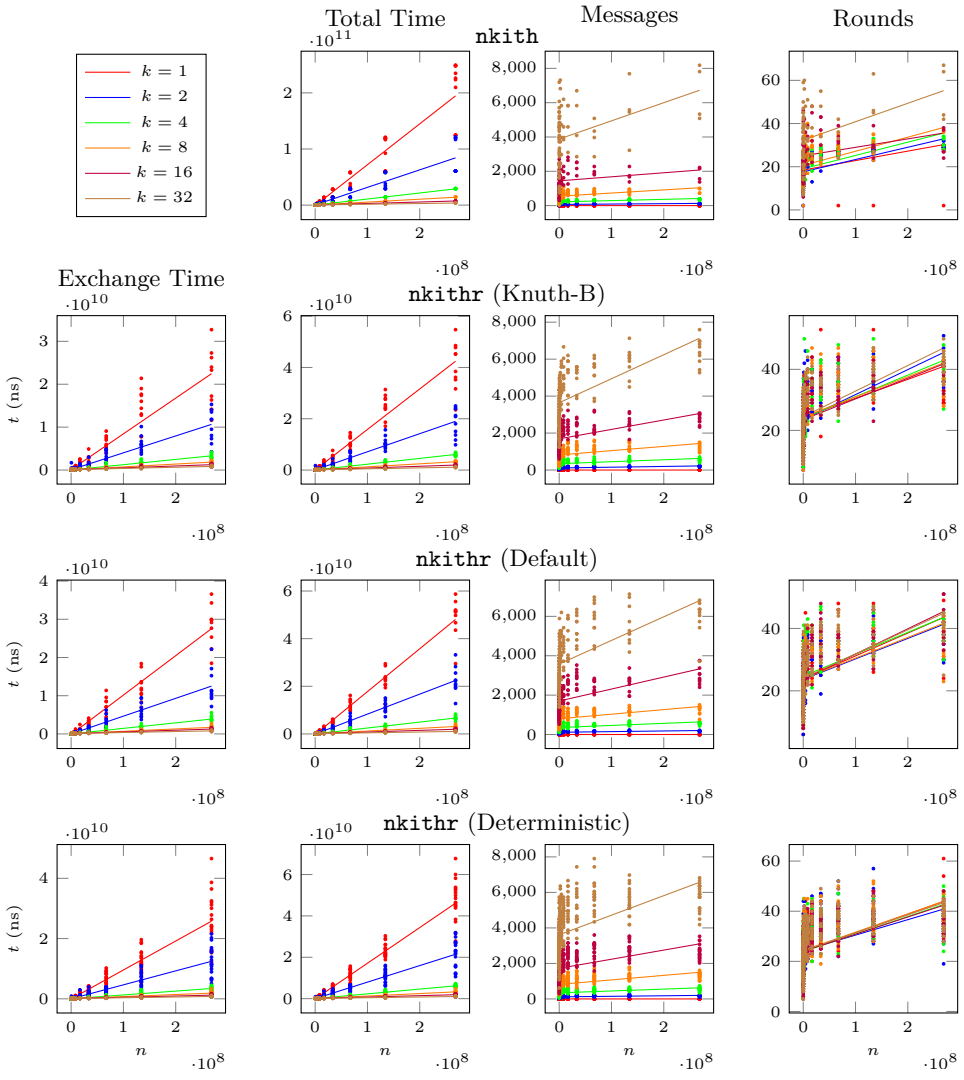

Figure 3: Empirical Complexity of Finding the Median

best (as yet unimplemented) deterministic pivoting method simply copies the behavior of the randomized pivot selection.

The results also serve as a clear evidence that no amount of "cleverness" in implementation can overcome the tyranny of scale. While familiarity with implementation details improved the performance of the randomized algorithm over more naive possible implementations of the same algorithm, it needed the solid algorithmic found upon which it could build.

The congress of ideas between the deterministic and randomized subfields of distributed algorithms is best kept free and fluid. Whether it is a deterministic algorithm informed by a randomized experiment or a recognition of a common special case that can be more rapidly decomposed in a deterministic fashion than the unbiased random approach would allow, the field as a whole will be richer for it.

References

[1] Anonymous. *Linux Programmer's Manual. qsort*. Linux Man Pages Project. Sept. 15, 2017.

[2] Anonymous. *Pseudo-random number generation*. cppreference.com. Mar. 4, 2019. URL: `https://en.cppreference.com/w/cpp/numeric/random` (visited on 03/20/2019).

[3] Anonymous. *std::discrete_distribution*. cppreference.com. June 15, 2018. URL: `https://en.cppreference.com/w/cpp/numeric/random/discrete_distribution` (visited on 03/21/2019).

[4] Anonymous. *std::mersenne_twister_engine*. cppreference.com. Feb. 25, 2019. URL: `https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine` (visited on 03/08/2019).

[5] Anonymous. *std::shuffle_order_engine*. cppreference.com. June 15, 2015. URL: `https://en.cppreference.com/w/cpp/numeric/random/shuffle_order_engine` (visited on 03/20/2019).

[6] Anonymous. *std::uniform_int_distribution*. cppreference.com. June 15, 2015. URL: `https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution` (visited on 03/20/2019).

[7] Richard Graham et al., eds. *MPI: A Message-Passing Interface Standard*. Version 2.1. Message Passing Interface Forum. June 3, 2008. URL: https://www.mpi-forum.org/docs/mpi-2.1/mpi21-report.pdf (visited on 03/08/2019).

[8] William Gropp. *Lecture 32: Introduction to MPI I/O*. URL: http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture32.pdf (visited on 04/09/2019).

[9] Gopal Pandurangan. *Algorithms*. Dec. 3, 2018. URL: https://sites.google.com/site/gopalpandurangan/algbook.pdf (visited on 03/25/2019).