9-1-1999

# An Improved Randomized Selection Algorithm With an Experimental Study

D.A. Bader

# DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

# SCHOOL OF ENGINEERING UNIVERSITY OF NEW MEXICO

**An Improved Randomized Selection Algorithm With an Experimental Study (*Extended Abstract*)**

David A. Bader [1]
dbader@eece.unm.edu
Department of Electrical and Computer Engineering
University of New Mexico, Albuquerque, NM 87131

UNM Technical Report: EECE-TR-99-005

Report Date: September 1999

[1]This work was supported in part by NSF CISE Postdoctoral Research Associate in Experimental Computer Science No. 96-25668 and U.S. Department of Energy Sandia-University New Assistant Professorship Program (SUNAPP) Award # AX-3006.

# Abstract

A common statistical problem is that of finding the median element in a set of data. This paper presents an efficient randomized high-level parallel algorithms for finding the median given a set of elements distributed across a parallel machine. In fact, our algorithm solves the general selection problem that requires the determination of the element of rank $k$, for an arbitrarily given integer $k$.

Our general framework is an SPMD distributed memory programming model that is enhanced by a set of communication primitives. We use efficient techniques for distributing and coalescing data as well as efficient combinations of task and data parallelism. The algorithms have been coded in the message passing standard MPI, and our experimental results from the IBM SP-2 illustrate the scalability and efficiency of our algorithm and improve upon all the related experimental results known to the authors.



## Keywords

Selection Algorithm, Randomized Algorithms, Parallel Algorithms, Experimental Parallel Algorithmics

# 1 Introduction

Given a set of data $X$ with $|X| = n$, the selection problem requires the determination of the element with rank $k$ (that is, the $k^{\text{th}}$ smallest element), for an arbitrarily given integer $k$. Median finding is a special case of selection with $k = \frac{n}{2}$. In previous work, we have designed deterministic and efficient parallel algorithms for the selection problem on current parallel machines [5, 6, 3]. In this paper, we discuss a new UltraFast Randomized algorithm for the selection problem which, unlike previous research (for example, [11, 12, 15, 8, 14, 13, 16, 19, 18, 17]), is not dependent on network topology or limited to the PRAM model which does not assign a realistic cost for communication. In addition, our randomized algorithm improves upon previous implementations on current parallel platforms, for example, Al-Furaih et al. [2] implement both our deterministic algorithm and the randomized algorithms due to Rajasekaran et al. [15, 13] on the TMC CM-5.

The main contributions of this paper are

1. New techniques for speeding the performance of certain randomized algorithms, such as selection, which are efficient with likely probability.
2. A new, practical randomized selection algorithm (UltraFast) with significantly improved convergence.

The remainder of this paper is organized as follows. Both our new and Rajasekaran's randomized selection algorithms are detailed in Section 2, followed by analysis and experimental results in Section 3. Additional information on Chernoff Bounds is located in Appendix A. More extensive statistics from our experiments are reported in [4].

# 2 Parallel Selection

The selection algorithm for rank $k$ assumes that input data $X$ of size $n$ is initially distributed evenly across the $p$ processors, such that each processor holds $\frac{n}{p}$ elements. Note that median finding is a special case of the selection problem where $k$ is equal to $\lceil \frac{n}{2} \rceil$. The output, namely the element from $X$ with rank $k$, is returned on each processor.

The randomized selection algorithm locates the element of rank $k$ by pruning the set of candidate elements using the following iterative procedure. Two *splitter* elements $(k_1, k_2)$ are chosen which partition the input into three groups, $G_0, G_1$, and $G_2$, such that each element in $G_0$ is less than $k_1$, each element in $G_1$ lies in $[k_1, k_2]$, and each in $G_2$ is greater than $k_2$. The desire is to have the middle group $G_1$ much smaller than the outer two groups ($|G_1| \ll |G_0|, |G_2|$) with the *condition* that the selection index lies within this middle group. The process is repeated iteratively on the group holding the selection index until the size of the group is "small enough," whereby the remaining elements are gathered onto a single processor and the problem is solved sequentially.

The key to this approach is choosing splitters $k_1$ and $k_2$ which minimize the size of the middle group while maximizing the probability of the *condition* that the selection index lies within this group. Splitters are chosen from a random sample of the input, by finding a pair of elements of certain rank in the sample (see Section 3). The algorithm of Rajasekaran and Reif [15, 13] takes a conservative approach which guarantees the condition with high probability. We have discovered a more aggressive technique for pruning the input space by choosing splitters closer together in the sample while holding the condition with likely probability. In practice, the condition almost always holds, and in the event of a failure, new splitters are chosen from the sample with a greater spread of ranks until the condition is satisfied.

In addition, we improve upon previous algorithms in the following ways.

1. **Stopping Criterion**. For utmost performance, current parallel machines typically require a coarse granularity, the measure of problem size per node, because communication is typically an order of magnitude

slower than local computation. In addition, machine configurations tend to be small to moderate in terms of number of processors ($p$). Thus, a stopping criterion of problem size $< p^2$ is much too fine grained for current machines, and we suggest, for instance, a stopping size of $\max(p^2, 4096)$. When $p$ is small and $n = \mathrm{O}(p^2)$, a second practical reason for increasing the stopping size is that the sample is very limited and might not yield splitters which further partition the input.

2. **Aggressive Convergence**. As outlined in Section 3, our algorithm converges roughly twice as fast as the best known previous algorithm.

3. **Algorithmic Reduction**. At each iteration, we use "selection" to choose the splitters instead of sorting, a computationally harder problem.

4. **Communication Aggregation**. Similar collective communication steps are merged into a single operation. For instance, instead of calling the **Combine** primitive twice to find the size of groups $G_0$ and $G_1$ ($|G_2|$ can be calculated from this information and the problem size), we aggregate these operations into a single step.

Next we outline our new UltraFast Randomized Selection Algorithm, followed by the Fast Randomized algorithm.

## 2.1 UltraFast Randomized Selection Algorithm

An SPMD algorithm on each processor $P_i$:

**Algorithm 1** *UltraFast Randomized Selection Algorithm*

**Input:**

$\{ n \}$ - Total number of elements
$\{ p \}$ - Total number of processors, labeled from 0 to $p-1$
$\{ L_i \}$ - List of elements on processor $P_i$, where $|L_i| = \frac{n}{p}$
$\{ C \}$ - A constant $\approx \max(p^2, 4096)$
$\{ \varepsilon \}$ - $\log_n$ of the sample size (e.g. 0.6)
$\{ \Delta^* \}$ - selection coefficient (e.g. 1.0)
$\{ \kappa \}$ - selection coefficient multiplier (e.g. 2.25)
*rank* - desired rank among the elements

**begin**
**Step 0.** Set $n_i = \frac{n}{p}$.
While $(n > C)$
**Step 1.** Collect a sample $S_i$ from $L_i$ by picking $n_i \frac{n^\varepsilon}{n}$ elements at random on $P_i$.
**Step 2.** $S = \textbf{Gather}(S_i, p)$.
Set $z =$ TRUE and $\Delta = \Delta^*$.
While $(z \equiv$ TRUE)
On $P_0$
**Step 3.** Select $k_1$, $k_2$ from $S$ with ranks $\left\lfloor \frac{i|S|}{n} - \Delta\sqrt{|S|} \right\rfloor$ and $\left\lfloor \frac{i|S|}{n} + \Delta\sqrt{|S|} \right\rfloor$.
**Step 4.** Broadcast $k_1$ and $k_2$.
**Step 5.** Partition $L_i$ into $< k_1$ and $[k_1, k_2]$, and $> k_2$, to give counts *less*, *middle*, (and *high*). Only save the elements which lie in the middle partition.
**Step 6.** $c_{less} = \textbf{Combine}(less, +)$; $\quad c_{mid} = \textbf{Combine}(middle, +)$;
**Step 7.** If $(rank \in (c_{less}\ ,\ c_{less} + c_{mid}])$
$n = c_{mid}$ ; $n_i = middle$ ; $rank = rank - c_{less}$ ; $z =$ FALSE
Else
On $P_0$: $\Delta = \kappa \cdot \Delta$
Endif
Endwhile
Endwhile
**Step 8.** $L = \textbf{Gather}(L_i)$.
**Step 9.** On $P_0$
Perform sequential selection to find element $q$ of *rank* in $L$;
$result = \textbf{Broadcast}(q)$.

**end**

## 2.2 Fast Randomized Selection Algorithm

This algorithm is due to Rajasekaran and Reif [15, 13], and implemented by Al-furaih et al. [2].

An SPMD algorithm on each processor $P_i$:

**Algorithm 2** *Fast Randomized Selection Algorithm*

**Input:**

$\{ n \}$ - Total number of elements
$\{ p \}$ - Total number of processors, labeled from 0 to $p-1$
$\{ L_i \}$ - List of elements on processor $P_i$, where $|L_i| = \frac{n}{p}$
$\{ \varepsilon \}$ - $\log_n$ of the sample size (e.g. 0.6)
*rank* - desired rank among the elements
$l = 0$ ; $r = \frac{n}{p} - 1$

**begin**
*while* $(n > p^2)$
**Step 0.** Set $n_i = r - l + 1$
**Step 1.** Collect a sample $S_i$ from $L_i[l,r]$ by picking $n_i \frac{n^\varepsilon}{n}$ elements at random on $P_i$ between $l$ and $r$.
**Step 2.** $S =$ **ParallelSort**$(S_i, p)$.
On $P_0$
**Step 3.** Pick $k_1$, $k_2$ from $S$ with ranks $\left\lceil \frac{i|S|}{n} - \sqrt{|S| \log_e n} \right\rceil$ and $\left\lceil \frac{i|S|}{n} + \sqrt{|S| \log_e n} \right\rceil$.
**Step 4.** Broadcast $k_1$ and $k_2$. The *rank* to be found will be in $[k_1, k_2]$ with high probability.
**Step 5.** Partition $L_i$ between $l$ and $r$ into $< k_1$, $[k_1, k_2]$, and $> k_2$ to give counts *less*, *middle*, and *high*, and splitters $s_0$ and $s_1$.
**Step 6.** $c_{mid} =$ **Combine**$(middle, +)$.
**Step 7.** $c_{less} =$ **Combine**$(less, +)$.
**Step 8.** If $(rank \in (c_{less}, c_{mid}])$
$n = c_{mid}$ ; $l = s_1$ ; $r = s_2$ ; $rank = rank - c_{less}$
Else
If $(rank \leq c_{less})$
$r = s_1$ ; $n = c_{less}$
Else
$n = n - (c_{less} + c_{mid})$ ; $l = s_2$ ; $rank = rank - (c_{less} + c_{mid})$
Endif
Endif
Endwhile
**Step 9.** $L =$ **Gather**$(L_i[l,r])$.
**Step 10.** On $P_0$
Perform sequential selection to find element $q$ of *rank* in $L$,
*result* = **Broadcast**$(q)$.

**end**

# 3 Analysis

The following sampling lemma from Rajasekaran [15] will be used in the analysis.

Let $S = \{v_1, v_2, \ldots, v_s\}$ be a random sample from a set $X$ of cardinality $n$. Also, let $v'_1, v'_2, \ldots, v'_s$ be the sorted order of this sample. If $r_i$ is the rank of $k'_i$ in $X$, the following lemma provides a high probability confidence interval for $r_i$.

**Lemma 1** *For every* $\alpha$, $Pr\left(|r_i - i\frac{n}{s}| > \sqrt{3\alpha}\frac{n}{\sqrt{s}}\sqrt{\log_e n}\right) < n^{-\alpha}$.

Thus, if $k_1$ and $k_2$ are chosen as the splitters from sample set $S$ by selecting the elements with rank $\frac{is}{n} - d\sqrt{s\log_e n}$ and $\frac{is}{n} + d\sqrt{s\log_e n}$, respectively, and $d = \sqrt{4\alpha}$, then the element of desired rank will lie in the middle partition $(c_{less}\ ,\ c_{less} + c_{mid}]$ with high probability $(1 - n^{-\alpha})$.

A tradeoff occurs between the size of the middle partition $(r)$ and the confidence that the desired element lies within this partition. Note that in the Fast Randomized algorithm, with $d = 1$, this probability is $1 - n^{-\frac{1}{4}}$, and $r \leq 8\frac{n}{\sqrt{s}}\sqrt{\log_e n}$. Since $s \approx n^{\varepsilon}$, this can be approximated by $r \leq 8n^{1-\frac{\varepsilon}{2}}\sqrt{\log_e n}$.

Suppose now the bound is relaxed with probability no less than $1 - n^{-\alpha} = \rho$. Then $\alpha = -\frac{\log(1-\rho)}{\log n}$, and the splitters $k_1, k_2$ can be chosen with ranks $\frac{is}{n} - \Delta\sqrt{s}$ and $\frac{is}{n} + \Delta\sqrt{s}$, for $\Delta = 2\sqrt{-\log_e(1-\rho)}$ (see Table I). Then the size of the middle partition can be bounded similarly by $r \leq 16\frac{n}{\sqrt{s}}\sqrt{-\log_e(1-\rho)}$. This can be approximated by $r \leq 16n^{1-\frac{\varepsilon}{2}}\sqrt{-\log_e(1-\rho)}$. Thus, the middle partition size of the UltraFast algorithm is typically smaller than that of the Fast algorithm, whenever the condition $n > (1-\rho)^{-4}$.

| $\Delta$ | Lower bound of capture ($\rho$, in %) |
|---|---|
| 6.07 | 99.99 |
| 5.26 | 99.9 |
| 4.29 | 99.0 |
| 3.03 | 90.0 |
| 2.54 | 80.0 |
| 2.19 | 70.0 |
| 1.91 | 60.0 |
| 1.50 | 43.0 |
| 1.00 | 22.1 |
| 0.50 | 6.05 |

Table I: Lower bound of the capture probability ($\rho$) that the selection index is in the middle partition, where $\rho = 1 - e^{-\frac{\Delta^2}{4}}$.

A large value for $\varepsilon$ increases running time since the sample (of size $n^{\varepsilon}$) must be either sorted (in Fast) or have elements selected from it (in UltraFast). A small value of $\varepsilon$ increases the probability that both of the splitters lie on one side of the desired element, thus causing an unsuccessful iteration. In practice, 0.6 is an appropriate value for $\varepsilon$ [2].

## 3.1 Complexity

We use a simple model of parallel computation to analyze the performance of these two selection algorithms. Current hardware platforms can be viewed as a collection of powerful processors connected by a communication network that can be modeled as a complete graph on which communication is subject to the restrictions imposed by the latency and the bandwidth properties of the network. We view a parallel algorithm as a sequence of local

computations interleaved with communication steps, and we allow computation and communication to overlap. We account for communication costs as follows.

The transfer of a block consisting of $m$ contiguous words, assuming no congestion, takes $O(\tau+\sigma m)$ time, where $\tau$ is an bound on the latency of the network and $\sigma$ is the time per word at which a processor can inject or receive data from the network.

One iteration of the Fast randomized selection algorithm takes $O\left(n^{(j)}+(\tau+\sigma)\log p\right)$ time, where $n^{(j)}$ is the maximum number of elements held by any processor during iteration $j$. From the bound on the size of the middle partition, we find a recurrence on the problem size during iteration $i$,

$$\begin{array}{rcl} n_0 & = & n \\ n_{i+1} & \leq & 8n_i^{0.7}\sqrt{\log_e n_i}\ , \end{array} \tag{1}$$

which shows a geometric decrease in problem size per iteration, and thus, $O(\log\log n)$ iterations are required. Since $n^{(j)}=O\left(\frac{n}{p}\right)$, Fast selection requires

$$O\left(\tfrac{n}{p}\log\log n+(\tau+\sigma)\log p\log\log n\right) \tag{2}$$

time. (Assuming random data distribution, the running time reduces to $O\left(\frac{n}{p}+(\tau+\sigma)\log p\log\log n\right)$.) [2]

Each iteration of the UltraFast algorithm is similar to Fast, except sorting is replaced by sequential selection, which takes linear time [9]. Also, the problem size during iteration $i$ is bounded with the following recurrence,

$$\begin{array}{rcl} n_0 & = & n \\ n_{i+1} & \leq & 16n_i^{0.7}\sqrt{-\log_e(1-\rho)}\ , \end{array} \tag{3}$$

and similar to the Fast algorithm, UltraFast as well requires $O(\log\log n)$ iterations. Thus, UltraFast randomized selection has a similar complexity, with a worst case running time given in Eq. (2). As we will show later by empirical results in Table III, though, the constant associated with the number of iterations is significantly smaller for the UltraFast algorithm.

## 3.2 Experimental Data Sets

Empirical results for the selection algorithm use the following three inputs. Given a problem of size $n$ and a $p$ processors,

- **[I]** - Identical elements $\{0,1,\ldots,\frac{n}{p}-1\}$ on each processor,
- **[S]** - Sorted elements $\{0,1,\ldots,n-1\}$ distributed in $p$ blocks across the processors, and
- **[R]** - Random, uniformly distributed, elements, with $\frac{n}{p}$ elements per processor.
- **[N]** - This input is taken from the NAS Parallel Benchmark for Integer Sorting [7]. Keys are integers in the range $[0,2^{19})$, and each key is the average of four consecutive uniformly distributed pseudo-random numbers generated by the following recurrence:

  $$x_{k+1}=ax_k\ (\mathrm{mod}\ 2^{46})$$

  where $a=5^{13}$ and the seed $x_0=314159265$. Thus, the distribution of the key values is a Gaussian approximation. On a $p$-processor machine, the first $\frac{n}{p}$ generated keys are assigned to $P_0$, the next $\frac{n}{p}$ to $P_1$, and so forth, until each processor has $\frac{n}{p}$ keys.

### 3.3 Empirical Results

Results for a previous implementation of the Fast randomized selection algorithm on the TMC CM-5 parallel machine appear in [2]. However, this machine is no longer available and does not support the current message passing standard **MPI**. Therefore, we have recoded this algorithm into MPI.

| $n$ | $p$ | [R]andom Input | | [S]orted Input | |
|---|---|---|---|---|---|
| | | CM-5 | SP-2 | CM-5 | SP-2 |
| 512K | 4 | 174 | 68.0 | 194 | 104 |
| | 8 | 105 | 62.7 | 119 | 79.6 |
| | 16 | 69.5 | 39.5 | 86.7 | 61.9 |
| 2M | 4 | 591 | 153 | 601 | 229 |
| | 8 | 318 | 108 | 359 | 182 |
| | 16 | 193 | 74.4 | 237 | 136 |

Table II: Comparison of the execution time of the Fast Randomized Selection Algorithm on TMC CM-5 [1, 2] and IBM SP-2-TN (in milliseconds).

Table II compares the execution time of the Fast Randomized algorithm on both the CM-5 [1, 2] and the SP-2. Since selection is computation-bound, we would expect the performance to be closely related to the node performance of these two machines. The SP-2-TN 66MHz POWER2 processor is roughly twice as fast as the CM-5 33 MHz RISC processor. As expected, this factor of two performance improvement is apparent in the execution time comparison for equivalent machine and problem sizes. In actuality, the SP-2 is more than twice as powerful, since communication latency and bandwidth are improved roughly by a factor of three.

We conducted experiments with our UltraFast and the known Fast randomized selection algorithms on an IBM SP-2 with four, eight, and sixteen processors, by finding the median of each input in the previous section for various problem sizes (ranging between $16K$ to $16M$ elements)[1]. A comparison of the empirical execution times for machine configurations of $p = 4, 8,$ and 16 processors are graphed using log-log plots in Figures 1, 2, and 3, respectively. In all cases, the UltraFast algorithm is substantially faster than the Fast randomized selection algorithm, typically by a factor of two. Running time can be characterized mainly by $\frac{n}{p}\log p$ and is only slightly dependent on input distribution.

For $p = 8$, Table III provides a summary of the number of times each algorithm iterates. While the Fast algorithm typically iterates in the neighborhood of about 25 times, there are cases when it iterates hundreds or even thousands of times. However, the UltraFast algorithm never iterates more then three times. This is due to two reasons. First, UltraFast converges roughly twice as fast as the Fast algorithm. Second, the algorithm stops iterating by using a more realistic stopping criterion matched to the coarse granularity of current parallel machines. In addition, when $p$ is small and $n = \mathrm{O}(p^2)$, the Fast algorithm's sample is very limited and sometimes does not yield splitters which further partition the input. Thus, in this situation, the Fast algorithm might iterative from tens to thousands of times before pruning any additional elements from the solution space.

Detailed results from the UltraFast and Fast algorithms (for the **[I]**, **[S]**, and **[R]** inputs) for $n =$ 512K, 1M, 2M, 4M, and 8M, and further statistics from the **[N]** input, are available in [4]

## 4 Future Directions

We are investigating other combinatorial algorithms that may have significant practical improvement by relaxing the probabilistic bounds, as demonstrated by our UltraFast randomized selection.

In addition, our UltraFast parallel, randomized selection algorithm, here designed and analyzed for a message-passing platform, would also be suitable for shared-memory multiprocessors (SMP's). Each communication step

[1] Throughout this paper, $K$ and $M$ refer to $2^{10}$ and $2^{20}$, respectively.

| n | Input | Fast Algorithm | UltraFast Algorithm |
|---|---|---|---|
| 512K | I | 19 | 2 |
| | S | 17 | 2 |
| | R | 29 | 2 |
| | N | 19 | 2 |
| 1M | I | 24 | 2 |
| | S | 17 | 2 |
| | R | 22 | 2 |
| | N | 32 | 2 |
| 2M | I | 26 | 2 |
| | S | 22 | 3 |
| | R | 21 | 2 |
| | N | 38 | 3 |
| 4M | I | 37 | 3 |
| | S | 23 | 3 |
| | R | 21 | 3 |
| | N | 4095 | 3 |
| 8M | I | 28 | 3 |
| | S | 24 | 3 |
| | R | 21 | 3 |
| | N | 866 | 3 |

Table III: Total number of iterations of the Fast and UltraFast Randomized Selection Algorithms. For this table, the number of processors used $p = 8$.

can be eliminated, simplified, or replaced with a shared-memory primitive. For instance, the SMP algorithm

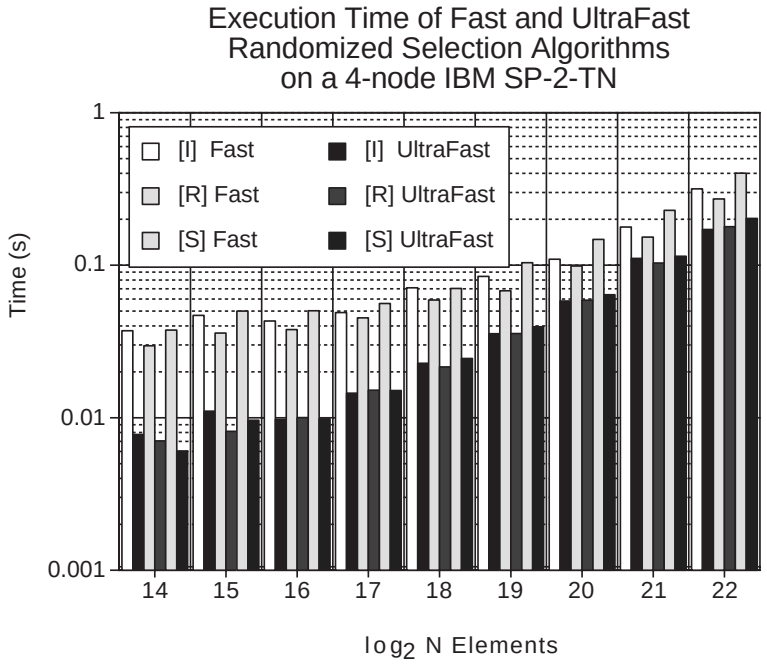


Figure 1: Empirical Performance of Fast versus UltraFast Randomized Selection Algorithms with $p = 4$ nodes of an IBM SP-2-TN.

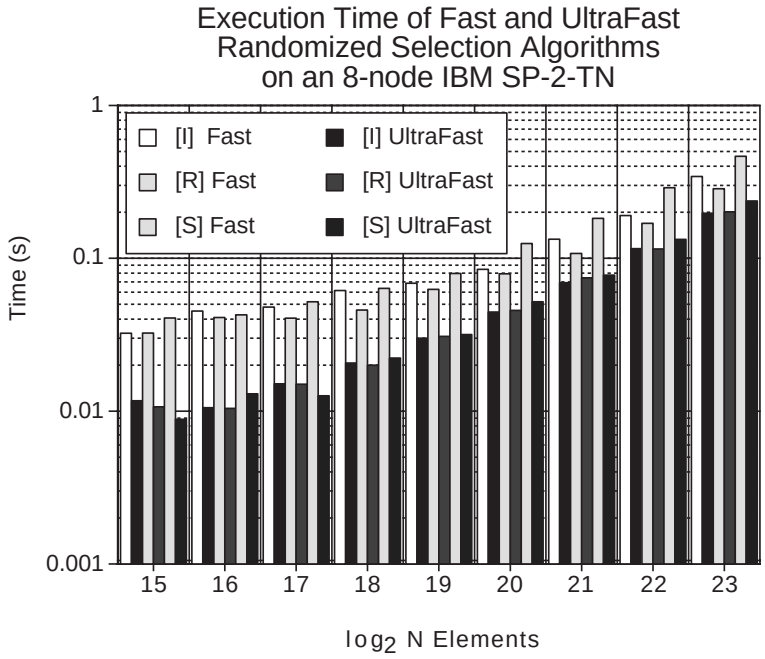


Figure 2: Empirical Performance of Fast versus UltraFast Randomized Selection Algorithms with $p = 8$ nodes of an IBM SP-2-TN.

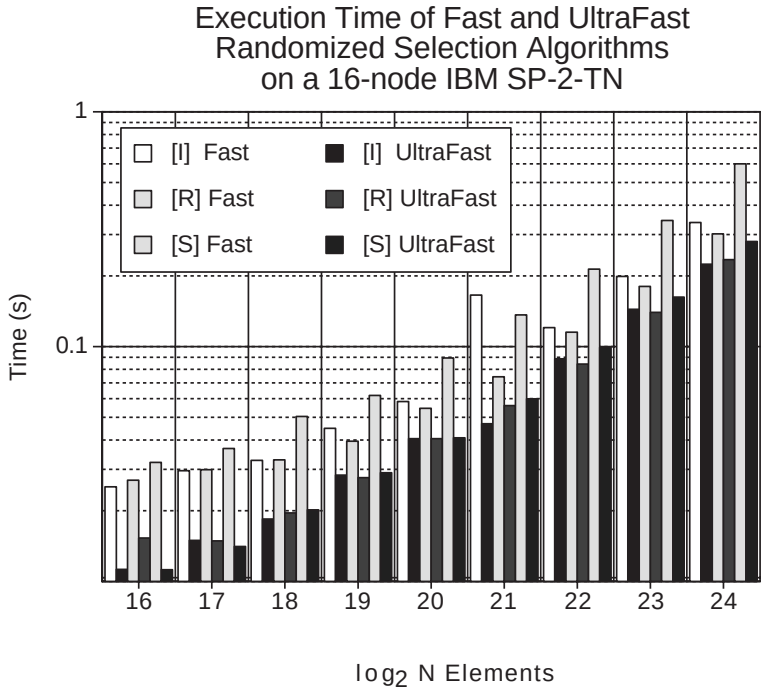


Figure 3: Empirical Performance of Fast versus UltraFast Randomized Selection Algorithms with $p = 16$ nodes of an IBM SP-2-TN.

would be as follows. Each processor collects its portion of the sample from the corresponding block of the input and writes the sample to a shared-memory array. Thus, the second step, a **Gather** communication, is eliminated. After a single processor determines the splitters $k_1$ and $k_2$ from the sample, the **Broadcast** communication in step four simplifies into a memory read by each processor. The **Combine** in step six may be replaced by the corresponding shared-memory primitive. The **Gather** in step eight can be replaced with a shared-memory gather. We are currently investigating the performance of this SMP approach.

# A Chernoff Bounds

The following inequalities are useful for bounding the tail ends of a binomial distribution with parameters $(n, p)$. If $X$ is a binomial with parameters $(n, p)$, then the tail distributions, known as Chernoff bounds [10], are as follows.

$$\Pr(X \leq (1-\varepsilon)np) \leq e^{-\frac{\varepsilon^2 np}{2}} \tag{4}$$

$$\Pr(X \geq (1+\varepsilon)np) \leq e^{-\frac{\varepsilon^2 np}{3}} \tag{5}$$

for all $0 < \varepsilon < 1$.

# References


[1] I. S. Al-Furaih. Timings of Selection Algorithm. Personal communication, April 1996.

[2] I. Al-furiah, S. Aluru, S. Goil, and S. Ranka. Practical Algorithms for Selection on Coarse-Grained Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 8(8):813–824, 1997.

[3] D. A. Bader. *On the Design and Analysis of Practical Parallel Algorithms for Combinatorial Problems with Applications to Image Processing*. PhD thesis, University of Maryland, College Park, Department of Electrical Engineering, April 1996.

[4] D. A. Bader. An Improved Randomized Selection Algorithm With an Experimental Study. Technical report, Electrical and Computer Engineering Department, The University of New Mexico, Albuquerque, NM, September 1999.

[5] D. A. Bader and J. JáJá. Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection. Technical Report CS-TR-3494 and UMIACS-TR-95-74, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, July 1995.

[6] D. A. Bader and J. JáJá. Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 292–301, Honolulu, HI, April 1996.

[7] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical Report RNR-94-007, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Moffett Field, CA, March 1994.

[8] P. Berthomé, A. Ferreira, B. M. Maggs, S. Perennes, and C. G. Plaxton. Sorting-Based Selection Algorithms for Hypercubic Networks. In *Proceedings of the 7th International Parallel Processing Symposium*, pages 89–95, Newport Beach, CA, April 1993. IEEE Computer Society Press.

[9] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time Bounds for Selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[10] H. Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations. *Annals of Math. Stat.*, 23:493–509, 1952.

[11] E. Hao, P. D. MacKenzie, and Q. F. Stout. Selection on the Reconfigurable Mesh. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation*, pages 38–45, McLean, VA, October 1992. IEEE Computer Society Press.

[12] D. Krizanc and L. Narayanan. Optimal Algorithms for Selection on a Mesh-Connected Processor Array. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 70–76, Arlington, TX, December 1992.

[13] S. Rajasekaran. Randomized Selection on the Hypercube. *Journal of Parallel and Distributed Computing*, 37(2):187–193, 1996.

[14] S. Rajasekaran, W. Chen, and S. Yooseph. Unifying Themes for Network Selection. In *Proceedings of the 5th International Symposium on Algorithms and Computation (ISAAC'94)*, pages 92–100, Beijing, China, August 1994. Springer-Verlag.

[15] S. Rajasekaran and J. H. Reif. Derivation of Randomized Sorting and Selection Algorithms. In R. Paige, J. Reif, and R. Wachter, editors, *Parallel Algorithms Derivation and Program Transformation*, chapter 6, pages 187–205. Kluwer Academic Publishers, Boston, MA, 1993.

[16] S. Rajasekaran and S. Sahni. Sorting, Selection and Routing on the Array with Reconfigurable Optical Buses. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1123–1132, 1997.

[17] S. Rajasekaran and S. Sahni. Randomized Routing, Selection, and Sorting on the OTIS-Mesh. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):833–840, 1998.

[18] S. Rajasekaran and D. S.L. Wei. Selection, Routing, and Sorting on the Star Graph. *Journal of Parallel and Distributed Computing*, 41:225–233, 1997.

[19] R. Sarnath and X. He. On Parallel Selection and Searching in Partial Orders: Sorted Matrices. *Journal of Parallel and Distributed Computing*, 40:242–247, 1997.