

# Practical Algorithms for Selection on Coarse-Grained Parallel Computers

Ibraheem Al-furaih, Srinivas Aluru, Sanjay Goil, Sanjay Ranka <sup>\*†</sup>  
School of Computer and Information Science and Northeast Parallel Architectures Center  
Syracuse University, Syracuse, NY 13244

## Abstract

In this paper, we consider the problem of selection on coarse-grained distributed memory parallel computers. We discuss several deterministic and randomized algorithms for parallel selection. Experimental results on the CM-5 demonstrate that randomized algorithms are superior to their deterministic counterparts.

## 1 Introduction

Given a set of  $N$  elements, a total order defined on the elements, and a number  $k$ , the selection problem is to find the  $k^{\text{th}}$  smallest element in the given set of elements. The problem has several applications in computer science and statistics. A special case of the problem, often found useful, is to find the median of the given data. The median of  $N$  elements is defined to be the element with rank  $\lfloor \frac{N}{2} \rfloor$ .

Parallel selection algorithms are useful in such practical applications as dynamic distribution of multidimensional data sets, parallel graph partitioning and parallel construction of multidimensional binary search trees. Many parallel algorithms for selection have been designed for the PRAM model and for various network models including trees, meshes, hypercubes and reconfigurable architectures [4]. More recently, Bader et.al. [3] implement a parallel deterministic selection algorithm on several distributed memory machines. In this paper, we consider and evaluate parallel selection algorithms for coarse-grained distributed memory parallel computers.

<sup>\*</sup>Ibraheem Al-furaih is supported by scholarship from King AbdulAziz City for Science and Technology (KACST), Riyadh, Saudi Arabia. Sanjay Goil is supported in part by NASA under subcontract #1057L0013-94 issued by the LANL. Sanjay Ranka is supported in part by NSF under ASC-9213821 and AFMC and ARPA under contract #F19628-94-C-0057. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

<sup>†</sup>We are grateful to Northeast Parallel Architectures Center and Minnesota Supercomputing Center for allowing us to use their CM-5. We would like to thank David Bader for providing us a copy of his paper and the corresponding code.

## 2 Model of Computation

We model a coarse-grained parallel machine as follows: A coarse-grained machine consists of several relatively powerful processors connected by an interconnection network. Rather than making specific assumptions about the underlying network, we assume a two-level model of computation. Communication between processors has a start-up overhead of  $\tau$ , while the data transfer rate is  $\frac{1}{\mu}$ . For our complexity analysis we assume that  $\tau$  and  $\mu$  are constant and independent of the link congestion and distance between two processors. These assumptions closely model the behavior of the CM-5 on which our experimental results are presented. The complexity analysis for other interconnection networks is discussed in a detailed version.

In the following, we describe some important parallel primitives that are repeatedly used in our algorithms and implementations (assuming  $p$  processors). For the analysis of the run times, the reader is referred to [6].

1. **Broadcast** In a Broadcast operation, one processor has an element of data to be broadcast to all other processors. This operation can be performed in  $O((\tau + \mu) \log p)$  time.
2. **Combine** Given an element of data on each processor and a binary associative and commutative operation, the Combine operation computes the result of combining the elements stored on all the processors using the operation and stores the result on every processor. This operation can also be performed in  $O((\tau + \mu) \log p)$  time.
3. **Parallel Prefix** Suppose that  $x_0, x_1, \dots, x_{p-1}$  are  $p$  data elements with processor  $P_i$  containing  $x_i$ . Let  $\otimes$  be a binary associative operation. The Parallel Prefix operation stores the value of  $x_0 \otimes x_1 \otimes \dots \otimes x_i$  on processor  $P_i$ . This operation can be performed in  $O((\tau + \mu) \log p)$  time.
4. **Gather** Given an element of data on each processor, the Gather operation collects all the data and stores it in one of the processors. This can be accomplished in  $O(\tau \log p + \mu p)$  time.

5. **Global Concatenate** This is same as Gather except that the collected data should be stored on all the processors. This operation can also be performed in  $O(\tau \log p + \mu p)$  time.
6. **Transportation Primitive** performs many-to-many personalized communication with possibly high variance in message size. If the total length of the messages being sent out or received at any processor is bounded by  $t$ , the time taken for the communication is  $2\mu t$  (+ lower order terms) when  $t \geq O(p^2 + p\tau/\mu)$ . If the outgoing and incoming traffic bounds are  $r$  and  $c$  instead, the communication takes time  $2\mu(r+c)$  (+ lower order terms) when either  $r \geq O(p^2 + p\tau/\mu)$  or  $c \geq O(p^2 + p\tau/\mu)$  [8].

### 3 Parallel Algorithms for Selection

Parallel algorithms for selection are iterative and work by reducing the number of elements to be considered from iteration to iteration. The elements are distributed across processors and each iteration is performed in parallel by all the processors. Let  $n$  be the number of elements and  $p$  be the number of processors. To begin with, each processor is given  $\lceil \frac{n}{p} \rceil$  or  $\lfloor \frac{n}{p} \rfloor$  elements. Let  $n_i^{(j)}$  be the number of elements in processor  $P_i$  at the beginning of iteration  $j$ . let  $n^{(j)} = \sum_{i=0}^{p-1} n_i^{(j)}$ . Let  $k^{(j)}$  be the rank of the element we need to identify among these  $n^{(j)}$  elements.

**Median of Medians Algorithm** The median of medians algorithm is a straightforward parallelization of the deterministic sequential algorithm [5] and has recently been suggested and implemented by Bader et. al. [3]. This algorithm requires load balancing at the beginning of each iteration.

At the beginning of iteration  $j$ , each processor finds the median of its  $n_i^{(j)} = \lceil \frac{n^{(j)}}{p} \rceil$  or  $\lfloor \frac{n^{(j)}}{p} \rfloor$  elements using the sequential deterministic algorithm. All such medians are gathered on one processor, which then finds the median of these medians. The median of medians is then estimated to be the median of all the  $n^{(j)}$  elements. The estimated median is broadcast to all the processors. Each processor scans through its set of points and splits them into two subsets containing elements less than or equal to and greater than the estimated median, respectively. A Combine operation and a comparison with  $k^{(j)}$  determines which of these two subsets to be discarded and the value of  $k^{(j+1)}$  needed for the next iteration.

Selecting the median of medians as the estimated median ensures that the estimated median will have at least a guaranteed fraction of the number of elements

below it and above it. This ensures that the worst case number of iterations required is  $O(\log n)$ . Let  $n_{max}^{(j)} = \max_{i=0}^{p-1} n_i^{(j)}$ . Thus, finding the local median and splitting the set of points into two subsets based on the estimated median each requires  $O(n_{max}^{(j)})$  time in the  $j^{th}$  iteration. The remaining work is one Gather, one Broadcast and one Combine operation. Therefore, the worst-case running time of this algorithm is  $\sum_{j=0}^{\log n-1} O(n_{max}^{(j)} + \tau \log p + \mu p)$ . Since  $n_{max}^{(j)} = O(\frac{n}{p})$ , the running time is  $O(\frac{n}{p} \log n + \tau \log p \log n + \mu p \log n)$ . This is the running time of the Median of Medians algorithm assuming load balancing but ignoring the cost of it.

**Bucket-Based Algorithm** The bucket-based algorithm [7] attempts to reduce the worst-case running time of the above algorithm without requiring load balance. As before, local medians are computed on each processor. However, the estimated median is taken to be the weighted median of the local medians, with each median weighted by the number of elements on the corresponding processor. This will again guarantee that a fixed fraction of the elements is dropped from consideration every iteration. The number of iterations of the algorithm remains  $O(\log n)$ .

The dominant computational work in the median of medians algorithm is the computation of the local median and scanning through the local elements to split them into two sets based on the estimated median. In order to reduce this work which is repeated every iteration, the bucket-based approach preprocesses the local data into  $\log p$  buckets such that for any  $0 \leq i < j < \log p$ , every element in bucket  $i$  is smaller than any element in bucket  $j$ . This requires  $O(\frac{n}{p} \log \log p)$  time. The cost of finding the local median reduces from  $O(\frac{n}{p})$  to  $O(\log \log p + \frac{n}{p \log p})$ . To split the local data into two sets based on the estimated median, first identify the bucket that should contain the estimated median. Only the elements in this bucket need to be split. Thus, this operation also requires only  $O(\log \log p + \frac{n}{p \log p})$  time.

After preprocessing, the worst-case run time for selection is  $O(\log \log p \log n + \frac{n}{p \log p} \log n + \tau \log p \log n + \mu p \log n) = O(\frac{n}{p \log p} \log n + \tau \log p \log n + \mu p \log n)$  for  $n > p^2 \log \log p$ . Therefore, the worst-case run time of the bucket-based approach is  $O(\frac{n}{p} (\log \log p + \frac{\log n}{\log p}) + \tau \log p \log n + \mu p \log n)$  without any load balancing.

**Randomized Selection Algorithm** Sequentially, the randomized selection algorithm works as follows. A random element is selected to be the estimated median. The set is split into two subsets  $S_1$  and  $S_2$  of

elements smaller than or equal to and greater than the estimated median. If  $|S_1| \geq k$ , recursively find the element with rank  $k$  in  $S_1$ . If not, recursively find the element with rank  $(k - |S_1|)$  in  $S_2$ . This can be parallelized easily. All processors use the same random number generator with the same seed so that they can produce identical random numbers. Consider the behavior of the algorithm in iteration  $j$ . First, a parallel prefix operation is performed on the  $n_i^{(j)}$ 's. All processors generate a random number between 1 and  $n^{(j)}$  to pick an element at random, which is taken to be the estimate median. From the parallel prefix operation, each processor can determine if it has the estimated median and if so broadcasts it. Each processor scans through its set of points and splits them into two subsets containing elements less than or equal to and greater than the estimated median, respectively. A Combine operation and a comparison with  $k^{(j)}$  determines which of these two subsets to be discarded and the value of  $k^{(j+1)}$  needed for the next iteration.

It can be shown that the number of iterations is  $O(\log n)$  with high probability. Let  $n_{max}^{(j)} = \max_{i=0}^{p-1} n_i^{(j)}$ . Thus, splitting the set of points into two subsets based on the median requires  $O(n_{max}^{(j)})$  time in the  $j^{th}$  iteration. The remaining work is one Parallel Prefix, one Broadcast and one Combine operation. Therefore, the total expected running time of the algorithm is  $\sum_{j=0}^{\log n - 1} O(n_{max}^{(j)}) + (\tau + \mu) \log p$ .

With load balancing and ignoring the cost of it, the running time of the algorithm reduces to  $O(\frac{n}{p} + (\tau + \mu) \log p \log n)$ . Even without this load balancing, assuming that the initial data is randomly distributed, the running time is expected to be  $O(\frac{n}{p} + (\tau + \mu) \log p \log n)$ .

**Fast Randomized Selection Algorithm** The expected number of iterations required for the randomized median finding algorithm is  $O(\log n)$ . In this section we discuss an approach due to Rajasekharan et. al. [7] that requires only  $O(\log \log n)$  iterations for convergence with high probability.

Suppose we want to find the  $k^{th}$  smallest element among a given set of  $n$  elements. Sample a set  $S$  of  $o(n)$  keys at random and sort  $S$ . The element with rank  $m = \lceil \frac{k|S|}{n} \rceil$  in  $S$  will have an expected rank of  $k$  in the set of all points. Identify two keys  $l_1$  and  $l_2$  in  $S$  with ranks  $m - \delta$  and  $m + \delta$  where  $\delta$  is a small integer such that with high probability the rank of  $l_1$  is  $< k$  and the rank of  $l_2$  is  $> k$  in the given set of points. With this, all the elements that are either  $< l_1$  or  $> l_2$  can be eliminated. Recursively find

the element with rank  $k - \text{rank}(l_1)$  in the remaining  $(\text{rank}(l_2) - \text{rank}(l_1) - 1)$  elements. If the number of elements is sufficiently small, they can be directly sorted to find the required element.

In iteration  $j$ , Processor  $P_i^{(j)}$  randomly selects  $n_i^{(j)} \frac{n^\epsilon}{n^{(j)}}$  of its  $n_i^{(j)}$  elements. The selected elements are sorted using a parallel sorting algorithm. Once sorted, the processors containing the elements  $l_1^{(j)}$  and  $l_2^{(j)}$  broadcast them. Each processor finds the number of elements less than  $l_1^{(j)}$  and greater than  $l_2^{(j)}$  contained by it. Using Combine operations, the ranks of  $l_1^{(j)}$  and  $l_2^{(j)}$  are computed and the appropriate action of discarding elements is undertaken by each processor. A large value of  $\epsilon$  increases the overhead due to sorting. A small value of  $\epsilon$  increases the probability that both the selected elements ( $l_1^{(j)}$  and  $l_2^{(j)}$ ) lie on one side of the element with rank  $k^{(j)}$ , thus causing an unsuccessful iteration. By experimentation, we found a value of 0.6 to be appropriate.

Rajasekharan et. al. show that the expected number of iterations of this median finding algorithm is  $O(\log \log n)$  and that the expected number of points decreases geometrically after each iteration. If  $n^{(j)}$  is the number of points at the start of the  $j^{th}$  iteration, only a sample of  $o(n^{(j)})$  keys is sorted. Thus, the cost of sorting,  $o(n^{(j)} \log n^{(j)})$  is dominated by the  $O(n^{(j)})$  work involved in scanning the points.

As in the randomized median finding algorithm, one iteration of the median finding algorithm takes  $O(n_{max}^{(j)} + (\tau + \mu) \log p)$  time. Assuming load balancing and ignoring the cost of load balancing, the running time of median finding is  $O(\frac{n}{p} + (\tau + \mu) \log p \log \log n)$ . Even without this load balancing, the running time is expected to be  $O(\frac{n}{p} + (\tau + \mu) \log p \log \log n)$  for random distribution of data.

## 4 Algorithms for load balancing

In order to ensure that the computational load on each processor is approximately the same during every iteration of a selection algorithm, we need to dynamically redistribute the data such that every processor has nearly equal number of elements. We have developed and compared several algorithms for load balancing for the median finding problems. In the following, we describe only one of them for brevity.

Initially, processor  $P_i$  has two integers  $s_i$  and  $r_i$  and  $s_i$  elements of data such that  $\sum_{i=0}^{p-1} s_i = \sum_{i=0}^{p-1} r_i$ . Let  $s_{max} = \max_{i=0}^{p-1} s_i$  and  $r_{max} = \max_{i=0}^{p-1} r_i$ . The objective is to redistribute the data such that processor  $P_i$  contains  $r_i$  elements. Every processor retains  $\min\{s_i, r_i\}$  of its original elements. If  $s_i > r_i$ , the pro-

cessor has  $(s_i - r_i)$  elements in excess and is labeled a source. Otherwise, the processor needs  $(r_i - s_i)$  elements and is labeled a sink. The excessive elements in the source processors and the number of elements needed by the sink processors are transferred using the transportation primitive. The maximum number of messages sent out by a processor is  $O(p)$  and the maximum number of elements sent is  $(n_{max} - n_{avg})$ , where  $n_{avg} = \lfloor \frac{n}{p} \rfloor$ .

The maximum number of elements received by a processor is  $n_{avg}$ . The worst-case running time is  $O(\tau p + \mu(n_{max} - n_{avg}))$ .

We call this algorithm non-order maintaining load balance algorithm as it does not maintain the ordering of the data. We also considered two other load balancing algorithms: the dimension exchange method and the global exchange method [2].

## 5 Implementation Results

We have implemented all the selection algorithms and the load balancing techniques on the CM-5. To experimentally evaluate the algorithms, we have chosen the problem of finding the median of a given set of numbers. We ran each selection algorithm with and without any load balancing (except for the bucket-based approach which does not use load balancing). In the following we briefly summarize a subset of the results. For details, the reader is referred to [2].

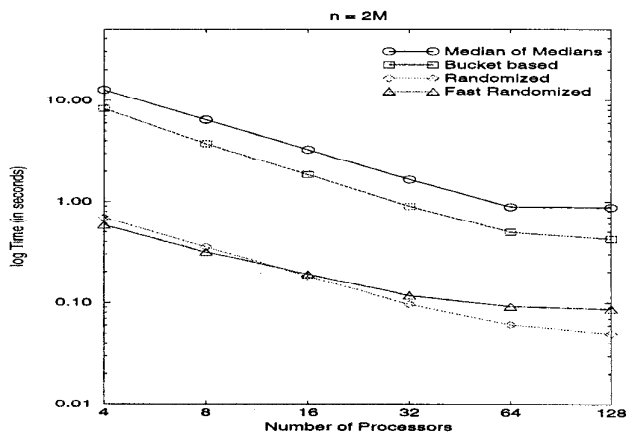


Figure 1: Performance of selection algorithms without load balancing (except for the median of medians algorithm for which load balancing is used) on random data sets.

The algorithms are run until the total number of elements falls below  $p^2$ , at which point the elements are gathered on one processor and the problem is solved by sequential selection. This was used to provide a consistent comparison between the different schemes.

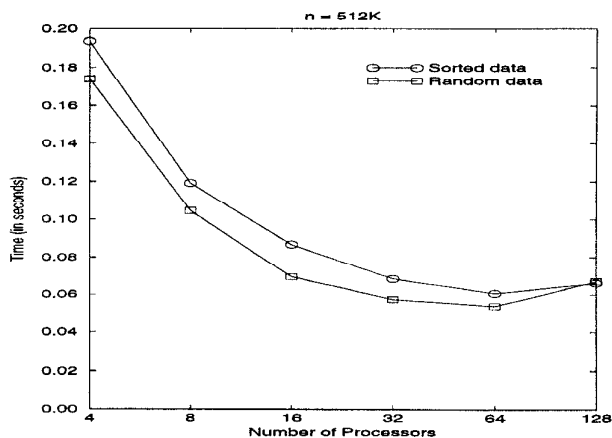


Figure 2: Performance of fast randomized selection algorithm using random and sorted data sets.

For each value of the total number of elements, we have run each of the algorithms on two types of inputs - random and sorted. We ran each experiment on five different random sets of data and used the average running time. Random data sets constitute close to the best case input for the selection algorithms. The sorted input is a close to the worst-case input for the selection algorithms.

The execution times of the four different selection algorithms without using load balancing for random data (except for median of medians algorithm requiring load balancing) with 2M numbers is shown in Figure 1. An immediate observation is that the randomized algorithms are superior to the deterministic algorithms by an order of magnitude. The factor of improvement in randomized parallel selection algorithms over deterministic parallel selection is due to improvements in both the sequential and parallel parts. Among the deterministic algorithms, the bucket-based approach consistently performed better than the median of medians approach by about a factor of two for random data. For sorted data, the bucket-based approach which does not use any load balancing ran only about 25% slower than median of medians approach with load balancing [2].

The effect of the various load balancing techniques on the fast randomized algorithm is given in Figure 2. The execution times are consistently better for sorted data without using any load balancing. For the randomized selection the cost of load balancing offset the improvements resulting in approximately the same overall cost. Load balancing for random data almost always had a negative effect on the total execution time for both the randomized methods [2]. Consider the variance in the running times between random and sorted data for both the randomized algo-

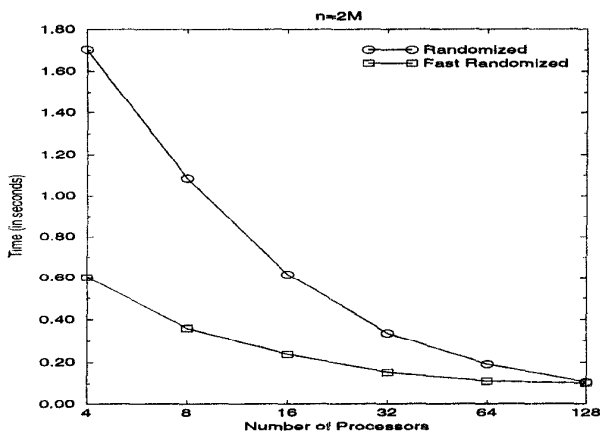
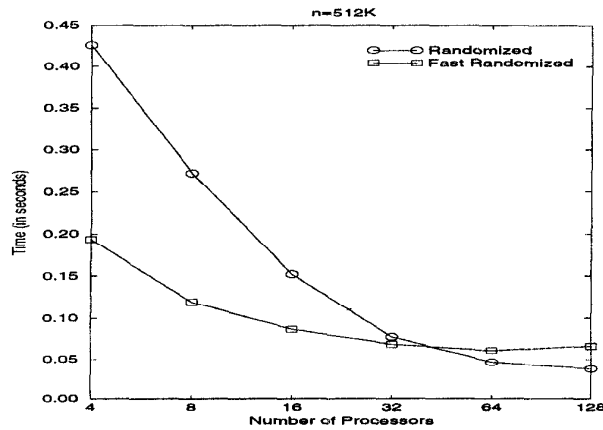


Figure 3: Performance of the two randomized selection algorithms on sorted data sets using the best load balancing strategies for each algorithm

gorithms. The randomized selection algorithm ran 2 to 2.5 times faster for random data than for sorted data [2]. The fast randomized selection with load balancing performs equally well on both best and worst-case data.

In Figure 3, we see a comparison of the two randomized algorithms for sorted data with the best load balancing strategies for each algorithm – no load balancing for randomized selection and non-order maintaining load balancing for fast randomized algorithm (which performed slightly better than other strategies). We see that, for large  $n$ , fast randomized selection is superior. For small data sets the cost of sorting the sample offsets the benefits of reduction in number of iterations.

## 6 Conclusions

We conclude that randomized algorithms are faster by an order of magnitude. If determinism is desired, the bucket-based approach is superior to the median

of medians algorithm. Of the two randomized algorithms, fast randomized selection with load balancing delivers good performance for all types of input distributions with very little variation in the running time. The overhead of using load balancing with well-behaved data is insignificant. Any of the load balancing techniques described can be used without significant variation in the running time. Randomized selection performs well for well-behaved data. There is a large variation in the running time between best and worst-case data. Load balancing does not improve the performance of randomized selection irrespective of the input data distribution.

## References

- [1] M. Ajtai, J. Komlos, W.L. Steiger and E. Szemerédi, Deterministic selection in  $O(\log \log N)$  parallel time, *Proceedings of the 18<sup>th</sup> Annual ACM Symposium on Theory of Computing* (1986) 188-195.
- [2] Ibraheem Al-furaih, Srinivas Aluru, Sanjay Goil and Sanjay Ranka, Practical Algorithms for Selection on Coarse-Grained Parallel Computers, Technical report available at <http://www.cis.ufl.edu/~ranka/>.
- [3] D. A. Bader and J. J'aj'a, Practical parallel algorithms for dynamic data redistribution, median finding and selection, Technical Report CS-TR-3494, School of Computer Science, University of Maryland, July 1995.
- [4] Berthomi, A. Ferreira, B.M. Maggs, S. Perennes, and C.G. Plaxton, Sorting-based selection algorithms for hypercubic networks, *Proc. 7<sup>th</sup> International Parallel Processing Symposium* (1993) 89-95.
- [5] 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) (1972) 448-461.
- [6] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin Cummings Publishing Company, California, 1994.
- [7] S. Rajasekharan, W. Chen and S. Yooseph, Unifying themes for network selection, *Proc. 5<sup>th</sup> International Symposium on Algorithms and Computation*, Beijing, China, 1994, Springer-Verlag Lecture Notes in CS 834, 92-100.
- [8] S. Ranka, R.V. Shankar and K.A. Alsabti, Many-to-many communication with bounded traffic, *Proc. Frontiers of Massively Parallel Computation* (1995), to appear.