Selection Algorithms for Parallel Disk Systems

Sanguthevar Rajasekaran*
Dept. of CISE, Uniersity of Florida
Gainesville FL 32611

Abstract

With the widening gap between processor speeds and disk access speeds, the I/O bottleneck has become critical. Parallel Disk Systems (PDS) have been introduced to alleviate this bottleneck. In this paper we present deterministic and randomized selection algorithms for parallel disk systems. The algorithms to be presented, in addition to being asymptotically optimal, have small underlying constants in their time bounds and hence have the potential of being practical.

1 Introduction

Given a sequence of n keys and an integer i, $1 \leq i \leq n$, the problem of selection is to identify the ith smallest of the n keys. This important comparison problem has been extensively studied. Numerous asymptotically optimal sequential algorithms have been discovered. Asymptotically optimal algorithms have been presented for varying parallel models as well.

Floyd and Rivest's sequential algorithm [9], in addition to being asymptotically optimal, is simpler than the optimal deterministic algorithm of Blum et al. [7].

*This research was supported in part by an NSF Award CCR-95-03-007 and an EPA Grant R-825-293-01-0.

Reischuk's randomized algorithm for the parallel comparison tree model takes $O(1)$ time using n processors and hence is clearly optimal. Meggido's algorithm finds the maximum in $O(1)$ time using n parallel comparison tree processors. Both of these algorithms are based on the idea of [9]. A constant time randomized algorithm for maximum selection for the CRCW PRAM has been given by Rajasekaran and Sen [18].

It is easy to obtain a randomized logarithmic time algorithm for selection using $\frac{n}{\log n}$ CRCW PRAM processors (see e.g., [10]). A deterministic logarithmic time asymptotically optimal algorithm is also known [11]. Cole's algorithm for the CRCW PRAM runs in $O(\log n \log^* n / \log\log n)$ time using an optimal number of processors. Chaudhuri, Hagerup, and Raman's algorithm [2] runs in $O(\log n / \log\log n)$ time on the CRCW PRAM using $n \log\log n / \log n$ CRCW PRAM processors.

Optimal algorithms have also been designed for models such as the mesh, the hypercube, meshes with buses, etc. For a survey of parallel selection algorithms, the reader is referred to [16].

We present two selection algorithms for the PDS. The PDS have been proposed with the widening gap between processor speeds and disk access speeds in mind. The first algorithm is randomized and the second algorithm is deterministic. The number of parallel I/O read operations needed

for either is $O\left(\frac{N}{DB}\right)$, where N is the number of input keys, D is the number of disks, and B is the block size. Thus the algorithms are asymptotically optimal. Due to the small underlying constants, the algorithms have the potential of being practical as well.

Many problems such as sorting, graph problems, etc. have been studied on the PDS. Any sorting algorithm can clearly be used to perform selection. The known lower bound on the number of passes through the data for sorting on the PDS is $\Omega\left(\frac{\log(N/B)}{\log(M/B)}\right)$, where N is the number of input keys, M is the internal memory size, and B is the block size. In practice one can assume that N is a polynomial in M and M is a polynomial in B and hence this lower bound simply means a constant number of passes through the data.

The PDS model we use is the one suggested by Vitter and Shriver in their pioneering paper [20]. Several asymptotically optimal sorting algorithms have been proposed for the PDS. All these algorithms, though theoretically important, have rather large constants in their time bounds. Recently, Rajasekaran has proposed a sorting algorithm called the (l, m)-merge sort (LMM) [15]. An implementation of this algorithm on the PDS makes no more than $\left[\frac{\log(N/M)}{\log\left(\min\{(M/B),\sqrt{M}\}\right)}+1\right]^2$ passes through the data. It has been shown that when D is large, LMM performs better than the disk-striped merge sort (DSM) algorithm that is used in practice.

An interesting question is if we can perform selection on the PDS in time better than sorting time. This paper answers this question in the affirmative. In Section 2 we introduce the PDS model. In Sections 3 and 4 we present our randomized and deterministic algorithms, respectively. Section 5 concludes the paper.

2 Parallel Disk Systems

Several models for parallel disks have been investigated in the literature. The model employed in this paper is the one introduced in one of the pioneering papers of Vitter and Shriver [20]. In this model there are D distinct and independent disk drives. The disks can simultaneously transmit a block of data. A block consists of B records. If M is the internal memory size, then one usually requires that $M \geq 2DB$. For the algorithms presented in this paper, a choice of $M = 3DB$ suffices. Of this, DB amount of memory is used to prefetch data in order to overlap computation and data access. From hereon, we use M to denote DB.

The problem of disk sorting was first studied by Aggarwal and Vitter in their foundational paper [5]. In the model they considered, each I/O operation results in the transfer of D blocks each block having B records. A more realistic model was envisioned in [20]. Several asymptotically optimal algorithms have been given for sorting on this model. Nodine and Vitter's optimal algorithm [13] involves solving certain matching problems. Aggarwal and Plaxton's optimal algorithm [4] is based on the Sharesort algorithm of Cypher and Plaxton. Vitter and Shriver gave an optimal randomized algorithm for disk sorting [20]. All these results are highly nontrivial and theoretically interesting. However, the underlying constants in their time bounds are high.

In practice the simple disk-striped mergesort (DSM) is used [6], even though it is not asymptotically optimal. DSM has the advantages of simplicity and a small constant. Data accesses made by DSM is such that at any I/O operation, the same portions of the D disks are accessed. This has the effect of having a single disk which can

transfer DB records in a single I/O operation. An $\frac{M}{DB}$-way mergesort is employed by this algorithm. To start with, initial runs are formed in one pass through the data. At the end the disk has N/M runs each of length M. Next, $\frac{M}{DB}$ runs are merged at a time. Blocks of any run are uniformly striped across the disks so that in future they can be accessed in parallel utilizing the full bandwidth. Each phase of merging involves one pass through the data. There are $\frac{\log(N/M)}{\log(M/DB)}$ phases and hence the total number of passes made by DSM is $\frac{\log(N/M)}{\log(M/DB)}$. In other words, the total number of I/O read operations performed by the algorithm is $\frac{N}{DB}\left(1+\frac{\log(N/M)}{\log(M/DB)}\right)$. The constant here is just 1.

The known lower bound on the number of passes for parallel disk sorting is $\Omega\left(\frac{\log(N/M)}{\log(M/B)}\right)$. If one assumes that N is a polynomial in M and that B is small (which are readily satisfied in practice), the lower bound simply yields $\Omega(1)$ passes. All the abovementioned optimal algorithms make only $O(1)$ passes. So, the challenge in the design of parallel disk sorting algorithms is in reducing this constant. If $M = 2DB$, the number of passes made by DSM is $1+\log(N/M)$, which indeed can be very high.

Recently, several works have been done that deal with the practical aspects. Pai, Schaffer, and Varman [14] analyzed the average case performance of a simple merging algorithm, employing an approximate model of average case inputs. Barve, Grove, and Vitter [6] have presented a simple randomized algorithm (SRM) and analyzed its performance. The analysis involves the solution of certain occupancy problems. The expected number $Read_{SRM}$ of I/O read operations made by their algorithm is such that

$$Read_{SRM} \leq \frac{N}{DB} + \frac{N}{DB}\frac{\log(N/M)}{\log kD}\frac{\log D}{k\log\log D} \quad (1)$$

$$* \left(1+\frac{\log\log\log D}{\log\log D}+\frac{1+\log k}{\log\log D}+O(1)\right)$$

The algorithm merges $R = kD$ runs at a time, for some integer k. When $R = \Omega(D\log D)$, the expected performance of their algorithm is optimal. However, in this case, the internal memory needed is $\Omega(BD\log D)$. They have also compared SRM with DSM through simulations and shown that SRM performs better than DSM.

Recently, Rajasekaran [15] has presented an algorithm (called (l,m)-merge sort (LMM)) which is asymptotically optimal under the assumptions that N is a polynomial in M and B is small. The algorithm is as simple as DSM. LMM makes less number of passes through the data than DSM when D is large.

The selection algorithms to be presented are as simple as the DSM. Data accesses are such that in any I/O operation, the same portions of the disks are accessed. Alive keys (i.e., keys that have not yet been eliminated) in any stage are uniformly striped across the disks.

3 A Randomized Selection Algorithm

In this section we present a randomized selection algorithm for the PDS. The number of I/O read operations made by the algorithm is $O\left(\frac{N}{DB}\right)$ with high probability.

Almost all the selection algorithms proposed in the literature, be they deterministic or randomized, sequential or parallel, are based on sampling.

For example, Floyd and Rivest's randomized algorithm [9] is based on random sampling. The algorithm consists of the following steps. 1) Select a random sample S of s elements from the input set X; 2) Sort the sample S and find two elements

ℓ_1 and ℓ_2 from S such that the ith smallest element of X will have a value in between ℓ_1 and ℓ_2 and also the number of keys from X that have a value in between ℓ_1 and ℓ_2 is 'small'; 3) Eliminate keys of X that do not have a value in the range $[\ell_1, \ell_2]$; and 4) Perform an appropriate selection in the set of remaining keys.

Sampling techniques have been repeatedly used to develop selection algorithms for a variety of parallel models of computing. Though these algorithms employ sampling as a common theme, they have model-dependent innovations and employ additional techniques. Our randomized algorithm is also based on the above theme.

A Sampling Lemma. Let Y be a sequence of n numbers from a linear order and let $S = \{k_1, k_2, \ldots, k_s\}$ be a random sample from Y. Also let $k'_1, k'_2, \ldots, k'_s$ be the sorted order of this sample. If r_i is the rank of k'_i in Y, the following lemma provides a high probability confidence interval for r_i. (The rank of any element k in Y is one plus the number of elements $< k$ in Y.)

Lemma 3.1 *For every* $\alpha > 0$, *Prob.* $\left(|r_i - i\frac{n}{s}| > \sqrt{3\alpha}\frac{n}{\sqrt{s}}\sqrt{\log n}\right) < n^{-\alpha}$.

A proof of the above lemma can be found in [17].

We say a randomized algorithm uses $\widetilde{O}(g(n))$ amount of any resource (like time, space, etc.) if there exists a constant c such that the amount of resource used is no more than $c\alpha g(n)$ with probability $\geq 1 - n^{-\alpha}$ on any input of length n and for any α (see e.g., [10]). Similar definitions apply to $\widetilde{o}(g(n))$ and other such 'asymptotic' functions.

By *high probability* we mean a probability of $\geq 1 - n^{-\alpha}$ for any fixed $\alpha \geq 1$ (n being the input size of the problem at hand). Let $B(n, p)$ denote a binomial random variable with parameters n and p.

One of the most frequently used facts in analyzing randomized algorithms is *Chernoff bounds*. These bounds provide close approximations to the probabilities in the tail ends of a binomial distribution. Let X stand for the number of heads in n independent flips of a coin, the probability of a head in a single flip being p. X is also known to have a binomial distribution $B(n, p)$. The following three facts (known as Chernoff bounds) will be used in the paper (and were discovered by Chernoff [3] and Angluin & Valiant [1]):

$$\text{Prob.}[X \geq m] \;\leq\; \left(\frac{np}{m}\right)^m \; e^{m-np},$$

$$\text{Prob.}[X \geq (1+\epsilon)np] \;\leq\; exp(-\epsilon^2 np/3), \text{and}$$

$$\text{Prob.}[X \leq (1-\epsilon)np] \;\leq\; exp(-\epsilon^2 np/2),$$

for any $0 < \epsilon < 1$, and $m > np$.

Now we are ready to describe our algorithm. Let $N = M^c$. In practice c can be assumed to be a constant. In today's PC market, M is of the order of megabytes and the disk space is of the order of gigabytes. So it is perhaps safe to assume that c is no more than 3. To begin with each key is *alive* and $n = N$.

Algorithm RSelect

repeat

Step 1. Let n be the number of alive keys. If $n \leq M$ then goto Step 6. Each alive key is included in the sample S with probability $\frac{M}{n}$. The expected number of sample keys is M. We can show that the actual number of keys in S is $M + \widetilde{o}(M)$. Count the number s of sample keys.

Step 2. Sort S and pick two keys ℓ_1 and ℓ_2 from S whose ranks in S are $i\frac{s}{n} - \delta$ and $i\frac{s}{n} + \delta$, respectively, for $\delta \geq \sqrt{3\alpha s \log n}$, for any fixed $\alpha \geq 1$.

Step 3. Compute the number n_1 of alive keys that are less than ℓ_1 and the number n_2 of alive keys that have a value in the range $[\ell_1, \ell_2]$.

Step 4. If $i < n_1$, or $i > n_1 + n_2$, or $n_2 > \frac{n}{M^{0.4}}$, goto Step 1.

Step 5. Any alive key whose value lies outside the range $[\ell_1, \ell_2]$ dies. Set $i = i - n_1$ and $n = n_2$.

forever

Step 6. Sort the alive keys and output the ith smallest key.

Analysis. In Step 1, the number of sample keys has a binomial distribution $B(n, M/n)$. An application of Chernoff bounds shows that $s = M + \widetilde{o}(M)$. Also Step 1 takes $O(\frac{n}{DB})$ I/O read operations.

Since S is kept in the internal memory, Step 2 does not involve any I/O operations.

In Step 3, an application of Lemma 3.1 implies that the number n_2 of keys surviving Step 5 is $\widetilde{O}\left(\frac{n}{\sqrt{s}}\sqrt{\log n}\right) = \widetilde{O}\left(\frac{n}{M^{0.4}}\right)$. As a result it follows that, the number of iterations of the *repeat* loop is $\widetilde{O}(c)$.

In Step 4, we can show that the probability of executing the goto statement is very small.

Step 5 involves $O(\frac{n}{DB})$ I/O read operations.

Note that the number of surviving keys from one iteration of the *repeat* loop to the next decreases by a factor of $M^{0.4}$ with high probability. Thus it follows that the total number of I/O read operations made by the entire algorithm is $\widetilde{O}\left(\frac{N}{DB}\right)$.

Thus we get the following Theorem.

Theorem 3.1 *Selection from out of N keys can be performed on the PDS using $\widetilde{O}\left(\frac{N}{DB}\right)$ I/O read operations.* □

4 A Deterministic Selection Algorithm

In this section we present our deterministic selection algorithm for the PDS. The number of I/O read operations performed by the algorithm is $O\left(\frac{N}{DB}\right)$. The underlying constant is small and hence this algorithm has the potential of being practical.

Sampling has also dominated as a technique useful in the design of deterministic selection algorithms. For example, Blum et al.'s algorithm [7] partitions the input such that there are 5 elements in each part, finds the median of each part, finds the median M of these medians, splits the input into two groups (those that are $\leq M$ and those that are greater than M), identifies the group that has the key to be selected, and finally performs a selection in the group that contains the key to be selected. The medians of the 5-element parts can be thought of as a sample of the input keys and hence the median M of these medians can be expected to be an approximate median of the input keys.

Consider a collection X of n keys whose ith smallest key we are interested in finding. We use the following strategy to identify two elements of X such that they will bracket the ith smallest element and also the number of keys of X that have a value in between these two keys is not large.

Partition the collection $X = R_0$ such that there are M keys in each part. Sort each part. From each part retain those keys that are at a distance of $\sqrt{M}$ from each other. That is, keep the keys whose ranks are $\sqrt{M}, 2\sqrt{M}, 3\sqrt{M}, \ldots$. Thus the number of keys in the retained set R_1 is $\frac{n}{\sqrt{M}}$. Now group the elements of R_1 such that there are M elements in each part, sort each part, and retain only every $\sqrt{M}$th element in each part. Call the

retained set R_2. Proceed to obtain R_i's in a similar fashion (for $i \geq 3$) until we reach a stage when $|R_j| \leq M$. If $n = M^c$, then clearly, $j = 2c - 2$.

This process can be represented by a tree of degree $\sqrt{M}$. Each leaf has M input elements. All the elements in the leaves constitute R_0. The root has R_j. Let the root be in level j. Let its children be in level $j-1$, and so on. The leaves are at level 0. There are $\sqrt{M}$ children to the root. Each such child has M elements. $\sqrt{M}$ elements are passed on from each child to its parent. In general each node in the tree has M elements. $\sqrt{M}$ elements from out of these will go to its parent. Each node $\sqrt{M}$ children.

Pick from R_j two elements ℓ_1 and ℓ_2 whose ranks are $i\frac{|R_j|}{n} - \delta$ and $i\frac{|R_j|}{n} + \delta$, respectively.

Without loss of generality assume that $|R_j| = M$. Then, $|R_i| = M(\sqrt{M})^{j-i}$. Consider an element x whose rank in R_j is q. Then the rank of q in R_{j-1} will be in the range $[q\sqrt{M}, q\sqrt{M} + \sqrt{M}(\sqrt{M} - 1)]$. This rank is also in the interval $[q\sqrt{M}, q\sqrt{M} + M]$. I.e., there is an uncertainty of M in the rank of x in R_{j-1}. Each child of the root contributes $\sqrt{M} - 1 \approx \sqrt{M}$ to this uncertainty. In general the uncertainty in the rank of x in R_i is contributed to by each node in level i. Note that there are $M^{(j-i)/2}$ nodes at level i. Each such node contributes $\sqrt{M} - 1$ to the uncertainty. Thus, if $U(i)$ is the maximum possible rank of x in R_i, then $U(i)$ satisfies:

$$U(i) \leq \sqrt{M}\, U(i+1) + M^{(j-i+1)/2}$$

which solves to $U(i) \leq M^{(j-i)/2}U(j) + (j - i)M^{(j-i+1)/2}$. When $i = 0$, we get $U(0) \leq M^{j/2}U(j) + jM^{(j+1)/2}$. In other words, $U(0) \leq qM^{c-1} + (2c-2)\frac{n}{\sqrt{M}}$. Note that $c = \frac{\log n}{\log M}$.
As a result, if we pick δ to be $(2c - 2 + \epsilon)\sqrt{M}$, for any $\epsilon > 0$, the rank of ℓ_1 in R_0 will be in the interval

$$\left[i - (2c - 2 + \epsilon)\frac{n}{\sqrt{M}}, i - \epsilon\frac{n}{\sqrt{M}}\right]$$

Also, the rank of ℓ_2 in R_0 will lie in the interval

$$\left[i + (2c - 2 + \epsilon)\frac{n}{\sqrt{M}}, i + (4c - 4 + \epsilon)\frac{n}{\sqrt{M}}\right]$$

Put together, we realize that the ith smallest element of R_0 will have a value in the interval $[\ell_1, \ell_2]$ and also the number of keys of R_0 that have a value in the interval $[\ell_1, \ell_2]$ is no more than $(6c - 6 + 2\epsilon)\frac{n}{\sqrt{M}}$.

Call the above process of starting from R_0 and obtaining $R_1, R_2, \ldots, R_j$ a *stage of sampling.* Note also that the number of I/O read operations needed for a stage is $O\left(\frac{n}{DB}\right)$ where $|R_0| = n$. One could see that a stage of sampling corresponds to one iteration of the *repeat* loop of RSelect. Similar sampling techniques have been employed by Munro and Paterson [12].

Now we present a detailed description of our algorithm. Let $K = k_1, k_2, \ldots, k_N$ be the input. Say we are interested in finding the ith smallest key. To begin with each key is alive and $n = N$.

Algorithm DSelect

repeat

Step 1. If $n \leq M$ goto Step 3. Perform a stage of sampling in the collection of alive keys. As a result, obtain two keys ℓ_1 and ℓ_2 that will bracket the key to be selected.

Step 2. Scan through the alive keys and kill the keys that have a value outside the range $[\ell_1, \ell_2]$. Count the number of keys surviving this step. Let this number be n.

forever

Step 3. Sort the alive keys and output the ith smallest element.

Analysis. Clearly, the number of alive keys reduces by a factor of $\Omega\left(\frac{\sqrt{M}\log M}{\log n}\right)$ from one iteration to the next of the *repeat* loop. Therefore, if $N = M^c$, the number of iterations of the *repeat* loop is $O(c)$.

Also, the number of I/O read operations needed in any iteration of the *repeat* loop is $O\left(\frac{n}{DB}\right)$ as has been discussed before. Since the number of alive keys decreases by a factor of $\Omega\left(\frac{\sqrt{M}\log M}{\log n}\right)$ from one iteration to the next, the total number of I/O read operations is only $O\left(\frac{N}{DB}\right)$. We arrive at the following Theorem.

Theorem 4.1 *We can perform selection from out of N given keys on the PDS using $O\left(\frac{N}{DB}\right)$ I/O read operations.* □

5 Conclusions

We have presented two selection algorithms for the PDS. Both are asymptotically optimal. The underlying constants in the time bounds are small and hence the algorithms have the potential of being practical.

References

[1] D. Angluin and L. G. Valiant, Fast Probabilistic Algorithms for Hamiltonian Paths and Matchings, *Journal of Computer and System Sciences* 18, 1979, pp. 155-193.

[2] S. Chaudhari, T. Hagerup, and R. Raman, Approximate and Exact Deterministic Parallel Selection, *Proc. 18th Annual Symposium on Mathematical Foundations of Computer Science*, 1993.

[3] H. Chernoff, A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations, *Annals of Mathematical Statistics* 23, 1952, pp. 493-507.

[4] A. Aggarwal and C. G. Plaxton, Optimal Parallel Sorting in Multi-Level Storage, *Proc. Fifth Annual ACM Symposium on Discrete Algorithms*, 1994, pp. 659-668.

[5] A. Aggarwal and J. S. Vitter, The Input/Output Complexity of Sorting and Related Problems, *Communications of the ACM*, 1988, 31(9):1116-1127.

[6] R. Barve, E. F. Grove, and J. S. Vitter, Simple Randomized Mergesort on Parallel Disks, Technical Report CS-1996-15, Department of Computer Science, Duke University, October 1996.

[7] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, R. E. Tarjan, Time Bounds for Selection, *Journal of Computer and System Sciences* 7, 1973, pp. 448-461.

[8] R. Cole, An Optimally Efficient Selection Algorithm, *Information Processing Letters* 26, 1988, pp. 295-299.

[9] R. W. Floyd and R. L. Rivest, Expected Time Bounds for Selection, *Communications of the ACM* 18(3), 1975, pp. 165-172.

[10] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms*, W. H. Freeman Press, 1998.

[11] J. J'a Já, *Parallel Algorithms: Design and Analysis*, Addison-Wesley Publishers, 1992.

[12] J. I. Munro and M. S. Paterson, Selection and Sorting with Limited Storage, *Theoretical Computer Science* 12, 1980, pp. 315-323.

[13] M. H. Nodine, J. S. Vitter, Large Scale Sorting in Parallel Memories, *Proc. Third Annual ACM Symposium on Parallel Algorithms and Architectures*, 1990, pp. 29-39.

[14] V. S. Pai, A. A. Schaffer, and P. J. Varman, Markov Analysis of Multiple-Disk Prefetching Strategies for External Merging, *Theoretical Computer Science*, 1994, 128(2):211-239.

[15] S. Rajasekaran, A Framework For Simple Sorting Algorithms On Parallel Disk Systems, *Proc. 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[16] S. Rajasekaran, Sorting and Selection on Interconnection Networks, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 21, 1995, pp. 275-296.

[17] S. Rajasekaran and J.H. Reif, Derivation of Randomized Sorting and Selection Algorithms, in *Parallel Algorithm Derivation and Program Transformation*, Edited by R. Paige, J.H. Reif, and R. Wachter, Kluwer Academic Publishers, 1993, pp. 187-205.

[18] S. Rajasekaran and S. Sen, Random Sampling Techniques and Parallel Algorithms Design, in *Synthesis of Parallel Algorithms*, edited by J. H. Reif, Morgan-Kaufmann Publishers, 1993, pp. 411-451.

[19] R. Reischuk, Probabilistic Parallel Algorithms for Sorting and Selection, *SIAM Journal of Computing*, 1985, 14(2):396-409.

[20] J. S. Vitter and E. A. M. Shriver, Algorithms for Parallel Memory I: Two-Level Memories, *Algorithmica*, 1994, 12(2-3):110-147.