1997

Practical Algorithms for Selection on Coarse-Grained Parallel Computers

Ibraheem Al-Furaih
Syracuse University

Srinivas Aluru
Syracuse University

Sanjay Goil
Syracuse University, School of Computer and Information Science

Sanjay Ranka
Syracuse University, School of Computer and Information Science

Practical Algorithms for Selection on Coarse-Grained Parallel Computers

Ibraheem Al-furiah*, Srinivas Aluru, Sanjay Goil †and Sanjay Ranka
Department of Computer and Information Science
Syracuse University, Syracuse, NY 13244-4100
email: *alfuraih, aluru, sgoil, ranka@top.cis.syr.edu*

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. We also consider several algorithms for load balancing needed to keep a balanced distribution of data across processors during the execution of the selection algorithms. We have carried out detailed implementations of all the algorithms discussed on the CM-5 and report on the experimental results. We demonstrate that the 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^{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 elment with rank $\lceil \frac{N}{2} \rceil$.

Sequentially, the selection problem can be solved in $O(N)$ time by using the deterministic algorithm of Blum et. al. [8] or in $O(N)$ expected time by using the randomized algorithm of Floyd et. al. [12]. Both the algorithms work as follows: First, an element of the set is estimated to be the 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 $|S_1| >= k$, recursively find the

*Supported by scholarship from King AbdulAziz City for Science and Technology (KACST), Riyadh, Saudi Arabia.

†The work of this author was supported in part by NASA under subcontract #1057L0013-94 issued by the LANL. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

element with rank k in S_1. If not, recursively find the elment with rank $(k-|S_1|)$ in S_2. Once the number of elements under consideration falls below a constant, the problem is solved directly by sorting and picking the appropriate element. The difference in the deterministic and randomized algorithms lies in the process of selecting the estimated median. In the deterministic algorithm, the set of elments is split into several subsets of constant size and the median of each subset is found directly. The median of the set of these medians is found using the deterministic selection algorithm, and is used as the estimated median. With this, the estimated median will have at least a guaranteed fraction of the number of elments below it and at least a guaranteed fraction of the elements above it, guaranteeing the $O(N)$ worst-case running time of the selection algorithm. In the randomized version, a random element is selected to be the estimated median. The randomized algorithm has a worst-case run time of $O(N^2)$ but has an expected run time of only $O(N)$ and is known to perform better in practice than its deterministic counterpart due to the low constant associated with the algorithm.

Many parallel algorithms for selection have been designed for the PRAM model [2, 3, 4, 9, 14] and for various network models including trees, meshes, hypercubes and reconfigurable architectures [6, 7, 13, 16, 21]. More recently, Bader et.al. [5] implement a parallel deterministic selection algorithm on several distributed memory machines including CM-5, IBM SP-2 and INTEL Paragon. In this paper, we consider and evaluate parallel selection algorithms for coarse-grained distributed memory parallel computers. A coarse-grained parallel computer consists of several relatively powerful processors connected by an interconnection network. Most of the commercially available parallel computers belong to this category. Examples of such machines include CM-5, IBM SP-1 and SP-2, nCUBE 2, INTEL Paragon and Cray T3D.

The rest of the paper is organized as follows: In Section 2, we describe our model of parallel computation and outline some primitives used by the algorithms. In Section 3, we present two deterministic and two randomized algorithms for parallel selection. Selection algorithms are iterative and work by reducing the number of elements to consider from iteration to iteration. Since we can not guarantee that the same number of elments are removed on every processor, this leads to load imbalance. In Section 4, we present several algorithms to perform such a load balancing. Each of the load balancing algorithms can be used by any selection algorithm that requires load balancing. In Section 5, we report and analyze the results we have obtained on the CM-5 by detailed implementation of the selection and load balancing algorithms presented. We conclude the paper in Section 6.

2 Preliminaries

2.1 Model of Parallel 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. The two-level model assumes a fixed cost for an off-processor access independent of the distance between the communicating processors. Communication between processors has a start-up overhead of τ, while the data transfer rate is $\frac{1}{\mu}$. For our complexity analysis we assume that τ and μ are constant and independent of the link congestion and distance between two processors. With new techniques, such as wormhole routing and randomized routing, the distance between communicating processors seems to be less of a determining factor on the amount of time needed to complete the communication. Furthermore, the effect of link contention is eased due to the presence of virtual channels and the fact that link bandwidth is much higher than the bandwidth of node interface. This permits us to use the two-level model and view the underlying interconnection network as a virtual crossbar network connecting the processors. These assumptions closely model the behavior of the CM-5 on which our experimental results are presented. Although the algorithms presented are analyzed under these assumptions, they are architecture-independent and can be efficiently implemented on meshes and hypercubes.

2.2 Parallel Primitives

In the following, we describe some important parallel primitives that are repeatedly used in our algorithms and implementations. We state the running time required for each of these primitives under our model of parallel computation. The analysis of the run times for the primitives described is fairly simple and is omitted in the interest of brevity. The interested reader is referred to [15]. In what follows, p refers to the number of processors.

1. **Broadcast**

   In a Broadcast operation, one processor has an element of data to be broadcasted 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 $x_0, x_1, \ldots, x_{p-1}$ 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 $x_0 \otimes x_1 \otimes \ldots \otimes x_i$ on processor P_i. This operation can be 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**

   The 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 $c \geq O(p^2 + p\tau/\mu)$ [20].

3 Parallel Algorithms for Selection

Parallel algorithms for selection are also 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. Otherwise, this can be easily achieved by using one of the load balancing techniques to be described in Section 4. 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. We use this notation to describe all the selection algorithms presented in this paper.

Algorithm 1 *Median of Medians selection algorithm*

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 $|L_i| = \frac{n}{p}$
$rank$ - desired rank among the total elements
$l = 0$; $r = \frac{n}{p} - 1$
On each processor P_i
while $n > p^2$

Step 1. Use **sequential selection** to find median m_i of list $L_i[l, r]$
Step 2. $M =$ **Gather**(m_i)
Step 3. On P_0

Find median of M, say MoM, and broadcast it to all processors.

Step 4. Partition L_i into $\leq MoM$ and $> MoM$ to give $index_i$, the split index
Step 5. $count =$ **Combine**$(index_i$, add) calculates the number of elements $< MoM$
Step 6. If $(rank \leq count\ )$

$n = count$; $r = index_i$; $rank = count$

else

$n = n - count$; $l = index_i$; $rank = rank - count$

Step 7. **LoadBalance**(L_i, n, p)

Step 8. $L =$ **Gather**$(L_i[l, r])$
Step 9. On P_0

Perform sequential selection to find element q of $rank$ in L
$result =$ **Broadcast**(q)

3.1 Median of Medians Algorithm

The median of medians algorithm is a straightforward parallelization of the deterministic sequential algorithm [8] and has recently been suggested and implemented by Bader et. al. [5]. 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 elments below it and at least a guaranteed fraction of the elements above it, just as in the sequential algorithm. This ensures that the worst case number of iterations required by the algorithm 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 algorithm requires the use of load balancing between iterations. With load balancing, $n_{max}^{(j)} = \frac{1}{2} n_{max}^{(j-1)}$. Assuming load balancing and ignoring the cost of load balancing itself, the running time of the algorithm reduces to $\sum_{j=0}^{\log n-1} O(\frac{n}{2^j p} + \tau \log p + \mu p) = O(\frac{n}{p} + \tau \log p \log n + \mu p \log n)$.

3.2 Bucket-Based Algorithm

The bucket-based algorithm [17] attempts to reduce the worst-case running time of the above algorithm without requiring load balance. First, in order to keep the algorithm deterministic without a balanced number of elements on each processor, the median of medians is replaced by the weighted median of medians. 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 elments 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 $O(\log p)$ 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 can be accomplished by finding the median of the local elements, splitting them into two buckets based on this median and recursively splitting each of these buckets into $\frac{\log p}{2}$ buckets using the same procedure. Thus, preprocessing the local data into $O(\log p)$ buckets requires $O(\frac{n}{p} \log \log p)$ time.

Bucketing the data simplifies the task of finding the local median and the task of splitting the local data into two sets based on the estimated median. To find the local median, identify the bucket containing the median and find the rank of the median in the bucket containing the median in $O(\log \log p)$ time using binary search. The local median can be located in the

Algorithm 2 *Bucket-based selection algorithm*

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 - is a constant
$rank$ - desired rank among the total elements
$l = 0$; $r = \frac{n}{p} - 1$
On each processor P_i
Step 0. Partition L_i on P_i into $\log p$ buckets of equal size such that if $r \in bucket_j$, and $s \in bucket_k$, then $r < s$ if $j < k$
while$(n > p^2)$

Step 1. Find median m_i of list $L_i[l, r]$ by finding the bucket bkt_k containing the median element using a binary search. This is followed by finding the appropriate rank in bkt_k. Let N_i be the number of remaining keys on P_i.
Step 2. $M =$ **Gather**(m_i) ; $Q =$ **Gather**(N_i)
Step 3. On P_0

$m_w =$ weighted median of M
$WM =$ **Broadcast**(m_w)

Step 4. Partition L_i into $\leq WM$ and $> WM$ using the $buckets$ to give $index_i$, the split index
Step 5. $count =$ **Combine**$(index_i$, add$)$ calculates the number of elements less than WM
Step 6. If $(rank \leq count\ )$

$n = count$; $r = index_i$; $rank = count$

else

$n = n - count$; $l = index_i$; $rank = rank - count$

Step 7. $L =$ **Gather**(L_i)
Step 8. On P_0

Perform sequential selection to find element q of $rank$ in L
$result =$ **Broadcast**(q)

Algorithm 3 *Randomized selection algorithm*

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 $|L_i| = \frac{n}{p}$
$rank$ - desired rank among the total elements
$l = 0$; $r = \frac{n}{p} - 1$
On each processor P_i
while$(n > p^2)$

Step 0. $n_i = r - l + 1$
Step 1. $s =$ **PrefixSum**(n_i, p)
Step 2. Generate a random number n_r (same on all processors) between 0 and $n-1$
Step 3. On P_k where $(n_r \in [s - n_i, s])$

$m_{guess} =$ **Broadcast**$(L_i[n_r - (s - n_i)])$

Step 4. Partition L_i into $\leq m_{guess}$ and $> m_{guess}$ to give $index_i$, the split index
Step 5. $count =$ **Combine**$(index_i$, add) calculates the number of elements less than m_{guess}
Step 6. If $(rank \leq count\ )$

$n = count$; $r = index_i$; $rank = count$
else
$n = n - count$; $l = index_i$; $rank = rank - count$

Step 7. $L =$ **Gather**$(L_i[l, r])$
Step 8. On P_0

Perform sequential selection to find element q of $rank$ in L
$result =$ **Broadcast**(q)

bucket by the sequential selection algorithm in $O(\frac{n}{p \log p})$ 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.

3.3 Randomized Selection Algorithm

The randomized median finding algorithm is a straightforward parallelization of the randomized sequential algorithm described in [12]. 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.

Since in each iteration approximately half the remaining points are discarded, the expected number of iterations is $O(\log n)$. 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 $\sum_{j=0}^{\log n-1} O(n_{max}^{(j)} + (\tau+\mu)\log p)$ time. Since $n_{max}^{(j)} = O(\frac{n}{p})$, the expected running time is $O(\frac{n}{p}\log n + (\tau+\mu)\log p \log n)$.

In practice, one can expect that $n_{max}^{(j)}$ reduces from iteration to iteration, perhaps by half. This is especially true if the data is randomly distributed to the processors, eliminating any order present in the input. In fact, by a load balancing operation at the end of every iteration, we can ensure that for every iteration j, $n_{max}^{(j)} = \frac{1}{2} n_{max}^{(j-1)}$. With load balancing and ignoring the cost of it, the running time of the algorithm reduces to $\sum_{j=0}^{\log n-1} O(\frac{n}{2^j p} + (\tau+\mu)\log p) = O(\frac{n}{p} + (\tau+\mu)\log p \log n)$. Even without this load balancing, assuming that the initial data is randomly distributed, the running time is expected to be $O(\frac{n}{p} + (\tau+\mu)\log p \log n)$.

3.4 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. [17] 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 δ 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 $k - l_1$ in the remaining $n - (l_1 + l_2)$ elements. If the number of elements is sufficiently small, they can be directly sorted to find the required element.

If the ranks of l_1 and l_2 are both $< k$ or both $> k$, the iteration is repeated with a different

Algorithm 4 *Fast randomized selection algorithm*

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 $|L_i| = \frac{n}{p}$
$rank$ - desired rank among the total elements
C - a constant
$l = 0$; $r = \frac{n}{p} - 1$
On each processor P_i
while$(n > p^2)$

Step 0. $n_i = r - l + 1$
Step 1. Collect a sample S_i from $L_i[l, r]$ by picking $n_i \frac{n^\epsilon}{n}$ 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 $\lceil \frac{i|S|}{n} - \sqrt{|S|logn} \rceil$ and $\lceil \frac{i|S|}{n} + \sqrt{|S|logn} \rceil$
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_1 and s_2.
Step 6. $c_{mid} =$ **Combine**$(middle,$ add$)$
Step 7. $c_{less} =$ **Combine**$(less,$ add$)$
Step 8. If $(rank \in (c_{less}, c_{mid}])$

$n = c_{mid}$; $l = s_1$; $r = s_2$; $rank = rank - c_{less}$
else
if($rank < c_{less}$)
$r = s_1$; $n = c_{less}$
else
$n = n - (c_{less} + c_{mid})$; $l = s_2$; $rank = rank - (c_{less} + c_{mid})$

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)

sample set. We make the following modification that may help improve the running time of the algorithm in practice. Suppose that the ranks of l_1 and l_2 are both $< k$. Instead of repeating the iteration to find element of rank k among the n elements, we discard all the elements less than l_2 and find the element of rank $k - rank(l_2)$ in the remaining $n - rank(l_2)$ elements. If the ranks of l_1 and l_2 are both $> k$, elements greater than l_1 can be discarded.

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.

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 ϵ increases the overhead due to sorting. A small value of ϵ 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.

As in the randomized median finding algorithm, one iteration of the median finding algorithm takes $O(n_{max}^{(j)} + (\tau + \mu) \log p)$ time. Since $n_{max}^{(j)} = O(\frac{n}{p})$ and $\log \log n$ iterations are required, median finding requires $O(\frac{n}{p} \log \log n + (\tau + \mu) \log p \log \log n)$ time.

As before, we can do load balancing to ensure that $n_{max}^{(j)}$ reduces by half in every iteration. Assuming this and ignoring the cost of load balancing, the running time of median finding reduces to $\sum_{j=0}^{\log \log n - 1} O(\frac{n}{2^j p} + (\tau + \mu) \log p) = O(\frac{n}{p} + (\tau + \mu) \log p \log \log n)$. Even without this load balancing, the running time is expected to be $O(\frac{n}{p} + (\tau + \mu) \log p \log \log n)$.

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 present three algorithms for performing such a load balancing. The algorithms can also be used in other problems that require dynamic redistribution of data and where there is no restriction on the assignment of data to processors.

We use the following notation to describe the algorithms for load balancing: Initially,

Algorithm 5 *Modified order maintaining load balance*

n - Number of total elements
p - Total number of processors labeled from 0 to $p-1$
L_i - List of elements on processor P_i of size n_i
On each processor P_i

Step 0. $n_{avg} = \lceil \frac{n}{p} \rceil$; if $p < n \bmod p$, increment n_{avg}
Step 1. $M =$ **Global_Concat**(n_i)
for $j \leftarrow 0$ to $p-1$

Step 2. $diff[j] = M[j]$ - n_{avg}
Step 3. If $diff[j]$ is positive P_j is labeled as a source. If $diff[j]$ is negative P_j is labeled as a sink.

Step 4. If P_i is a source calculate the prefix sum of the positive $diff[*]$ in an array p_src, else calculate the prefix sums for sinks using negative $diff[*]$ in p_snk.
if(source[P_i])

Step 5. $l_i = |p\_src[i]| - diff[i]$
Step 6. $r_i = |p\_src[i]| - 1$
Step 7. Calculate the range of destination processors $[P_l, P_r]$ using a binary search on p_snk.
Step 8. while($l \leq r$)
Send $[min(r_i, p\_snk[P_l]) - l_i]$ elements to P_l and increment l

if(sink[P_i])

Step 5. $l_i = p\_snk[i] - diff[i]$
Step 6. $r_i = p\_snk[i] - 1$
Step 7. Calculate the range of source processors $[P_l, P_r]$ using a binary search on p_src.
Step 8. while($l \leq r$)
Receive $[min(r_i, p\_src[P_l]) - l_i]$ elements from P_l and increment l

processor P_i contains n_i elements. n is the total number of elements on all the processors, i.e. $n = \sum_{i=0}^{p-1} n_i$. Let $n_{max} = max_{i=0}^{p-1} n_i$. Let $n_{avg} = \lfloor \frac{n}{p} \rfloor$.

4.1 Order Maintaining Load Balance

Suppose that each processor has its set of elements stored in an array. We can view the n elements as if they were globally sorted based on processor and array indices. For any $i < j$, any element in processor P_i appears earlier in this sorted order than any element in processor P_j. The order maintaining load balance algorithm is a parallel prefix based algorithm that preserves this global order of data after load balancing.

The algorithm first performs a Parallel Prefix operation to find the position of the elements it contains in the global order. The objective is to redistribute data such that processor P_i contains the elements with positions $n_{avg}i \ldots n_{avg}(i+1)-1$ in the global order. Using the parallel prefix operation, each processor can figure out the processors to which it should send data and the amount of data to send to each processor. Similarly, each processor can figure out the amount of data it should receive, if any, from each processor. Communication is generated according to this and the data is redistributed.

In our model of computation, the running time of this algorithm only depends on the maximum communication generated/received by a processor. The maximum number of messages sent out by a processor is $\lceil \frac{n_{max}}{n_{avg}} \rceil + 1$ and the maximum number of elements sent is n_{max}. The maximum number of elements received by a processor is n_{avg}. Therefore, the running time is $O(n_{avg} + \tau \frac{n_{max}}{n_{avg}} + \mu n_{max})$.

The order maintaining load balance algorithm may generate much more communication than necessary. For example, consider the case where all processors have n_{avg} elements except that P_0 has one element less and P_{p-1} has one element more than n_{avg}. The optimal strategy is to transfer the one extra element from P_p to P_0. However, this algorithm transfers one element from P_i to P_{i-1} for every $1 \leq i < p-1$, generating $(p-1)$ messages.

Since preserving the order of data is not important for the selection algorithm, the following modification is done to the algorithm: Every processor retains $min\{n_i, n_{avg}\}$ of its original elements. If $n_i > n_{avg}$, the processor has $(n_i - n_{avg})$ elements in excess and is labeled a source. Otherwise, the processor needs $(n_{avg} - n_i)$ elements and is labeled a sink. The excessive elements in the source processors and the number of elements needed by the sink processors are ranked separately using two Parallel Prefix operations. The data is transferred from sources to sinks using a strategy similar to the order maintaining load balance algorithm. This algorithm, which we call modified order maintaining load balance algorithm (modified OMLB), is implemented in [5].

The maximum number of messages sent out by a processor in modified OMLB is $O(p)$ and the maximum number of elements sent is $(n_{max} - n_{avg})$. The maximum number of elements received by a processor is n_{avg}. The worst-case running time is $O(n_{avg} + \tau p + \mu(n_{max} - n_{avg}))$.

4.2 Dimension Exchange Method

The dimension exchange method is a load balancing technique originally proposed for hypercubes [11]. In each iteration of this method, processors are paired to balance the load locally among themselves which eventually leads to global load balance. The algorithm runs in $\log p$ iterations. In iteration i $(0 \leq i < \log p)$, processors that differ in the i^{th} least significant bit position of their id's exchange and balance the load. After iteration i, for any $0 \leq j < \lfloor \frac{p}{2^i} \rfloor$,

Algorithm 6 *Dimension exchange method*

n - Number of total elements
p - Total number of processors labeled from 0 to $p-1$
L_i - List of elements on processor P_i of size n_i
On each processor P_i

for $j \leftarrow 0$ *to* $\log p - 1$

Step 1. $P_l = P_i$ XOR 2^j
Step 2. Exchange the count of elements between $P_i(n_i)$ and $P_l(n_l)$
Step 3. $n_{avg} = \lceil \frac{n_i + n_l}{2} \rceil$
if $(n_i > N_{avg})$
Step 4. **Send** $n_i - n_{avg}$ elements from $L_i[n_{avg}]$ to processor P_l
Step 5. $n_i = n_{avg}$
else
if ($N_l > N_{avg}$)
Step 4. **Receive** $n_l - n_{avg}$ elements from processor P_l at $L_i[n_i]$
Step 5. Increment n_i by $n_l - n_{avg}$

processors $P_{j2^i} \ldots P_{j2^{i+1}-1}$ have the same number of elements.

In each iteration, $\frac{p}{2}$ pairs of processors communicate in parallel. No processor communicates more than $\frac{n_{max}}{2}$ elements in an iteration. Therefore, the running time is $O(\tau \log p + \mu n_{max} \log p)$. However, since 2^j processors hold the maximum number of elements in iteration j, it is likely that either n_{max} is small or far fewer elements than $\frac{n_{max}}{2}$ are communicated. Therefore, the running time in practice is expected to be much better than what is predicated by the worst-case.

4.3 Global Exchange

This algorithm is similar to the modified order maintaining load balance algorithm except that processors with large amounts of data are directly paired with processor with small amounts of data to minimize the number of messages.

As in the modified order maintaining load balance algorithm, every processor retains $min\{n_i, n_{avg}\}$ of its original elements. If $n_i > n_{avg}$, the processor has $(n_i - n_{avg})$ elements in excess and is labeled a source. Otherwise, the processor needs $(n_{avg} - n_i)$ elements and is labeled a sink. All the source processors are sorted in non-increasing order of the number of excess elements each processor holds. Similarly, all the sink processors are sorted in non-increasing order of the number of elements each processor needs. The information on the number of excessive elements in each source processor is collected using a Global

Algorithm 7 *Global Exchange load balance*

n - Number of total elements
p - Total number of processors labeled from 0 to $p-1$
L_i - List of elements on processor P_i of size n_i
On each processor P_i

Step 0. $n_{avg} = \lceil \frac{n}{p} \rceil$; if $p < n \bmod p$, increment n_{avg}
Step 1. M = **Global_Concat**(n_i)
for $j \leftarrow 0$ *to* $p-1$

Step 2. $diff[j] = M[j]$ - n_{avg}
Step 3. If $diff[j]$ is positive P_j is labeled as a source. If $diff[j]$ is negative P_j is labeled as a sink.

Step 4. For $k \in [0, p-1]$ sort $diff[k]$ for sources in descending order maintaining appropriate processor indices. Also sort $diff[k]$ for sinks in ascending order.
Step 5. If P_i is a source calculate the prefix sum of the positive $diff[*]$ in an array p_src, else calculate the prefix sums for sinks using negative $diff[*]$ in p_snk.
Step 6. If P_i is a source calculate the prefix sum of the positive $diff[*]$ in an array p_src, else calculate the prefix sums for sinks using negative $diff[*]$ in p_snk.
if(source[P_i])

Step 7. $l_i = |p\_src[i]| - diff[i]$
Step 8. $r_i = |p\_src[i]| - 1$
Step 9. Calculate the range of destination processors $[P_l, P_r]$ using a binary search on p_snk.
Step 10. while($l \leq r$)

Send $[min(r_i, p\_snk[P_l]) - l_i]$ elements to P_l and increment l

if(sink[P_i])

Step 7. $l_i = p\_snk[i] - diff[i]$
Step 8. $r_i = p\_snk[i] - 1$
Step 9. Calculate the range of source processors $[P_l, P_r]$ using a binary search on p_src.
Step 10. while($l \leq r$)

Receive $[min(r_i, p\_src[P_l]) - l_i]$ elements from P_l and increment l

Selection Algorithm	Run-time
Median of Medians	$O(\frac{n}{p} + \tau \log p \log n + \mu p \log n)$
Bucket-based	$- - - -$
Randomized	$O(\frac{n}{p} + (\tau + \mu) \log p \log n)$
Fast randomized	$O(\frac{n}{p} + (\tau + \mu) \log p \log \log n)$

Table 1: The running times of various selection algorithm assuming but not including the cost of load balancing

Concatenate operation. Each processor locally ranks the excessive elements using a prefix operation according to the order of the processors obtained by the sorting. Another Global Concatenate operation collects the number of elements needed by each sink processor. These elements are then ranked locally by each processor using a prefix operation performed using the ordering of the sink processors obtained by sorting.

Using the results of the prefix operation, each source processor can find the sink processors to which its excessive elements should be sent and the number of element that should be sent to each such processor. The sink processors can similarly compute information on the number of elements to be received from each source processor. The data is transferred from sources to sinks. Since the sources containing large number of excessive elements send data to sinks containing large number of excessive elements, this may reduce the total number of messages sent.

In the worst-case, there may be only one processor containing all the excessive elements and thus the total number of messages sent out by the algorithm is $O(p)$. No processor will send more than $(n_{max} - n_{avg})$ elements of data and the maximum number of elements received by any processor is n_{avg}. The worst-case run time is $O(n_{avg} + \tau p + \mu(n_{max} - n_{avg}))$.

5 Implementation Results

The estimated running times of various selection algorithms are summarized in Table 1 and Table 2. Table 1 shows the estimated running times assuming that each processor contains approximately the same number of elements at the end of each iteration of the selection algorithm. This can be expected to hold for random data even without performing any load balancing and we also observe this experimentally. Table 2 shows the worst-case running times in the absence of load balancing.

We have implemented all the selection algorithms and the load balancing techniques

Selection Algorithm	Run-time
Median of Medians	$O(\frac{n}{p}\log n + \tau \log p \log n + \mu p \log n)$
Bucket-based	$O(\frac{n}{p}(\log\log p + \frac{\log n}{\log p}) + \tau \log p \log n + \mu p \log n)$
Randomized	$O(\frac{n}{p}\log n + (\tau + \mu)\log p \log n)$
Fast randomized	$O(\frac{n}{p}\log\log n + (\tau + \mu)\log p \log\log n)$

Table 2: The worst-case running times of various selection algorithms

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 without any load balancing and with each of the load balancing algorithms described (except for the bucket-based approach which does not use load balancing). We have run all the resulting algorithms on 32k, 64k, 128k, 256k, 512k, 1024k and 2048k numbers using 2, 4, 8, 16, 32, 64 and 128 processors. For each value of the total number of elements, we have run each of the algorithms on two types of inputs - random and sorted. In the random case, $\frac{n}{p}$ elements are randomly generated on each processor. To eliminate peculiar cases while using the random data, 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. In the sorted case, the n numbers are chosen to be the numbers $0 \ldots n-1$, with processor P_i containing the numbers $i\frac{n}{p} \ldots (i+1)\frac{n}{p} - 1$. The sorted input is a close to the worst-case input for the selection algorithms. For example, after the first iteration of a selection algorithm using this input, approximately half of the processors lose all their data while the other half retains all of their data. Without load balancing, the number of active processors is cut down by about half every iteration. The same is true even if modified order maintaining load balance and global exchange load balancing algorithms are used. After every iteration, about half the processors contain zero elements leading to severe load imbalance for the load balancing algorithm to rectify. Only some of the data we have collected is illustrated in order to save space.

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 for which global exchange is used) with 128k, 512k and 2048k numbers is shown in Figure 1. The graphs clearly demonstrate that all four selection algorithms scale well with the number of processors. An immediate observation is that the randomized algorithms are superior to the deterministic algorithms by an order of magnitude. For example, with $n = 2$M and $p = 32$, the median of medians algorithm ran at least 16 times slower and the bucket-based selection algorithm ran at least 9 times slower than either of the randomized algorithms. Such an order of magnitude difference is uniformly observed even using any of the load balancing techniques

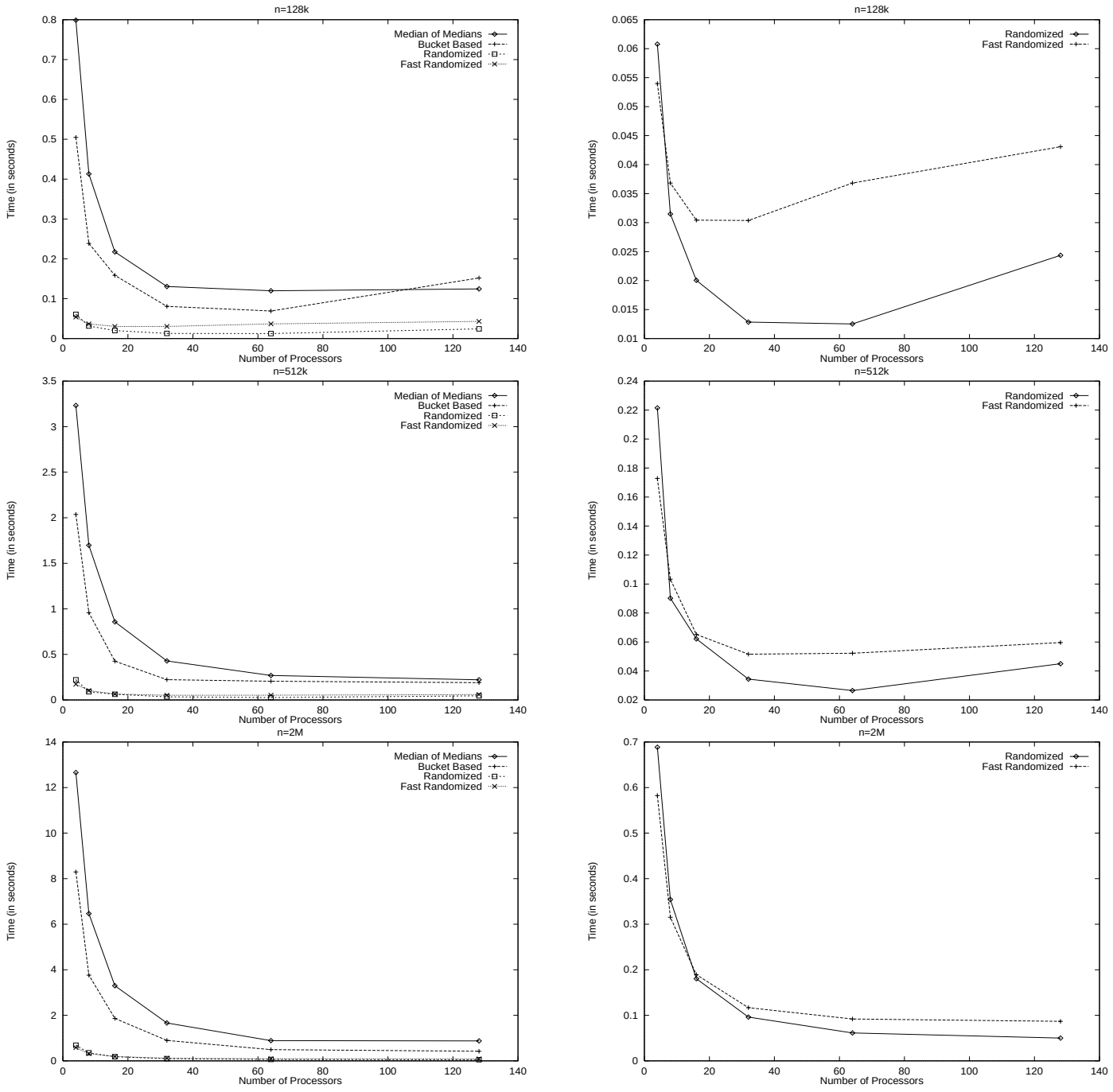

Figure 1: Performance of different selection algorithms without load balancing (except for median of medians selection algorithm for which global exchange is used) on random data sets.

and also in the case of sorted data. This is not surprising since the constants involved in the deterministic algorithms are higher due to recursively finding the estimated median. 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.

In each iteration of the parallel selection algorithm, each processor also performs a local selection algorithm. Thus the algorithm can be split into a parallel part where the processors combine the results of their local selections and a sequential part involving executing the sequential selection locally on each processor. In order to convince ourselves that randomized algorithms are superior in either part, we ran the following hybrid experiment. We ran both the deterministic parallel selection algorithms replacing the sequential selection parts of it by randomized sequential selection. The running time of the hybrid algorithms was in between the deterministic and randomized parallel selection algorithms. We made the following observation: The factor of improvement in randomized parallel selection algorithms over deterministic parallel selection is due to improvements in both the sequential and parallel parts. For large n, much of the improvement is due to the sequential part. For large p, the improvement is due to the parallel part. We conclude that randomized algorithms are faster in practice and drop the deterministic algorithms from further consideration.

To facilitate an easier comparison of the two randomized algorithms, we show their performance separately in Figure 1. Fast randomized selection is asymptotically superior to randomized selection for worst-case data. For random data, the expected running times of randomized and fast randomized algorithms are $O(\frac{n}{p} + (\tau + \mu)\log p \log n)$ and $O(\frac{n}{p} + (\tau + \mu)\log p \log\log n)$, respectively. Consider the effect of increasing n for a fixed p. Initially, the difference in $\log n$ and $\log\log n$ is not significant enough to offset the overhead due to sorting in fast randomized selection and randomized selection performs better. As n is increased, fast randomized selection begins to outperform randomized selection. For large n, both the algorithms converge to the same execution time since the $O(\frac{n}{p})$ computational time dominates. Reversing this point view, we find that for any fixed n, as we increase p, randomized selection will eventually perform better and this can be readily observed in the graphs.

The effect of the various load balancing techniques on the randomized algorithms for random data is shown in Figure 2 and Figure 3. The execution times are consistently better without using any load balancing than using any of the three load balancing techniques. Load balancing for random data almost always had a negative effect on the total execution time and this effect is more pronounced in randomized selection than in fast randomized selection. This is explained by the fact that fast randomized selection has fewer iterations

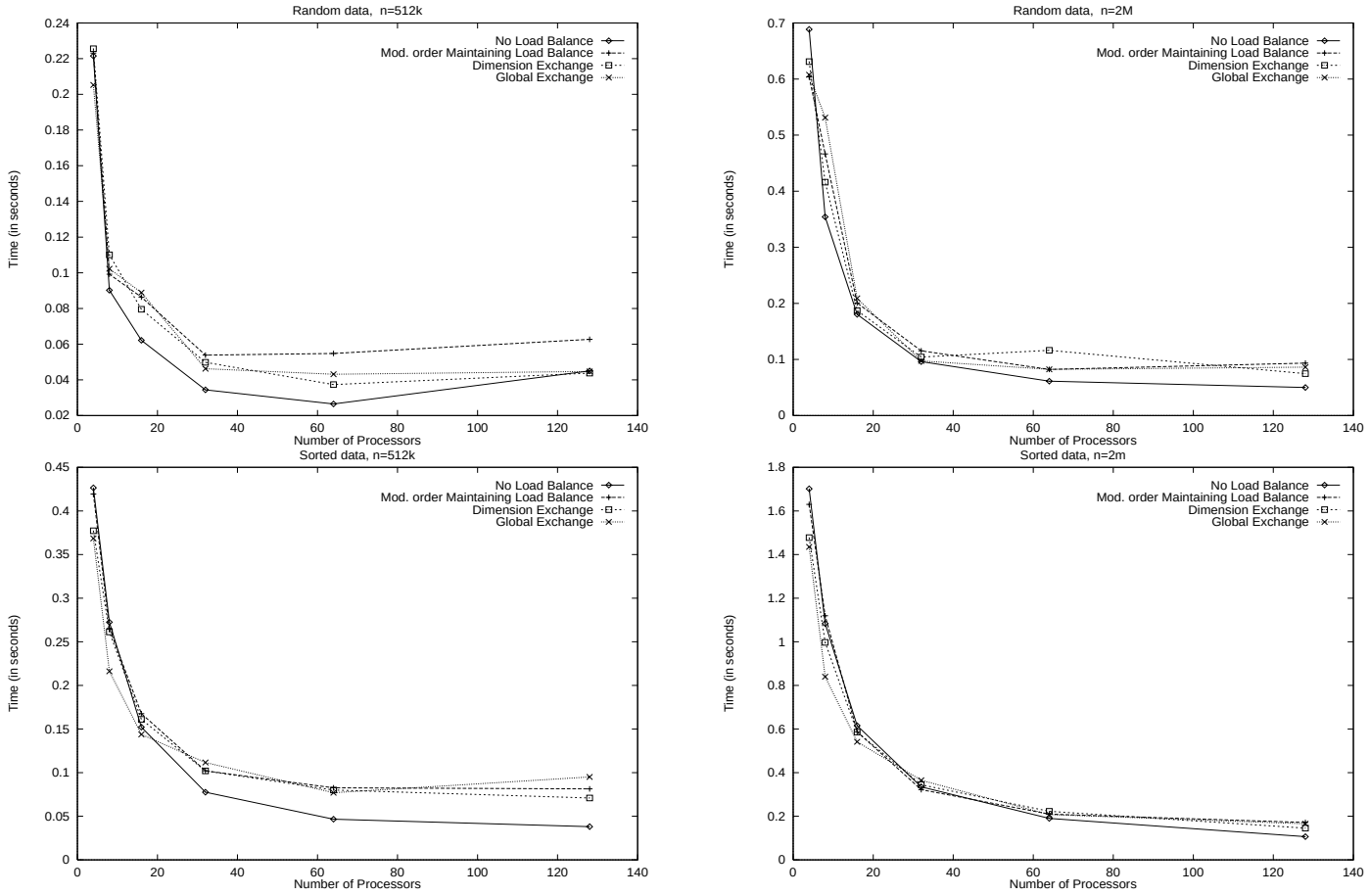

Figure 2: Performance of randomized selection algorithm with different load balancing strategies on random and sorted data sets.

($O(\log \log n)$ vs. $O(\log n)$) and less data in each iteration.

The observation that load balancing has a negative effect on the running time for random data can be easily explained: In load balancing, a processor with more elements sends some of its elements to another processor. The time taken to send the data is justified only if the time taken to process this data in future iterations is more than the time for sending it. Suppose that a process sends m elements to another processor. The processing of this data involves scanning it in each iteration based on an estimated median and discarding part of the data. For random data, it is expected that half the data is discarded in every iteration. Thus, the estimated total time to process this data is $O(m)$. The time for sending the data is $(\tau + \mu m)$, which is also $O(m)$. By observation, the constants involved are such that load balancing is taking more time than the reduction in running time caused by it.

Consider the effect of the various load balancing techniques on the randomized algorithms for sorted data (see Figure 2 and Figure 3). Even in this case, the cost of load balancing more than offset the benefit of it for randomized selection. However, load balancing significantly

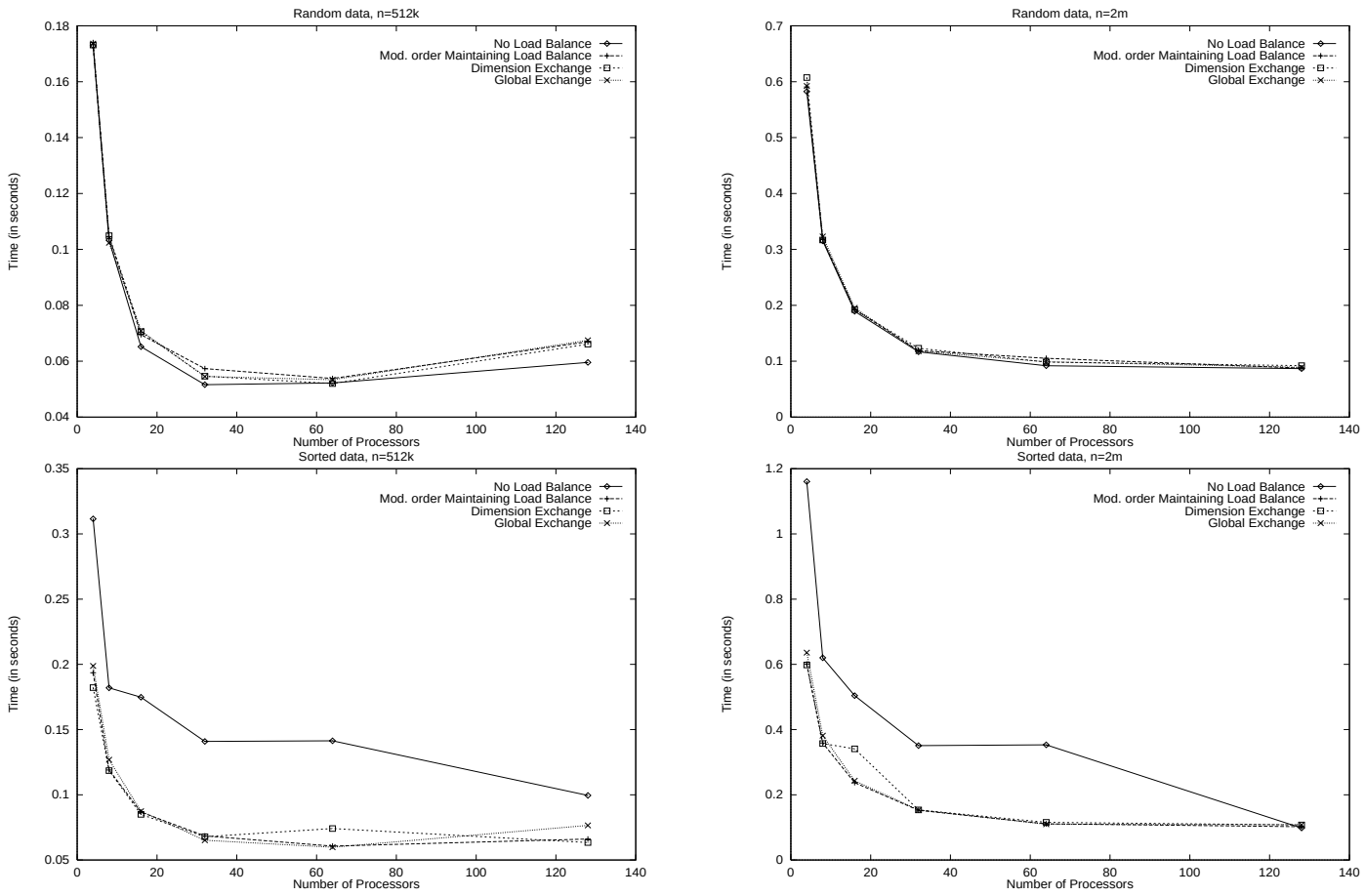

Figure 3: Performance of fast randomized selection algorithm with different load balancing strategies on random and sorted data sets.

improved the performance of fast randomized selection.

In Figure 4, 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 modified 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. We also observe (see Figure 4 and Figure 1) that the fast randomized selection has better comparative advantage over randomized selection for sorted data.

Finally, we consider the time spent in load balancing itself for the randomized algorithms on both random and sorted data (see Figure 5 and Figure 6). For both types of data inputs, fast randomized selection spends much less time than randomized selection in balancing the load. This is reflective of the number of times the load balancing algorithms are utilized ($O(\log \log n)$ vs. $O(\log n)$). Clearly, the cost of load balancing increases with the amount of imbalance and the number of processors. For random data, the overhead due to load balancing is quite tolerable for the range of n and p used in our experiments. For sorted

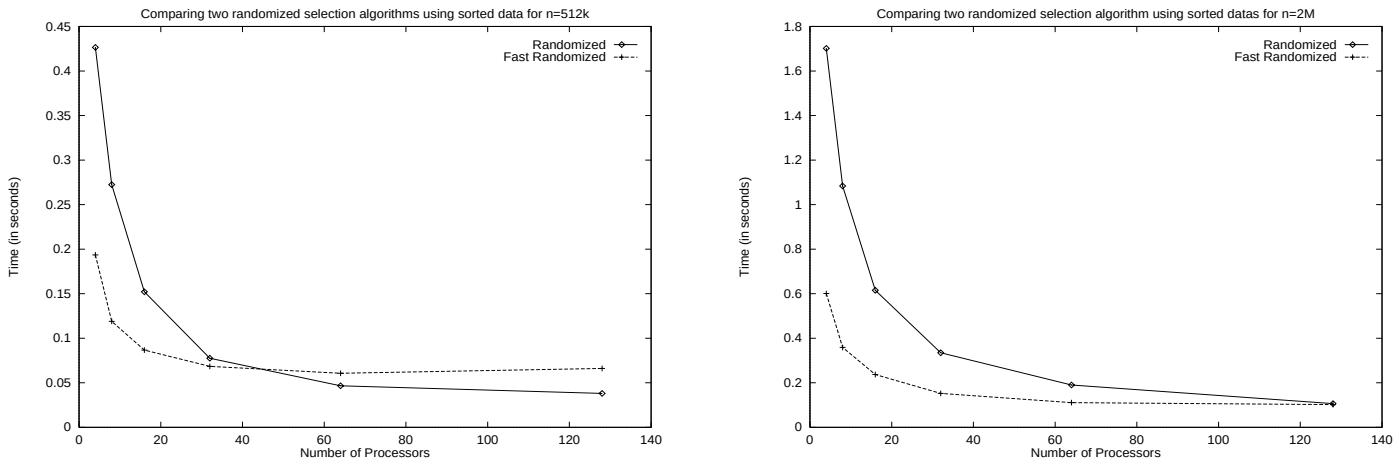

Figure 4: Performance of the two randomized selection algorithms on sorted data sets using the best load balancing strategies for each algorithm – no load balancing for randomized selection and modified order maintaining load balancing for fast randomized selection.

data, a significant fraction of the execution time of randomized selection is spent in load balancing. Load balancing never improved the running time of randomized selection. Fast randomized selection benefited from load balancing for sorted data. The choice of the load balancing algorithm did not make a significant difference in the running time.

Consider the variance in the running times between random and sorted data for both the randomized algorithms. The randomized selection algorithm ran 2 to 2.5 times faster for random data than for sorted data (see Figure 5). Using any of the load balancing strategies, there is very little variance in the running time of fast randomized selection (Figure 6). The algorithm performs equally well on both best and worst-case data.

6 Conclusions

In this paper, we have tried to identify the selection algorithms that are most suited for fast execution on coarse-grained distributed memory parallel computers. After surveying various algorithms, we have identified four algorithms and have described and analyzed them in detail. We also considered three load balancing strategies that can be used for balancing data during the execution of the selection algorithms.

Based on the analysis and experimental results, 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

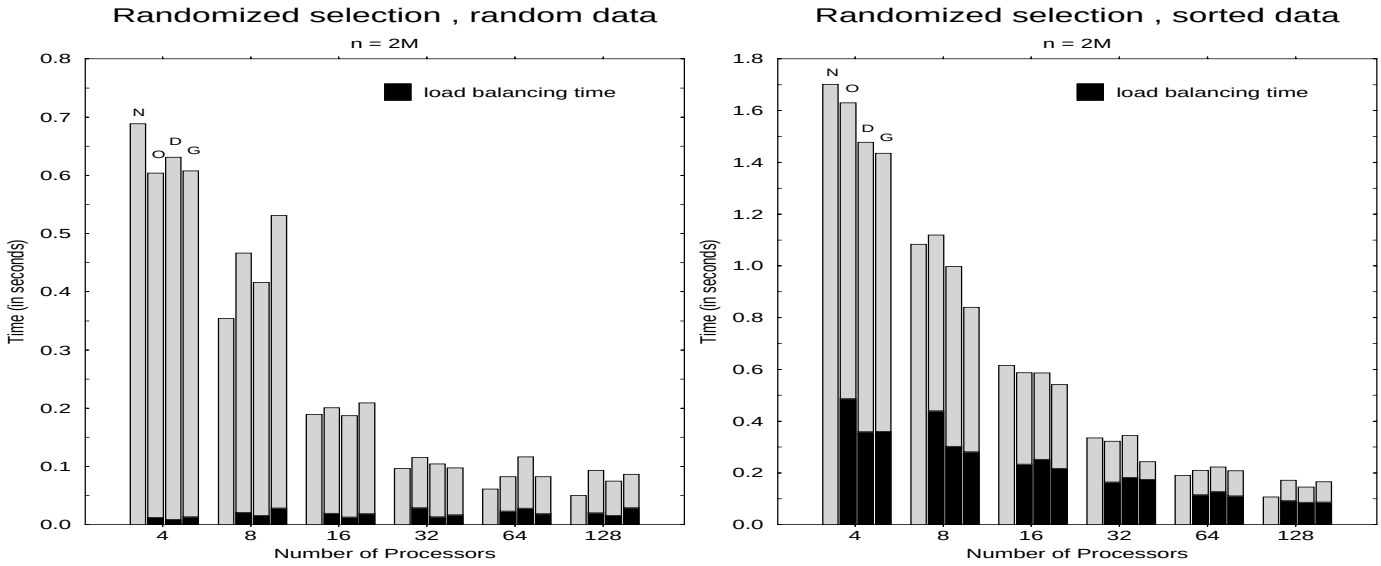

Figure 5: Performance of randomized selection algorithm with different load balancing strategies – No load balancing (N), Order maintaining load balancing (O), Dimension exchange method (D) and Global exchange (G).

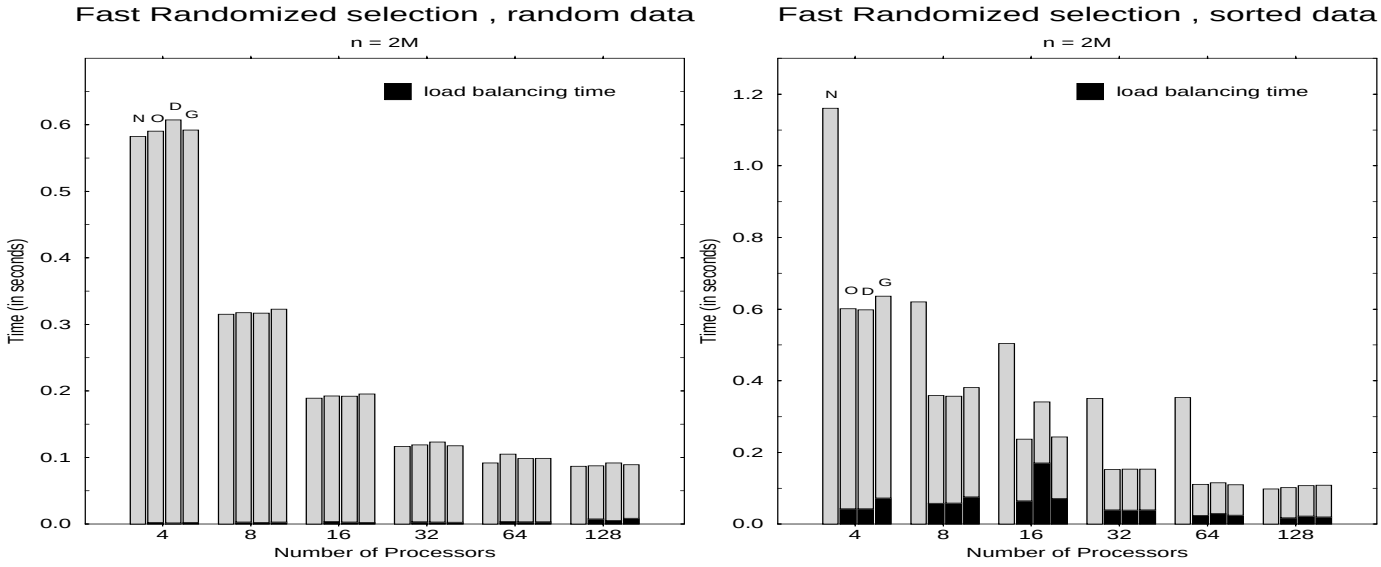

Figure 6: Performance of fast randomized selection algorithm with different load balancing strategies – No load balancing (N), Order maintaining load balancing (O), Dimension exchange method (D) and Global exchange (G).

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. Komlo's, W.L. Steiger and E. Szemere'di, Deterministic selection in $O(\log \log N)$ parallel time, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing* (1986) 188-195.

[2] S.G. Akl, *The design and analysis of parallel algorithms*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[3] S.G. Akl, Parallel selection in $O(\log \log n)$ time using $O(n/\log \log n)$ processors, Technical Report 88-221, Department of Computing and Information Science, Queen's University, Kingston, Ontario, March 1988.

[4] S.G. Akl, An optimal algorithm for parallel selection, *Information Processing Letters, 19(1)* (1984) 47-50.

[5] 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.

[6] Berthom, A. Ferreira, B.M. Maggs, S. Perennes, and C.G. Plaxton, Sorting-based selection algorithms for hypercubic networks, *Proc. 7th International Parallel Processing Symposium* (1993) 89-95.

[7] G.E. Blelloch, Prefix sums and their applications, Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[8] 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.

[9] R. Cole and C.K. Yap, A parallel median algorithm, *Information Processing Letters, 20(3)* (1985) 137-139.

[10] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to algorithms*. MIT Press, Cambridge, MA, 1990.

[11] G. Cybenko, Dynamic load balancing for distributed memory multiprocessors, *Journal of Parallel and Distributed Computing, 7* (1989) 279-301.

[12] R.W. Floyd and R.L. Rivest, Expected time bounds for selection, *Communications of the ACM, 18(3)* (1975) 165-172.

[13] E. Hao, P.D. MacLenzie and Q.F. Stout, Selection on the Reconfigurable Mesh, *Proceedings of the 4^{th} Symposium on the Frontiers of Massively Parallel Computation* (1992) 38-45.

[14] J. J'aJ'a, *An introduction to parallel algorithms*, Addison-Wesley Publishing Company, New York, 1992.

[15] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin Cummings Publishing Company, California, 1994.

[16] C.G. Plaxton, Efficient computation on sparse interconnection networks, Ph.D. Thesis, Department of Computer Science, Stanford University, 1989.

[17] S. Rajasekharan, W. Chen and S. Yooseph, Unifying themes for parallel selection, *Proc. 5^{th} International Symposium on Algorithms and Computation*, Beijing, China, 1994, Springer-Verlag Lecture Notes in CS 834, 92-100.

[18] S. Rajasekharan and J.H. Reif, Derivation of randomized sorting and selection algorithms, *Parallel Algorithm Derivation and Program Transformation*, edited by R. Paige, J.H. Reif and R. Watcher, Kluwer Academic Publishers, 1993, 187-205.

[19] S. Rajasekharan, Randomized parallel selection, *Proc. Symposium on Foundations of Software Technology and Theoretical Computer Science* (1990) 215-224.

[20] 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.

[21] R. Sarnath and X. He, Efficient parallel algorithms for selection and searching on sorted matrices, *Proc. 6^{th} International Parallel Processing Symposium* (1992) 108-111.

[22] A. Schonhage, M.S. Paterson and N. Pippenger, Finding the median, *Journal of Computer and System Sciences, 13* (1976) 184-199.

[23] J. Woo and S. Sahni, Load balancing on a hypercube, *Proc. 5^{th} International Parallel Processing Symposium* (1991) 525-530.