Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection (*Preliminary Draft*)

David A. Bader*
dbader@eng.umd.edu

Joseph JáJá†
joseph@umiacs.umd.edu

Institute for Advanced Computer Studies, and
Department of Electrical Engineering,
University of Maryland, College Park, MD 20742

July 14, 1995

Abstract

A common statistical problem is that of finding the median element in a set of data. This paper presents a fast and portable parallel algorithm for finding the median given a set of elements distributed across a parallel machine. In fact, our algorithm solves the general selection problem that requires the determination of the element of rank i, for an arbitrarily given integer i. Practical algorithms needed by our selection algorithm for the dynamic redistribution of data are also discussed. Our general framework is a single-address space, distributed memory programming model that is enhanced by a set of communication primitives. We use efficient techniques for distributing, coalescing, and load balancing data as well as efficient combinations of task and data parallelism. The algorithms have been coded in Split-C and run on a variety of platforms, including the Thinking Machines CM-5, IBM SP-1 and SP-2, Cray Research T3D, Meiko Scientific CS-2, Intel Paragon, and workstation clusters. Our experimental results illustrate the scalability and efficiency of our algorithms across different platforms and improve upon all the related experimental results known to the authors. More efficient implementations of the communication primitives will likely result in even faster execution times.

Keywords: Parallel Algorithms, Communication Primitives, Median Finding, Selection, Load Balancing, Data Redistribution, Parallel Performance.

1 Problem Overview

Consider the problem of finding the median of a set of n elements that are spread across a p-processor distributed memory machine, where $n \geq p^2$. The median is typically defined as the element that is the

*The support by NASA Graduate Student Researcher Fellowship No. NGT-50951 is gratefully acknowledged.
†Supported in part by NSF grant No. CCR-9103135 and NSF HPCC/GCAG grant No. BIR-9318183.

50^{th} quantile of a set, or the element of rank $\lceil \frac{n}{2} \rceil$ after the data has been sorted in ascending order. A more general problem is that of **selection**; namely, we have to find the element of rank i, for a given parameter i, $1 \leq i \leq n$. Parallel sorting trivially solves the selection problem, but sorting is known to be computationally harder than selection.

Previous parallel algorithms for selection ([9], [23], [31], [25]) and data redistribution ([28], [34]) tend to be network dependent or assume the PRAM model, and thus, are not efficient or portable to current parallel machines. In this paper, we present algorithms that are shown to be scalable and efficient across a number of different platforms.

The organization of this paper is as follows. Section 2 addresses the Block Distributed Memory model for analyzing shared memory style parallel algorithms. The Communication Library Primitive operations which are fundamental to the design of high-level algorithms are introduced in Section 3. A practical algorithm for the dynamic redistribution of data derived from these primitives is given in Section 4. A parallel selection algorithm is described and analyzed in Section 5, together with experimental results on a number of platforms.

2 The Block Distributed Memory Model

We use the Block Distributed Memory (BDM) Model ([26], [27]) as a computation model for developing and analyzing our parallel algorithms on distributed memory machines. This model allows the design of algorithms using a single address space and does not assume any particular interconnection topology. The model captures performance by incorporating a cost measure for interprocessor communication induced by remote memory accesses. The cost measure includes parameters reflecting memory latency, communication bandwidth, and spatial locality. This model allows the initial placement of data and prefetching.

The complexity of parallel algorithms will be evaluated in terms of two measures: the computation time $T_{comp}(n,p)$, and the communication time $T_{comm}(n,p)$. The measure $T_{comp}(n,p)$ refers to the maximum of the local computations performed on any processor as measured on the standard sequential model. The communication time $T_{comm}(n,p)$ refers to the total amount of communications time spent by the overall algorithm in accessing remote data. Using the BDM model, an access operation to a remote location takes $\tau + 1$ time, and l prefetch read operations can be executed in $\tau + l$ time, where τ is the normalized maximum latency of any message sent in the communications network. No processor can send or receive more than one word at a time.

We present several useful communication primitives in [3] and [4] for the transpose (also known as "index" or "all-to-all personalized" communication) and the broadcast data movements. Since these will be important primitives for analyzing our parallel algorithms, a summary of these communication primitives follows.

3 Communication Library Primitives

The following are our communication library primitives which are useful for routing most data movements. Our algorithms will be described as shared-memory programs that make use of these primitives and do not make any assumptions of how these primitives are actually implemented. In our analysis, we use the BDM model and the results of [26] and [27].

The basic data transport is a **read** or **write** operation. The remote read and write typically have both blocking and non-blocking versions. Also, when reading or writing more than one element, bulk data transports are provided with corresponding **bulk_read** and **bulk_write** primitives. The first hierarchy of collective communication primitives are similar to those for the IBM POWERparallel machines [8], the Cray MPP systems [15], standard message passing [29], and communication libraries for shared memory languages on distributed memory machines, such as Split-C [16], and include the following: **bcast**, **reduce**, **combine**, **scatter**, **gather**, **concat**, **transpose**, and **prefix**. A higher level primitive **redist** is described later for dynamic data redistribution.

Note that shared arrays are held in distributed memory across a set of processors. A typical array, $A[r:s]$ contains $s-r+1$ elements, each assigned to a location in a unique processor. Collective communications are defined on **process groups**, namely, the subset of processors which hold elements from array A. For example, the process group is defined to have $p = s-r+1$ processors, logically and consecutively ranked from 0 to $p-1$. In general, nothing is known about the physical layout of A, which is assumed to be arbitrary, i.e. $A[r]$ and $A[r+1]$ might reside on P_a and P_b, for any $a \neq b$. For ease of describing the primitives below, we normalize $A[r:s]$ by relabeling it as $A'[0:p-1]$, where p is defined as $s-r+1$. Note that this is just a change of variable to simplify the discussion, and not a physical remapping of the data.

3.1 Communication Primitive: READ$(A[r][x:x+q-1])$

Given a shared $k \times p$ matrix on a p processor partition, the **READ** primitive is an operation that allows an arbitrary processor to request and receive q elements $(1 \leq q \leq k)$ from a remote location on P_r. Note that many parallel platforms contain both blocking (one-phase) and non-blocking (two-phase) read function calls. In the BDM model, its complexity is defined to be

$$\begin{cases} T_{comm}(n,p) & \leq \quad \tau + q; \\ T_{comp}(n,p) & = \quad O(1). \end{cases} \tag{1}$$

3.2 Communication Primitive: WRITE$(A[r][x:x+q-1])$

The complementary data movement, the **WRITE** primitive, is called when an arbitrary processor writes q elements $(1 \leq q \leq k)$ from a local array to a remote location. Again, many parallel platforms contain both blocking and non-blocking write function calls. The BDM complexity is again given in Eq. (1).

3.3 Communication Primitive: CONCAT$(A[0:p-1])$

Given a shared input array $A[0:p-1]$ on a p processor partition, distributed with one element per processor, the **CONCAT** Communication Library Primitive returns a $p \times p$ array consisting of the rearrangement of data such that each processor holds a local copy of the $1 \times p$ array A. In the BDM model, this **CONCAT** communication algorithm has the following complexity:

$$\begin{cases} T_{comm}(n,p) & \leq & \tau + p - 1; \\ T_{comp}(n,p) & = & \mathrm{O}(1). \end{cases} \tag{2}$$

3.4 Communication Primitive: TRANSPOSE$(A[0:p-1][0:q-1])$

Given a $q \times p$ matrix on a p processor partition, where p divides q, the **TRANSPOSE** Communication Library Primitive consists of rearranging the data in the $q \times p$ array such that the first $\frac{q}{p}$ rows of elements are moved to the first processor, the second $\frac{q}{p}$ rows to the second processor, and so on, with the last $\frac{q}{p}$ rows of the matrix moved to the last processor. This primitive is also known as the **index** operation ([8], [11]). The BDM algorithm and analysis for the **TRANSPOSE** data movement is given in [3] and is similar to that of the LogP model [17]. This **TRANSPOSE** communication algorithm has the following complexity:

$$\begin{cases} T_{comm}(n,p) & \leq & \tau + \left(q - \frac{q}{p}\right); \\ T_{comp}(n,p) & = & \mathrm{O}(q). \end{cases} \tag{3}$$

3.5 Communication Primitive: BCAST$(A[r][x:x+q-1])$

Another useful data movement primitive is **BCAST** broadcasting primitive. An efficient BDM algorithm is given ([3], [26]) which takes q elements ($q \geq p$) on a single processor and broadcasts them to the other $p-1$ processors using just two **TRANSPOSE** Communication Primitives.

An efficient algorithm for broadcasting no greater than p elements from one processor (P_r) to the remaining $p-1$ processors is to perform the **CONCAT** communication primitive, such that processors only prefetch data when it is from processor r. This algorithm is identical in complexity to Eq. (2). On the other hand, this problem can be solved using k-ary balanced tree algorithm [26], in which case the communication would be $T_{comm}(n,p) \leq 2\,(2\tau \log_k p + p)$.

For larger q, a more efficient algorithm to broadcast the q elements from a single processor to p processors is based on the **TRANSPOSE** primitive. Processor r holds the q elements to be broadcast in the first column of matrix A. We perform the **TRANSPOSE**(A) primitive, thus, giving every processor $\frac{q}{p}$ elements. Each processor then locally rearranges the data so that an additional

TRANSPOSE data movement will result in each processor holding a copy of all the q elements in its column of A [26].

The analysis of this **BCAST** algorithm is simple. Since this algorithm just performs two **TRANSPOSE** Communication Primitives, the complexities of the **BCAST** Primitive are

$$\begin{cases} T_{comm}(n,p) & \le & 2\left(\tau + \left(q - \frac{q}{p}\right)\right); \\ T_{comp}(n,p) & = & \mathrm{O}(q). \end{cases} \tag{4}$$

See [3] and [4] for algorithmic details, performance analysis, and empirical results for these communication primitives.

3.6 Communication Primitive: PREFIX$(A[0:p-1], \oplus)$

Given an associative operator $\oplus$ (e.g. $+$, $\times$, min, max, etc.) and a shared input array $A[0:p-1]$ on a p processor partition, distributed with one element per processor, the **PREFIX** Communication Library Primitive coalesces the data such that each processor k contains a single element $PS[k] = A[0] \oplus A[1] \oplus \ldots \oplus A[k]$. Parallel computers can handle this efficiently when the element $PS[k]$ is assumed to reside on processor k [10], and Split-C implements this as a primitive library function. An analysis for this operation on the BDM model is given in [4]. Since these rounds can be realized with an **CONCAT** primitive operation followed by $\mathrm{O}(p)$ local computation of the prefix-sums, the resulting complexity is

$$\begin{cases} T_{comm}(n,p) & \le & \tau + p - 1; \\ T_{comp}(n,p) & = & \mathrm{O}(p). \end{cases} \tag{5}$$

Note that our algorithm can perform a stronger operation for the same complexity; namely, all p prefix-sums values can be made available as local elements on all processors. Thus, each processor k contains $PS[i] = A[0] \oplus A[1] \oplus \ldots \oplus A[i]$, for all $0 \le i \le p-1$. This is equivalent to calling **CONCAT**(**PREFIX**$(A[0:p-1])$).

3.7 Communication Primitive: REDUCE$(A[0:p-1], \oplus)$

The **REDUCE** Communication Primitive takes a shared input array $A[0:p-1]$ and an associative operator $\oplus$, and on a single processor, returns the value of $\sum_{i=0}^{p-1} A[i]$, where $\sum$ uses the associative operation $\oplus$. We implement this primitive by calling **PREFIX** with the array A and the operation $\oplus$. Instead of using all p prefix-sums, only the value of $PS[p-1]$ is returned. The BDM model complexity analysis is identical to Eq. (5).

3.8 Communication Primitive: COMBINE$(A[0:p-1], \oplus)$

As with **REDUCE**, the **COMBINE** Communication Primitive again takes a shared input array $A[0:p-1]$ and an associative operator $\oplus$, and returns another $1 \times p$ shared array, consisting of p

copies of the value of $\sum_{i=0}^{p-1} A[i]$, where $\sum$ uses the associative operation $\oplus$. A simple implementation of this primitive calls **BCAST**($p-1$, **REDUCE**($A[0:p-1], \oplus$)). Another implementation follows from the stronger **PREFIX** primitive. Instead of returning all p prefix-sums, only the value of $PS[p-1]$ is returned on each processor. Thus, the BDM model complexity analysis is identical to Eq. (5).

3.9 Communication Primitive: GATHER($A[0:p-1][0:s-1]$) (SCATTER($A[r][0:n-1]$))

Given an $s \times p$ matrix distributed across a p processor partition, where $q = sp$, the **GATHER** Primitive converts the data layout such that the entire sp elements are held in a $q \times 1$ array local to a single processor. A simple algorithm consists of logically replicating the input data such that there are p copies in contiguous memory, and then calling the **TRANSPOSE** Communication Primitive. Note that the inverse operation to this primitive is that of **SCATTER**, where a single column of q elements of data on one processor is divided into p equal-sized chunks and transposed to fill a $\frac{q}{p} \times p$ distributed layout. The analysis for these two primitives is given in Eq. (3).

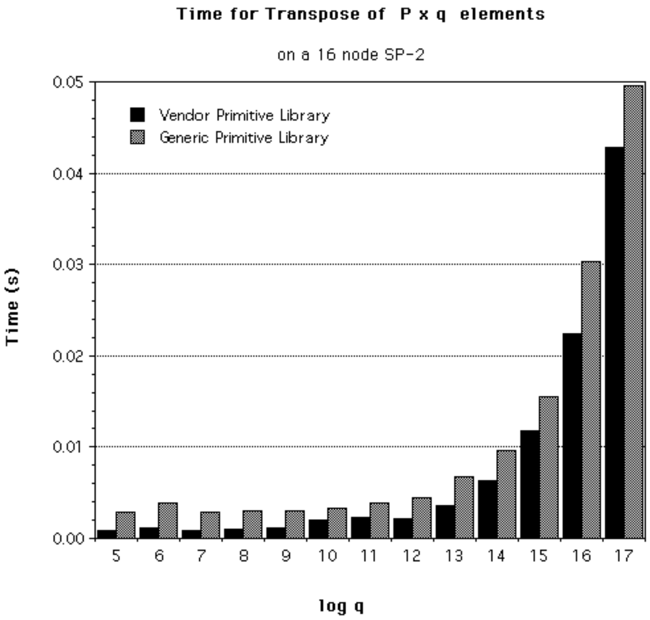
3.10 Implementation Issues

The implementation of the communication primitives presented in this section can be achieved by library code which need use only the basic **READ** and **WRITE** primitives. However, parallel machine vendors, realizing the importance of fast primitives ([8], [11], [29], and [15]), provide their own library calls which benefit from knowledge of and access to lower level machine specifics and optimizations.

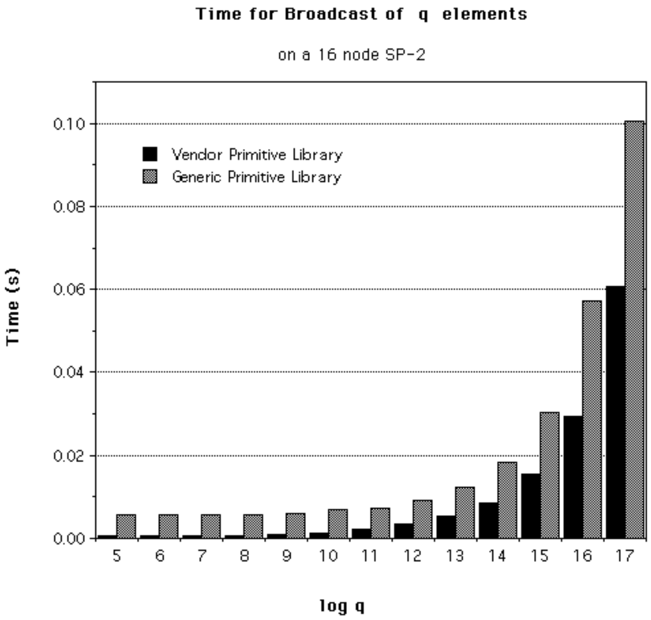
Communication primitives are considered to be a black box, where the implementation is unimportant from the user's perspective, as long as the primitives produce the correct results. Figure 1 provides an example using the **TRANSPOSE** and **BROADCAST** primitives on the IBM SP-2. Note that the "Vendor" primitive library corresponds to a primitive function implemented directly on top of the respective collective communication library function provided by IBM. The "Generic" primitive library uses our generic (and portable) implementation which call only the **READ** and **WRITE** primitives. Note that for both implementation methods, and for both primitives, execution time is similar, and making use of a vendor's library can improve performance.

4 Dynamic Redistribution of Data

The technique of dynamically redistributing data such that all processors have a uniform workload is an essential primitive to many irregular problems, such as computational adaptive graph (grid) problems ([30], [19], [13]) including finite element calculations, molecular dynamics [24], particle dynamics [18], plasma particle-in-cell [20], raytraced volume rendering [22], region growing and computer vision [33], and statistical physics [7], and, as we will show, the selection problem. The running time of



TRANSPOSE



BROADCAST

Figure 1: Performance of Communication Primitives

these parallel algorithms is categorized by the maximum running time of any of the p processors' subproblems. Equalizing the amount of work assigned to each processor is an attempt at minimizing the maximum single processor running time, and thus, reducing the overall execution time. Here, the input is distributed across p processors with a distribution that is irregular and not known a priori. We present two methods for the dynamic redistribution of data which remap the data such that no processor contains more than the average number of data elements. The first method is similar to a method presented in ([26], [27]), and only a brief sketch will be given. The second method, which is shown to be superior, will be presented in greater detail.

4.1 Dynamic Data Redistribution: Method A

A simple method for dynamic data redistribution ranks each element in order across the p processors, and assigns each set of q consecutively labeled elements to a processor, where $q = \left\lceil \frac{n}{p} \right\rceil$. Note that when p does not divide n evenly, the last processor will receive less than q elements. We refer to this as **Method A**.

Figure 2 shows a dynamic data redistribution example for **Method A**. This is a simple example for 8 processors and 63 elements, with an arbitrary initial distribution of $N = [10, 3, 2, 20, 0, 14, 6, 8]$. Here, $q_j = \left\lceil \frac{63}{8} \right\rceil = 8$, for $0 \le j \le 6$, while $q_7 = 7$, since P_7 receives the remainder of elements when p does not divide the total number of elements evenly.

An algorithm for **Method A** first calls the $\mathbf{CONCAT}(N[0:p-1])$ communication primitive and

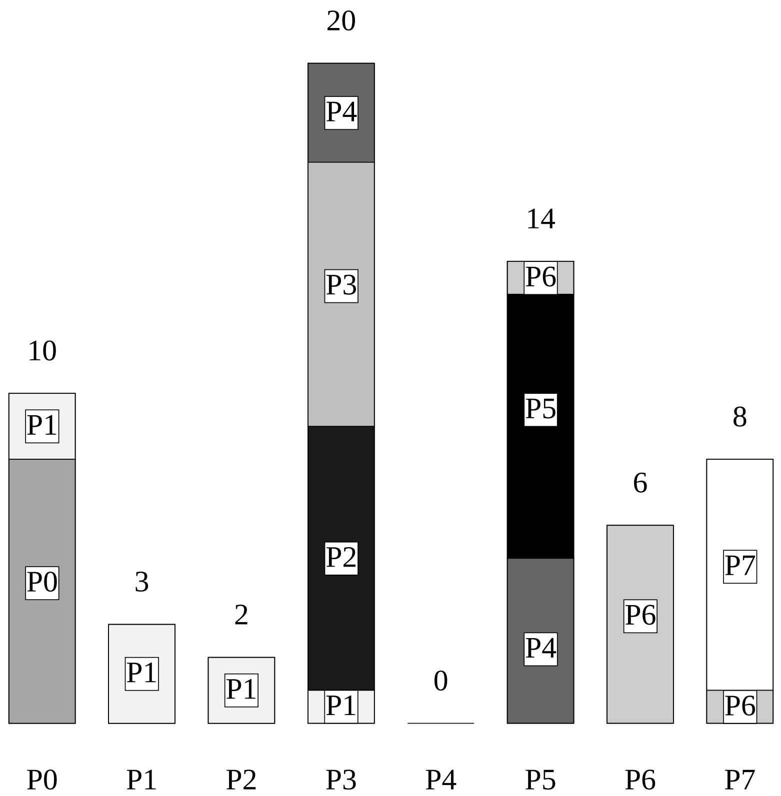


Figure 2: Example of Dynamic Data Redistribution (Method A) with $p = 8$ and $n = 63$

assigns it to array N', a $p \times p$ shared array. Another $p \times p$ shared array of prefix-sums of the values from N, say PS, is derived from N' by simple local running sum calculations. Thus, every processor contains local copies of all prefix-sums. Suppose elements are logically ranked in consecutive order from 1 to n. In the final layout, processor i will hold elements ranked from $qi+1$ to $q(i+1)$, inclusively. Using the prefix-sum information, each processor easily determines where these elements are located and issues **READ** primitives for the respective remote locations to fill the $\left\lceil \frac{n}{p} \right\rceil \times p$ distributed output array.

The analysis for the dynamic data redistribution algorithm using the BDM model is as follows. The **CONCAT** primitive requires communication $T_{comm}(n,p) \le \tau + p - 1$ and $T_{comp}(n,p) = \mathrm{O}(1)$ (Eq. (2)). The local prefix-sum calculation requires $\mathrm{O}(p)$. Determining the location of elements to be read using the prefix-sums has computational complexity of $T_{comp}(n,p) = \log p$. Assume that the maximum number of elements initially on a processor is m, i.e., $m = \max_i\{N[i]\}$. The **READ** primitive for actually issuing the remote read requests uses $T_{comm}(n,p) \le \tau + \max\left\{\frac{n}{p} + 1, m\right\}$ and $T_{comp}(n,p) = \mathrm{O}\left(\frac{n}{p} + m\right)$ since each processor fetches at most $\left\lceil \frac{n}{p} \right\rceil$ elements, but in the worst case, a processor is the source of m fetched elements. Since these requests are pipelined, only a single latency

τ is incurred. Since $m \geq \frac{n}{p}$, the dynamic data redistribution algorithm has the following complexity:

$$\begin{cases} T_{comm}(n,p) & \leq \quad 2\tau + \max_i\{N[i]\} + p; \\ T_{comp}(n,p) & = \quad \mathrm{O}(\max_i\{N[i]\}). \end{cases} \tag{6}$$

Note that the input distribution N for dynamic data redistribution can range from already balanced data ($N[i] = m, \forall i$) to the case where all data is located on a single processor ($N[i] = N, i = i'; N[i] = 0, \forall i \neq i'$). For a large class of irregular problems such that data are distributed with a certain class of distributions, it has been shown that the distribution is typically closer to the first scenario, ($N[i] \approx m, \forall i$) [28].

4.2 Dynamic Data Redistribution: Method B

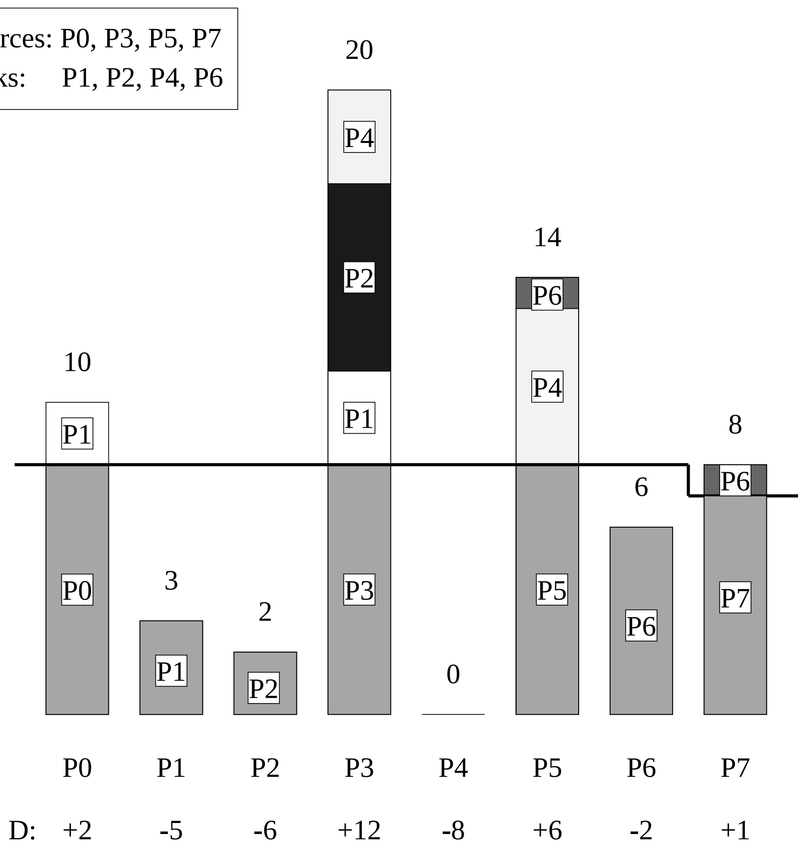


Figure 3: Example of Dynamic Data Redistribution (Method B) with $p = 8$ and $n = 63$

A more efficient dynamic data redistribution algorithm, here referred to as **Method B**, makes use of the fact that a processor initially filled with at least q elements should not need to receive any more elements, but instead, should send its excess to other processors with less than q elements. There are pathological cases for which **Method A** essentially moves all the data, whereas **Method B** only moves a small fraction. For example, if P_0 contains no elements, and P_1 through P_{p-2} each have q elements, with the remaining $2q$ elements held by the last processor, **Method A** will left shift all the data by one processor. However, **Method B** substantially reduces the communication traffic by taking only the q extra elements from P_{p-1} and sending them to P_0.

Dynamic data redistribution **Method B** calculates the differential D_j of the number of elements on processor P_j to the balanced level of q. If D_j is positive, P_j becomes a **source**; and conversely, if D_j is negative, P_j becomes a **sink**. The group of processors labeled as sources will have their excess elements ranked consecutively, while the processors labeled as sinks similarly will have their holes ranked. Since the number of elements above the threshold of q equals the number of holes below the threshold, there is a one-to-one mapping of data which is used to send data from a source to the respective holes held by sinks.

In addition to reduced communication, **Method B** performs data remapping **in-place**, without the need for a secondary array of elements used to receive data, as in **Method A**. Thus, **Method B** also has reduced memory requirements.

Figure 3 shows the same data redistribution example for **Method B**. The heavy line drawn horizontally across the elements represents the threshold q below which sinks have holes and sources contain excess elements. Note that P_{p-1} again holds the remainder of elements when p does not divide the total number of elements evenly.

The SPMD algorithm for **Method B** is described below. The following is run on processor j:

Algorithm 1 *Parallel Dynamic Data Redistribution Algorithm - Method B*
Shared Memory Model Algorithm.

Input:

$\{ j \}$ is my processor number;
$\{ p \}$ is the total number of processors, labeled from 0 to $p-1$;
$\{ A \}$ is the $M \times p$ input array of elements;
$\{ N \}$ is the $1 \times p$ input array of n_j's;

begin

1. $N' = \mathbf{CONCAT}(N)$;
2. Locally **calculate** the sum $n = \sum_{i=0}^{p-1} N'[j][i]$;
3. **Set** q_k, the equalized number of elements on P_k, equal to $\left\lceil \frac{n}{p} \right\rceil$, for $0 \le k \le p-2$;
   Set $q_{p-1} = n - (q_0 * (p-1))$; P_{p-1} receives the remainder of elements when p does not evenly divide n;
4. **Set** $D[k] = N'[j][k] - q_k$, for $0 \le k \le p-1$; This is the differential of elements on P_k;
5. **If** $D[k] > 0$ **then** $\mathrm{SRC}[k] = 1$ **else** $\mathrm{SRC}[k] = 0$, for $0 \le k \le p-1$;
6. **If** $D[k] < 0$ **then** $\mathrm{SNK}[k] = 1$ **else** $\mathrm{SNK}[k] = 0$, for $0 \le k \le p-1$;
7. **For all** $\{k | \mathrm{SRC}[k]\}$,
   Set SRC_RANK[k] equal to the prefix sum of the corresponding $D[k]$ values;
   This ranks the excess elements;
8. **For all** $\{k | \mathrm{SNK}[k]\}$,
   Set SNK_RANK[k] equal to the prefix sum of the corresponding $-D[k]$ values;
   This ranks the holes for elements;

9. **If** SRC[j] **then**
   9.1 **Set** l_j = SRC_RANK[j] - D[j] + 1; the rank of my first element;
   9.2 **Set** r_j = SRC_RANK[j]; the rank of my last element;
   9.3 **Set** $s_j = min\,\{\alpha|\text{SNK}[\alpha] \wedge (l_j \leq \text{SNK\_RANK}[\alpha])\}$;
      the label of the processor holding the hole with rank l_j;
   9.4 **WRITE** $min(\text{SNK\_RANK}[s_j], r_j)$ excess elements from P_j to P_{s_j},
      offset in $A[s_j][\star]$ by $N'[j][s_j] + (l_j - (\text{SNK\_RANK}[s_j] + D[s_j] + 1))$;
   9.5 **If** P_j still contains excess elements **then**
      9.5.1 **Set** $t_j = min\,\{\alpha|\text{SNK}[\alpha] \wedge (r_j \leq \text{SNK\_RANK}[\alpha])\}$;
         the label of the processor holding element with rank r_j;
      9.5.2 **If** $t_j > s_j + 1$, **then WRITE** excess elements to all holes in A in
         processors $s_j + 1, \ldots, t_j - 1$;
      9.5.3 **WRITE** the remaining excess elements to P_{t_j}, offset in $A[t_j][\star]$ by $N'[j][t_j]$.
10. **Update** $N[j]$.

end

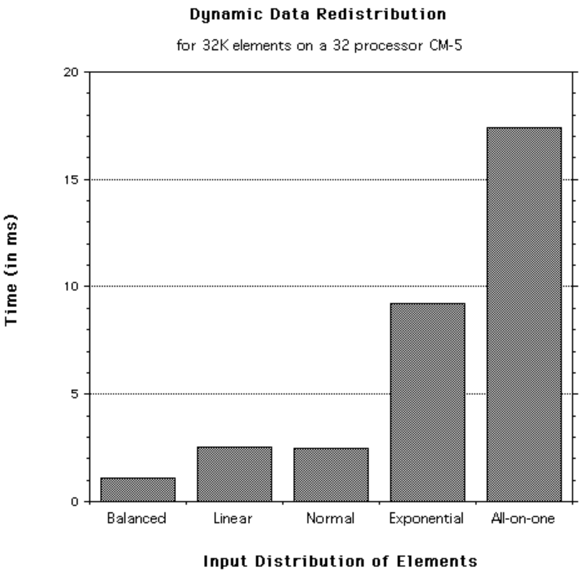
The analysis for **Method B** of the parallel dynamic data redistribution algorithm is identical to that of **Method A**, and is given in Eq. (6). Note that both methods have theoretically similar complexity results, but **Method B** is superior for the reasons stated earlier.

Figure 4 shows the running time of **Method B** for dynamic data redistribution. The top left-hand plate contains results from the CM-5, the top right-hand from the SP-2. The bottom plate contains results from the Cray T3D. In the five experiments, on the 32 processors CM-5, the total number of elements n is is $32K$. On the SP-2, the 8 node partition has $n = 32K$ elements, while the 16 node partition has results using both $n = 32K$ and $64K$ elements. The T3D experiment also uses 16 nodes and a total number of elements $n = 32K$ and $64K$. Let j represent the processor label, for $0 \leq j \leq p-1$. Then the five input distributions are defined as

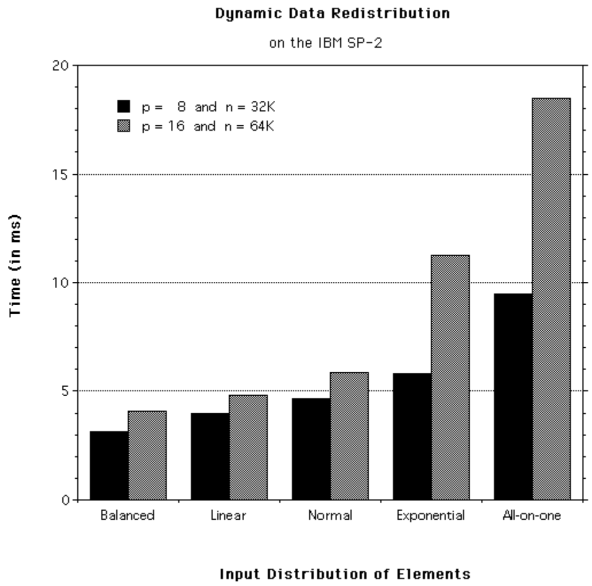
- **Balanced:** Each processor initially holds $\frac{n}{p}$ elements and hence $m = \frac{n}{p}$;
- **Linear:** Each processor initially holds $j\frac{2n}{p(p-1)}$ elements and hence $m = 2\frac{n}{p}$;
- **Normal:** Elements are distributed in a Gaussian curve [1] and hence $m \approx 2.4\frac{n}{p}$ for $p \geq 8$;
- **Exponential:** P_j contains $\frac{n}{2^{j+1}}$ elements, for $j \neq p-1$, and P_{p-1} contains $\frac{n}{2^{p-1}}$ elements and hence $m = \frac{n}{2}$;
- **All-on-one:** An arbitrary processor contains all n elements and hence $m = n$.

The complexity stated in Eq. (6) indicates that the amount of local computation depends only on m (linearly) while the amount of communication increases with both parameters m and p. In particular, for fixed p and a specific machine, we expect the total execution time to increase linearly with m. The results shown in Figure 4 confirm this latter observation.

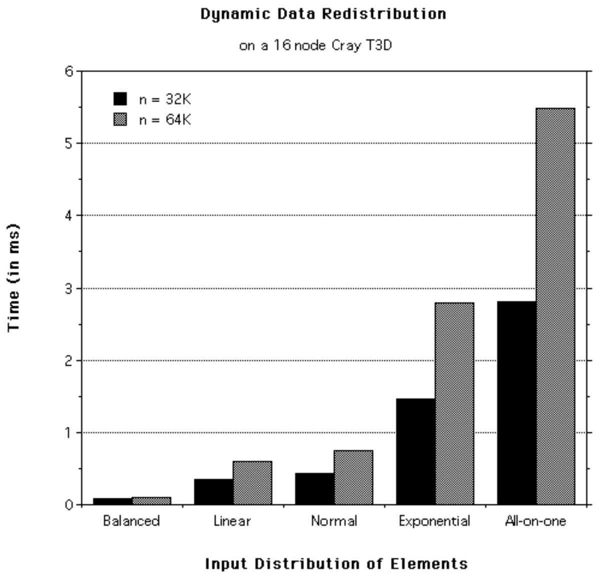
[1] We sample a mean zero, s.d. one, Gaussian curve at the center of p intervals equally spaced along $[-3, 3]$. The sample values are normalized to sum to n by multiplying each by $\frac{n}{\text{sum of the } p \text{ samples}}$. The value of m can be verified empirically.



TMC CM-5



IBM SP-2

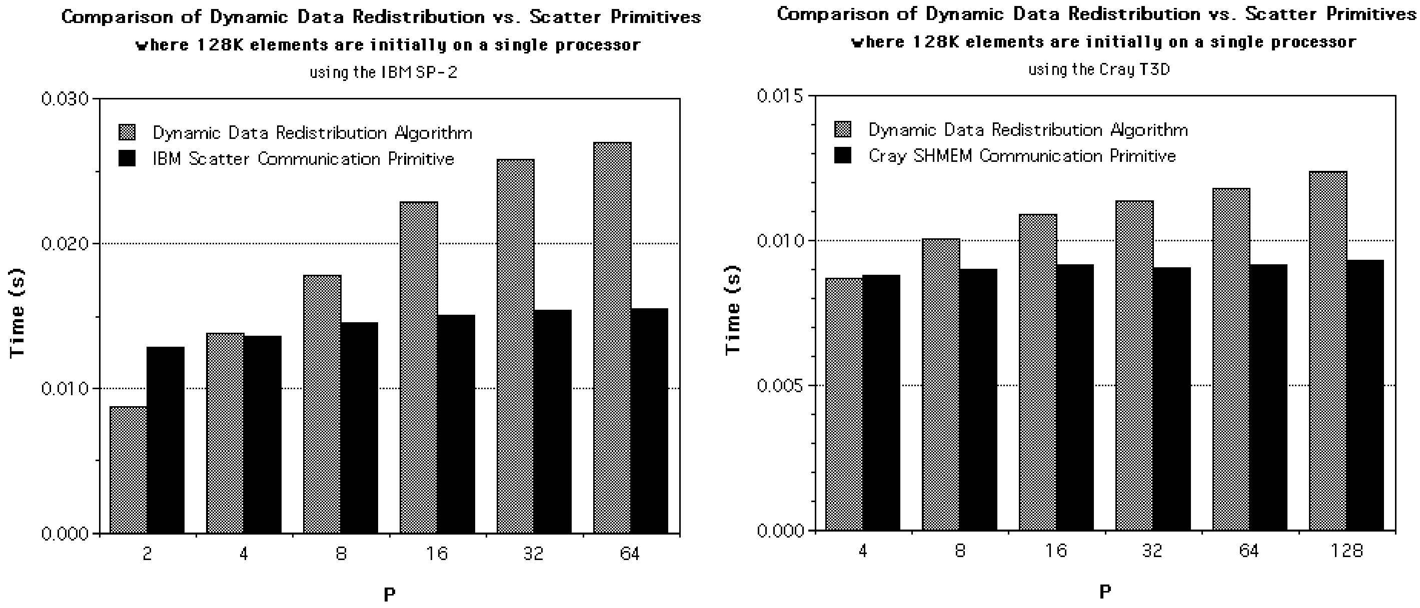


Cray T3D

Figure 4: Dynamic Data Redistribution Algorithms - Method B. The complexity of our algorithm is essentially linear in $m = \max_i\{N[i]\}$

Note that for the **All-on-one** input distribution, the dynamic data redistribution results in the same loading as would calling a **scatter** primitive. In Figure 5 we compare the dynamic data redistribution algorithm performance with that of directly calling a **scatter** IBM communication primitive on the IBM SP-2, and calling **SHMEM** primitives on the Cray T3D. In this example, we have used from 2 to 64 wide nodes of the SP-2 and 4 to 128 nodes of the T3D. Note that the performance of our portable redistribution code is close to the low-level vendor supplied communication primitive for the scatter operation. As anticipated by the complexity of our algorithm stated in Eq. (6), the communication overhead increases with p.

Using this dynamic data redistribution algorithm, which we call **REDIST**, we can now describe the parallel selection algorithm.



IBM SP-2 Cray T3D

Figure 5: Comparison of REDIST vs. Scatter Primitives

5 Parallel Selection - Overview

The selection algorithm makes no initial assumptions about the number of elements held by each processor, nor the distribution of values on a single processor or across the p processors. We define n_j to be the number of elements initially on processor j, for $0 \leq j \leq p-1$, and hence the total number n of elements is $n = \sum_{j=0}^{p-1} n_j$.

The input is a shared memory array of elements $A[0 : p-1][0 : M-1]$, and $N[0 : p-1]$, where $N[j]$ represents n_j, the number of elements stored in $A[j][\star]$, and the selection index i. Note that the median finding algorithm is a special case of the selection problem where i is equal to $\lceil \frac{n}{2} \rceil$. The output is the element from A with rank i.

The parallel selection algorithm is motivated by similar sequential ([14], [32]) and parallel ([1], [25]) algorithms. We use recursion, where at each stage, a "good" element from the collection is chosen to split the input into two partitions, one consisting of all elements less than or equal to the splitter and the second consisting of the remaining elements. Suppose there are t elements in the lower partition. If the value of the selection index i is less than or equal to t, we recurse on that lower partition with the same index. Otherwise, we recurse on the higher partition looking for index $i' = i - t$.

The choice of a good splitter is as follows. Each processor finds the median of its local elements, and the median of these p medians is chosen.

Since no assumptions are made about the initial distribution of counts or values of elements before calling the parallel selection algorithm, the input data can be heavily skewed among the processors. We use a dynamic redistribution technique which tries to equalize the amount of work assigned to each processor.

5.1 Parallel Selection - Implementation and Analysis

The parallel algorithm for selection can now be presented, and makes use of the Dynamic Data Redistribution algorithm given in Section 4. The following is run on processor j:

Algorithm 2 *Parallel Selection Algorithm*

Shared Memory Model Algorithm.

Input:

{ j } is my processor number;
{ p } is the total number of processors, labeled from 0 to $p-1$;
{ A } is the $M \times p$ input array of elements;
{ N } is the $1 \times p$ input array of n_j's;

begin

1. **If** $n < p^2$ **then**
   {
   1.1 $A' =$ **GATHER**(A);
   1.2 Processor 0 **calls a sequential selection algorithm** to find x, the i^{th} value of A'.
   1.3 Result $=$ **BCAST**(x).
   }
2. **REDIST** (A, N, p);
3. **Radixsort** local elements $A[j][0:N[j]-1]$, and **find** the local median;
4. $B =$ **GATHER** of the p median elements, distributed one per processor;
5. Processor 0 **calculates** the median of the medians m, and
   5.1 $x =$ **BCAST**(m);
6. Each processor j **finds** the position k, where $k = max\{l | A[l,j] \leq x\}$,
   using the binary search technique, and **sets** $T[j] = k$;
7. $t =$ **COMBINE**$(T, +)$;

This returns the sum $t = \sum_{j=0}^{p-1} T[j]$, i.e. the number of elements on the low side of the partition;

8. **If** $i \leq t$, **then** $N[j] = k$ and the selection algorithm is called recursively on the first k elements held in A on each processor.
   Otherwise, $i > t$, and selection is called recursively on the last $N[j] - k$ elements held in A on each processor with the selection index $i - k$.

end

The analysis of the parallel selection algorithm is as follows. For $n < p^2$, in step 1, we solve the problem sequentially in linear time. For larger n, dynamic data redistribution algorithm is called in step 2 to ensure that there are $\lceil \frac{n}{p} \rceil$ elements on processors 0 through $p-2$, and processor $p-1$ holds the remaining $n - (p-1)\lceil \frac{n}{p} \rceil$ elements. At least half of the medians found in step 3 are less than or equal to the splitter. Thus, at least half of the p groups contribute at least $\lceil \frac{n}{2p} \rceil$ elements that are less than the splitter, except for the last group and the group containing the splitter element. Therefore, the total number of elements less than or equal to the splitter is at least

$$\left\lceil \frac{n}{2p} \right\rceil \left(\left\lceil \frac{1}{2} p \right\rceil - 2 \right) \geq \frac{n}{4} - \frac{n}{p}$$

Similarly, the number of elements that are greater than the splitter is at least $\frac{n}{4} - \frac{n}{p}$. Thus, in the worst case, the selection algorithm is called recursively on at most

$$n - \left\lceil \frac{n}{4} - \frac{n}{p} \right\rceil = \frac{3}{4} n + \frac{n}{p}$$

elements.

Using the complexity of the communication primitives as stated in Section 3, it is easy to to derive the recurrence equations for the parallel complexity of our algorithm. Solving these recurrences yields the following complexity:

$$\begin{cases} T_{comm}(n,p) & \leq & O\Big((\tau + p)\log \frac{n}{p^2} + m\Big), \; n \geq p^2; \\ T_{comp}(n,p) & = & O\Big(\frac{n}{p} + m\Big), \end{cases} \tag{7}$$

where m is defined in Eq. (6) to be $\max_j\{N[j]\}$, the maximum number of elements initially on any of the processors. For fixed p, the communication time increases linearly with m and logarithmically with n, while the computation time grows linearly with both m and n.

The running time of the median algorithm on the TMC CM-5 using both methods of dynamic data redistribution is given in Figure 6. Similar results are given in Figure 7 for the IBM SP-2. In all data sets, initial data is balanced.

5.2 Data Sets

The input sets are defined as follows. If the set's tag ends with 8, 16, 32, 64, or 128, there are initially 8192, 16384, 32768, 65536, or 131072 elements per processor, respectively. The values of these elements

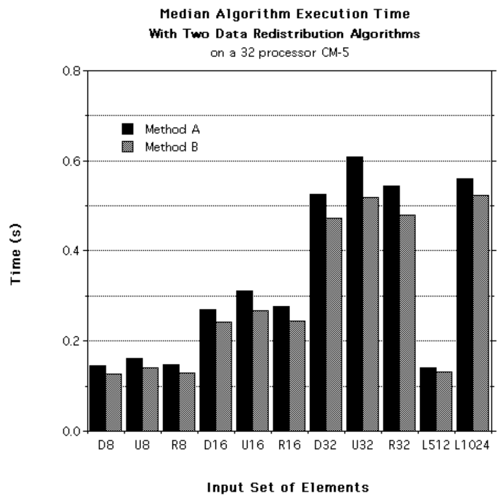


Figure 6: Performance of Median Algorithm

are chosen by the method represented by the first letter. If the total number of elements per processor is q, and the processor is labeled j, for $0 \leq j \leq p-1$, then

- **D: Duplicate**. Each processor holds values $[0, q-1]$;
- **U: Unique**. Each processor holds values $[jq, (j+1)q-1]$;
- **R: Random**. Each processor holds uniformly random values in the range $[0, 2^{31}-1]$.

The last two input sets correspond to an intermediate problem set from a computer vision algorithm for segmenting images [4]. Set $L512$ (derived from band 5 of a 512×512 Landsat TM image) contains a total of 2^{18} elements, which is the same size as the input sets ending with tag 8 on a 32 processor machine. Set $L1024$, with a total of 2^{20} elements, is derived from a similar 1024×1024 image, and has the same number of elements as an input set ending with tag 32 on a 32 processor machine.

On the SP-2, results given in Figure 7 are only for **Method B**, with each timing bar broken into two partitions showing the portion of the total running time spent performing data redistribution versus the remaining selection time. As these empirical data show, dynamic data redistribution is only a small fraction of the total running time, which implies that the data is fairly balanced after each iteration. Also, in every case, **Method B** outperforms **Method A**.

We benchmark our selection algorithm in Table I. The input for this problem, taken from the NAS Parallel Benchmark for Integer Sorting [5], is 2^{23} integers in the range $[0, 2^{19})$, spread out evenly across the processors. Each key is the average of four consecutive uniformly distributed pseudo-random numbers generated by the following recurrence:

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

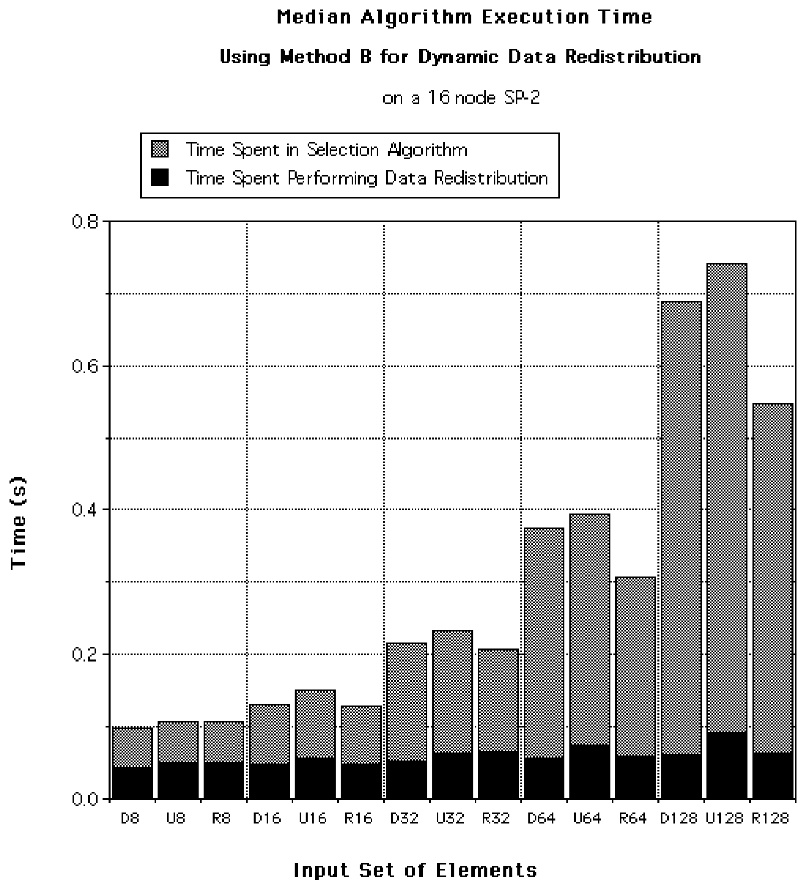
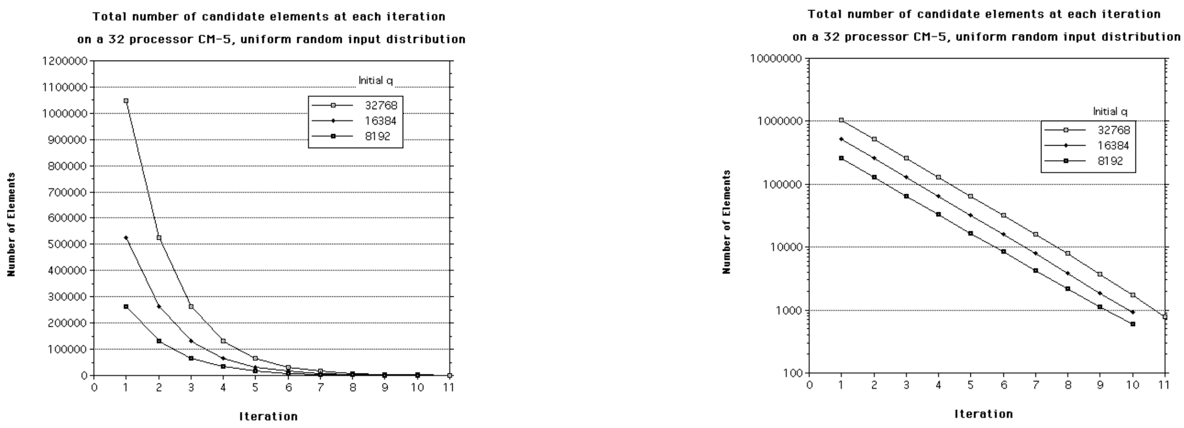


Figure 7: Performance of Median Algorithm on the SP-2

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

The empirical results presented in Table I clearly show that the selection algorithm is scalable with respect to machine size, since doubling the number of processors solves the problem in about half the time. This is consistent with the BDM analysis given in Eq. (7). For $n = 2^{23}$ and machine sizes typically in the order of tens or hundreds of processors, computation dominates the selection algorithm, and execution time scales as $\frac{1}{p}$. (For verification, the median of the NAS input set is 262198.) Our code for selection, written in the high-level parallel language of Split-C, is ported to the parallel machines with absolutely no modifications to the source code. Even without machine-specific (low-level) code optimizations that are typically needed for superior parallel performance, we have an algorithm which performs extremely well across a variety of current parallel machines such as the Cray T3D, IBM SP-2, TMC CM-5, and Meiko CS-2.

Next we compare our selection algorithm to that of the trivial method of selection by parallel integer sorting on the TMC CM-5. As shown in Table II, our high-level selection algorithm beats the



linear scale

log scale

Figure 8: Number of candidates per iteration

fastest sorting results for the NAS input that are known to the authors. Note that the algorithm in [6] is machine-specific and does not actually result in a sorted list.

Figure 8 shows that the parallel selection algorithm for $R8$, $R16$, and $R32$, reduces the candidate elements by approximately one-half during each successive iteration. In this plot, $p = 32$; thus, when the data sets shrinks to a size less than p^2, i.e. smaller than 1024, a sequential algorithm is employed to solve the corresponding selection problem.

6 Acknowledgements

We would like to thank the CASTLE/Split-C group at UC Berkeley, especially the help and encouragement from David Culler, Arvind Krishnamurthy, and Lok Tin Liu. Computational support on UC Berkeley's 64-processor TMC CM-5 was provided by NSF Infrastructure Grant number CDA-8722788. We also thank Toby Harness and the Numerical Aerodynamic Simulation Systems Division of NASA's Ames Research Center for use of their 160-node IBM SP-2-WN.

Also, Klaus Schauser, Oscar Ibarra, and David Probert of the University of California, Santa Barbara, provided access to the UCSB 64-node Meiko CS-2. The Meiko CS-2 Computing Facility was acquired through NSF CISE Infrastructure Grant number CDA-9218202, with support from the College of Engineering and the UCSB Office of Research, for research in parallel computing.

Arvind Krishnamurthy provided additional help with his port of Split-C to the Cray Research T3D [2]. The Jet Propulsion Lab/Caltech 256-node Cray T3D Supercomputer used in this investigation was provided by funding from the NASA Offices of Mission to Planet Earth, Aeronautics, and Space Science. Use of the University of Alaska - Arctic Region Supercomputing Center's 128-node Cray T3D was supported by a grant from the Strategic Environmental Research and Development Program under the sponsorship of the U.S. Army Corps of Engineers, Waterways Experiment Station. The

Machine	PE's	BDM Selection Algorithm
IBM-SP2-TN2	4	4.88
	8	2.40
	16	1.17
IBM-SP2-WN	4	4.05
	8	1.98
	16	1.01
	32	0.571
	64	0.367
Cray T3D	4	7.05
	8	3.55
	16	1.81
	32	0.929
	64	0.483
	128	0.275
Meiko CS-2	16	3.03
	32	1.55
TMC CM-5	16	5.57
	32	2.77
	64	1.68

Table I: Execution Times for the High-Level BDM Selection (in seconds) on the NAS IS input set

Researchers	Time (in seconds)	Notes
Bader & JáJá	2.77	BDM Selection
Dusseau [21]	7.67	Radix Sort
TMC [6]	4.31	Ranking without permuting the data

Table II: Execution Time for Selection on a 32-processor CM-5 on the NAS IS input set

content of this paper does not necessarily reflect the position or the policy of the government and no official endorsement should be inferred.

We also acknowledge William Carlson and Jesse Draper from the Center for Computing Science (formerly Supercomputing Research Center) for writing the parallel compiler AC (version 2.6) [12] on which the T3D port of Split-C has been based.

We would like to acknowledge the use of a 16-node IBM SP-2-TN2, which was provided by an IBM Shared University award and an NSF Research Infrastructure Initiative Grant No. CDA9401151.

The discussions with David Helman and Simon Hawkin proved helpful, and we greatly appreciate their suggestions which improved this research.

Please see `http://www.umiacs.umd.edu/~dbader` for additional performance information.

References

[1] S.G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.

[2] R.H. Arpaci, D.E. Culler, A. Krishnamurthy, S.G. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In ACM Press, editor, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 320–331, Santa Margherita Ligure, Italy, June 1995.

[3] D. A. Bader and J. JáJá. Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study. Technical Report CS-TR-3384 and UMIACS-TR-94-133, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, December 1994. To be presented at the Fifth ACM SIGPLAN Symposium of Principles and Practice of Parallel Programming, Santa Barbara, CA, July 1995.

[4] D. A. Bader, J. JáJá, D. Harwood, and L.S. Davis. Parallel Algorithms for Image Enhancement and Segmentation by Region Growing with an Experimental Study. Technical Report CS-TR-3449 and UMIACS-TR-95-44, Institute for Advanced Computer Studies (UMIACS), University of Maryland, College Park, MD, May 1995. Submitted to *Journal of Supercomputing*.

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

[6] D.H. Bailey, E. Barszcz, L. Dagum, and H.D. Simon. NAS Parallel Benchmark Results 10-94. Report NAS-94-001, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Moffett Field, CA, October 1994.

[7] C.F. Baillie and P.D. Coddington. Cluster Identification Algorithms for Spin Models - Sequential and Parallel. *Concurrency: Practice and Experience*, 3(2):129–144, 1991.

[8] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis, and M. Snir. CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 6:154–164, 1995.

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

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

[11] J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby. Efficient Algorithms for All-to-All Communications in Multi-Port Message-Passing Systems. In *6th Annual ACM Symposium on Parallel Algorithms and Architectures*, volume 6, pages 298–309, Cape May, NJ, June 1994. ACM Press.

[12] W.W. Carlson and J.M. Draper. AC for the T3D. Technical Report SRC-TR-95-141, Supercomputing Research Center, Bowie, MD, February 1995.

[13] A. Choudhary, G. Fox, S. Ranka, S. Hiranandani, K. Kennedy, C. Koelbel, and J. Saltz. Software Support for Irregular and Loosely Synchronous Problems. *International Journal of Computing Systems in Engineering*, 3(1-4), 1992.

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

[15] Cray Research, Inc. *SHMEM Technical Note for C*, October 1994. Revision 2.3.

[16] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, S. Luna, T. von Eicken, and K. Yelick. *Introduction to Split-C.* Computer Science Division - EECS, University of California, Berkeley, version 1.0 edition, March 6, 1994.

[17] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[18] L. Dagum. Three-Dimensional Direct Particle Simulation on the Connection Machine. RNR Technical Report RNR-91-022, NASA Ames, NAS Division, August 1991.

[19] J. De Keyser and D. Roose. Load Balacing Data Parallel Programs on Distributed Memory Computers. *Parallel Computing*, 19:1199–1219, 1993.

[20] K. Dincer. Particle-in-cell simulation codes in High Performance Fortran. Report SCCS-663, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY, November 1994.

[21] A.C. Dusseau. Modeling Parallel Sorts with LogP on the CM-5. Technical Report UCB//CSD-94-829, Computer Science Division, University of California, Berkeley, 1994.

[22] S. Goil and S. Ranka. Dynamic Load Balancing for Raytraced Volume Rendering on Distributed Memory Machines. Report SCCS-693, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY, February 1995.

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

[24] Y.-S. Hwang, R. Das, J. Saltz, B. Brooks, and M. Hodoscek. Parallelizing Molecular Dynamics Programs for Distributed Memory Machines: An Application of the CHAOS Runtime Support

Library. Technical Report CS-TR-3374 and UMIACS-TR-94-125, Department of Computer Science and UMIACS, Univ. of Maryland, 1994.

[25] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, New York, 1992.

[26] J. JáJá and K.W. Ryu. The Block Distributed Memory Model. Technical Report CS-TR-3207, Computer Science Department, University of Maryland, College Park, January 1994.

[27] J.F. JáJá and K.W. Ryu. The Block Distributed Memory Model for Shared Memory Multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 752–756, Cancún, Mexico, April 1994. (Extended Abstract).

[28] K. Mehrotra, S. Ranka, and J.-C. Wang. A Probabilistic Analysis of a Locality Maintaining Load Balancing Algorithm. In *Proceedings of the 7th International Parallel Processing Symposium*, pages 369–373, Newport Beach, CA, April 1993. IEEE Computer Society Press.

[29] Message Passing Interface Forum. A Message Passing Interface Standard. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 1994.

[30] C.-W. Ou and S. Ranka. Parallel Remapping Algorithms for Adaptive Problems. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 367–374, McLean, VA, February 1995. IEEE Computer Society Press.

[31] R. Sarnath and X. He. Efficient parallel algorithms for selection and searching on sorted matrices. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 108–111, Beverly Hills, CA, March 1992. IEEE Computer Society Press.

[32] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, MA, 1988.

[33] C. Weems, E. Riseman, A. Hanson, and A. Rosenfeld. The DARPA Image Understanding Benchmark for Parallel Computers. *Journal of Parallel and Distributed Computing*, 11:1–24, 1991.

[34] J. Woo and S. Sahni. Load Balancing on a Hypercube. In *Proceedings of the 5th International Parallel Processing Symposium*, pages 525–530, Anaheim, CA, April 1991. IEEE Computer Society Press.