Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection (Extended Abstract)

David A. Bader* Joseph JáJá†

Institute for Advanced Computer Studies, and
Department of Electrical Engineering,
University of Maryland, College Park, MD 20742
E-mail: {dbader, joseph}@umiacs.umd.edu

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 distributed memory programming model 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.*

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 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 (e.g., [10, 20, 28, 22]) 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.

2. The Block Distributed Memory Model

We use the Block Distributed Memory (BDM) Model ([23, 24]) as a computation model for developing and analyzing our parallel algorithms on distributed memory machines. Each of our hardware platforms can be viewed as a collection of powerful processors connected by a communication network that can be modeled as a complete graph on which communication is subject to the restrictions imposed by the latency and the bandwidth properties of the network. We view a parallel algorithm as a sequence of local computations interleaved with communication steps, and we allow computation and communication to overlap. 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 communication time $T_{comm}(n,p)$ refers to the total amount of communications time spent by the overall algorithm in accessing remote data. The transfer of a block consisting of m contiguous words between two processors, assuming no congestion, takes $\tau + \sigma m$ time, where τ is the latency of the network and σ is the time per word at which a processor can inject or receive data from the network. In addition to the basic **read** and **write** primitives, we assume

*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.

1063-7133/96 $5.00 © 1996 IEEE
Proceedings of IPPS '96

the existence of a collection of collective communication primitives that include **concat, transpose, prefix, reduce, combine, gather,** and **scatter** [2, 3, 4, 5]. A brief description of some of the primitives used by our algorithms are as follows. The **transpose** primitive is an all-to-all personalized communication in which each processor has to send a unique block of data to every processor, and all the blocks are of the same size. The **bcast** primitive is called to broadcast a block of data from a single source to all the remaining processors. When an array is distributed among the processors with a single element per processor, the **concat** collective communication primitive creates a local copy of this array on each processor, and the **combine** primitive (along with an associative operator) provides each processor with a local copy of the reduction of the distributed array. The primitives **gather** and **scatter** are companion primitives whereby **scatter** divides a single array residing on a processor into equal-sized blocks which are then distributed to the remaining processors, and **gather** coalesces these blocks residing on the different processors into a single array on one processor. The cost of each collective communication primitive will be modeled by $\tau + \sigma \max(m, p)$, where m is the maximum amount of data transmitted or received by a processor. Our cost measure can be justified by using our earlier work [23, 24, 2, 3, 4]. Using this cost model, we can evaluate the communication time $T_{comm}(n, p)$ of an algorithm as a function of the input size n, the number of processors p, and the parameters τ and σ.

We define the computation time $T_{comp}(n, p)$ as the maximum time it takes a processor to perform all the local computation steps. In general, the overall performance $T_{comp}(n, p) + T_{comm}(n, p)$ involves a tradeoff between $T_{comm}(n, p)$ and $T_{comp}(n, p)$. Our aim is to develop parallel algorithms that achieve $T_{comp}(n, p) = O\left(\frac{T_{seq}}{p}\right)$ such that $T_{comm}(n, p)$ is minimum, where T_{seq} is the complexity of the best sequential algorithm. Such optimization has worked very well for the problems we have looked at, but other optimization criteria are possible. The important point to notice is that, in addition to scalability, our optimization criterion requires that the parallel algorithm be an efficient sequential algorithm (i.e., the total number of operations of the parallel algorithm is of the same order as T_{seq}).

2.1. Implementation Issues

The implementation of the collective communication primitives presented in detail in [4] and listed above can be achieved by library code which need use only the basic **read** and **write** primitives. While we have developed our own portable implementation of the primitives, parallel machine vendors, realizing the importance of fast primitives ([9, 11, 26, 14]), have started to provide their own library calls which benefit from knowledge of and access to lower level machine specifics and optimizations.

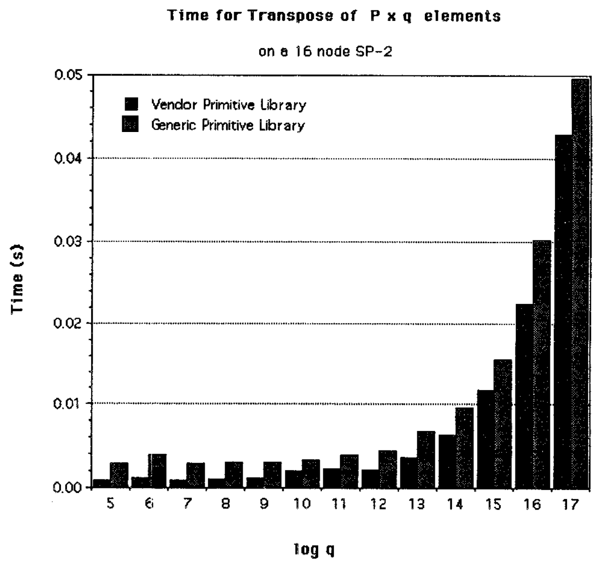


Figure 1. Performance of the transpose Communication Primitive

For our purposes, 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** primitive 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, execution time is similar, and making use of a vendor's library can improve performance.

3. Dynamic Redistribution of Data

The technique of dynamically redistributing data such that each processor has a uniform workload is an essential operation in many irregular problems, such as computational adaptive graph (grid) problems ([27, 16, 12]) including finite element calculations, molecular dynamics [21], particle dynamics [15], plasma particle-in-cell [17], raytraced volume rendering [19], region growing and computer vision [30], and statistical physics [8]. 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 ([23, 24]), and only a brief sketch will be given. The second method, which is shown to be superior, will be presented in greater detail.

3.1. Dynamic Data Redistribution: Method A

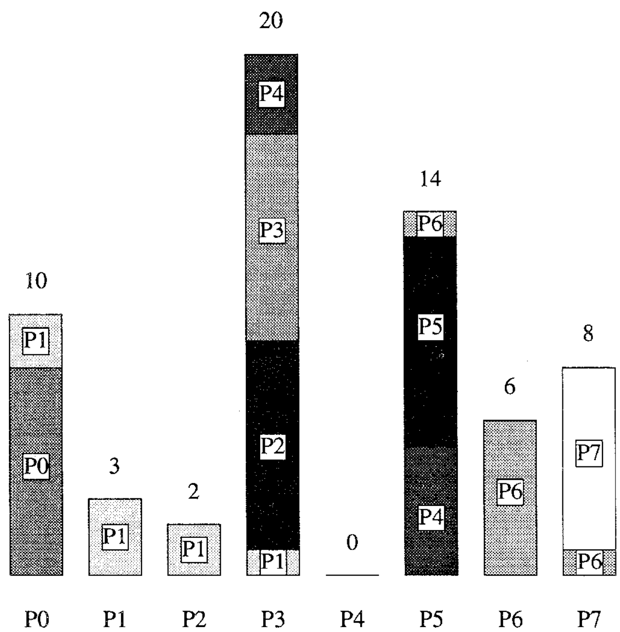


Figure 2. Example of Dynamic Data Redistribution (Method A) with $p = 8$ and $n = 63$

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, $q_j = \lceil \frac{63}{8} \rceil = 8$, for $0 \leq j \leq 6$, 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 **concat** communication primitive and 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 simply from N' by 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 shows that [4]

$$\begin{cases} T_{comm}(n,p) & \leq & 2\tau + \max_i\{N[i]\} + p; \\ T_{comp}(n,p) & = & O(\max_i\{N[i]\}). \end{cases} \qquad (1)$$

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, ($N[i] \approx m, \forall i$) [25].

3.2. Dynamic Data Redistribution: Method B

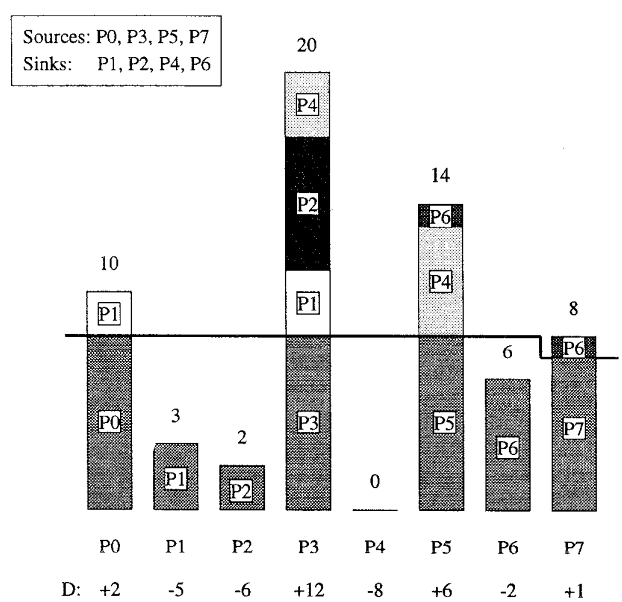


Figure 3. Example of Dynamic Data Redistribution (Method B) with $p = 8$ and $n = 63$

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 above which 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*

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. $N' =$ **concat**(N);
2. Locally **calculate** the sum $n = \sum_{i=0}^{p-1} N'[j][i]$;
3. **Set** $q_k = \left\lceil \frac{n}{p} \right\rceil$, for $0 \le k \le p-2$; and $q_{p-1} = n - (q_0 * (p-1))$; ($P_{p-1}$ 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** SRC$[k] = 1$ **else** SRC$[k] = 0$, for $0 \le k \le p-1$;
6. **If** $D[k] < 0$ **then** SNK$[k] = 1$ **else** SNK$[k] = 0$, for $0 \le k \le p-1$;
7. **For all** $\{k|\text{SRC}[k]\}$, **Set** SRC_RANK$[k]$ equal to the prefix sum of the corresponding $D[k]$ values; (This ranks the excess elements;)
8. **For all** $\{k|\text{SNK}[k]\}$, **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** $s_j = min \{\alpha|\text{SNK}[\alpha] \wedge (l_j \le \text{SNK\_RANK}[\alpha])\}$; (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** $t_j = min \{\alpha|\text{SNK}[\alpha] \wedge (r_j \le \text{SNK\_RANK}[\alpha])\}$; (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. (1). Note that both methods have theoretically similar complexity results, but **Method B** is superior in practice for the reasons stated earlier.

Figure 4 shows the running time of **Method B** for dynamic data redistribution. The top plate contains results from the SP-2, and the bottom from the Cray T3D. In the five experiments, on the SP-2, the 8 node partition contains $n = 32K$ elements, and the 16 node partition contains $n = 64K$ 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 $0 \le j \le p-1$. Then the five input distributions are defined as follows.

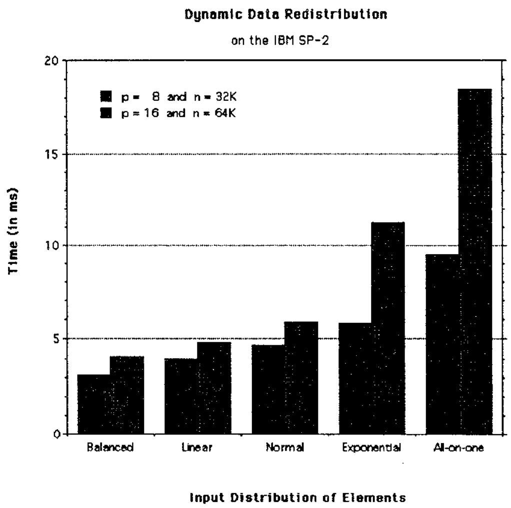
- **Balanced:** Each processor initially holds $\frac{n}{p}$ elements and hence $m = \frac{n}{p}$;

- **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 $m = n$.

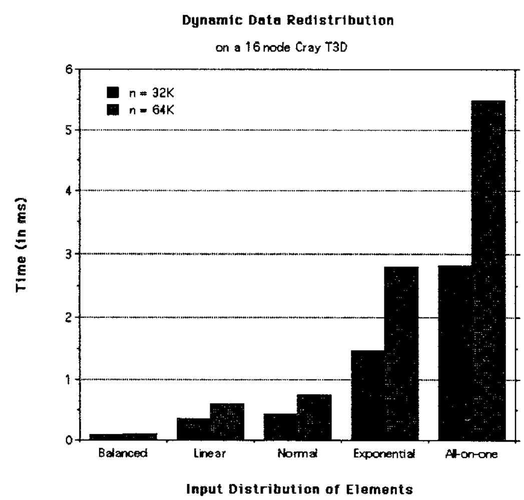
The complexity stated in Eq. (1) indicates that the amount of local computation depends only on m (linearly) while the amount of communication increases with both parameters m and p. In particular, for fixed p and a specific machine, we expect the total execution time to increase linearly with m. The results shown in Figure 4 confirm this latter observation.

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. (1), 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

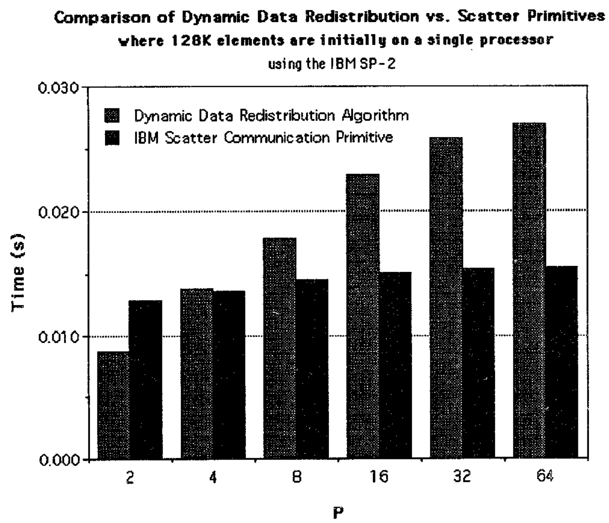
Figure 4. Dynamic Data Redistribution Algorithms - Method B. The complexity of our algorithm is essentially linear in $m = \max_i\{N[i]\}$.

4. Parallel Selection - Overview

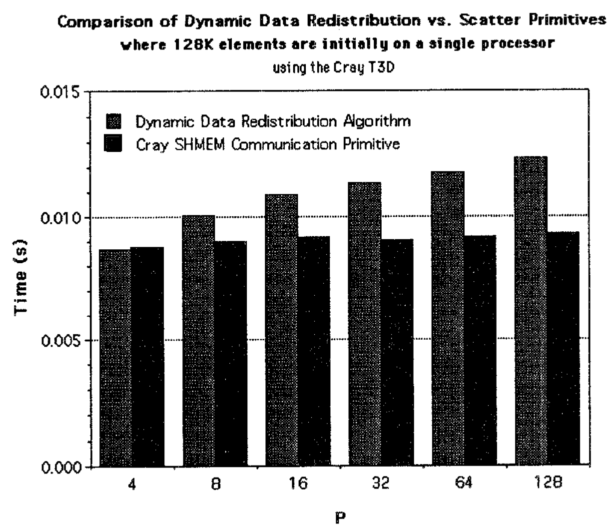
The selection algorithm makes no initial assumptions about the number of elements held by each processor, nor the distribution of values on a single processor or across the p processors. We define n_j to be the number of elements initially on processor j, for $0 \leq j \leq p-1$, and hence the total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

[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.



IBM SP-2



Cray T3D

Figure 5. Comparison of redist vs. scatter Primitives

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 ([13, 29]) and parallel ([1, 22]) 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.

4.1. Parallel Selection - Implementation and Analysis

We now present the parallel algorithm for selection, making use of the Dynamic Data Redistribution algorithm given in Section 3. The following is run on processor j:

Algorithm 2 *Parallel Selection Algorithm*

Block Distributed 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 shows that [4]

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

where m is defined in Eq. (1) 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.

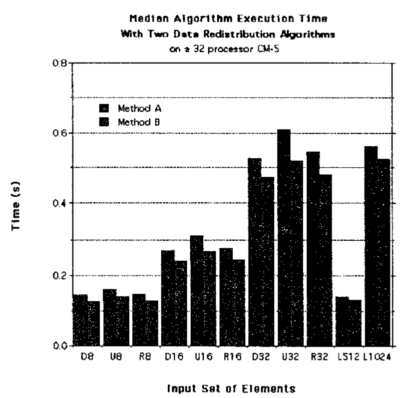


Figure 6. Performance of Median Algorithm

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.

4.2. 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 are chosen by

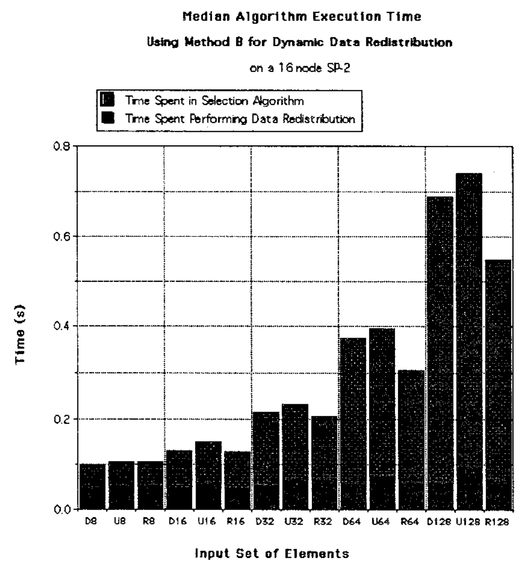


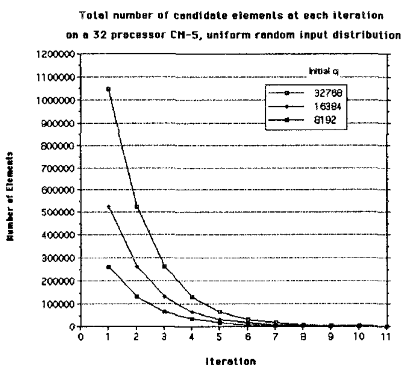
Figure 7. Performance of Median Algorithm on the SP-2

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

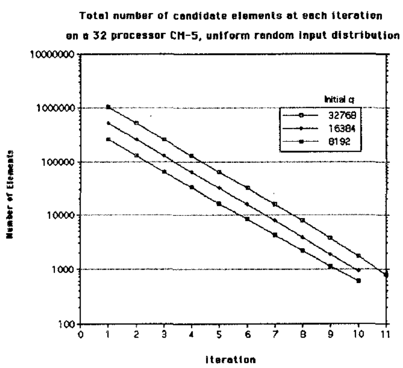
- **D: Duplicate**. Each processor holds the values $[0, q - 1]$;
- **U: Unique**. Each processor holds the values $[jq, (j + 1)q - 1]$;
- **R: Random**. Each processor holds uniformly random values in the range $[0, 2^{31} - 1]$.

The last two input sets correspond to an intermediate problem set from a computer vision algorithm for segmenting images [5]. 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 parts 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**.



linear scale



log scale

Figure 8. Number of candidates per iteration

We benchmark our selection algorithm in Table I. The input for this problem, taken from the NAS Parallel Benchmark for Integer Sorting [6], 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:

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

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

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

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

the problem in about half the time. This is consistent with the BDM analysis given in Eq. (2). 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 with 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 fastest sorting results on the CM-5 for the NAS input. Note that the algorithm in [7] 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.

Researchers	Time (s)	Notes
Bader & JáJá	2.77	BDM Selection
Dusseau [18]	7.67	Radix Sort
TMC [7]	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

5. Acknowledgements

We would like to thank the CASTLE/Split-C group at UC Berkeley, especially for 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.

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

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

Please see `http://www.umiacs.umd.edu/~dbader` for additional performance information. In addition, all the code used in this paper is freely available for interested parties from our anonymous ftp site, `ftp://ftp.umiacs.umd.edu/pub/dbader`. We encourage other researchers to compare with our results for similar inputs.

References

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

[2] D. 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, Dec. 1994. To appear in *Journal of Parallel and Distributed Computing*.

[3] D. Bader and J. JáJá. Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study. In *Fifth ACM SIGPLAN Symposium of Principles and Practice of Parallel Programming*, pages 123–133, Santa Barbara, CA, July 1995.

[4] D. Bader and J. JáJá. Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection. Technical Report CS-TR-3494 and UMIACS-TR-95-74, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, July 1995. To be presented at the 10th *International Parallel Processing Symposium*, Honolulu, HI, April 15-19, 1996.

[5] D. Bader, J. JáJá, D. Harwood, and L. 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. To be presented at the 10th *International Parallel Processing Symposium*, Honolulu, HI, April 15-19, 1996.

[6] 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.

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

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

[9] 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.

[10] P. Berthomé, A. Ferreira, B. Maggs, S. Perennes, and C. 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.

[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] 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.

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

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

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

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

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

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

[19] 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.

[20] E. Hao, P. MacKenzie, and Q. 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.

[21] 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.

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

[23] J. JáJá and K. Ryu. The Block Distributed Memory Model. Technical Report CS-TR-3207, Computer Science Department, University of Maryland, College Park, January 1994. To appear in *IEEE Transactions on Parallel and Distributed Systems*.

[24] J. JáJá and K. 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).

[25] 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.

[26] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 1994. Version 1.0.

[27] 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.

[28] 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.

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

[30] 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.