Random Data Accesses on a Coarse-grained Parallel Machine I. One-to-one Mappings

Sanjay Ranka
Ravi Shankar

CRPC-TR94530-S
October 1994

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Random Data Accesses on a Coarse-grained Parallel Machine I. One-to-one Mappings *

Ravi V. Shankar Sanjay Ranka
School of Computer and Information Science
Syracuse University, Syracuse, NY 13244-4100
e-mail: rshankar, ranka@top.cis.syr.edu

October 1994

Abstract

This paper describes deterministic communication-efficient algorithms for performing dynamic permutations on a coarse-grained parallel machine. Our analysis shows that the general permutation operation can be completed in $C\mu n/p$ (+ lower order terms) time and is optimal and scalable provided $n \geq O(p^3+p^2\tau/\mu)$ (n is the size of the permutation or the number of elements distributed across the p processors, τ is the start-up overhead and $1/\mu$ is the data transfer rate). C is a small constant typically between 2 and 3 for write permutations, slightly higher for read permutations. Modifications to exploit locality of access are presented. Special classes of permutations that are optimal for smaller sizes are also described. The dynamic permutation operation provides the framework for the communication-efficient simulation of an EREW PRAM on a coarse-grained distributed memory parallel machine. A companion paper [20] deals with the problem of random data accesses with hot spots.

*A preliminary version of this paper titled *Performing Dynamic Permutations on a Coarse-grained Parallel Machine* is to be presented at the First International Workshop on Parallel Processing, Bangalore, December 1994.

1 Introduction

Let n be the number of elements distributed across p processors. A permutation is an operation that rearranges data associated with some or all of the n elements. Permutations can also be defined as follows. Let each element i $(0 \leq i < n)$ have a pointer (i.e., destination/source number) $P(i)$ $(0 \leq P(i) < n)$ and data $D(i)$ associated with it. In a *write* permutation, each element i $(0 \leq i < n)$ sends its data to element $P(i)$. In a *read* permutation, each element i gets data from element $P(i)$. In both cases, it is imperative that no two elements have the same value of $P(i)$. In High Performance Fortran (HPF), a forall statement such as

```
forall (i=0:n-1)  Result(P(i)) = D(i)
```

results in a write permutation, while a statement such as

```
forall (i=0:n-1)  Result(i) = D(P(i))
```

results in a read permutation, assuming in both cases that $P(i) \neq P(j)$ for any $i \neq j, 0 \leq i, j < n$. Figure 1 illustrates the permutation primitives through examples. The issues involved in the design of algorithms for read permutations are very similar to those for write permutations. These are outlined in section 9. The rest of the paper deals with write permutations only.

Element index	0	1	2	3	4	5	6	7
Pointer P	.	2	0	7	1	6	3	5
Read Result	-	D(2)	D(0)	D(7)	D(1)	D(6)	D(3)	D(5)
Write Result	D(2)	D(4)	D(1)	D(6)	-	D(7)	D(5)	D(3)
Processor #	0	0	1	1	2	2	3	3

Figure 1: Read and Write Permutations

Efficient parallelization of a large number of applications on coarse-grained machines requires minimizing off processor accesses. Data structures and the corresponding computations are distributed such that most of the computations can be performed using local data. Several distributions for arrays have been found to be useful in practice and have been incorporated into data parallel languages like High Performance Fortran [9]. However, efficient data distribution for one phase of computation may in general be different from the next phase. In such cases performance improvement can be achieved by redistribution of data. Redistribution of data elements in HPF can be viewed as permutations [1].

Sample based sorting algorithms go through several stages [22]. First, a small sample of all the data elements in each processor is sorted to find approximate partitioners. These partitioners are used to move local data to appropriate destinations such that the data elements in each processor are smaller than the data elements in the processor to its right and larger than those in the processor

to its left. The data movement stage can viewed as a permutation. The scalability of sample sort critically depends on the cost of this permutation [12].

The execution of array assignment statements in HPF requires data movements in which only a subset of elements of the source are mapped to a subset of the elements of a destination array (excluding special cases such as the array assignment statement having a scalar as the right hand side). The permutation primitive can be generalized to deal with such cases of array *reformatting*.

2 Modeling a Coarse-grained Parallel Machine

We model a coarse-grained parallel machine as follows. A coarse-grained machine consists of several 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. A unit computation local to a processor has a cost of δ. Communication between processors has a start-up overhead of τ, while the data transfer rate is $1/\mu$. The time taken to send a message from one processor to another is modeled as $\tau + \mu m$, where m is the size of the message. For our complexity analysis we assume that τ and μ are constant, independent of the link congestion and distance between two nodes. With new techniques such as wormhole routing and randomized routing [13, 12, 7, 15], the distance between communicating processors seems to be less of a determining factor on the amount of time needed to complete the communication. Further, the effect of link contention (due to several messages traversing common links along their routes) is limited due to presence of virtual channels and the fact that link bandwidths are much larger than node interface bandwidths. This permits us to use the two-level model and view the underlying interconnection network as a virtual crossbar network connecting the processors. The logP [6] model and the postal model [2] are theoretical models, based on the above philosophy, for coarse-grained machines.

Although our algorithms are analyzed under these assumptions, most of them are architecture independent and can be efficiently implemented on meshes and hypercubes.

3 The Dynamic Permutation Problem

We are primarily interested in optimal communication-efficient algorithms for performing permutations. This section presents algorithms for permutations available in the literature and their limitations.

Linear permutation is a simple algorithm for performing all-to-all personalized communication where the messages exhibit a low variance in size. The algorithm is shown in figure 2. Linear permutation is deterministic and takes $O(sp)$ time, where s is the upper bound on the sizes of the messages exchanged. When linear permutation is used for performing dynamic permutations,

Linear_Permutation

For all processors P_i, $0 \leq i \leq p-1$, *in parallel do*

Generate receive vector *recvl* from the send vectors *sendl* in all the processors;

for $k = 1$ to $p-1$ do

$j = i \oplus k$;

if $sendl^j > 0$ then P_i sends a message of size $sendl^j$ to P_j

if $recvl^j > 0$ then P_i receives a message of size $recvl^j$ from P_j

Barrier synchronize with all processors;

endfor

Figure 2: The Linear Permutation Algorithm

message sizes could vary between 0 and $\lceil n/p \rceil$ and, in the worst case, a dynamic permutation could take $O(n)$ time and is hence non-optimal.

Sorting all the elements based on the destination element indices is one way of performing a dynamic permutation optimally. Such sorting based algorithms are highly communication inefficient, since the elements are moved around through many intermediate processors during the sort. See [19] for details. Using sample based sorting algorithms for performing permutations is not an option since these algorithms themselves require a permutation for data movement. Algorithms using randomization also have large constants and are not very practical for coarse-grained machines.

Our objective is to design a deterministic algorithm for permutation that eliminates node contention at the destination processors. In other words, all communication needs to scheduled such that no processor will receive more than one message at any time. This allows us to give a worst-case analysis of the time taken. The dynamic nature of the problem rules out the use of expensive communication scheduling algorithms to eliminate or reduce node contention. The dynamic permutation problem has the property that each processor sends out no more than $\lceil n/p \rceil$ elements and receives no more than $\lceil n/p \rceil$ elements. Since the outgoing/incoming traffic at any processor is upper-bounded by $\lceil n/p \rceil$, this is a bounded transportation problem [19]. Underlying each permutation is a many-to-many personalized communication problem. The communication matrices in figure 3 show the underlying communication pattern for the permutations in figure 1.

A minimal restriction on the size of the permutations for optimality and communication-efficiency is derived next. Consider algorithms that perform permutations by transferring data elements directly to the destination processors. This includes the linear permutation algorithm as well as non-deterministic asynchronous communication routines that are commonly used in practice. When every data element is directly transferred to its destination processor, each processor may have to

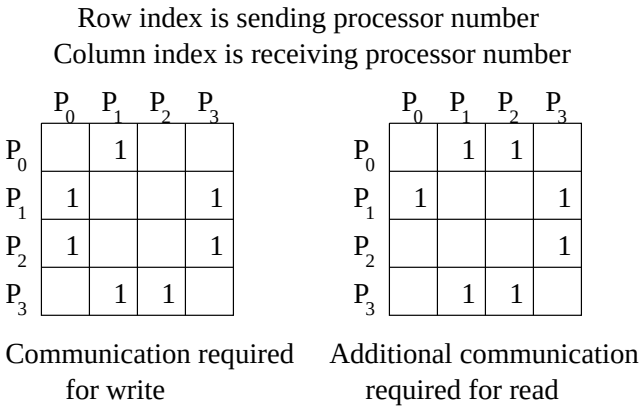


Figure 3: Communication Patterns for the Permutation Examples

communicate with $min(n/p, p)$ processors, and send out up to $\lceil n/p \rceil$ elements when performing a permutation. This would take $\mu n/p$ communication time along with a start-up overhead of $min(n/p, p)\tau$ time. Local computation in the processors takes at least $\delta n/p$ time. Thus, the total time required for performing a general permutation is atleast $\delta n/p + \tau min(n/p, p) + \mu n/p$. In practice n/p is at least as much as p, that is, $min(n/p, p) = p$. For an efficient implementation, the start-up cost should not be allowed to dominate the time required. This implies that $\mu n/p \geq \tau p$, that is, $n \geq p^2\tau/\mu$.

4 The Dynamic Permutation Algorithm

The many-to-many personalized communication with possibly high variance in message sizes, that underlies a dynamic permutation, can be performed in two stages. Each of these stages involves an all-to-all personalized communication with low variance in message sizes. Each of the p processors may have up three roles to play in this algorithm: as *source* processors when they have elements to be sent out, as *intermediate* processors, and as *destination processors* when they have elements to be received.

The n elements taking part in the permutation are distributed across the p processors. Let a_{ij} $(0 \leq i, j < p)$ be the number of elements sent from processor P_i to processor P_j. The number of elements in each processor before and after the permutation is at most $\lceil n/p \rceil$. Therefore, $\sum_{i=0}^{p-1} a_{ij} \leq \lceil n/p \rceil$ for $0 \leq j < p$ and $\sum_{j=0}^{p-1} a_{ij} \leq \lceil n/p \rceil$ for $0 \leq i < p$.

4.1 Message Splitting

The two-stage algorithm replaces the direct sending of a_{ij} elements from P_i to P_j by sending them through processors P_k $(0 \leq k < p)$ which act as intermediaries. If every a_{ij} is a multiple of p, this message splitting is trivial, since the a_{ij} elements can be equally divided among the p processors. The size of a message exchanged between any pair of source and intermediate processors is of size

no more than $\lceil n/p^2 \rceil$. The same is true of the size of messages exchanged between intermediate and destination processors. Thus the permutation can be completed in $2p(\tau + \mu n/p^2)$ time. The algorithm is optimal when $n \geq O(p^2\tau/\mu)$.

The splitting of messages, which is central to the algorithm, is shown in figure 4 for the general case where every a_{ij} may not be a multiple of p. The splitting is illustrated by extending the 2-D communication matrix along a third dimension. This dimension is indexed by k, the number of the intermediate processor through which the messages are sent (i and j represent the source and destination processor number, respectively). Of the a_{ij} elements sent from source processor P_i to destination processor P_j, a_{ij} *div* p elements will be sent through each intermediate processor. The bigger problem is deciding which of the p intermediate processors get the remaining a_{ij} *mod* p elements. We use a round-robin technique for the assignment of these excess elements which are allocated one by one to each intermediate processor. The allocation is started at the intermediate processor where the allocation of excess elements from the same source processor (to destination processors with smaller indices) left off. Figure 4 shows $a_{00} = 11$ split into (3,3,3,2), $a_{01} = 1$ into (0,0,0,1), $a_{02} = 4$ into (1,1,1,1) and $a_{03} = 1$ into (1,0,0,0). This assignment of excess elements to intermediate processors can be done in parallel with no communication. If the round-robin assignment of excess elements is to continue across source processors (as shown in the example in the figure) a global prefix-sum-scan of the quantity $\sum_{j=0}^{p-1} a_{ij}$ is needed. This scan, if used, takes $2\tau \log p$ time.

4.2 Communication Time

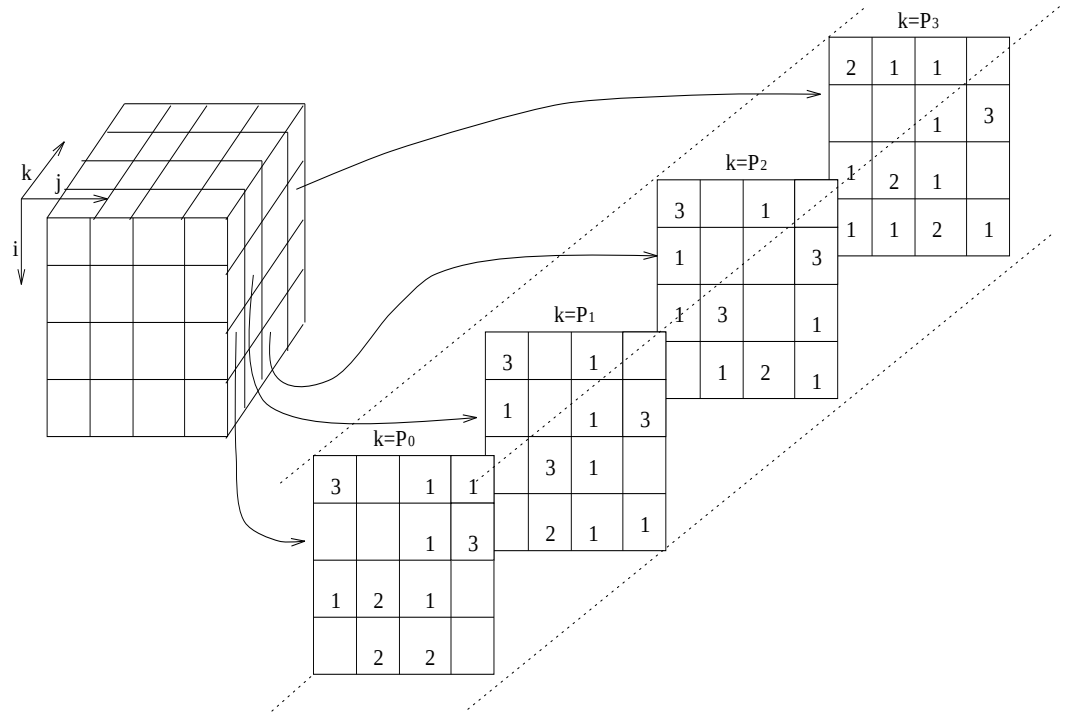
Figure 4 also shows the original communication matrix, and the communication matrices for the first and second stages. These were obtained by summation along dimensions k, j, and i, respectively. The entries in the communication matrices for the first and second stages cannot be greater than $\lceil n/p^2 \rceil$ and $\lceil n/p^2 + p \rceil$, respectively.

The round-robin assignment technique ensures that no source processor sends messages of size more than $\lceil n/p^2 \rceil$ to any intermediate processor. However, the messages sent from the intermediate processors to the destination processors could be of size nearly $\lceil n/p^2 + p \rceil$. The communication in the first stage can be completed in $p\tau + p\mu(n/p^2 + 1)$ time using linear permutation, while the second stage could take $p\tau + p\mu(n/p^2 + p)$. The two-stage algorithm takes time $2\mu n/p$ and is optimal when $n/p \geq O(p^2 + p\tau/\mu)$.

4.3 Message Coalescing

The message sent from a source processor to an intermediate processor, or from an intermediate processor to a destination processor, could have as many as p parts each. Message coalescing is done to ensure that these parts are sent out as a single message. This implies that no more than p messages [1] will be sent out of any source processor in the first stage and no more than p messages

[1] In reality, this can be no more than $p-1$ messages, since one of the messages is sent to the sending processor itself. This paper, for the sake of simplicity, continues to refer to p as the maximum number of messages being sent out.



Original Communication Matrix

	P_0	P_1	P_2	P_3	
P_0	11	1	4	1	17
P_1	2		3	12	17
P_2	3	10	3	1	17
P_3	1	6	7	3	17
	17	17	17	17	

Communication Matrices for Stage 1 and Stage 2

	P_0	P_1	P_2	P_3
P_0	5	4	4	4
P_1	4	5	4	4
P_2	4	4	5	4
P_3	4	4	4	5

	P_0	P_1	P_2	P_3
P_0	4	4	5	4
P_1	4	5	4	4
P_2	5	4	3	5
P_3	4	4	5	4

Figure 4: Splitting of Messages in the Permutation Algorithm

will be sent out of any intermediate processor in the second stage. Message coalescing can be done in one of two ways depending upon the size of each element. If each element is small in size, a local reshuffling (figure 5) is done to arrange all data being sent to the same processor into one contiguous message. If each element is multiple words long, such reshuffling could get prohibitively expensive. Instead, each message being sent is described using at most $\lceil n/p \rceil$ pointers and p associated lengths (figure 6). The actual coalescing is done when the elements are sent out of the processor. Such a send primitive that avoids local copying by allowing access to data from non-contiguous areas is available in MPI (Message Passing Interface [14]) and can be easily implemented in low-level software. In either case, if element i needs to send its data $D(i)$ to destination $P(i)$, the destination address needs to be split into a processor address ($P(i)$ div p) and a local address within that destination processor ($P(i)$ mod p). This entire computation is local to each processor and takes $O(n/p)\delta$ time.

5 Unbalanced Dynamic Permutations

The dynamic permutation algorithm can be generalized to deal with *unbalanced* permutations, where the number of elements leaving a processor is different from the number of elements entering it. This occurs, for instance, during array redistributions. If a maximum of x elements leave a processor and a maximum of y elements enter a processor, the permutation can be done in time $(x+y)\mu$ + lower order terms, provided, either $x \geq O(p\tau/\mu)$ and $y \geq O(p^2 + p\tau/\mu)$ or $x \geq O(p^2 + p\tau/\mu)$ and $y \geq O(p\tau/\mu)$.

The condition $y \geq O(p^2+p\tau/\mu)$ follows from the two-stage algorithm presented in the last section, since it was only the second stage that required the constraint $n/p \geq O(p^2+p\tau/\mu)$ for optimality. An alternate message splitting scheme can be used to reduce the second stage's communication time to $p\tau+\mu(y+p)$ while increasing the first stage's communication time to $p\tau+\mu(x+p^2)$. For optimality, this imposes the constraint $x \geq O(p^2 + p\tau/\mu)$. This new message splitting scheme is illustrated in figure 7.

The new scheme also assigns excess elements from source processors in a round-robin fashion, one by one to each intermediate processor. The allocation is started at the at the intermediate processor where the allocation of elements to the same destination processor (from source processors with smaller indices) left off. Figure 7 shows $a_{00} = 11$ split into (3,3,3,2), $a_{10} = 2$ into (1,0,0,1), $a_{20} = 3$ into (0,1,1,1) and $a_{30} = 1$ into (1,0,0,0). This assignment of excess elements to intermediate processors can be done in parallel upon completion of a global prefix-sum-scan with the vector a_{ij} $(0 \leq j < p)$ in each processor P_i. The vector scan takes $2\tau \log p + 2\mu p$ time. If the round-robin assignment of excess elements is to continue across destination processors (not shown in the example in the figure) a global prefix-sum-combine of the vector a_{ij} $(0 \leq j < p)$ would be needed in each processor P_i. The vector combine takes the same amount of time as the vector scan, and the two can even be done together. This new message splitting scheme ensures that messages exchanged in the second stage are of size no more than $\lceil y/p \rceil$, while those in the first stage are of size no more

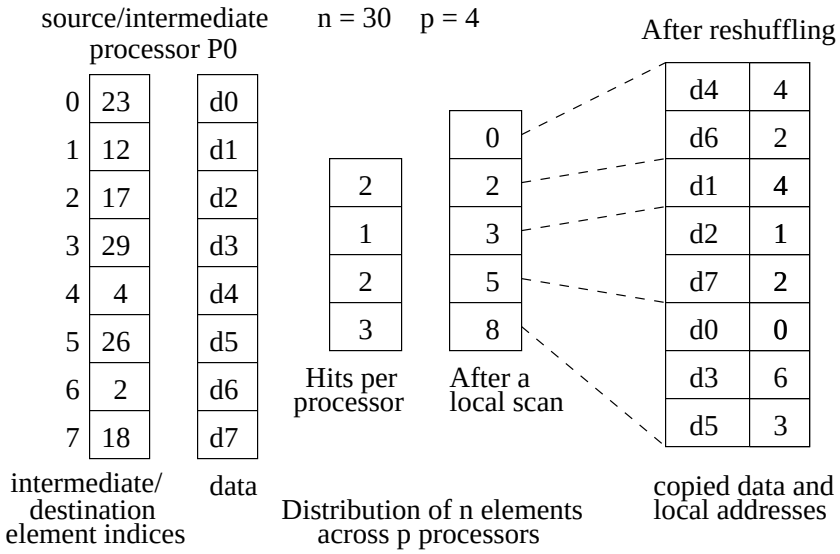


Figure 5: Reshuffling for message coalescing

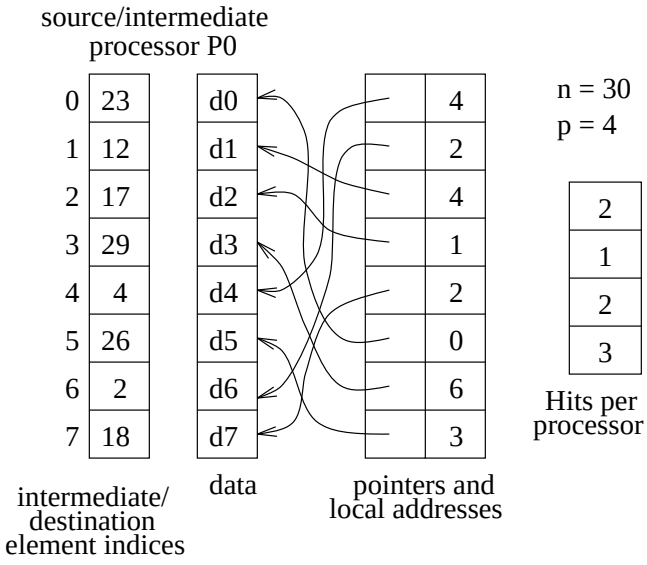


Figure 6: Message coalescing without reshuffling

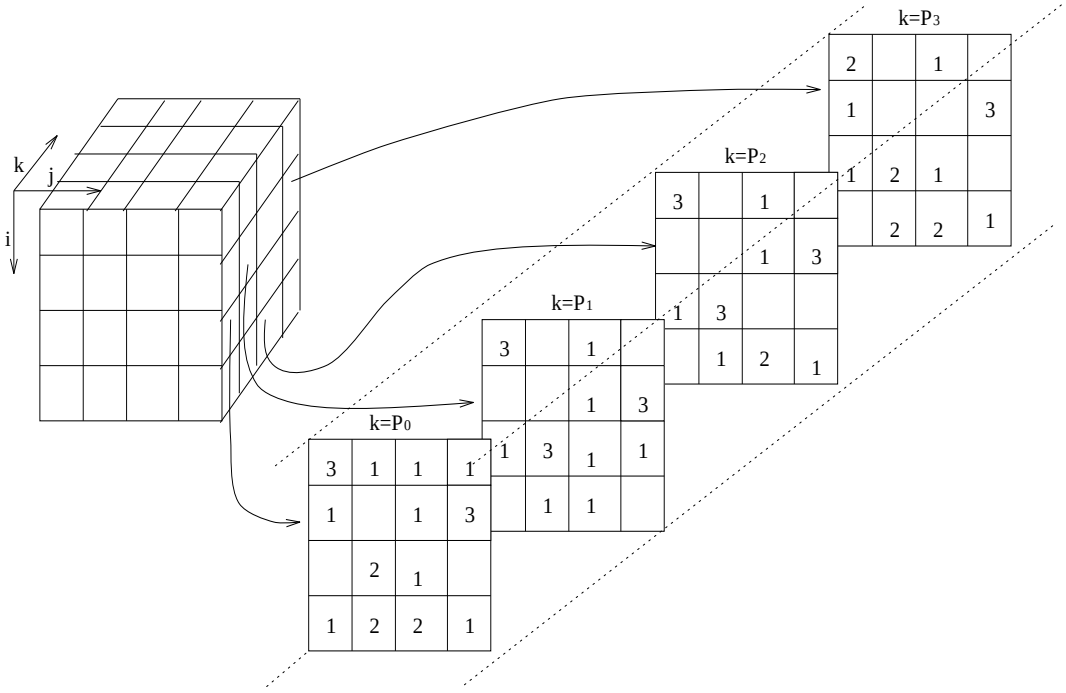


Figure 7: An Alternate Message Splitting Scheme

than $\lceil x/p + p \rceil$.

The $O(p^2)$ terms in the constraints for x, y (n/p in the balanced permutation case) represent worst case requirements. A probabilistic analysis [19] indicates that these requirements can be brought down to $O(p\sqrt{p \ln p})$ from $O(p^2)$. The $O(p\tau/\mu)$ terms in the constraints are added to ensure that start-up overheads do not dominate the time taken.

6 Exploiting Locality of Access

In the last two sections we presented an algorithm that performed dynamic permutations in two stages. The time taken by the algorithm depended only on the maximum number of elements sent out/received at every processor. However, the following points need to be noted:

1. If the destination pointers of some source elements point to the same processor in which the elements reside, such elements need not participate in the two-stage algorithm.

2. If the communication underlying the permutation is nearly *uniform*, that is, if the messages to be exchanged between source and destination processors are roughly of the same size, a linear permutation algorithm can complete this permutation in one stage rather than two.

Exploiting simplicity in the access patterns (as in case 2) and locality of access (as in case 1) improves the time taken by the permutation algorithm and is important for any practical implementation.

7 Monotonic Dynamic Permutations

Monotonic permutations, or permutations in which the pointers are sorted, can be performed using the algorithms described earlier in this paper. However, when the underlying architecture is a virtual crossbar the constants in the communication time for monotonic permutations can be reduced further. In fact, a single stage algorithm is sufficient to perform a monotonic permutation optimally and deterministically with no node contention on a virtual crossbar. In this section we have chosen to present the monotonic permutation algorithm through two important primitives, concentrate and distribute [16], where the pointers of the permutation are sorted. These primitives are useful, for instance, when working with sparse arrays, where the concentrate primitive can be used to convert the array from a dense representation to a compact representation [21]. After working with the compact representation, the distribute primitive can be used to convert the array back to the dense representation.

Concentrate

In the *concentrate* primitive there are an uneven number of selected elements in each processor and these have to be reassigned equally to the p processors as follows. Each selected element i has data $D(i)$ and the number $R(i)$ of selected elements with lower indices. The objective is to to set $Result(R(i))$ to $D(i)$. Figure 8 illustrates the concentrate operation. The number of elements sent out of a processor could exhibit a large variation, but the difference between the number of elements received by each processor cannot be more than 1. Let s_{max} and s_{min} be the maximum and minimum number of elements sent out of any processor. Let q be the maximum number of messages sent out or received at any processor ($q < p$), and let r be the total number of elements sent out from all the processors. The concentrate primitive results in each processor receiving either $\lceil r/p \rceil$ or $\lfloor r/p \rfloor$ elements. An algorithm that can perform this concentrate in time μs_{max} is optimal. To design an optimal algorithm, we need to ensure that node contention is eliminated.

While performing a concentrate, the destinations $R(i)$ in all the sending processors are sorted to begin with. This property is used to divide the elements in the sending processors into segments, where each segment contains elements being sent to the same processor. Consider a single sending processor with s elements ($s_{min} \leq s \leq s_{max}$). These s elements could be divided into as many as q segments. The segments in this processor (see figure 9) are of three kinds: *preceding partial* segments that continue from the previous processor, *succeeding partial* segments that (start in this processor and) continue into the next processor, and *full* segments that start and terminate in this processor. Let α_i, β_i, and γ_i be the number of elements in these three kinds of segments respectively ($\alpha_i + \beta_i + \gamma_i = s$) in sending processor P_i. Each such processor first sends the γ_i elements in its succeeding partial segment followed by the β_i elements in its full segments. To avoid node contention, this is followed by a wait until other processors sharing the preceding partial segment send out the their elements from the same segment. Finally, the α_i elements in the preceding partial

D	-	D(1)	-	-	D(4)	D(5)	-	D(7)
R	.	0	.	.	1	2	.	3
Result	D(1)	D(4)	D(5)	D(7)	-	-	-	-

Figure 8: The Concentrate Primitive

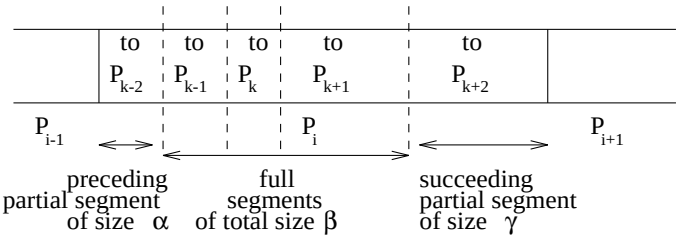


Figure 9: Segments in a processor during concentrate

segment are sent out. On a virtual crossbar this algorithm takes $q\tau + \mu s_{max}$ time. The steps in the concentrate algorithm are outlined in figure 10.

Distribute

The *distribute* primitive is the inverse of the concentrate primitive. An equal number of elements from each processor have to be reassigned to the p processors. The distribute primitive is illustrated in figure 11. While performing a distribute, the destinations $R(i)$ in all the sending processors are also in sorted order to begin with.

The algorithm for distribute is identical to the concentrate algorithm, although the various quantities used in the complexity analysis now represent different things. The sending processors now send out r elements, each processor contributing either $\lceil r/p \rceil$ or $\lfloor r/p \rfloor$ elements. The number of elements received could exhibit a large variation. Let s_{max} and s_{min} be the maximum and minimum number of elements received at any processor, and let q be the maximum number of messages sent out or received at any processor ($q < p$). The algorithm presented avoids node contention and can perform the distribute optimally on the virtual crossbar in time $q\tau + \mu s_{max}$.

8 Multiple Permutations

In this section we deal with a special class of permutations where the permutation of n elements can be decomposed into n/p permutations of p elements, with one element per processor participating in each permutation. One such permutation is illustrated in figure 12.

If $n = p$, any permutation of elements falls under this class. The time taken to perform such a permutation (on the virtual crossbar) is $\tau + \mu\ m$ where m is the size of each element. The $n = p$

Concentrate

For all processors P_i, $0 \leq i \leq p-1$, *in parallel do*

Determine whether the first and last segments are preceding partial segments and succeeding partial segments respectively, using right and left shifts by one element

If processor has a succeeding partial segment or a full segment,
then set s_bit to 1 and s_data to γ_i *else* set s_bit to 0 and s_data to α_i

Perform a segmented +scan (upward, inclusive) using s_bit to indicate the start of scan segments, and s_data as the element to be scanned. Each processor contributes just one element to the scan.

A right shift by one gives each preceding partial segment the number of elements r_{prec} preceding it in the same segment ($r_{prec} \leq \lceil r/p \rceil$).

Send elements in succeeding partial segment as a single message to the appropriate destination processor. Set *traffic_sent* to γ_i.

Send elements from each full segment to appropriate destination processor. Add β_i to *traffic_sent*.

Wait for a period of time corresponding to the sending of a message of size r_{prec} − *traffic_sent*.

Send elements in preceding partial segment as a single message to the appropriate destination processor.

Figure 10: The Concentrate Algorithm

D	D(0)	D(1)	D(2)	D(3)	-	-	-	-
R	1	4	5	7	.	.	.	.
Result	-	D(0)	-	-	D(1)	D(2)	-	D(3)

Figure 11: The Distribute Primitive

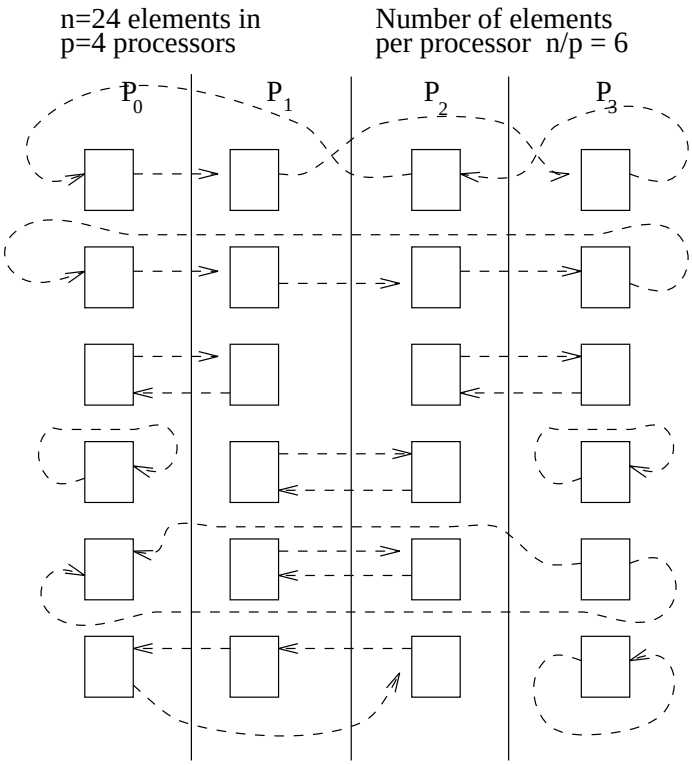


Figure 12: Multiple permutations example

condition is fairly unrealistic for coarse-grained architectures and start-up overheads are likely to dominate the time taken, unless the size of the elements being permuted is large.

If $n \neq p$, the n/p permutations can be performed one at a time. This takes $\delta n/p + \tau n/p + \mu mn/p$ time. If $n/p \leq p$, this is the minimal time needed to complete the permutations. If $n/p > p$, the two-stage algorithm that does message coalescing may perform better, since that could reduce the start-up overhead from $\tau n/p$ to τp. For the two-stage algorithm to perform better than the sequence of permutations algorithm, the following condition must hold:

$$\frac{n}{p}\tau + \frac{n}{p}\mu m \geq p\tau + 2\frac{n}{p}\mu m$$

$$\Rightarrow \frac{n-p^2}{p}\tau \geq \frac{n}{p}\mu m$$

$$\Rightarrow n \geq \frac{p^2\tau}{\tau - \mu m}$$

When comparing architectures, the above constraint implies that the two-stage algorithm is better if $\tau \geq \frac{\mu mn}{n-p^2}$. When comparing problems, the two-stage algorithm is better if the size of each element $m \leq \frac{(n-p^2)\tau}{\mu n}$ or if the size of the permutation $n \geq \frac{p^2\tau}{\tau - \mu m}$, other conditions remaining the same.

9 Read Permutations

In a read permutation, the n pointers give the indices of elements from which data is to be obtained. In the general case, algorithms for a read permutation go through two times the kind of communication that a write permutation goes through. The first communication involves the sending of the requesting elements' addresses to the source processors containing data. The second communication involves the sending of the requested elements by the source processors. If the data elements are of size m and the addresses are of size 1, the first communication takes $2p\tau + \mu(2n/p + p^2)$ time while the second takes $2p\tau + \mu(2mn/p + p^2)$ time. Thus the read permutation takes $4p\tau + 2\mu((m+1)n/p + p^2)$ time, or $2\mu(m+1)n/p$ time when $n \geq O(p^3/m + p^2\tau/\mu m)$. In comparison, the write permutation with elements of size m takes $2p\tau + \mu(2mn/p + p^2)$ time or $2\mu mn/p$ time under similar restrictions on n.

The effects of larger element size m on the time taken for read/write permutations are the following:

1. The minimal requirement on the size of the permutation n for optimality is scaled down by a factor of m. That is, the requirement $n \geq O(p^3)$ is reduced to $n \geq O(p^3/m)$.

2. Similarly, the requirement on the size of the permutation to avoid domination by start-up overheads is also scaled down by the factor m. That is, the requirement $n \geq O(p^2\tau/\mu)$ is reduced to $n \geq O(p^2\tau/\mu m)$.

10 EREW PRAM Simulation

The EREW PRAM is a shared-memory parallel programming model which allows only exclusive reads and exclusive writes. Simultaneous access of a single memory location by more than one-processor is not allowed. Dynamic permutations form the basic communication primitives for simulating an EREW PRAM. The algorithms described in this paper can thus be used to simulate an n processor EREW PRAM on a p processor machine. A wide variety of parallel algorithms have been described in the literature for the theoretical EREW PRAM model [10]. The PRAM simulation would provide a transparent method for implementing these algorithms on a real machine. The EREW PRAM simulation is communication-efficient and optimal provided $n \geq O(p^3 + p^2\tau/\mu)$.

11 Conclusions

In this paper we have presented communication-efficient algorithms for performing dynamic permutations on a coarse-grained parallel machine. Any dynamic permutation of size n such that the sources and destinations are equally divided can be completed in $C\mu n/p$ time, when the number of elements $n \geq O(p^3 + p^2\tau/\mu)$ and C is a small constant. The algorithm was generalized to deal with the case of unbalanced dynamic permutations. Algorithms for special cases such as monotonic permutations, concentrate/distribute, and multiple permutations were also presented. Scheduling of static permutations has been discussed in [17, 18].

The constants in the communication time complexity of the algorithms presented in this paper are very small. This is a necessary requirement for effective utilization of typical coarse-grained machines. When message sizes are small, latency becomes a dominating issue. Reduction in latency cost at the expense of sending the message to the final destination processor through several intermediate processors has been successfully achieved for all-to-all personalized communication with uniform messages by using a multiphase approach [4]. These techniques reduce on the latency requirements by transferring the data through several intermediate processors (where several messages are combined). These methods are equally applicable to our algorithms.

Although our algorithms were presented for a virtual crossbar model, they are relatively architecture independent and can be efficiently implemented on wide variety of interconnection networks. In particular, the dynamic permutation algorithm requires just two phases of all-to-all personalized communication with equal sized messages. Several algorithms for the all-to-all personalized communication exist, with time requirements proportional to traffic for hypercubes with cut through routing [3] or multiport communication [11], and with time requirements proportional to cross-section bandwidth for meshes [8] with cut-through routing.

The algorithms performing dynamic permutations take time proportional to (the maximum of) the total number of participating elements in a processor instead of p times (the maximum of) the number of elements that could be exchanged between any two processors. This result is of significance in the analysis of the time complexity of many algorithms. For instance, the sample sort algorithm's

worst-case time complexity is reduced from $O(n)$ [12, page 246] and matrix transpose (with checkerboard partitioning, i.e., block-block distribution) can be shown to be optimal on hypercubes and meshes with cut-through routing [12, page 158].

By formalizing one-to-one random data accesses, this paper provides a framework for solving irregular and unstructured applications such as graph problems in which accesses can be arbitrary/irregular and one-to-one. It also provides a framework for runtime support for languages such as HPF, specifically for data redistributions and array reformatting[1] through assignment statements. Conversions between any two regular distributions (block, cyclic,block-cyclic) in HPF, between any two irregular distributions, or between a regular distribution and an irregular distribution can all be viewed as dynamic permutations. Optimizations can be added if the distribution statements are known at compile time or when the accesses have inherent locality. These issues are under investigation.

References

[1] Seungjo Bae, Sanjay Ranka, Ravi V. Shankar. The Reformat Primitive - Runtime Support for Data Redistribution and Array Assignment Statements in HPF, (in preparation).

[2] A. Bar-No. and S. Kipnis. Designing Broadcasting Algorithms for the Postal Model for Message-Passing Systems, *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, 1992, pp. 13–22.

[3] Shahid H. Bokhari. Complete Exchange on the iPSC/860, ICASE Technical Report No. 91-4, NASA Langley Research Center, January 1991.

[4] Shahid H. Bokhari. Muliphase Complete Exchange on a circuit-switched hypercube, *Proceedings of 1991 International Conference on Parallel Processing*, pp. 525-529, 1991.

[5] Zeki Bozkus, Sanjay Ranka, Geoffrey C. Fox. Benchmarking the CM-5 Multicomputer, Proceedings of the Frontiers of Massively Parallel Computation, pp. 100-107, October 1992.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken, LogP: Towards a Realistic Model of Parallel Computation, *Proceedings of 4th ACM Symposium on Principles and Practices of Parallel Programming*, pp. 1–12, 1993.

[7] Willian J. Dally and Chuck L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks, *IEEE Trans. on Computers*, 36(5):pp. 547–553, May 1987.

[8] S. E. Hambrusch, F. Hameed, and A. A. Khokhar, Communication Operations on Coarse-Grained Mesh Architectures, Technical Report, Department of Computer Science, Purdue University.

[9] High Performance Fortran Forum, *High Performance Fortran Language* Specification, March 1994.

[10] Joseph Jaja. *An Introduction to Parallel Algorithms* Addison-Wesley, 1992.

[11] S. L. Johnsson, and C. T. Ho, Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38 (9), pp. 1249-1268, September 1989.

[12] Vipin Kumar, Ananth Grama, Anshul Gupta, George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin-Cummings, 1994.

[13] C. Leiserson et al. The Network Architecture of the Connection Machine CM-5, *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, 1992.

[14] MPI Forum. The Message-Passing Interface Standard, University of Tennessee, Knoxville.

[15] Lionel M. Ni and Philip K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks, *IEEE Computer*, 26(2):62–76, February 1993.

[16] D. Nassimi and S. Sahni. Data Broadcasting in SIMD Computers, *IEEE Transactions on Computers* C-30(2):101-107 (1981).

[17] S. Ranka, J. C. Wang, and G. C. Fox. Static and Runtime Scheduling of All-to-Many Personalized Communication on Permutation Networks, *IEEE Trans. on Parallel and Distributed Systems*. To appear.

[18] S. Ranka, J. C. Wang and M. Kumar. All-to-many communication avoiding node contention, *Journal of Parallel and Distributed Computing*. To appear.

[19] Ravi V. Shankar, Khaled A. Alsabti, Sanjay Ranka. The Transportation Primitive, CIS Technical Report, Syracuse University, August 1994.

[20] Ravi V. Shankar, Sanjay Ranka. Random Data Accesses on a Coarse-Grained Parallel Machine - II. One-to-many and Many-to-one Mappings, October 1994.

[21] Ravi V. Shankar, Sanjay Ranka. Parallel Vision Algorithms Using Sparse Array Representations, *Pattern Recognition*, 1993, vol. 26, No. 10, pp. 1511–1519.

[22] H. Shi, J. Schaeffer. Parallel Sorting by Regular Sampling. Journal of Parallel and Distributed Computing, Vol.14, pp.361-372, 1990.