Random Data Accesses on a Coarse-grained Parallel Machine II. One-to-many and Many-to-one Mappings

Sanjay Ranka
Ravi Shankar

CRPC-TR94531-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 II. One-to-many and Many-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 random data accesses with hot spots on a coarse-grained parallel machine. The general random access read/write operations with hot spots 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 number of elements distributed across p processors, τ is the start-up overhead and $1/\mu$ is the data transfer rate). C is a small constant between 3 and 4 for the random access write operation, slightly higher for the random access read operation. Monotonic random access reads/writes can be completed with smaller constants and are optimal for smaller n as well. The random access read/write operations provide the framework for the communication-efficient simulation of CREW and CRCW PRAMs on a coarse-grained distributed memory parallel machine. A companion paper [24] deals with the problem of performing dynamic permutations.

1 Introduction

Let n be the number of elements distributed across p processors. In a Random Access Read (RAR), each of the n elements may need to read data from another element [22]. The data is available in array D. Each element has the index of the element from which data is needed in array P. That is, element i needs $D(P(i))$. Figure 1 shows an example of a RAR.

Element index	0	1	2	3	4	5	6	7
Pointer P	7	.	0	7	1	6	3	0
After RAR	D(7)	-	D(0)	D(7)	D(1)	D(6)	D(3)	D(0)

Figure 1: RAR example

In a Random Access Write (RAW) each of the n elements may need to write data to another element [22]. The data is available in array D. Each element has, in array P, the index of the element to which it has to send its data. Unlike the RAR case, it is possible to have collisions during a RAW. This happens when two or more data elements are written to the same destination. When collisions are bound to occur, one of the following things can be done: (i) if the colliding data values are the same, use that value; otherwise, report an error, or (ii) choose one of the colliding values using a pre-defined rule, or (iii) combine the colliding data values using a pre-defined binary associative operator. Figure 2 shows an example of a RAW where collisions are resolved using a binary associative operator (shown as a + in the figure).

A forall statement of the following type in High Performance Fortran (HPF) [12] results in a RAR:

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

while the loop shown below when executed in parallel results in a RAW:

```
do (i=0:n-1)
   A(P(i)) = A(P(i)) + D(i)
```

In a RAW, the n elements writing data can be viewed as n threads of control in the p processors. These n threads write data to n elements of shared memory. In the RAW, the n threads and the

Element index	0	1	2	3	4	5	6	7
Pointer P	7	.	0	7	1	6	3	0
After RAW	D(2)+D(7)	D(4)	-	D(6)	-	-	D(5)	D(0)+D(3)

Figure 2: RAW example

shared memory of size n are equally divided among the p processors. Similarly, in the RAR, the n threads reading data and the n elements of shared memory from which data is read are equally divided among the p processors.

Random access reads/writes represent the basic operations that perform random data accesses where the access patterns could be *one-to-many* or *many-to-one* mappings. We use the term *hot spots* to refer to collisions that occur during a RAW or multiple reads from the same element during the RAR. Parallelization of many applications require the distribution of the data structures over all processors (either due to the lack of sufficient memory on one node or due to limited scalability when keeping it replicated). Each processor requires a subset of data structures in order to perform its computations locally. Some parts of the data structure are required by only one processor, while other parts are required by several processors (and are hot spots). A simple example of this is the generation of a locally essential tree on every processor, as is done for the parallelization of the Barnes-Hut-based n-body implementations [28, 3, 26]. The inverse of the above scenario is one in which data items are written into an accumulation array. Different entries within the accumulation array receive different numbers of writes. Often hot spots tend to dictate the time taken to perform the communication, and their distribution is available only at run-time. Such hot spots are inherent features of many algorithms such as those for histogramming, geometric hashing [21, 25], and database searching.

Our algorithms alleviate the hot spots problem by dynamically stretching and shrinking them as necessary. The algorithms are communication-efficient and the number of data movements between processors is kept very low. This was motivated by the fact that the cost of unit communication is usually much higher than the cost of unit computation, as evident from existing machines.

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 $1/\mu$. 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 [15, 14, 8, 17], 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 [7] model and the postal model [2] are

theoretical models based on the above philosophy, for coarse-grained machines.

- **Sending a Message**
  Assuming no node contention, 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.

- **Global Combine and Prefix Scans**
  Assume that each processor contains a vector $V_i[0 \cdots \frac{n}{p} - 1]$. Let p be the number of processors. The global combine operation computes an element-wise sum of the local list in each processor. The resultant vector $R[0 \cdots \frac{n}{p} - 1]$ is stored in all the processors.

$$R[j] = \sum_{i=0}^{p-1} V_i[j]$$

  The global vector prefix-scan performs an element-wise prefix-scan of the local list in each processor. Assuming that $+$ is the scan operator, the resultant vector $R_q[0 \cdots \frac{n}{p} - 1]$ in processor q $(0 \leq q < p)$ is given by:

$$R_q[j] = \sum_{i=0}^{q-1} V_i[j]$$

  We assume that these operations can be completed in $2\mu\frac{n}{p}$ time with a start-up overhead of $2\tau \log p$.

- **Complete Exchange**
  The complete exchange primitive performs all-to-all personalized communication with equal sized messages. It t is total length of all the messages sent out and received at every processor, complete exchange can be performed in time $p\tau + \mu t$.

- **Transportation**
  The transportation primitive performs many-to-many personalized communication with possibly high variance in message sizes. If the total length of the messages being sent out or received at any processor is upper bounded by t, the time taken for the communication is $2\mu t$ (+ lower order terms) when $t \geq O(p^2 + p\tau/\mu)$. If the outgoing and incoming traffic bounds were r and c instead, the communication takes time $2\mu(r + c)$ (+ lower order terms) when either $r \geq O(p^2 + p\tau/\mu)$ or $c \geq O(p^2 + p\tau/\mu)$. These asymptotic results are based on a worst case analysis. A probabilistic analysis brings the $O(p^2)$ term in the traffic requirement down to $O(p\sqrt{p \ln p})$ [23].

Although our algorithms are analyzed under the assumptions of a virtual crossbar, most of them are architecture independent and can be efficiently implemented on meshes and hypercubes. For example, complete exchange is a well-studied problem for which several algorithms are available in the literature using hypercubes [5] (time requirements proportional to traffic) and meshes [9] (time

Example (a)

	P_0				P_1				P_2				P_3			
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
P	4	5	5	1	5	5	0	5	8	15	8	9	10	13	13	8	16	20	21	21	20	16	22	17	29	29	27	28	31	26	24	29

Example (b)

	P_0				P_1				P_2				P_3			
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
P	4	5	25	9	25	16	22	15	1	18	8	4	30	18	31	9	16	25	31	13	0	6	22	14	31	2	17	13	3	26	14	19

Example (c)

	P_0				P_1				P_2				P_3			
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
P	4	5	25	9	15	15	12	15	1	28	2	4	30	28	31	6	16	25	31	13	0	6	22	14	21	22	17	13	31	16	23	19

Communication matrices for examples (a), (b) and (c)

	P_0	P_1	P_2	P_3	
P_0	8				8
P_1		8			8
P_2			8		8
P_3				8	8
	8	8	8	8	

	P_0	P_1	P_2	P_3	
P_0	2	2	2	2	8
P_1	2	2	2	2	8
P_2	2	2	2	2	8
P_3	2	2	2	2	8
	8	8	8	8	

	P_0	P_1	P_2	P_3	
P_0	2	5		1	8
P_1	4			4	8
P_2	2	2	2	2	8
P_3		1	6	1	8
	8	8	8	8	

Figure 3: RAW with Limited Hot Spots

requirements based on the cross-section bandwidth). The global combine and prefix scans can be completed on a hypercube in the time specified above. The presence of specialized hardware networks like the control network on the CM-5 can speed up the operation considerably for small vectors [6].

3 RAW with Limited Hot Spots

Special cases of the hot spots problem arise when the hot spots are distributed uniformly across the processors. This is illustrated through three examples in figure 3. In example (a) the hot spots could be resolved through simple local processing and no communication. The communication pattern underlying example (b) is a complete exchange. The time taken for this case is $p\tau + \mu n/p$. Example (c) is characterized by an upper bound on the number of elements received or sent out by any processor. As shown in the accompanying communication matrix [1], the underlying communication could involve messages with widely varying sizes. This is a bounded transportation problem, similar to the communication that underlies a dynamic permutation. It can be performed in $2\mu n/p$ time for $n \geq O(p^3 + p^2\tau/\mu)$ (see [24] for details).

[1] In all the communication matrices that appear in this paper, the row and column numbers give the indices of the sending and receiving processors respectively. The row sums and column sums give the total number of elements sent out and received at a processor.

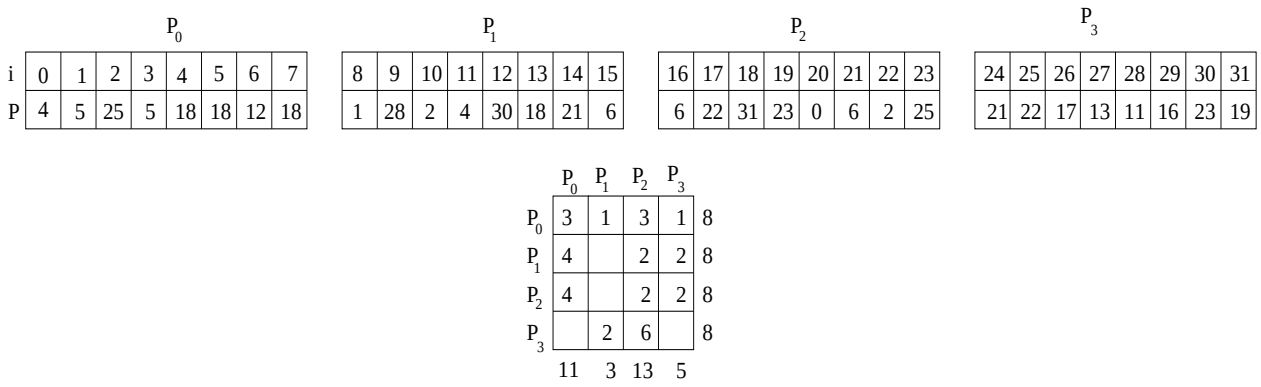

Figure 4: RAW with Arbitrary Hot Spots

4 RAW with Arbitrary Hot Spots

Hot spots, in general, need not be uniformly distributed across the processors. An example is shown in figure 4. In an extreme case where the sum of all the n elements is needed at a single destination, the incoming traffic bound at one of the processors is $O(n)$. The algorithm described in this section splits the communication in the RAW into two stages [2] , bringing the traffic bound down to $O(n/p)$. Processors sending elements during the first stage are referred to as *source* processors, while those that receive elements during the second stage are *destination* processors. Processors that receive data during the first stage and send data during the second stage are referred to as *intermediate* processors. The RAW algorithm that deals with arbitrary hot spots includes two major communication stages and involves:

- pre-processing at the source processors,
- communication between the source and intermediate processors (which resembles the communication underlying a dynamic permutation [24]),
- pre-processing at the intermediate processors (which includes prefix scans that resolve some of the collisions at hot spots),
- communication between the intermediate and destination processors (which resembles the communication underlying a monotonic dynamic permutation [24]), and
- post-processing at the destination processors.

Dividing the threads writing data and the elements in shared memory being written to, equally among the p processors, results in n/p threads and n/p amount of shared memory per processor. An important feature of the parallel RAW algorithm presented here is the grouping of elements into

[2]The two stages in the RAW algorithm are entirely different from the two stages in the algorithms for dynamic permutations [24] or transportation [23].

buckets (as explained below). The parallel RAW algorithm takes $O(kp + n/kp)\mu + 3\mu n/p$ time, k being the number of buckets per processor. The algorithm is optimal with a complexity of $3\mu n/p$ when $n \geq O(p^3 + p^2\tau/\mu)$. The requirement $n \geq O(p^3 + p^2\tau/\mu)$ is not overly restrictive, because the underlying architecture is coarse-grained and p is usually not too large. A sequential RAW algorithm that works on n elements in a single processor would take $O(n)$ time.

4.1 Pre-processing at the Source Processor

The shared memory at each of the p destination processors is initially viewed as a collection of k *buckets*. Pre-processing first needs to be done in order to minimize the amount of communication between the processors. In every processor, an array of size kp is created to record the number of threads writing to each bucket. This is done by examining the n/p destination indices associated with the n/p threads local to each processor. The local count is followed by a global vector sum-combine, which results in an identical vector in every processor (figure 5). This vector gives the total number of threads, from all the processors, writing to each bucket. These kp entries give exactly the contention at each bucket. This predetermined contention is used to *stretch* (or shrink) buckets in order to balance the communication to follow. The new extent of each bucket is easily obtained by performing a local sequential prefix-sum-scan. Since threads from every processor could potentially write to a newly stretched bucket, an additional step is used to allocate different portions of the bucket to threads from different processors. This step basically repeats the global operation done earlier with the local bucket hits array, but performs a global vector prefix-sum-scan instead of a reduce. The prefix scan is exclusive, and as a result each processor knows exactly which portions of the kp buckets it will write to. These are the intermediate locations to which data elements are to be sent.

When the actual communication is performed, all data elements intended for the same destination processor should be sent together as a single message. This *message coalescing* helps to minimize communication overhead. Message coalescing is preceded by a sequential reshuffling of the data local to each processor, if the element size is small. If the element size is large, local reshuffling can be avoided through the use of a set of pointers to describe each message [24].

4.2 Communication Between Source and Intermediate Processors

The communication between the source and intermediate processors, as determined by the pre-processing stage, is a bounded transportation problem, similar to the communication that underlies a dynamic permutation. Figure 6 shows the communication in the first stage for the example introduced in figure 4. The initial assignment of buckets to the destination processors and the extent of the buckets after stretching are also shown. The numbers below each bucket gives the new length of that bucket. Each processor has no more than n/p data elements to send out in the form of p or less messages, and each receiving processor gets no more than n/p data elements due to the bucket stretching. This communication can be performed in $2\mu n/p$ time when $n \geq O(p^3 + p^2\tau/\mu)$.

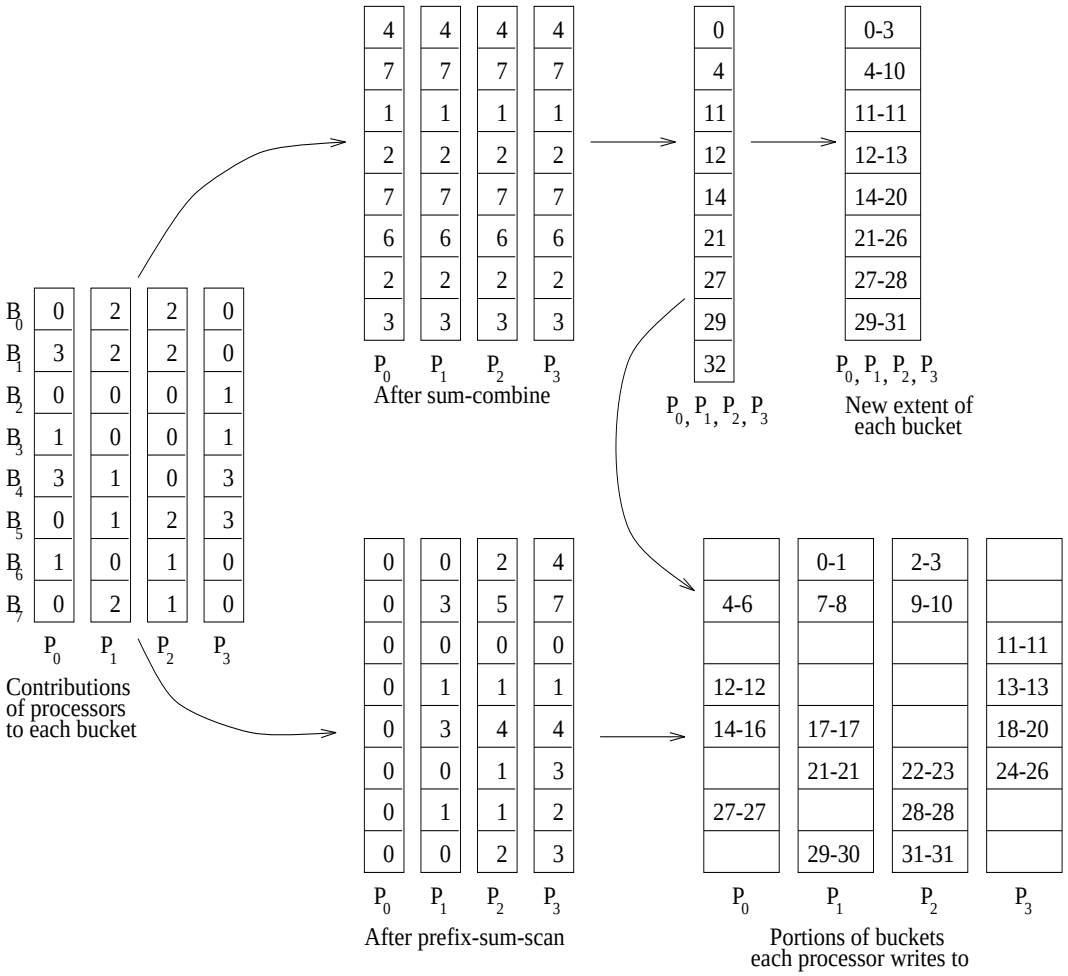

Figure 5: Pre-processing at the Source Processor

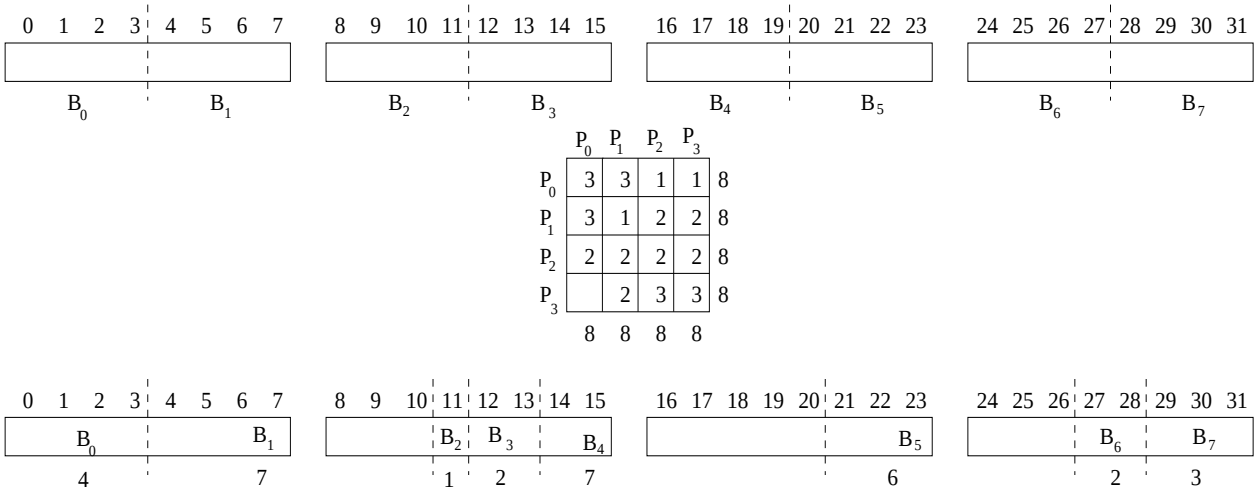

Figure 6: Communication between Source and Intermediate Processors

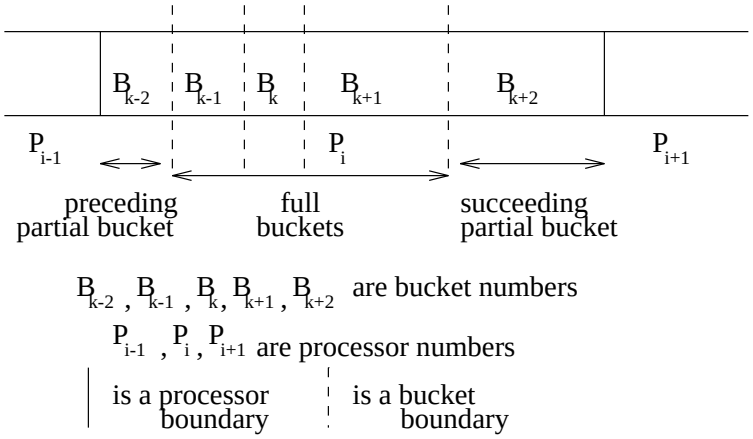

Figure 7: Ordering of Buckets in an Intermediate Processor

4.3 Processing at the Intermediate Processors

The maximum number of data elements received at each intermediate processor is n/p. These elements belong to a set of consecutive buckets. The elements that contribute to a single bucket are, in general, not contiguous unless a reshuffling is performed. The reshuffling would take under $kp + n/p$ time and would involve two passes: the first through a maximum of kp buckets to find out the number of elements contributing to each bucket, and the second through the n/p elements, to move them to the right position. This does not alter the algorithm's asymptotic time complexity, but in practice could get expensive especially if the element size is large. If reshuffling is done, the buckets in a processor will be explicitly ordered as shown in figure 7.

If reshuffling is not done, the ordering shown in figure 7 is not explicit. A sequential pass through the n/p or less elements in each processor identifies the bucket that each element has to go to. The buckets that are contained entirely in a processor are called *full* buckets, while those that stretch across processor boundaries are called *partial* buckets. A processor cannot have more than two partial buckets.

4.3.1 Processing of Full Buckets

Let b be the number of elements that have contributed to a bucket, that is, the number of threads writing to the bucket. This bucket is classified as being *sparse* if $b \leq n/kp$ [3] and as *dense* if $b > n/kp$. Sparse buckets require storage of size b, and dense buckets require n/kp. The latter follows from the fact that when more than n/kp elements contribute to a bucket of size n/kp, there must be collisions and these collisions could be resolved using the given RAW collision resolving operator. The processing of full buckets involves performing this collision resolution. In the intermediate processors shown in the example in figure 8, B_0, B_2, B_3, B_6 and B_7 are full buckets that are sparse.

[3] Sparse buckets could have alternately been defined as buckets for which $b \leq$ a constant times n/kp

4.3.2 Processing of Partial Buckets

Of the buckets in each processor, only the first and last buckets could be partial. If they exist, they are called the *preceding* partial bucket and the *succeeding* partial bucket, respectively. Decisions made as to whether partial buckets are sparse or dense will be incomplete with just local information. For this reason, partial buckets are always stored using the dense representation. Identifying the preceding and succeeding partial buckets in each processor, if this information is not already available, takes $2 \log kp$ time using binary search on the kp sized array that holds the new extent of each bucket. Once identified, processing is done in order to combine information from parts of buckets spread across processors. This is done using a global segmented-sum-scan with vectors of size n/kp. Each segment represents a stretched bucket. The only problems that arise are at processors that have a succeeding and a preceding partial bucket (such as intermediate processors P_1 and P_2 in the example in figure 8), since this implies that such a processor should be part of two segments. This problem is taken care of by temporarily ignoring the preceding partial buckets in such processors, and completing the segmented scan. The results of the scan are augmented by having each processor with ignored preceding partial buckets send its contents to the processor where that bucket started. This is done by simple one-to-one communications with no node contention using vectors of size n/kp. The segmented-scan and the multiple one-to-one communications take $O(\log p)\tau + O(n/kp)\mu$ time combined and the constants involved are very small.

4.4 Communication Between Intermediate and Destination Processors

The communication between the intermediate and destination processors is again a bounded transportation problem, similar to the communication that underlies a monotonic dynamic permutation [24]. Figure 8 shows the second stage's communication for the example introduced in figure 4. Consider a bucket B_b that is spread across many processors. A processor that contains B_b as a preceding partial bucket does not send out any elements of B_b. A processor that contains B_b as a succeeding partial bucket sends out all the elements of B_b. This amounts to $\lceil n/kp \rceil$ elements, since partial buckets are always stored using the dense representation. Due to this, the upper bound on the number of elements sent out of any intermediate processor is $\lceil n/p \rceil + \lceil n/kp \rceil$. The upper bound on the number of elements received at any destination processor is $\lceil n/p \rceil$ since each of the k buckets at a destination receives no more than n/kp elements. This communication can be completed in $\mu(n/p + n/kp)$ time [24]. The algorithm for performing the communication is very similar to the algorithm presented in figure 11 later in this paper.

4.5 Time taken by the RAW Algorithm

- The time taken for pre-processing at the source processor is $O(n/p+kp)\delta + O(\log p)\tau + O(kp)\mu$ since it involves sequential passes through arrays of size n/p and kp, and a global prefix-scan and a global combine with an array of size kp.

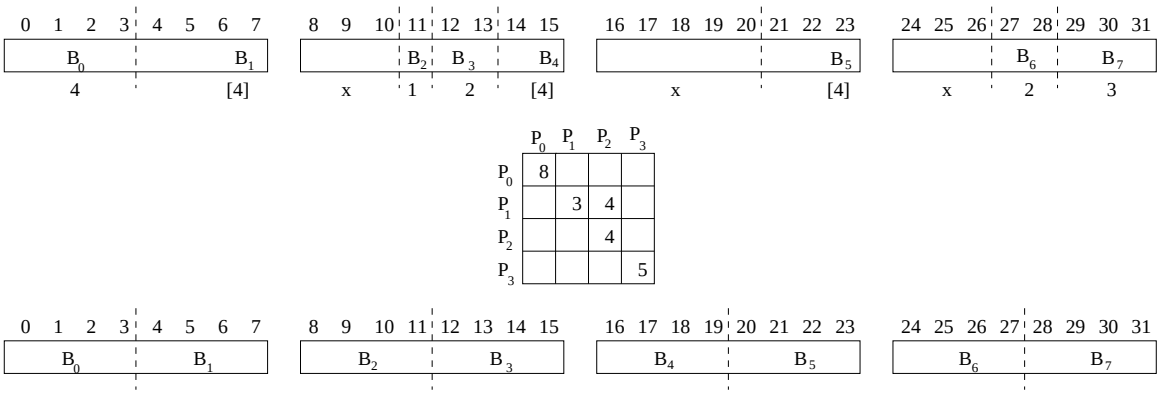

Figure 8: Communication between Intermediate and Destination Processors

- The communication between the source and intermediate processors takes $2p(\tau + \mu(n/p^2 + p))$ time, that is, $2\mu n/p$ time when $n \geq O(p^3 + p^2\tau/\mu)$.
- Processing of full buckets at the intermediate processors takes $O(n/p + kp)\delta$ time, while the processing of partial buckets takes an additional $O(\log p)\tau + O(n/kp)\mu$ for the prefix-scan with an array of size n/kp.
- The time taken for communication between the intermediate and destination processors is $p\tau + \mu(n/p + n/kp)$.
- Rearranging elements at the destination processors needs $O(n/p)\delta$ time.

Overall, the two-stage RAW algorithm takes $O(n/p + n/kp + kp)\delta + O(p + \log p)\tau + O(n/kp + kp)\mu + 3\mu n/p$ time. We choose the number of buckets k per processor to satisfy $n/p \geq n/kp$ and $n/p \geq O(kp)$, that is, $1 \leq k < p$ and $n \geq O(kp^2)$. There is a lot of leverage in choosing the value of k. k being close to 1 reduces the time for the prefix-scan/combine with the array of size kp, but increases the time for the prefix-scan with the array of size n/kp. k being close to p does the opposite. An intermediate value of k brings down the time taken by the prefix-scans, leaving $3\mu n/p$ (+ lower order terms) as the time taken by the two-stage RAW algorithm. The algorithm imposes a restriction of $n \geq kp^2$ while the underlying transportation primitive requires $n \geq O(p^3 + p^2\tau/\mu)$ for optimality.

5 A Hybrid RAW Algorithm

A hybrid algorithm, incorporating features from both the limited hot spots RAW algorithm and the arbitrary hot spots RAW algorithm, is illustrated through an example in figure 9. First, elements whose final destination is the source processor itself, are not unnecessarily sent to intermediate processors and received back. This transforms the communication matrix in figure 4 to the one showed in figure 9(i). Second, buckets that are sparse are not processed using the two-stage algorithm

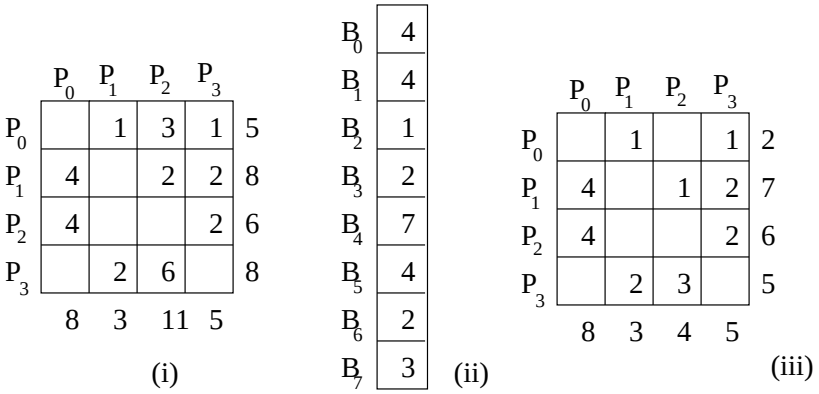

Figure 9: The Hybrid RAW Algorithm

for the following reason: Sparse buckets, by definition, do not receive more than n/kp elements. Each destination processor has k buckets initially, and therefore, receives no more than n/p elements for its sparse buckets. In any RAW, no source processor sends out more than n/p elements. Thus, the processing of sparse buckets alone is a bounded transportation problem and can be completed in $2\mu n/p$ time when $n \geq O(p^3 + p^2\tau/\mu)$. Figure 9(ii) shows the number of elements received by each bucket. In the example, all buckets receiving 4 or less elements are sparse. Figure 9(iii) shows that the processing of sparse buckets has bounded incoming and outgoing traffic at every processor. The processing of dense buckets is done through the two-stage algorithm.

The processing of sparse buckets and the first stage in the processing of dense buckets can be combined by sending out elements from the source processors to both the intermediate and the destination processors together. In this case, the traffic bound at the source processors is n/p while that at the receiving (intermediate/destination) processors is $2n/p$. Communication can be completed in $\mu(n/p + 2n/p)$ time. The predetermined contention at the sparse buckets can be used to modify the initial stretching of buckets such that the traffic the bound at the receiving processors is brought down to n/p. Communication for the first stage of the RAW now takes only $2\mu n/p$ time (Optimizations described in [24] can also be applied to the first stage). Thus the RAW algorithm retains the small constants associated with the communication time even when the sparse buckets are not processed using the two-stage algorithm.

Another advantage in processing sparse buckets separately is that the incoming and outgoing traffic bounds for the communication in the second stage is likely to be lower in practical implementations because of the following reasons:

- The total outgoing traffic at any processor is bounded by n/p minus the traffic associated with sparse buckets.
- The maximum outgoing traffic is proportional to the number of hot spot buckets handled by any intermediate processor. If the number of hot spot buckets is small, the outgoing traffic from any intermediate processor would also be small.

- The maximum incoming traffic is proportional to the number of hot spot buckets owned by any destination processor. If the number of such hot spot buckets is small, the incoming traffic at any destination processor would also be small.

6 Monotonic RAW

A special case of the RAW primitive arises when the destinations $P(i)$ for element i are in sorted order. This type of RAW can be performed in time $q\tau + \mu n/p$ (+ lower order terms) on a virtual crossbar, where q is the maximum number of processors that any processor communicates with ($q < p$). In a monotonic RAW, collisions can be resolved using a segmented scan initially. After this scan the number of elements sent out or received by any processor is upper-bounded by $\lceil n/p \rceil$.

The property that the destinations $P(i)$ in all the sending processors are sorted to begin with, is first used to divide the elements in the sending processors into segments. Each segment contains all elements i for which $P(i)$ is identical. A segmented scan is performed using the given RAW collision resolving operator. In each segment, all elements but the last need not participate in the RAW any more. The elements are now resegmented, with each segment now defined as all the remaining elements sending data to the same processor. Consider a single sending processor with s elements ($s \leq \lceil n/p \rceil$). These s elements could be divided into as many as q segments. The segments in this processor (see figure 10) 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. The steps in the algorithm for a monotonic RAW are outlined in figure 11. The initial element-wise segmentation takes $\delta n/p + \tau \log p$ time. The remainder of the steps can be performed in $\tau q + \mu n/p$, since each processor sends out no more than q messages and these q messages contain a total of n/p elements. Therefore, a monotonic RAW can be performed in $\tau q + \mu n/p$ time.

The *generalize* primitive [22] shown in figure 12, can also be done using an algorithm similar to the monotonic RAW algorithm. Here, $D(i)$, the data from element i ($0 \leq i < n$) has to be sent to elements $P(i-1)+1$ to $P(i)$ (assume $P(-1) = -1$). Two companion primitives, the *concentrate* and the *distribute* also work with sorted destinations but, unlike generalize, perform only one-to-one mappings. These primitives were originally defined in [22] and algorithms for these for a coarse-grained parallel machine were presented in [24].

On a coarse-grained architecture, the generalize primitive involves each of the sending processors sending out either $\lceil r/p \rceil$ or $\lfloor r/p \rfloor$ elements. The total number of elements sent out $r \leq n$. Each of the receiving processors receives no more than $\lceil n/p \rceil$ elements from no more than q processors ($q < p$). Unlike the monotonic RAW, the generalize primitive does not have any collisions. Therefore, the element-wise segmented scan that was performed at the start of the monotonic RAW algorithm is not needed. On the other hand, generalize involves multiple destination elements receiving copies

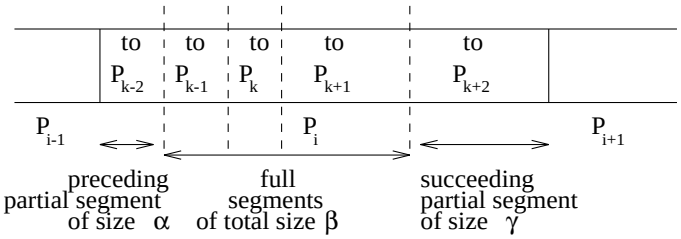

Figure 10: Segments in a processor during the Monotonic RAW algorithm

---

Monotonic RAW

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

Define a segment to be all the elements sending data to the same destination element. Perform a segmented scan (upward, inclusive) using the given RAW collision resolving operator. In each segment, all elements but the last, need not participate in the RAW any more.

Redefine a segment to be all the remaining elements sending data to the same processor.

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 ($r_{prec} \leq \lceil n/p \rceil$).

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 11: Algorithm for Monotonic RAW

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

Figure 12: The Generalize Primitive

of the same element. For this reason, an element-wise segmented copy-scan is performed at the end of the generalize algorithm. For this segmented scan, segment i denotes elements $P(i-1)+1$ to $P(i)$. On the virtual crossbar, the algorithm for generalize takes $q\tau + \mu n/p + \tau \log p + \delta n/p$, that is, $q\tau + \mu n/p$ time (assuming $q\tau$ is larger than $\tau \log p$).

7 Changes Required for RAR

Algorithms for the RAR primitive can be designed based on the RAW algorithms presented in the previous sections. RAR does not involve collisions between elements. However, in the case of multiple elements reading from the same source element there could be hot spots. An extreme case is the broadcast primitive, where each of the n elements needs data from a single source element. The RAR could involve limited hot spots, in which case inter-processor communication may not be required or communication can be done using a complete exchange or bounded transportation. With arbitrary hot spots, a two-stage algorithm can be used. A hybrid algorithm incorporating features from both, the limited hot spots and the arbitrary hot spots cases, would be useful in practice.

The RAW algorithm involved *source* processors writing data to *destination* processors. In a RAR algorithm it's again the source processors sending their data to destination processors, but that occurs only after the destination processors have requested them for the elements needed. The RAR algorithm hence goes through twice the kind of communication that a RAW goes through. However the first communication from source to destination processor involves the sending of just addresses, while the second communication involves the sending of the elements. If the data elements are of size m and the addresses of size 1, the first communication takes $3(p\tau + \mu n/p)$ time, while the second takes $3(p\tau + \mu mn/p)$ time. Thus, the RAR algorithm takes $6p\tau + 3\mu(m+1)n/p$ time when $n \geq O(p^3/m + p^2\tau/\mu m)$. In comparison, the RAW algorithm took $3p\tau + 3\mu mn/p$ time under similar restrictions on n.

The time required for a RAR can be reduced to $5p\tau + \mu(3m+2)n/p$ by replacing the two way communication between the source and intermediate processors by a one way communication from the source to the intermediate processors with elements of size m. This eliminates the sending of requested element addresses from the intermediate to source processors, reducing time taken by $p\tau + \mu n/p$.

8 CREW/CRCW PRAM Simulation

The PRAM is a shared-memory parallel programming model that has been widely used for the design of parallel algorithms [13]. It is an abstract model which does not differentiate between the cost of unit computation and unit communication. By simulating a PRAM on a more realistic (but still sufficiently general) model, an efficient and reasonably architecture-independent implementation of shared memory on distributed memory machines can be provided. The various algorithms presented

in the companion paper [24] described the simulation of the exclusive read and exclusive write capabilities. The algorithms presented in this paper describe the simulation of the concurrent read and concurrent write capabilities. The PRAM simulation is communication-efficient and is optimal provided the size of the PRAM $n \geq O(p^3 + p^2\tau/\mu)$.

9 Conclusions

In this paper we have presented communication-efficient algorithms for performing random data accesses involving hot spots on a coarse-grained parallel machine. This was done through the design of algorithms for the general random access write/read (RAW/RAR) primitives and for special cases of the same. The algorithms for RAW were described in detail, and modifications required for the RAR were outlined.

The main contribution of this paper is the presentation of algorithms that perform random data accesses in time independent of the presence and severity of the hot spots. This is achieved by dynamically stretching and shrinking the hot spots as necessary. Further, the constants involved in the communication time complexity are very small. This is a necessary requirement for effective utilization of typical coarse-grained machines where the number of processors is small and communication overheads are high. An important feature of the algorithms presented is that they are relatively architecture independent and can be efficiently implemented on a wide variety of interconnection networks.

We believe that the above primitives will be of crucial importance in the implementation of data parallel constructs such as those found in High Performance Fortran [1]. These primitives are also important for the implementation of the BSP model [27] on coarse-grained machines. The parallelization of data parallel applications on coarse-grained machines have been limited to regular applications or irregular applications with relatively static structure and limited hot spots. We believe that the results presented in this paper are the first steps towards implementing dynamic data parallel applications on coarse-grained machines. Another important requirement for efficient data parallel implementations is the exploitation of structural as well as spatial locality in applications. Preliminary results for exploiting spatial locality for adaptive and irregular applications have been presented in [18, 19, 20]. The H-PRAM model [10, 11] presented a framework for the exploitation of structural locality. Structural locality has also been exploited in nested data parallel languages such as NESL [4] and in Proteus [16]. Our algorithms for parallel random data accesses were designed to incorporate such additions in the immediate future.

References

[1] Seungjo Bae, Sanjay Ranka, Ravi V. Shankar. The HARD Primitive - Scalable Parallelization of the FORALL statement and the gather/scatter Primitives in HPF, (in preparation).

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

[3] S. Bhatt, M. Chen, C. Lin, and P. Liu. Abstractions for parallel n-body simulation, Scalable High-Performance Computing Conference SHPCC, 1992.

[4] G. E. Blleloch. NESL: A nested data parallel language, Technical Report CMU-CS-93-129, Carnegie Mellon University, Jan. 1992.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] T. Heywood and S. Ranka. A Practical Hierarchical Model of Parallel Computation: The Model, *Journal of Parallel and Distributed Computing*, November 1992, vol. 3, pp. 212–232.

[11] T. Heywood and S. Ranka. A Practical Hierarchical Model of Parallel Computation: Binary Tree and FFT Graph Algorithms, *Journal of Parallel and Distributed Computing*, November 1992, vol. 3, pp. 233–249.

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

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

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

[15] 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.

[16] P. Mills, L. Nyland, J. Prins, J. Reif, and R. Wagner. Protoyping Parallel and Distributed Programs in Proteus, *Proceedings of Symposium on Parallel and Distributed Processing*, 1991.

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

[18] Chao-Wei Ou and Sanjay Ranka. Parallel Remapping Algorithms for Adaptive Problems, *Frontiers '95*. To appear.

[19] Chao-Wei Ou and Sanjay Ranka. Parallel Incremental Graph Partitioning Using Linear Programming, *Supercomputing '94*, November 1994. To appear.

[20] Chao-Wei Ou, Sanjay Ranka, and Geoffrey Fox. Fast Mapping And Remapping Algorithm For Irregular and Adaptive Problems, *Proceedings of the 1993 International Conference on Parallel and Distributed Systems*, Taipei, Taiwan, December 1993.

[21] Victor K. Prasanna, Cho-Li Wang, Scalable Data Parallel Object Recognition using Geometric Hashing on the CM-5. Scalable High Performance Computing Conference, SHPCC, 1994.

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

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

[24] Ravi V. Shankar, Sanjay Ranka. Random Data Accesses on a Coarse-Grained Parallel Machine - I. One-to-one Mappings, CIS Technical Report, Syracuse University, October 1994.

[25] Ravi V. Shankar, Sanjay Ranka. Histogramming based Algorithms on a Coarse-grained Parallel Machine, (in preparation).

[26] J. P. Singh. *Parallel Hierarchical N-body Methods and Their Implications for Multiprocessors*, Ph.D. thesis, Stanford University, 1993.

[27] L. G. Valiant, A Bridging Model for Parallel Computation, *Communication of the ACM*, vol. 2, No. 8, 1990, pp. 103–111. *SIAM Journal of Sci. and Stat. Computation*, 1991.

[28] M. Warren and J. Salmon. Astrophysical N-body simulations using hierarchical tree data structures, *Supercomputing*, 1992.