The Transportation Primitive

Khaled Alsabti
Sanjay Ranka
Ravi Shankar

CRPC-TR94529-S
August 1994

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

The Transportation Primitive *

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

August 1994

Abstract

This paper presents algorithms for implementing the transportation primitive on a distributed memory parallel architecture. The transportation primitive performs many-to-many personalized communication with bounded incoming and outgoing traffic. We present a two-stage deterministic algorithm that decomposes the communication with possibly high variance in message size into two communication stages with low message size variance. If the maximum outgoing or incoming traffic at any processor is t, transportation can be done in $2t\mu$ time (+ lower order terms) when $t \geq O(p^2 + p\tau/\mu)$ (μ is the inverse of the data transfer rate, τ is the startup overhead). If the maximum outgoing and incoming traffic are r and c respectively, transportation can be done in $(r+c)\mu$ time when either $r \geq O(p^2)$ or $c \geq O(p^2)$. Optimality and scalability are thus achieved when the traffic is large, a condition that is usually satisfied in practice. The algorithm was implemented on the Connection Machine CM-5. The implementation used the low latency communication primitives (active messages) available on the CM-5, but the algorithm as such is architecture-independent. An alternate single-stage algorithm using distributed random scheduling was implemented on the CM-5 and the performance of the two algorithms were compared.

*A preliminary version of this paper titled *Many-to-many Personalized Communication with Bounded Traffic* is to presented at Frontiers '95,Mclean, Virginia, February 1995.

1 Basic Communication Primitives

Communication between processors on a parallel machine can generally be described as x-to-y communication where x and y can be substituted by *one*, *all*, or *many*. "Communication" implies processors sending and receiving messages: x being *one*, *all*, and *many* respectively, indicates that only one of the p processors sends data, that all processors send data, and that only some processors send data. Similarly, y being *one*, *all*, and *many* indicate that from each of the senders, one, all, and many processors receive data respectively. Communication can be further distinguished as a *broadcast/accumulation* or as *personalized communication*. For example, one-to-all communication could be either a one-to-all broadcast (*single-node broadcast*) where a single processor sends out the same message to all processors, or a one-to-all personalized communication (*single-node scatter*) where a single processor sends out different messages to each processor. This classification is fairly standard in the literature. See, for instance, [7]. Algorithms for performing broadcasts are comparatively easier than those for performing personalized communication. All further discussion in this paper deals with personalized communication. Communication with multiple senders and multiple receivers is also referred to as *collective* communication.

2 Collective Communication Parameters

Any type of communication in a machine with p processors can be represented using a communication matrix, a p x p matrix M where the addresses of the sending and receiving processors are used as row and column indices. The matrix entry m_{ij} denotes the size of the message being sent by processor P_i to processor P_j. The rows of the matrix are called send vectors and the columns are called receive vectors. The outgoing traffic r_i is the sum of the sizes of the messages being sent by processor P_i, while the incoming traffic c_j is the sum of the sizes of the messages being received by processor P_j. The outgoing traffic bound r is the maximum outgoing traffic at any processor, and the incoming traffic bound c is the maximum incoming traffic at any processor. The overall traffic bound t is the maximum traffic, incoming or outgoing, at any processor.

$$r_i = \sum_j m_{ij} \qquad c_j = \sum_i m_{ij}$$

$$r = \max_i \ r_i \qquad c = \max_j \ c_j \qquad t = maximum(r, c)$$

The fan-out f_i is the number of messages sent by processor P_i, while the fan-in g_j is the number of messages received by processor P_j. The fan-out bound f is the maximum fan-out at any processor, and the fan-in bound g is the maximum fan-in at any processor. The overall fan-in/fan-out bound h is the maximum number of messages, being sent or received, at any processor.

$$f_i = \sum_j sgn(m_{ij}) \qquad g_j = \sum_i sgn(m_{ij})$$

$$f = \max_i \ f_i \qquad g = \max_j \ g_j \qquad h = maximum(f, g)$$

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	
P_0		3	1		2	1	2	1	10
P_1	1		2	2	1	1	3		10
P_2	4	1		2		2		1	10
P_3		2				3	1	4	10
P_4	3		4			2	1		10
P_5	1	2	1	2				4	10
P_6		2	1		7				10
P_7	1		1	4		1	3		10
	10	10	10	10	10	10	10	10	

Figure 1: A matrix illustrating all-to-many communication with equal traffic

The *sgn* function returns +1,0,-1 depending on whether its argument is positive, zero, or negative.

The relation between the parameters just defined and the different kinds of collective communication is as follows. If $f_i = p$ for all i ($0 \leq i < p$), the communication is *all-to-all*. This also implies that $g_j = p$ for all j ($0 \leq j < p$). If $f_i > 0$ for all i, the communication is *all-to-many*. If $f_i = 0$ or $f_i = p$ for each i, and $g_j > 0$ for each j, the communication is *many-to-all*. The general case, where $f_i \geq 0$ for all i is *many-to-many* communication. Collective communication can be further classified based on the sizes of the messages being sent and received. Messages could be *uniform* (of the same size) or *non-uniform* (of different sizes). The variance in message size is an important factor that affects the performance of an algorithm for collective communication. Most algorithms presented in the literature deal only with all-to-all communication with uniform message sizes.

3 The Transportation Primitive

The transportation primitive is a general communication primitive that performs many-to-many personalized communication in which message sizes could be highly non-uniform. It encompasses all the basic communication primitives outlined in the last two sections.

In the most general case, the transportation problem could have differing incoming and outgoing traffic at every processor. The incoming and outgoing traffic bounds determine the amount of time the transportation takes. The transportation problem is illustrated in the communication matrices shown in figures 1 and 2. The entry beyond the right margin of row P_i is the outgoing traffic r_i, while the entry below column P_j gives the incoming traffic c_j. Figure 1 illustrates the case where the incoming and outgoing traffic at each processor is equal to the overall traffic bound, a special case which will be considered in the description of the algorithms. Figure 2 illustrates a case where the incoming and outgoing traffic at each processor are not equal, and the incoming and outgoing traffic bounds are different.

Transportation with an overall traffic bound of t, illustrated in figure 3, cannot be done in time less

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	
P_0		3	1			1		1	6
P_1	1		2	2		1	1		7
P_2	1	1		2		2		1	7
P_3		1				3	1		5
P_4	3					2	1		6
P_5	1	1	1	2				2	7
P_6									0
P_7	1		1	4		1			7
	7	6	5	10	0	10	3	4	

Figure 2: A matrix illustrating many-to-many communication with bounded traffic

than $O(t)$. When the outgoing and incoming traffic bounds r and c are different, transportation cannot be done in time less than $O(r + c)$. The two-stage algorithms presented in this paper achieve these times and are optimal under large traffic conditions. Bounded transportation appears in a wide variety of parallel algorithms such as matrix transpose on a rectangular grid, in the final phase of sorting algorithms like sample sort, in transformations between any two distributions (like block, cyclic, and block-cyclic) that distribute data equally among all processors, etc. We are using them for performing dynamic permutations [11] and for dealing with highly irregular data accesses involving hot-spots [12] on coarse-grained parallel machines.

4 CM-5 System Overview

4.1 Node/Network Architecture

The Connection Machine Model CM-5 [13] is a synchronized MIMD distributed-memory parallel machine available in configurations of 32 to 1024 processing nodes. Each node contains a 33 MHz SPARC microprocessor with 32 megabytes of memory, and is rated at 22 Mips and 5 Mflops. Four optional floating-point vector units can be added to each node, and this increases the node's peak performance to 128 Mips and 128 Mflops.

The CM-5 interconnection network has three components: a data network, a control network, and a diagnostic network. The data network has a fat-tree topology and provides high-performance data communication between the system components. The network has a peak bandwidth of about 5 megabytes per second for node-to-node communication. However, if the destination is within the same same cluster of 4 or 16 nodes in the fat-tree, a peak bandwidth of 20 megabytes per second and 10 megabytes per second, respectively, can be achieved [13]. The control network handles operations requiring the cooperation of many or all processors. This includes broadcasting, combining, and global operations. The diagnostic network helps in the detection and isolation of errors throughout the system.

	P_0	P_1	P_2	P_3	
P_0	1	5		3	9
P_1	2		5	2	9
P_2		4	1	4	9
P_3	6		3		9
	9	9	9	9	

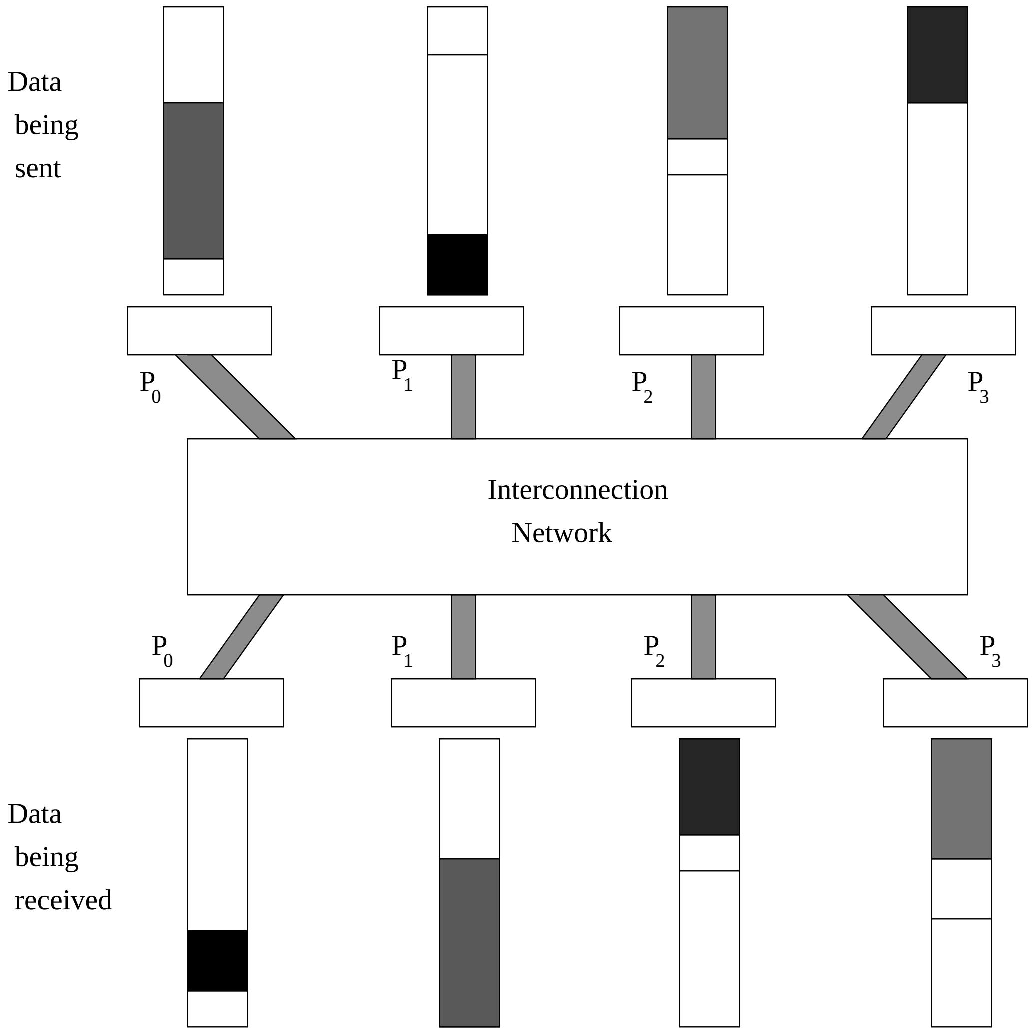

Figure 3: The Bounded Transportation Problem

Both, the control network and the diagnostic network, have a binary tree topology.

Our implementations were performed on a 32-node CM-5 using active messages for low latency communication. Each 20-byte active message packet can carry up to 16 bytes of payload. Sending and receiving a single-packet active message on the CM-5 takes 1.6 μs and 1.7 μs respectively [6]. We used the CMMD message passing library and CMAML (the CMMD active messages layer) [14]. Two other implementations of active messages on the CM-5 exist: the original CMAM library [6] from UC Berkeley and the Strata library from MIT [3].

4.2 Modeling the CM-5

- **Sending a Message**
  The time taken to send a message from one node on the CM-5 to another can be modeled as $O(\tau + \mu M)$, where τ is the startup overhead, μ is the inverse of the data transfer rate and M is the size of the message. As mentioned earlier, the value of μ depends on whether the destination belongs to a specific subgroup and whether other nodes are sending messages. For our complexity analysis we will assume that τ and μ are constant, independent of the congestion and distance between two nodes.

- **Global Combine**
  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 (also referred to as the global reduce 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.

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

  This operation can be completed in $O(\phi_2 \frac{n}{p})$ time on the CM-5, where ϕ_2 is a small constant [2].

- **Global Vector Scan**
  Let each processor contain a vector $V_i[0 \cdots \frac{n}{p} - 1]$. The global vector prefix-sum-scan operation computes an element-wise prefix-sum-scan of the local list in each processor. The resultant vector $R[0 \cdots \frac{n}{p} - 1]$ in processor q $(0 \le q < p)$ is given by:

  $$R[j] = \sum_{i=0}^{q} V_i[j]$$

  This operation can be completed in $O(\phi_3 \frac{n}{p})$ time on the CM-5, where ϕ_3 is a small constant.

5 Collective Communication with Low Message Size Variance

The simplest version of collective communication involves all processors exchanging messages of the same size s. This is all-to-all personalized communication with uniform messages. Under these conditions, a linear permutation algorithm [1] can be used to perform the communication. A linear permutation algorithm goes through $p - 1$ iterations, and in iteration k processor P_i $(0 \le i < p, 0 < k < p)$

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 4: The Modified Linear Permutation Algorithm

exchanges data with processor $P_{i \oplus k}$ ($\oplus$ is the bitwise exclusive OR operator). The time complexity of linear permutation is $O(sp)$.

A slightly modified linear permutation algorithm can be used when the messages are not uniform but exhibit only a small variation in size. Here, processors no longer send messages of exactly the same length. Instead they exchange send and receive vectors, and exchange only messages of the required lengths. The algorithm is shown in figure 4, where $sendl^j$ is the size of the message sent to processor P_j (from P_i) and $recvl^j$ is the size of the message received from processor P_j (by P_i). The implicit synchronization in the linear permutation algorithm is replaced by an explicit barrier synchronization, and the algorithm retains the deterministic time complexity of O(sp) where s is the upper bound on the sizes of the messages exchanged. The barrier also prevents the communication network from getting congested and this has been shown to improve performance [4]. This is the algorithm referred to when "linear permutation" is mentioned in the rest of this paper.

6 Collective Communication with High Message Size Variance

Dealing with communication in which message sizes show a large variation is a difficult problem. A linear permutation algorithm could take as much as O(tp) time. Sorting messages by size is not guaranteed to improve performance either. We use a distributed random scheduling algorithm using spin locks to deal with such a situation. The distributed scheduling algorithm [15] was chosen over other graph based techniques because its low overhead enables scheduling to be done dynamically.

The algorithm is presented in figure 5. Each processor maintains a status bit that indicates whether the processor is busy or free. Processors which have messages to send perform a test-and-set operation to determine whether the intended destination is free. If the destination is free, its status bit is set to busy, and data is transferred as a single message. If the destination is busy, the sending processor tries

Distributed_Random_Scheduling

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;

Pre-allocate receiving buffers according to receive vector *recvl*;

Repeat

Select a destination node from send vector *sendl*, use active messages to test-and-set destination node's busy_lock;

If the destination node is free to receive message,

Send message to the destination node;

Upon completion, reset destination node's busy_lock to free;

Reset the corresponding entry in send vector *sendl*;

Until send vector *sendl* is empty

Wait until all incoming messages arrive at their proper buffers.

Figure 5: The Distributed Random Scheduling Algorithm

another intended destination using the same procedure. The test-and-set inquiry operation is shown in figure 6.

We re-implemented the distributed scheduling algorithm using active messages on the CM-5. Two improvements were incorporated into the new implementation, which also replaced the interrupts in the earlier implementation with polling. First, a busy destination processor when replying to the sender of an inquiry gives a measure of how busy it is. The sender notes down this measure and makes sure that the destination will not be disturbed for this measure of time. If the sender receives busy signals from all the intended destinations, it goes to sleep for the amount of time indicated by the minimum of the measures returned by the destinations. The second improvement allowed busy destination processors to give the go-ahead for a new message transfer when the current message transfer is about to get over.

7 Two-stage Algorithm for the Bounded Transportation Problem

We have developed a two-stage algorithm that decomposes the transportation problem involving communication with high message size variance, into two communication stages with low message size variance. In the general case, the fan-out and fan-in at each processor is less than or equal to p and the traffic bound is t. Results are given separately for the equal traffic case, where the incoming traffic and the outgoing traffic at each processor is exactly equal to the overall traffic bound t. Each processor takes on three roles in this two-stage algorithm. First, each processor P_i for which the fan-out f_i is non-zero

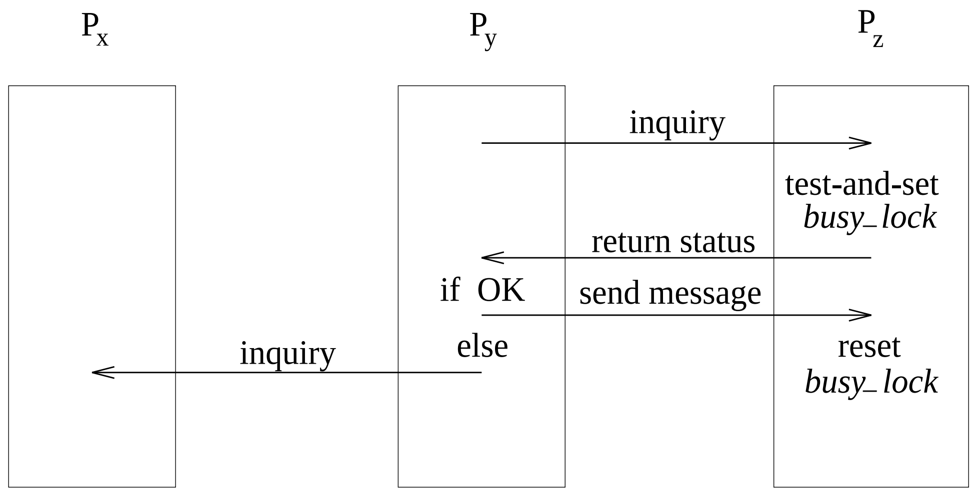

Figure 6: The inquiry operation in Distributed Scheduling

acts as a *source* processor, sending out data during the first stage. Second, each processor participates as an *intermediary*, receiving data during the first stage, and sending data during the second stage. Third, each processor P_j for which the fan-in f_j is non-zero acts as a *destination* processor, receiving data during the second stage. The organization of data in the source, intermediate and destination processors is shown in figure 7.

7.1 The First Stage

Local pre-processing

In source processor P_i $(0 \leq i < p)$ let a_{i0},a_{i1},...,$a_{i(p-1)}$ be the number of elements being sent to destination processors P_0, P_1,...,P_{p-1} respectively. In stage 1, each of the a_{ij} elements is divided into p parts (each of size either $\lceil a_{ij}/p \rceil$ or $\lfloor a_{ij}/p \rfloor$) to be sent to processors P_0 to P_{p-1}. [1] At the end of stage 1, processor P_k acting as an intermediary could receive messages of size up to $t/p + p$, since

$$\sum_{i=0}^{p-1} \lceil a_{ik}/p \rceil \ \leq \ \sum_{i=0}^{p-1} a_{ik}/p + p \ \leq \ c_k/p + p \ \leq \ t/p + p$$

The lower bound for the message size is 0, unless we are dealing with the equal traffic case, when the lower bound becomes $t/p - p$, since

$$\sum_{i=0}^{p-1} \lfloor a_{ik}/p \rfloor \ \geq \ \sum_{i=0}^{p-1} a_{ik}/p - p \ \geq \ c_k/p - p \ \geq \ t/p - p$$

Our goal is to achieve communication with low variance in message sizes for both stages. A simple change in the algorithm can achieve the balance we desire for the first stage. At any processor P_i,

[1]In reality, this is only $p-1$ messages, since one of the messages is to be sent to the sending processor itself. Our implementations take this into account, but this paper, for the sake of simplicity, continues to refer to p as the number of messages being sent out.

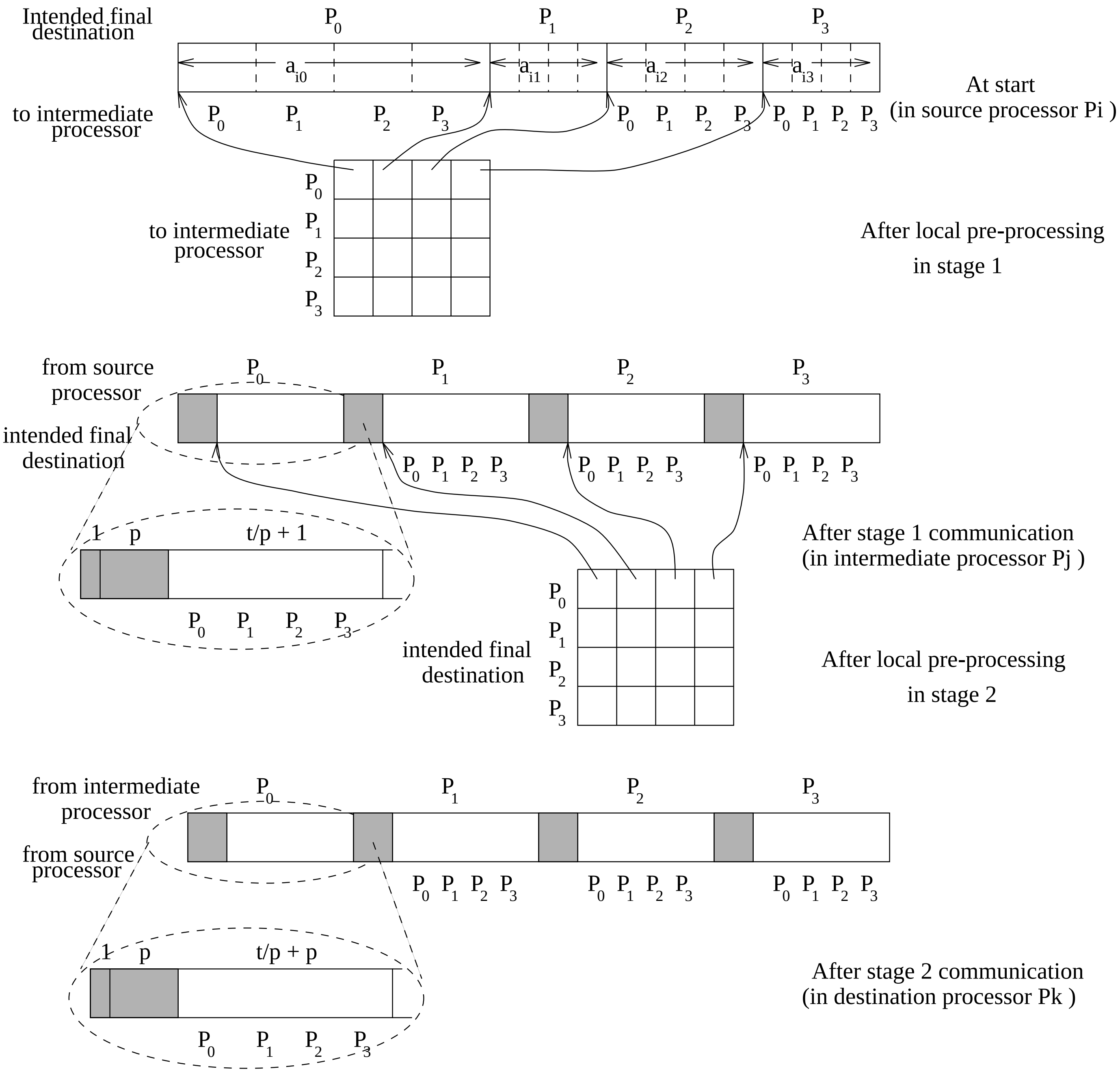

Figure 7: Organization of Data in the Two-stage Algorithm

when dividing the a_{ij} elements into p parts for sending, the last a_{ij} mod p elements are assigned to the p intermediate processors in a round robin fashion. This ensures that each intermediate processor P_k receives messages of size no more than $\lceil t/p \rceil$. In the equal traffic case, the message sizes can vary by only one element since the smallest message size is $\lfloor t/p \rfloor$.

Figure 8 gives the details of algorithm used for local pre-processing in stage 1. The overall time required is $O(p^2)$.

Communication

In an initial version of the implementation, local reshuffling was done at the source processors in order to get all the data elements being sent to the same intermediate processor into contiguous memory locations. Such reshuffling gets prohibitively expensive when t is large. Our current implementation requires that the communication routines take as arguments pointers to p memory locations in the source processor and p associated lengths for each message being sent, as shown in figure 7. Note that this does not increase the communication startup latency by a factor of p.

In the equal traffic case, since the communication is balanced with message lengths differing by just one element, linear permutation works best. In the general case, distributed scheduling for the first stage's communication may perform better, but linear permutation gives an upper bound on the time taken for communication. A maximum of p messages of length no more than $\lceil t/p \rceil$ may need to be sent. In addition, each of these messages has to be padded with p lengths (and the sum of these p lengths, see figure 7) to help the intermediate processor determine the message portions to be sent to each destination processor. The time required for communication is $O(p(\tau + \mu(t/p + p)))$.

Figure 9 illustrates the two-stage algorithm through an example, showing how messages from a particular source processor to a particular destination processor are split and sent through the intermediate processors. This example uses an additional step to ensure that the total length of the messages reaching an intermediate processor in stage 1 is not greater than $p\lceil t/p \rceil$, that is, $t + p$. In the algorithm description above the total length was upper-bounded by $t + p^2$. (However, if $t/p < O(p)$, communication in stage 1 takes only $O(p(\tau + \mu(t/p + 1)))$, as explained later. The additional step reduces the total length of messages reaching an intermediate processor from $t + p$ to t). This additional step involves a global prefix-sum-scan on the quantity $(\sum_{j=0}^{p-1} a_{ij})$ mod p in each processor P_i. The result of the scan indicates the intermediate processor at which the round-robin assignment of excess elements should begin. If the cost of a prefix-sum-scan is less than the savings obtained through the tighter bound on the total length of the messages, the additional step should be used.

7.2 The Second Stage

Local pre-processing

At the intermediate processors, each of which receives p messages, local pre-processing is done as preparation for the second stage. An initial implementation performed reshuffling. Our current implementation sets up, for each message sent out in the second stage, two arrays containing p pointers and

```
procedure Stg1lpp(sendl, send_msg_start, send_msg_len)
/* This is code that runs in every processor.
 * sendl[0..P-1] is the send vector
 * index j gives destination processor #, index k gives intermediate proc #
 *
 * send_msg_start[0..P-1][0..P-1] gives the index of the element from the
 * input array marking the start of each of the P parts of the P
 * messages sent out from this source processor;
 *
 * send_msg_len[0..P-1][0..P] gives the length of those parts; in
 * particular, the entry send_msg_len[0..P-1][0] gives the total
 * length of messages to each intermediate processor
 */
begin

  for j := 0 to P-1 do
    for k := 0 to P-1 do
       send_msg_len[k][j+1] := sendl[j] div P;
        /* (sendl[j] div P) is the # of elements originally meant for
            processor j now being sent to every intermediate processor */

  k := 0;
  for j := 0 to P-1 do
    for x := 1 to (sendl[j] mod P) do
    /* (sendl[j] mod P) is the # of elements meant for destination processor
       j that could not be divided equally among the intermediate processors */

      begin
        send_msg_len[k][j+1] := send_msg_len[k][j+1] + 1;
        k := (k+1) mod P;
      end;

  data_ptr := 0;
  for j := 0 to P-1 do
    for k := 0 to P-1 do
      begin
        send_msg_start[k][j] := data_ptr;
        data_ptr := data_ptr + send_msg_len[k][j+1];
        send_msg_len[k][0] := send_msg_len[k][0] + send_msg_len[k][j+1];
        /* send_msg_len[k][0] is current message size for interm. proc k */
      end

end;
```

Figure 8: Local Pre-processing in Stage 1

destination processor

source processor	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	

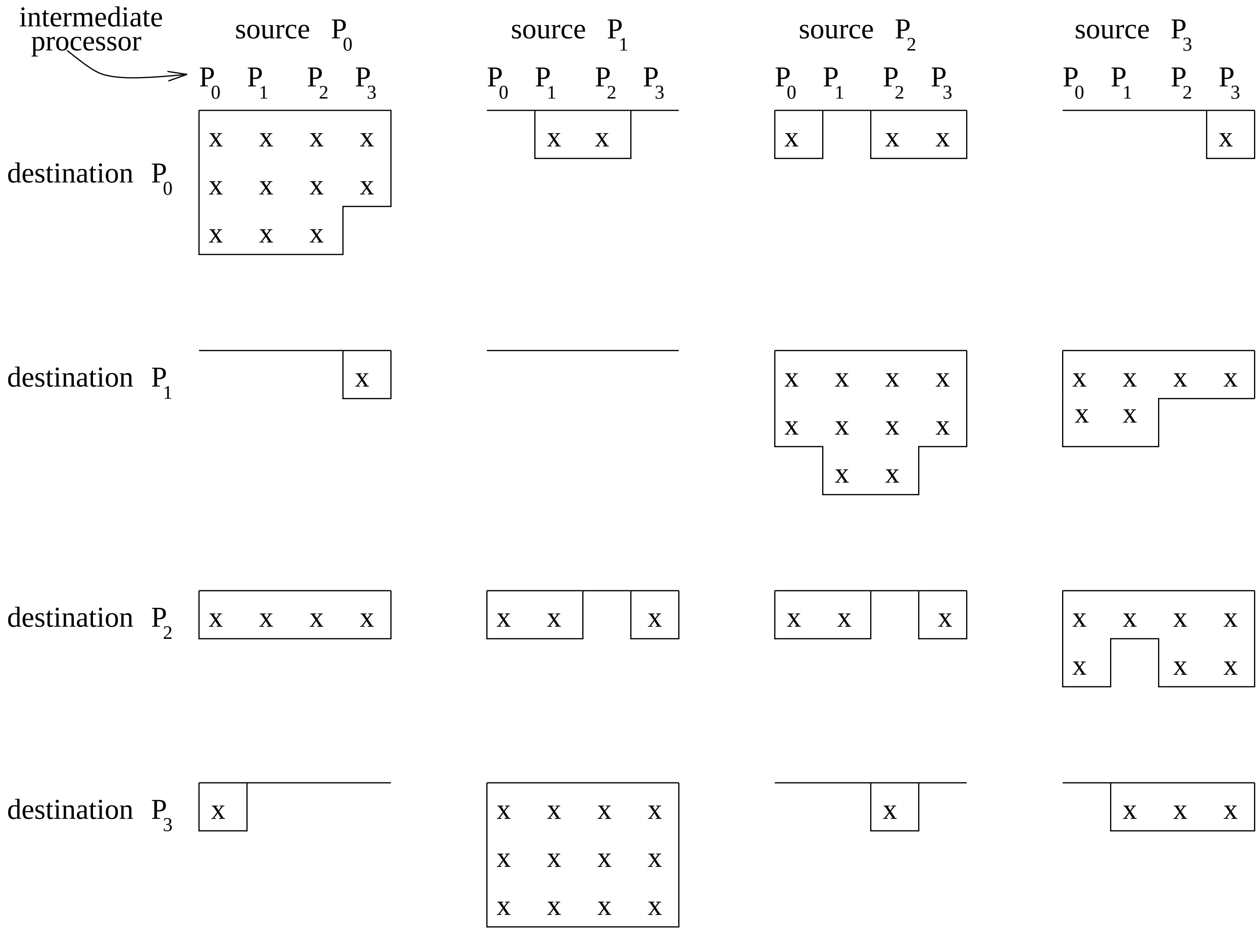

Figure 9: Splitting of Messages in the Two-Stage Algorithm

```
procedure Stg2pp (send_msg_start, send_msg_len)
/* send_msg_start[0..P-1][0..P-1] gives the index of the element from the
 * input array marking the start of each of the P parts of the P
 * messages sent out from this intermediate processor;
 *
 * send_msg_len[0..P-1][0..P] gives the length of those parts; in
 * particular, the entry send_msg_len[0..P-1][0] gives the total
 * length of messages to each destination processor
 */
begin
  /* initializing the total length of each message */
  for i := 0 to P-1 do
    send_msg_len[i][0] := 0;

  for i := 0 to P-1 do
    begin
      start := (T/P)*i + 1 + P;        /* start position of current message */
      for j := 0 to P-1 do                         /* T is the traffic bound */
        begin
          total := data_int[start*i+j+1];  /* length of current sub-message */
          send_msg_start[j][i] := start;
          start := start + total;
          send_msg_len[j][0] := send_msg_len[j][0] + total;
          send_msg_len[j][i] := total;
        end
    end
end;
```

Figure 10: Local Pre-processing in Stage 2

p lengths. Since a maximum of p messages could be sent out, this takes $O(p^2)$ time. Figure 10 gives the steps used for local pre-processing in stage 2.

Communication

Messages sent out in stage 2 could be of size up to $t/p + p$. In the general case, the lower bound on message size is 0, but in the equal traffic case, message size cannot be lower than $t/p - p$. Lowering the variance in message size, as was done in stage 1, is not as easy any more. The total size of the messages received at a destination processor is upper bounded by $t + p^2$. The upper bound on the communication time required in stage 2 is $O(p(\tau + \mu(t/p + p)))$. In practice, a random reshuffling of messages at the source processor, as explained in the appendix, could reduce the expected length of the messages in stage 2. The expected upper bound on the communication time required in stage 2 would then be $O(p(\tau + \mu(t/p + \sqrt{p \ln p})))$.

7.3 Analysis of Deterministic Time Complexity

- The local pre-processing needed for the two-stage algorithm takes $O(p^2)$ time. The two communication stages take $O(p(\tau + \mu(t/p + p)))$ time. Thus the two-stage algorithm has a deterministic time complexity of $O(p^2 + p\tau + \mu(t + p^2))$. The constants associated with the O notation in the analysis are small, typically between 2 and 3. The algorithm takes time $O(t)$ and is optimal when traffic $t \geq O(p^2 + p\tau/\mu)$.

- For $O(p\tau/\mu) \leq t < O(p^2 + p\tau/\mu)$, local pre-processing becomes a bottleneck to achieving optimality. This bottleneck can be overcome and the pre-processing time can be reduced to $O(t)$ by working with sparse representations (storing just the non-zero entries and their indices) of the p^2 sized arrays used in pre-processing. Further, the padding with p lengths done during stage 1 can be replaced by padding with $\lceil t/p \rceil$ lengths, making $O(p(\tau + \mu(t/p + 1)))$ the communication time required for the first stage. Time taken for the second stage's communication remains as the algorithm's bottleneck for achieving optimality when $O(p\tau/\mu) \leq t < O(p^2 + p\tau/\mu)$.

- In the case where every a_{ij} is a multiple of p, that is, if the message sent by any source processor to any destination processor is a multiple of p, optimality is achieved for $t \geq O(p\tau/\mu)$. This result is significant because it says that transportation with highly non-uniform messages can be performed in using a theoretically optimal and a very practical algorithm, if the message sizes are divisible by p. The constraint $t \geq O(p\tau/\mu)$ is satisfied when startup time does not dominate the time taken for communication.

An algorithm for transportation based on sorting can provide a better asymptotic time complexity in the general case when the traffic is small. Since the destination processors are numbers from a fixed range, local sorting done using a radix-sort takes just $O(t)$ time. Data movement between processors can be achieved using an adaptation of rotate-sort [8]. All communication between processors can be done as fixed (or static) permutations. Such a combination was used to perform sorting for geometric hashing in [10]. This rotate-sort and radix-sort combination performs transportation in $O(t)$ time, but requires nine local radix-sorts, six rotates (fixed permutations), and three row-wise sorts. Performing each row-wise sort through a transpose, followed by a radix-sort and another transpose, implies that the three row-wise sorts require three radix-sorts and six transposes (fixed permutations). The sorting based algorithm requires that the data be moved about 9 to 12 times between the processors, compared to the 2 movements required in the two-stage algorithm. The difference in the constants associated with the communication term in the time complexity of both algorithms is significant, especially for coarse-grained architectures. This makes the sorting based algorithm highly impractical in spite of its better asymptotic time complexity for smaller traffic.

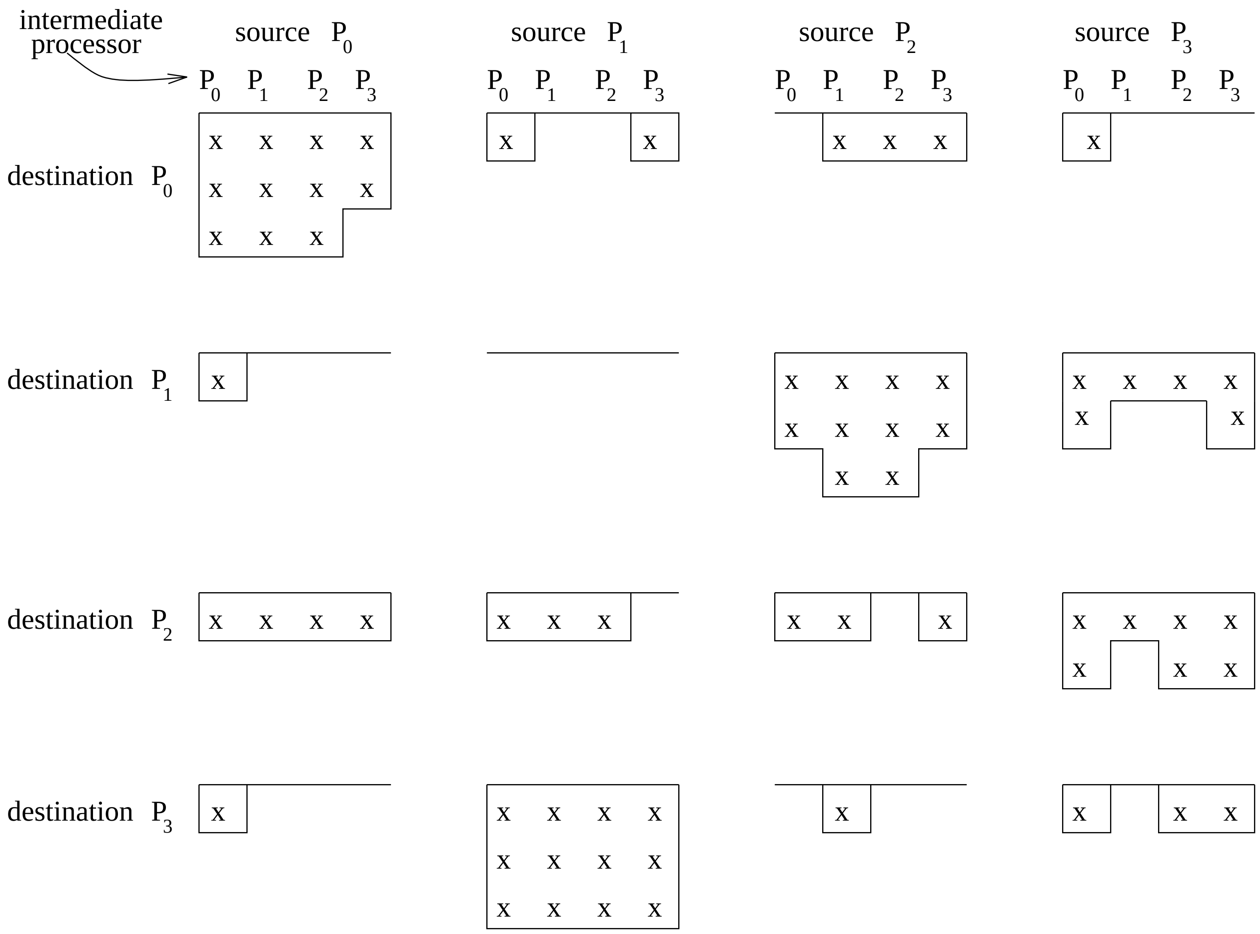

Figure 11: An Alternate Scheme for the Splitting of Messages

8 Transportation with Differing Incoming and Outgoing Traffic Bounds

With an outgoing traffic bound r and an incoming traffic bound c, the two-stage algorithm takes $minimum(p^2, r)$ and $minimum(p^2, c)$ for the local pre-processing in the two stages. Communication in stage 1 takes $\mathrm{O}(p(\tau + \mu(r/p + 1)))$ time and communication in stage 2 takes $\mathrm{O}(p(\tau + \mu(c/p + p)))$ time. The algorithm takes $\mathrm{O}(r + c)$ time and is optimal for $r \geq O(p)$ (unless the startup time is high) and $c \geq O(p^2)$. Incorporating the additional step shown in figure 9 reduces the stage 1 communication time to $\mathrm{O}(p\tau + \mu r/p)$ making the algorithm optimal for $c \geq O(p^2)$. The only constraint on r is that the startup time should not dominate the time taken by stage 1, that is, $r \geq O(p\tau/\mu)$.

If, on the other hand, the outgoing traffic bound r is higher than the incoming traffic bound c, an alternate scheme can be used for message splitting. This scheme is illustrated through an example in figure 11. The change in the stage 1 local pre-processing algorithm to accomodate the new scheme is shown in figure 12. At any processor P_i, when dividing the a_{ij} elements into p parts for sending during the first stage, the last a_{ij} mod p elements are assigned to the p intermediate processors in a round

⋮

```
for j := 0 to P-1 do
  sendl[j] := sendl[j] mod P;
Global_Vector_Prefix_Sum_Scan(sendl, sendl_new);
for j := 0 to P-1 do
  sendl_new[j] := sendl_new[j] mod P;

for j := 0 to P-1 do
  begin
    k := sendl_new[j];
    for x := 1 to sendl[j] do
    /* sendl[j] is the # of elements meant for destination processor j
       that could not be divided equally among the intermediate processors */

      begin
        send_msg_len[k][j+1] := send_msg_len[k][j+1] + 1;
      end;
  end
```

⋮

Figure 12: Changes to the Stage 1 Local Pre-processing Algorithm

robin fashion. In the earlier message splitting scheme, we ensured that each intermediate processor receives messages of size no more than $\lceil t/p \rceil$ from any source processor. The new scheme ensures that each intermediate processor sends messages of size no more than $\lceil t/p \rceil$ to any destination processor. This is achieved by performing the round-robin assignment across all the source processors rather than inside each source processor. A global prefix-sum scan with the vector (a_{i0} *mod* p, a_{i0} *mod* p, ... , $a_{i(p-1)}$ *mod* p) in each source processor P_i is needed. This takes O(p) time and does not affect the time complexity of pre-processing. Communication in stage 1 takes O($p(\tau + \mu(r/p + p))$) time and communication in stage 2 takes O($p(\tau + \mu(c/p + 1))$) time. The algorithm takes O($r + c$) time and is optimal for $r \geq O(p^2)$ and $c \geq O(p)$. As with the earlier scheme, an additional step (not shown in figures 11 and 12) can reduce the upper bound on the total length of messages leaving an intermediate processor from $p\lceil t/p \rceil$ to t. The additional step involves a global sum-combine with vectors of size p. This additional step reduces the stage 2 communication time to O($p\tau + \mu c/p$) making the algorithm optimal for $r \geq O(p^2)$. The only constraint on c is that the startup time should not dominate the time taken by stage 2, that is, $c \geq O(p\tau/\mu)$).

9 Performance Results

The two-stage algorithm and the single-stage algorithm were implemented on the CM-5 using the CMMD message passing library with CMAML active message routines. Communication matrices were generated such that message sizes were non-uniform while the traffic was bounded. Three parameters were used to control the kind of matrix that was generated. The fan-out parameter k specified the number of processors that each processor communicates with ($k \leq p$). The sum of the messages being sent out and received at each processor was fixed at t, the traffic parameter. A parameter l was used to control the non-uniformity of messages sent out by the processors. It was used as follows: Of the k processors receiving messages from a single processor, the fraction lt of the traffic reached $(1-l)k$ processors, while the remaining $(1-l)t$ traffic reached lk processors.

Figure 13 compares the performances of the single-stage distributed scheduling algorithms with and without the improvements. The horizontal axis gives the traffic (in words) at each processor and the vertical axis gives the time taken in seconds. The parameter k was varied from 2 to 32 and the parameter l was varied from $1/2$ to $1/k$. The algorithm with the improvements performed better. The variation in the time taken for different values of k and l is large for both algorithms. Figure 14 compares the performance of the two-stage algorithm with that of the single-stage algorithm with the improvements. The single-stage algorithm consistently performed better than the two-stage algorithm, although it exhibited a much larger variance in the time taken. The two-stage algorithm timings were within a factor of 1.5 times the single-stage readings. It should be noted that the two-stage algorithm is fairly architecture-independent, while the single-stage algorithm (particularly the one with the improvements) is architecture-dependent. The latter is also highly dependent on the availability of low latency communication primitives.

Sample values of k and l were chosen to highlight a best-case and a worst-case performance of the two-stage algorithm among the trials that were conducted. Figure 15 illustrates the best case in which the two-stage algorithm performed as well as the single-stage algorithms, even out-performing the single-stage algorithm without the improvements. In this trial k and l were fixed at 32 and $1/32$ respectively, which indicates that 1 out of 32 processors received $31/32$ of the total traffic, while the other 31 processors received in total $1/32$ of the traffic. It was a trial in which the messages were highly non-uniform in size. Figure 16 illustrates a worst case for the two-stage algorithm. Both the single-stage algorithms out-performed the two-stage one. In this trial k and l were fixed at 2 and $1/2$ respectively. This indicates that only 2 processors receive data from a single processor, and both of them receive exactly the same amount of traffic. It was a trial in which the messages were uniform in size. The two-stage algorithm's performance remained roughly close to its best-case performance, but the single-stage algorithm's performance improved considerably.

10 Conclusions

We have presented a variety of solutions for the transportation problem on a distributed memory parallel machine. A two-stage algorithm that takes time no more than $2t\mu$ (+ lower order terms) when traffic $t \geq O(p^2 + p\tau/\mu)$ was presented. For smaller traffic ($t \geq O(p\sqrt{p \ln p})$), the two-stage algorithm is expected to work well, as shown in the probabilistic analysis in the appendix. An algorithm using sorting can improve the result to O(t) time for $t \geq O(p)$, but the associated constants make this algorithm less desirable for implementation. The two-stage algorithm can also be used when any processor is receiving at most c amount of data and sending at most r amount of data. Time taken is no more than $(r+c)\mu$

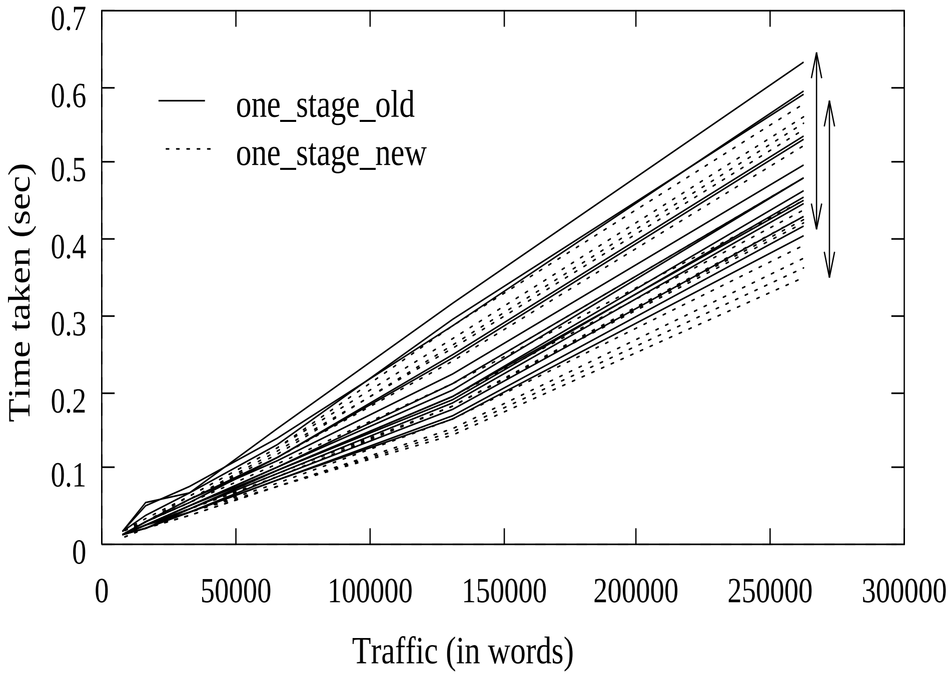

Figure 13: Comparison between the two single-stage algorithms

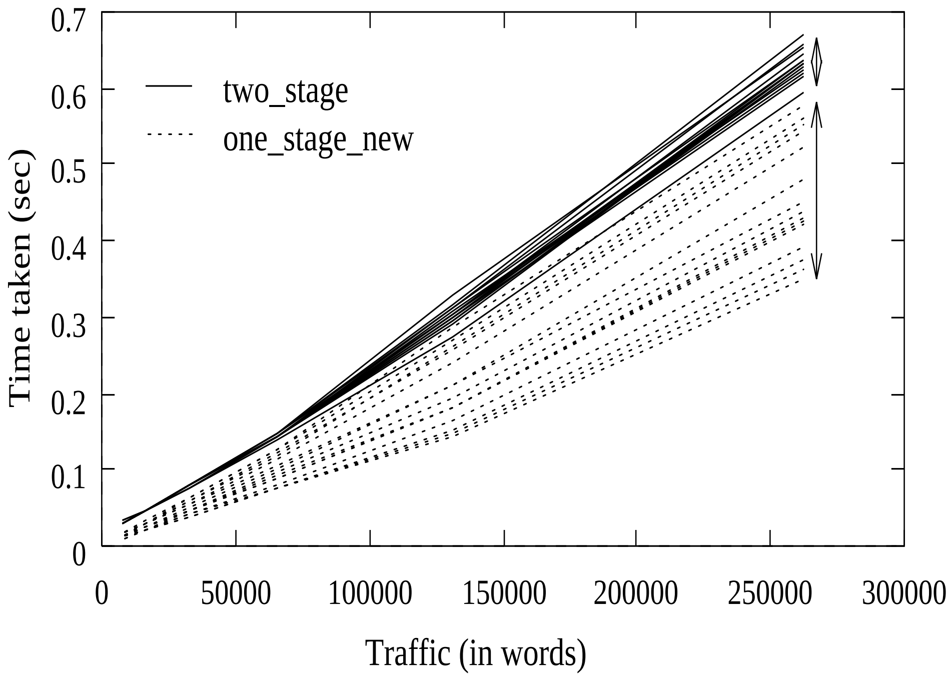

Figure 14: Comparison of the two-stage and single-stage algorithms

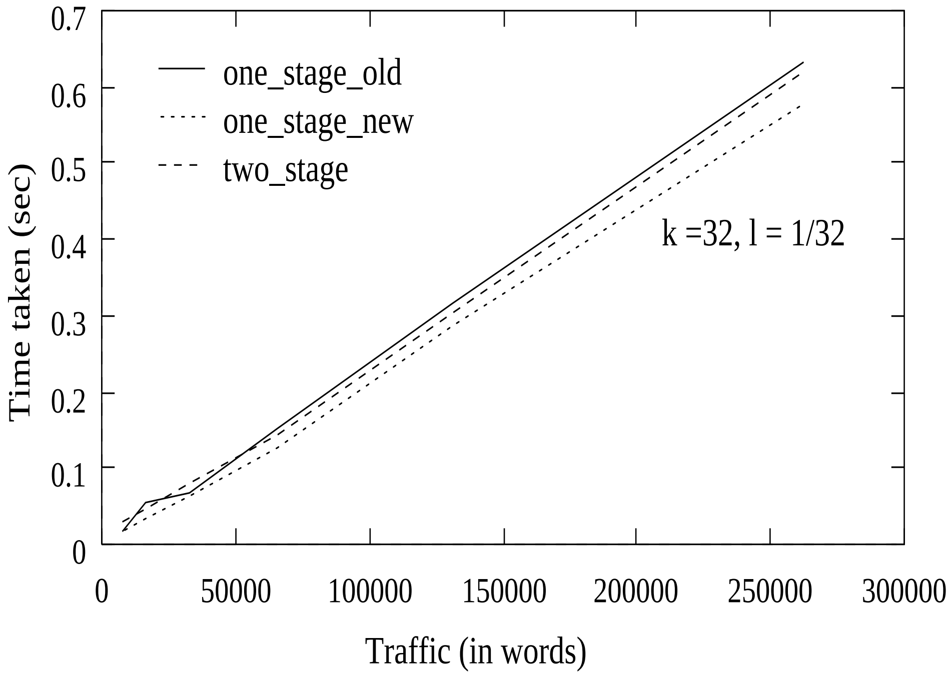

Figure 15: One of the good performances of the two-stage algorithms

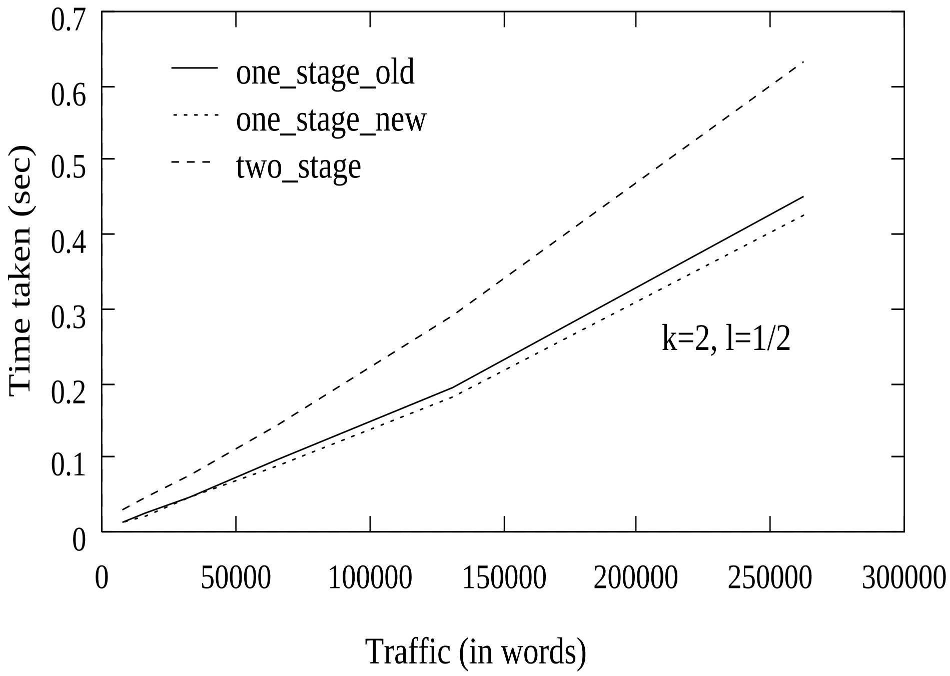

Figure 16: One of the bad performances of the two-stage algorithms

(+ lower order terms) when one of the bounds is O(p^2). A single-stage algorithm using distributed random scheduling was implemented and compared with an implementation of the two-stage algorithm. The distributed scheduling algorithm performed better on the CM-5, but this result is not expected to apply to other architectures. Besides, the single-stage algorithm is not deterministic, and that makes it difficult to ascertain its time complexity.

We have shown that many-to-many personalized communication with non-uniform messages can be performed using two stages of all-to-all personalized communication with uniform messages. Thus, the performance of the two-stage algorithm is roughly half that of an all-to-all personalized communication with the same amount of traffic. The latter problem has been widely investigated in the literature for a variety of interconnection networks (meshes, hypercubes, etc), message passing strategies (wormhole routing, store-and-forward routing, etc), single-port vs. multi-port communication. This makes the two-stage decomposition method useful for a wide variety of architectures. We are currently investigating the performance of these algorithms on other parallel architectures (Intel Paragon, iPSC 860, and the IBM SP1).

References

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

[2] 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.

[3] Eric A. Brewer and Robert Blumofe, Strata: A Multi-Layer Communications Library, MIT Laboratory of Computer Science Technical Report, February 1994.

[4] Eric A. Brewer, Bradley C. Kuszmaul, How to Get Good Performance from the CM-5 Data Network, Proceedings of the 8th International Parallel Processing Symposium, April 1994.

[5] Herbert A. David, *Order Statistics*, John Wiley and Sons, New York, 1981.

[6] T. von Eicken, D.E. Culler, S.C. Goldstein, K.E.Schauser. Active Messages: a mechanism for integrated communication and computation. Proceedings of the ISCA '92, Gold Coast, Australia, May 1992.

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

[8] J. Marberg, E.Gafni. Sorting in Constant Number of Row and Column Phases on a Mesh. Algorithmica, Vol.3, pp.561-572, 1988.

[9] K. Mehrotra, S. Ranka, J.C. Wang. A Probabilistic Analysis of a Locality Maintaining Load Balancing Algorithm, Proc. 7th International Parallel Processing Symposium, April 1993.

[10] 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.

[11] 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.

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

[13] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, October 1991.

[14] Thinking Machines Corporation. *CMMD Reference Manual Version 3.0*, October 1991.

[15] Jhy-chun Wang, Tseng-Hui Lin, Sanjay Ranka. Distributed Scheduling of Unstructured Collective Communication on the CM-5. Hawaii International Conference on System Sciences, 1993.

A Probabilistic Analysis of Time Taken

The purpose of the first stage in the two-stage algorithm was to spread out data leaving the source processors evenly among the intermediate processors. The intended intermediate processor numbers for the p messages leaving a source processor can be shuffled randomly within groups of messages of size $\lceil t/p \rceil$ and $\lfloor t/p \rfloor$, without affecting the algorithm. This would still preserve the upper bound derived earlier for total number of data elements sent or received in the first stage. The stage 1 communication now needs to include an extra array of length $minimum(p, \lceil t/p \rceil)$ tagged on to each outgoing message. This array gives the permutation that was performed locally before the send. It is needed at the destination processors since the p parts of a message reaching a destination processor must be put back together in order to complete the transportation.

A probabilistic analysis of the improvement in time due to the above modification follows. Let $\mid \frac{a_{0j}}{p} \mid, \mid \frac{a_{1j}}{p} \mid, ..., \mid \frac{a_{(p-1)j}}{p} \mid$ be the p parts of a message of length m reaching destination processor P_j. The notation $\mid \frac{a_{ij}}{p} \mid$ stands for $\lceil \frac{a_{ij}}{p} \rceil$ (with probability $\frac{a_{ij}\ mod\ p}{p}$) or $\lfloor \frac{a_{ij}}{p} \rfloor$ (with probability $1 - \frac{a_{ij}\ mod\ p}{p}$). This assumes that $a_{ij}\ mod\ p$ being $0, 1, ..., p-1$ is equally likely. In the deterministic analysis, the length m of the message reaching destination processor P_j was taken to be $t/p+p$ to accomodate the worst case.

The expected value of $\mid \frac{a_{ij}}{p} \mid$ is

$$\begin{aligned}
& \lceil \frac{a_{ij}}{p} \rceil (\frac{a_{ij}\ mod\ p}{p}) + \lfloor \frac{a_{ij}}{p} \rfloor (1 - \frac{a_{ij}\ mod\ p}{p}) \\
= \;& (\frac{a_{ij}}{p} + 1 - \frac{a_{ij}\ mod\ p}{p})(\frac{a_{ij}\ mod\ p}{p}) + (\frac{a_{ij}}{p} - \frac{a_{ij}\ mod\ p}{p})(1 - \frac{a_{ij}\ mod\ p}{p}) \\
= \;& \frac{a_{ij}}{p}
\end{aligned}$$

The variance of $\mid \frac{a_{ij}}{p} \mid$ is

$$\begin{aligned}
& (\lceil \frac{a_{ij}}{p} \rceil - \frac{a_{ij}}{p})^2 (\frac{a_{ij} \ mod \ p}{p}) + (\lfloor \frac{a_{ij}}{p} \rfloor - \frac{a_{ij}}{p})^2 (1 - \frac{a_{ij} \ mod \ p}{p}) \\
= & \ (1 - \frac{a_{ij} \ mod \ p}{p})^2 (\frac{a_{ij} \ mod \ p}{p}) + (\frac{-a_{ij} \ mod \ p}{p})^2 (1 - \frac{a_{ij} \ mod \ p}{p}) \\
= & \ (1 - \frac{a_{ij} \ mod \ p}{p})(\frac{a_{ij} \ mod \ p}{p})
\end{aligned}$$

The expected value of m is

$$\sum_{i=0}^{p-1} \frac{a_{ij}}{p} = t/p$$

The variance of m is

$$\sum_{i=0}^{p-1} (1 - \frac{a_{ij} \ mod \ p}{p})(\frac{a_{ij} \ mod \ p}{p})$$

The expected value for the variance of m is

$$\begin{aligned}
& \frac{1}{p^2} \sum_{i=1}^{p} (p-i)i \\
= & \ \frac{1}{p^2} (\frac{p^2(p+1)}{2} - \frac{p(p+1)(2p+1)}{6}) \\
= & \ \frac{1}{6p^2} p(p+1)(p-1) \\
= & \ \frac{p^2-1}{6p} \\
\approx & \ \frac{p}{6}
\end{aligned}$$

Let X_1, X_2, ..., X_n be independent and identically distributed random variables with mean 0 and variance 1. Let $X = max\{X_1, X_2, ..., X_n\}$. Then, for large n, the distribution for the normalized X is given by the *extreme-value-distribution* [5, 9]. Using the extreme-value-distribution assumption gives us $E(X) = a_n + \frac{\gamma}{b_n}$ where γ = Euler's constant = 0.5772, and $Var(X) = \frac{\pi^2}{6{b_n}^2}$. In particular, if the X_is are normally distributed, then both a_n and b_n are approximately equal to $\sqrt{2 \ln n}$. If the mean and variance of the n random variables are μ and σ^2, rather than 0 and 1 respectively, the values of $E(X)$ and $Var(X)$ are given by $E(X) = \mu + \sigma(\sqrt{2 \ln n} + \frac{\gamma}{\sqrt{2 \ln n}})$ and and $Var(X) = \frac{\pi^2 \sigma^2}{6{b_n}^2} = \frac{\pi^2 \sigma^2}{12 \ln n}$.

In two-stage algorithm, the length of a message reaching a destination processor, from a particular intermediate processor, has a mean of t/p and an expected variance of $p/6$. Each destination processor could receive such messages from each intermediate processor. This is done through the p iterations in the linear permutation algorithm used to perform communication. The time taken by any iteration of the linear permutation algorithm is dictated by the longest of the messages that need to be sent during that iteration. The length of a message in any iteration is given by the sum of p uniform distributions. We approximate this by a normal distribution. We can now use the properties of the extreme-value-distribution to obtain the expected value of the upper bound on the length of messages sent out during any iteration. This expected value is

$$\begin{aligned}
& \mu + \sigma(\sqrt{2 \ln p} + \frac{\gamma}{\sqrt{2 \ln p}}) \\
= \quad & t/p + \sqrt{\frac{p}{6}}(\sqrt{2 \ln p} + \frac{\gamma}{\sqrt{2 \ln p}}) \\
\approx \quad & t/p + \sqrt{\frac{p}{6}}\sqrt{2 \ln p}
\end{aligned}$$

The expected value of the maximum time needed for the communication in the second stage is $\mathrm{O}(p\tau + \mu(t + p\sqrt{p \ln p}))$.