1995

Irregular Personalized Communication on Distributed Memory Machines

Sanjay Ranka
Syracuse University, School of Computer and Information Science, ranka@top.cis.syr.edu

Jhy-Chun Wang
Syracuse University, School of Computer and Information Science, jcwang@cs.uiuc.edu

Irregular Personalized Communication on Distributed Memory Machines

Sanjay Ranka Jhy-Chun Wang[1]

4-116 Center for Science and Technology
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244-4100
Email: *ranka@top.cis.syr.edu*

Manoj Kumar

IBM T.J. Watson Research Center
Hawthorne, NY

[1]Jhy-Chun Wang's current address is: Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801. Email: jcwang@cs.uiuc.edu

Abstract

In this paper we present several algorithms for performing all-to-many personalized communication on distributed memory parallel machines. We assume that each processor sends a different message (of potentially different size) to a subset of all the processors involved in the collective communication. The algorithms are based on decomposing the communication matrix into a set of partial permutations. We study the effectiveness of our algorithms both from the view of static scheduling and from runtime scheduling.

Index Terms: Loosely synchronous communication, node contention, non-uniform message size, personalized communications, runtime scheduling, static scheduling.

1 Introduction

Load balancing and reduction of communication are two important issues for achieving good performance on distributed memory parallel computers. It is important to map the program such that the total execution time is minimized; the mapping typically can be performed statically or dynamically. For most regular and synchronous problems [10], this mapping can be performed at the time of compilation by giving directives in the language to decompose the data and its corresponding computations (based on the owner computes rule—where each processor only computes values of data it owns [6, 7, 23, 28]). This typically results in regular collective communication between processors. Many such primitives have been developed in [2, 21].

For a large class of scientific problems that are irregular in nature, achieving a good mapping is considerably more difficult [8]. Further, the nature of this irregularity may not be known at the time of compilation and can be ascertained only at runtime. The handling of irregular problems requires the use of runtime information to optimize communication and load balancing [13, 18, 20]. These packages derive the necessary communication information based on the nonlocal data required for performing local computations.

Consider the parallelization of a single concurrent computational phase of an explicit unstructured mesh fluids calculation. This step is typically executed repeatedly without change in computational structure. The computational structure of the above code is given in Figure 1. Similar examples of such computations are iterative solvers using sparse matrix-vector multiplications [24]. Further, a multiple phase computation consists of a series of dissimilar, loosely synchronous computational phases where each individual phase is a single concurrent computational phase. Examples of these computations include unstructured multigrid [17], parallelized sparse triangular solver [1, 4], and particle-in-cell codes [15, 26].

The key problem in efficiently executing these programs is partitioning the data and computation such that the load on each node is balanced and the communication is minimized. Figure 2 describes a decomposition of such a problem. The x and y arrays in Figure 1 represent the nodes in Figure 2, while the nde array represents the edges. This partitioning then dictates the program's synchronization and communication requirements, which must also be computed. The computational pattern may only be available at runtime and may not be done directly by the compiler; instead, calls to a runtime environment need to be generated to do the partitioning. Several algorithms are available in the literature to perform this partitioning (see [16] for a detailed list of such references).

The partitioning described in Figure 2 generates an 8×8 communication matrix COM (Table 1). A "1" in the (i, j) entry represents the fact that processor P_i needs to communicate to processor P_j. Each message is of different size and each processor may send a different number of messages. In our example, P_0 sends only three messages while P_4 sends five messages. If we allow processors to arbitrarily send their outgoing messages, it may happen that at one stage processors P_0, P_1, P_3, P_4 and P_6 will all try to send messages to processor P_2. Since the receiving processor typically can receive messages from only one processor

C This is a simplified sweep over edges of a mesh. A flux across a
C mesh edge is calculated. Calculation of the flux involves
C flow variables stored in array x. The flux is accumulated to array y.

do $i = 1, N$
  S1 $n1 = nde(i, 1)$
  S2 $n2 = nde(i, 2)$
  S3 $flux = f(x(n1), x(n2))$
  S4 $y(n1) = y(n1) + flux$
  S5 $y(n2) = y(n2) - flux$
end do

Figure 1: Code representing a simple explicit unstructured fluid calculation.

at a time, one or more of the sending processors may have to wait for other processors to complete their communication. We use the term *node contention* to refer to this situation. We will show that node contention has a deteriorating effect on the total time required for communication.

In this paper, we develop several simple methods of scheduling all-to-many personalized communication. The cost of the scheduling algorithm can be amortized over several iterations, as the same schedule can be used several times. In the above unstructured mesh example, the same iteration is typically repeated several times.

In general, assuming a system with n processors, our algorithms take as input an $n \times n$ communication matrix COM. $COM(i, j)$ is equal to a positive integer m if processor P_i needs to send a message (of m unit) to P_j, $0 \leq i, j \leq n-1$. Our algorithms decompose the communication matrix COM into a set of partial permutations, $pm_1, pm_2, \cdots, pm_l$, where l is a positive integer and pm_k^i represents the i^{th} entry in vector pm_k. The decomposition is made such that if $COM(i, j) \neq 0$, then there exists a k, $1 \leq k \leq l$, such that $pm_k^i = j$.

The communication matrix of Table 1 may be decomposed into the following permutations:

$pm_1 = (6, 7, 0, 1, 2, 3, 4, 5)$,
$pm_2 = (2, 3, 6, 5, 7, 4, 0, 1)$,
$pm_3 = (-, 0, 1, 2, 3, 7, -, 4)$,
$pm_4 = (1, 2, 3, 4, 5, -, 7, 6)$, and
$pm_5 = (-, -, 4, -, 6, -, 2, -)$.

where in each permutation every processor both sends and receives at most one message.

Assuming that the processors perform their operation in a synchronous fashion, the time taken to complete a permutation depends on the largest message in the permutation. Since the message sizes in one permutation may vary widely, we develop several schemes to reduce

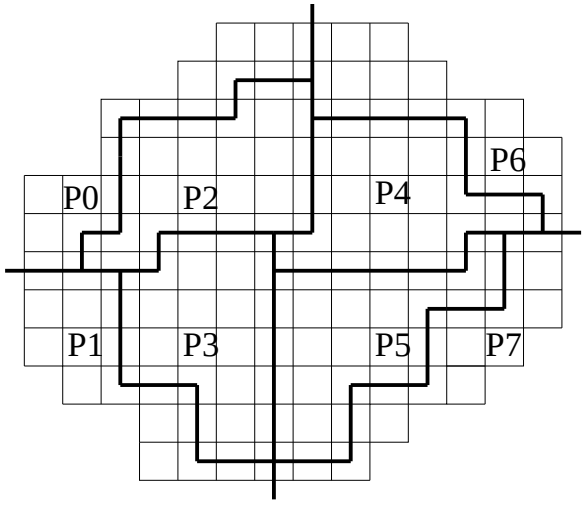


Figure 2: The partitioning of irregular mesh.

	0	1	2	3	4	5	6	7
0		1	1				1	
1	1		1	1				1
2	1	1		1	1		1	
3		1	1		1	1		
4			1	1		1	1	1
5				1	1			1
6	1		1		1			1
7		1			1	1	1	

Table 1: An 8×8 communication matrix (blank entries imply no communication).

the variance of message size within one permutation. This is done by splitting large messages into smaller pieces, each of which is sent in different phases.

With the advent of new routing methods [9, 19, 25], the distance to which a message is sent is becoming relatively less and less important [3]. Thus, assuming no link contention, permutation is an efficient collective communication primitive. For an architecture like the CM-5, the data transfer rate seems to be bounded by the speed at which data can be sent or received by any processor [5]. Thus, if a particular processor receives more than one message or has to send out more than one message in one phase, then the time will be lower bounded by the time required to remove the messages from the network by the processor receiving the maximum amount of data.

Clearly, this is not going to be the case for all architectures. The paths of two messages may have a common link. This may sequentialize the transfer of the two messages (especially for machines that use circuit switching routing). Assuming that routing is static in nature (i.e., the path to send a message from one node to another node can be predetermined), we can build partial permutations that satisfy the property that no two messages interact; however, this would depend on the topology and routing methodology and would increase the cost of obtaining a good schedule.

The algorithms described in this paper do not take link contention into account. A main reason for this is that message routing is randomized on the CM-5 [14, 25], it is not possible to statically schedule messages in such a fashion that link contention can be avoided, although randomization alleviates that problem to a large extent. The variation of time required for different random permutations (in which each node sends a data to a random, but different node) is very small on a 32-node CM-5 (cf. Section 3.2). The algorithms developed in this paper can be extended to the architectures where link contention is an important issue by decomposing communication matrix into partial permutations which avoid link contention. The cost of these algorithms would depend on the topology as well as the routing method.

We show that our algorithms are inexpensive enough to be suitable for static as well as runtime scheduling. If the number of times the same communication schedule is used is large (which happens for a large class of problems [7]), the fractional cost of the scheduling algorithm is quite small. Further, compared to naive algorithms, our algorithm can result in a significant reduction in the total amount of communication.

The rest of this paper is organized as follows. Notations, definitions, and general communication properties used throughout are given in Section 2. Section 3 provides an overview of CM-5. Section 4 presents a simple asynchronous communication algorithm. Section 5 describes algorithms that avoid node contention. Section 6 proposes approaches to reduce the variance of message size in one permutation. Section 7 presents experimental results on a 32-node CM-5. Finally, conclusions are given in Section 8.

2 Preliminaries

The communication matrix COM is an $n \times n$ matrix where n is the number of processors. $COM(i,j)$ is equal to a positive integer m if processor P_i needs to send a message (of m units) to P_j, otherwise $COM(i,j) = 0$, $0 \leq i,j < n$. Thus, row i of COM represents the sending vector, $sendl_i$, of processor P_i, which contains information about the destination node and the size of outgoing messages. Column i of COM represents the receiving vector, $recvl_i$, of processor P_i, which contains information about the source node and the size of incoming messages. The entry $sendl_i^j$ ($recvl_i^j$) represents the j^{th} entry in the vector $sendl_i$ ($recvl_i$). Assuming $COM(i,j) = m$, then $sendl_i^j = recvl_j^i = m$. We will use $sendl$ and $recvl$ to represent each processor's sending vector and receiving vector when there is no ambiguity.

COM can be decomposed into a set of communication phases, cp_k, $1 \leq k \leq l$, l, a positive integer, such that

$$COM(i,j) = m, \;\; m > 0 \quad \Rightarrow \quad \exists! k, \;\; 1 \leq k \leq l, \;\; cp_k^i = j \; .$$

We define the k^{th} communication phase as

$$cp_k^i = j, \;\; i = 0, 1, \ldots, n-1, \;\; and \;\; 0 \leq j < n$$

if processor P_i needs to send a message to processor P_j at the k^{th} phase, otherwise $cp_k^i = -1$.

Thus, *node contention* can be formally defined as

$$\exists k, \;\; 1 \leq k \leq l, \;\; cp_k^{i_1} = j_1 \;\; and \;\; cp_k^{i_2} = j_2 \Rightarrow i_1 \neq i_2 \;\; and \;\; j_1 = j_2 \neq -1 \; ,$$

where $i_1, i_2 = 0, 1, \ldots, n-1$ and $0 \leq j_1, j_2 < n$.

A partial permutation pm_k is a communication phase that

$$pm_k^{i_1} = j_1 \;\; and \;\; pm_k^{i_2} = j_2, \;\; i_1, i_2 = 0, 1, \ldots, n-1 \;\; and \;\; 0 \leq j_1, j_2 < n \; ,$$

$$i_1 = i_2 \quad \Leftrightarrow \quad j_1 = j_2 \; ;$$

$pm_k^i = -1$ if P_i does not send a message at this permutation.

Since permutation has the useful property that every processor both sends and receives at most one message, it does not cause any node contention. In this paper we will use permutation as our underlying communication scheme.

2.1 Notation and Assumptions

We categorize scheduling algorithms into several categories:

1. *Uniformity of message*—Uniform messages mean all messages are of equal size. In this paper we assume that messages are of non-uniform size. In case the messages are of the same size, the algorithms developed in [22] have considerably smaller scheduling overhead.

2. *Density of communication matrix*—If the communication matrix is nearly dense, then all processors send data to all other processors. If the communication matrix is sparse, then every processor sends to only a subset of processors. Our algorithms assume that the latter is true. There are a number of algorithms for the totally dense cases [2, 12].

3. *Static or runtime scheduling*—Communication scheduling must be performed statically or dynamically.

For the reasons mentioned in the previous section, the algorithms described in this paper do not take link contention into account. We also make the following assumptions for developing our algorithms and their complexity analysis.

1. Every permutation can be completed in $(\tau + M\varphi)$ time, where τ is the communication latency, M is the maximum size of any message sent in this permutation, and φ represents the inverse of data transmission rate.

2. In case communication is sparse, all nodes send and receive an approximately equal number of messages. Let density d represent the number of messages sent or received by every processor.

3. We assume that each processor can send only one message and receive only one message at a time. If the density is d, then at least d permutations are required to send all messages.

3 CM-5 System Overview

This section gives a brief overview of the CM-5 system that we used to conduct our experiments. The CM-5 is available in configuration of 32 to 1024 processing nodes, each node being a SPARC microprocessor with 32M bytes of memory and optional vector units. The node operates at 33 MHz and is rated at 22 Mips and 5 MFlops. When equipped with vector units, each node of the machine is rated at 128 Mips (peak) and 128 MFlops (peak).

The CM-5 internal networks include two components, data network and control network. The CM-5 has a separate diagnostics network to detect and isolate errors throughout the system.

The data network provides high-performance data communications among all system components. The network has a peak bandwidth of 5M bytes/sec for node-to-node communication. However, if the destination is within the same cluster of 4 or 16, it can give a peak bandwidth of 20M bytes/sec and 10M bytes/sec, respectively [5]. Figure 3 shows the data network with 16 nodes.

The control network handles operations that require the cooperation of many or all processors. It accelerates cooperative operations such as broadcast and integer reduction, and system management operations such as error reporting.

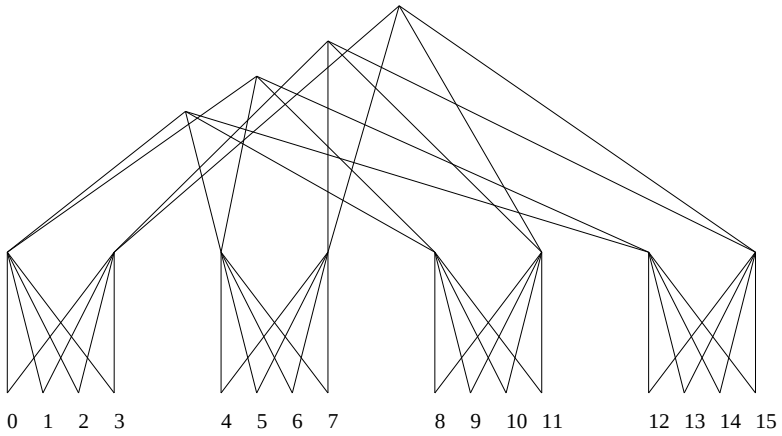


Figure 3: CM-5 data network with 16 nodes.

3.1 Node Contention on CM-5

Table 2 shows the impact of node contention on a 32-node CM-5. In these experiments, processor P_{31} is the receiving node, and processors P_i, $0 \leq i < d$ are sending nodes. In every phase, each sending node sends an equal amount of data (256 bytes or 4K bytes) to P_{31} simultaneously. We record the time (in milliseconds) taken for P_{31} to complete receiving all incoming data, and the maximum, minimum, and average time taken among sending nodes to complete sending data.

The results reveal that when the number of messages sent to the same node (at the same time) increases, the average time each sending node needs to complete sending its message increases (the same holds true for the maximum time and minimum[1] time among the sending processors). Thus it is inefficient to allow more than one node to send a message to the same processor simultaneously.

These observations suggest that node contention will result in overall performance degradation. Avoiding node contention should therefore be considered as an important factor when we conduct communication scheduling.

3.2 Cost of Random Permutations

We randomly generated 2 test sets, each containing 5000 random permutations. The sizes of the message used in each of these permutations are 1K bytes and 256K bytes, respectively. The communication cost distribution (in terms of average communication cost) is given in Figures 4 and 5. The results depict that for most cases the communication cost is within $\pm 10\%$ of average cost (the average communication costs for message of size 1K bytes and 256K bytes are 0.543 milliseconds and 62.923 milliseconds, respectively). Thus the perfor-

[1]One exception to the time increase is that when all 31 nodes send messages to processor P_{31}, nodes P_{28}, P_{29}, P_{30}, and P_{31} are in the same 4-node cluster, which can provide higher communication bandwidth, so the minimum time taken in this case is less compared with the 16-node case.

d	256 bytes				4096 bytes			
	recv	send			recv	send		
		max	min	ave		max	min	ave
1	0.089	0.131	0.050	0.061	0.516	0.504	0.485	0.488
2	0.125	0.150	0.070	0.081	1.083	1.048	1.023	1.038
4	0.205	0.199	0.098	0.116	2.189	2.124	2.085	2.097
8	0.375	0.298	0.173	0.210	4.693	4.844	4.353	4.502
16	0.731	0.575	0.302	0.394	9.865	10.065	9.155	9.476
31	1.396	1.279	0.151	0.815	19.485	19.544	2.847	15.550

Table 2: The impact of node contention on a CM-5.

dist	1	2	4	8	16	*ave*
comm	47.136	47.143	47.320	62.582	68.006	62.923

$*$ *comm*: communication cost in milliseconds.
$**$ *ave*: average communication cost of 5000 randomly generated permutation samples.

Table 3: Communication cost for permutations with message of length 256K bytes within different cluster sizes.

mance of our algorithms, which use permutation as the underlying communication scheme, are not significantly affected by a given sequence of permutation instances. The bandwidth achieved for these permutations is approximately 4M bytes/sec, which is close to the peak bandwidth of 5M bytes per second provided by the underlying hardware for long distance messages.

There are some permutations for which the performance is expected to be better than random permutations. One such class of permutations is when processor P_i exchanges messages with processor $P_{i \oplus dist}$[2], $0 \leq i < n$ and $dist = 1, 2, 4, 8, 16$. Each permutation represents a communication pattern where processors communicate with processors within the clusters of 2, 4, 8, 16, and 32, respectively. The results for these permutations are given in Table 3. These results show that these specialized permutations, in which every processor sends a message to another processor within the same group of 8 nodes, take approximately 25% less time over random permutations. However, our algorithms do not exploit these special cases.

[2] $\oplus$ represents bitwise exclusive OR operator.

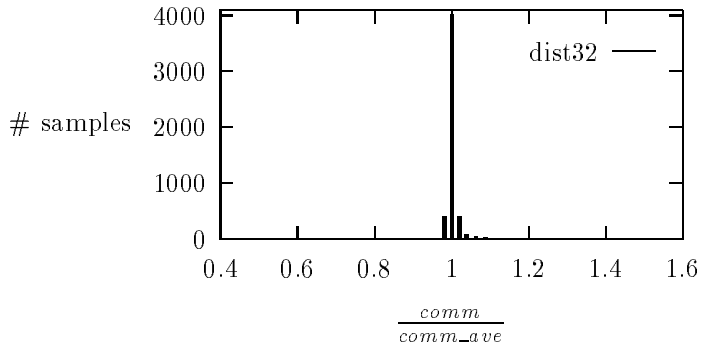


Figure 4: Communication cost distribution for 5000 permutation samples with message of length 1K bytes on a 32-node CM-5.

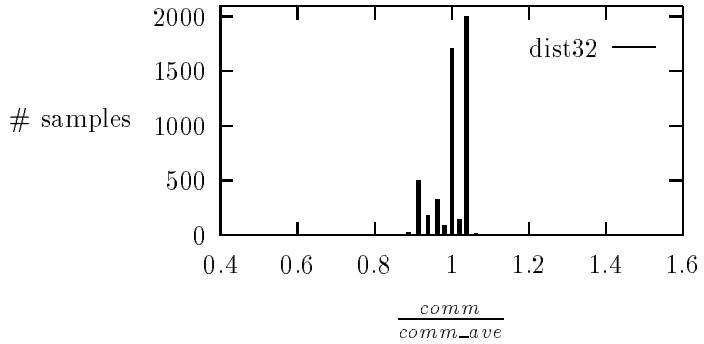


Figure 5: Communication cost distribution for 5000 permutation samples with message of length 256K bytes on a 32-node CM-5.

4 Asynchronous Communication (AC)

The most straightforward approach is to use asynchronous communication. The algorithm is divided into three phases:

1. Each processor first posts requests for expected incoming messages (this operation will pre-allocate buffers for those messages).

2. Each processor sends all of its outgoing messages to other processors.

3. Each processor checks and confirms incoming messages (some of which may already have arrived at their receiving buffer(s)) from other processors.

Asynchronous_Send_Receive()
For all processors P_i, $0 \leq i \leq n-1$, *in parallel do*
  allocate buffers and post requests for incoming messages;
  send out all outgoing messages to other processors;
  check and confirm incoming messages from other processors.

Figure 6: Asynchronous Communication Algorithm.

During the send-receive process, the sending processor need not wait for a completion signal from the receiving processor, but can keep sending outgoing messages until they are all done. This naive approach is expected to perform well when density d is small. The asynchronous algorithm is given in Figure 6.

The worst case time complexity of this algorithm is difficult to analyze, as it will depend on the congestion and contention on the nodes and network. Also, each processor may have only limited space in message buffers. In such cases, when the system buffer space is fully occupied by unconfirmed messages, further messages will be blocked at the sender processors' side. The overflow may block processors from doing further processing (including the receiving of messages) because processors are waiting for other processors to consume and empty their buffers in order to receive new incoming messages. This situation may never be resolved and a deadlock may occur among processors.

In case the sources of incoming messages are not known in advance or there is no buffer space available for pre-allocation, we may replace the *post-send-confirm* operation by the *send-detect-receive* operation, where we use *busy waiting* to detect incoming messages and copy them into the application buffer. Buffer copying is very costly and should generally be avoided. The experimental results described in this paper use the approach given in Figure 6.

5 Methods Avoiding Node Contention

Our scheduling algorithms assume the availability of a global communication matrix COM. A concatenation operation [5] can be performed on the sending vector (of length n) of each processor to derive this matrix at runtime. For an n-node CM-5, performing a concatenate operation with each node contributing a message of size n is efficient and can be completed in $O(n^2 + \tau \log n)$ amount of time [5]. Concatenate operation has efficient implementation on other architectures like hypercubes and meshes [2, 21]. In case the communication matrix COM is sparse in nature, each processor will send and receive d messages in a system with n processors ($d < n$), we can reduce the total time to $O(dn + \tau \log n)$ by using a sparse representation for the sending vector. In such a case, the communication matrix would be an $n \times d$ matrix such that each row is a sparse representation of the corresponding sending vector.

Linear_Permutation()
For all processors P_i, $0 \leq i \leq n-1$, *in parallel do*
  for k = *1 to n-1 do*
    $j = i \oplus k$;
    if $COM(i,j) > 0$ *then* P_i sends a message to P_j;
    if $COM(j,i) > 0$ *then* P_i receives a message from P_j;
  endfor

Figure 7: Linear Permutation Algorithm.

5.1 Linear Permutation (LP)

In this algorithm (Figure 7), each processor P_i sends a message to processor $P_{(i \oplus k)}$ and receives a message from $P_{(i \oplus k)}$, where $0 < k < n$. When $COM(i,j) = 0$, processor P_i will not send a message to processor P_j, but will receive a message from P_j if $COM(j,i) > 0$. The entire communication uses pairwise exchange ($j = i \oplus k \Leftrightarrow i = j \oplus k$). A simple variation of LP is that each processor P_i sends a message to processor $P_{(i+k) \bmod n}$ and receives a message from $P_{(i-k) \bmod n}$, where $0 < k < n$. The experimental results show that, for the CM-5, the former approach performs slightly better.

This algorithm assumes that the number of processors, n, is a power of 2, and the algorithm can easily be extended when n is not a power of 2.

5.2 Random Scheduling Using Heaps (RS_NH)

During the communication scheduling, the worst case time complexity to access each entry of COM is $O(n^2)$. In order to reduce this overhead, the first step of this algorithm is to compress the COM into an $n \times d$ matrix $CCOM$ by a simple compressing procedure (Appendix A). This procedure will improve the worst case time to access each active element (of $CCOM$) to $O(dn)$.

If we perform this compression statically, the time complexity is $O(n(n+d)) = O(n^2)$. When performing this operation at runtime, each processor compacts one row, and then all processors participate in a concatenate operation to combine individual rows into an $n \times d$ matrix. The cost of this parallel scheme is $O(n + (dn + \tau \log n)) = O(dn + \tau \log n)$ (assuming the concatenate operation can be completed in $O(dn + \tau \log n)$ time).

The vector prt is used as a pointer whose elements point to the maximum number of positive columns in each row of $CCOM$. In order to schedule the communication in such a way that each processor will try to send out larger messages first, we sort the active entries in $CCOM$ by message size. A heap (denoted by $heap_k$ in row k) is embedded such that the root entry $CCOM(k,0)$ contains the largest message size among all the entries in row k. Three heap procedures are needed in the algorithm: *Heap_Extract_Max()* returns the location of the

RS_NH()

1. Use matrix COM to create an $n \times d$ matrix $CCOM$;
2. In each row k, $0 \leq k < n$, build a heap $heap_k$ based on the entries $CCOM(k, j)$'s corresponding message size, where $0 \leq j < d$;
3. *Generate_Permutations()*.

Figure 8: Random Scheduling using Heaps (RS_NH) Algorithm.

entry with largest message size within a heap; *Heap_Remove()* removes the specified entry from the heap; and *Heap_Insert()* inserts an entry into the heap. Each of these procedures can be completed in $O(\log d)$ time [11].

The vectors *send* and *receive* are used to record the destination of each outgoing message and the source of each incoming message in one permutation, respectively; $send(i) = j$ denotes processor P_i needs to send a message to processor P_j, and $receive(j) = i$ denotes processor P_j will receive a message from processor P_i. These two vectors are initialized to -1 at the beginning of each iteration. We assume that $CCOM(i, j) = -1$ if no message is to be sent. After the compressing procedure, the first d columns of each row may contain active entries. When searching for a available entry along row i, the first column j with $CCOM(i, j) = k$ and $receive(k) = -1$ will be chosen. We then set $send(i) = k$ and $receive(k) = i$. Since the messages are non-uniform, the message sizes in one permutation may vary in a wide range. If we allow every processor to completely send its message, then the communication time in each step is upper bounded by the maximum message size in each step. (Although RS_NH is executed in a loosely synchronous fashion, processors with small messages may be idle while waiting for processors with large messages to complete.) In order to eliminate idle time for processors, we introduce several approaches to choose a reasonable message size in each communication phase such that processors with small messages will send their messages completely, while processors with large messages will send only part of their messages.

The RS_NH algorithm is described in Figure 8.

Step 1 (Figure 8) takes $O(n^2)$ time to complete sequentially. When used at runtime, each processor creates one row of $CCOM$, then all processors participate in a concatenate operation. The time required for this step is $O(dn + \tau \log n)$. The time required for Step 2 is $O(dn)$.

Step 1 (Figure 9) takes $O(n)$ time. Step 3 requires a sort operation (we use merge sort in our experiments, which has a time complexity of $O(n \log n)$). This sort operation can be approximated by using a histogram-based approach to reduce the scheduling time. The time required for communication in Step 4 is $O(\tau + \varphi M^k_{thresh})$ time where M^k_{thresh} is the

Generate_Permutations()

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

1. Set all entries of vectors *send* and *receive* to -1;

2. $x = random(1..n)$;
   for $y = 0$ *to* n-1 *do*
   $i = (x + y) \ mod \ n$; $j = 0$; $S = \phi$;
   while (send(i) = -1 AND $j \leq prt(i)$*) do*
   $k = CCOM(i, l)$, where $l = Heap\_Extract\_Max(heap_i)$;
   if (receive(k) = -1) then;
   $send(i) = k$; $receive(k) = i$;
   endif
   $S = S \cup CCOM(i, l)$; $Heap\_Remove(heap_i, l)$; $j = j + 1$;
   endwhile
   For all entries, $CCOM(i, k)$, in S (except the last one), $Heap\_Insert(heap_i, M_{ik})$;
   /* M_{ik} is $CCOM(i, k)$'s corresponding message size */
   endfor

3. $M_{thresh} = Decide\_Size()$;

4. *if (send(i)* $\neq$ *-1) then* P_i sends a message, no bigger than M_{thresh}, to $P_{send(i)}$;
   if (receive(i) $\neq$ *-1) then* P_i receives a message from $P_{receive(i)}$;

5. For each row k which sent a complete message at this iteration, decreases $prt(k)$ by 1; For each row l which only sent partial message, add the remainder of the message back to its proper location in $heap_l$;

Until all messages are sent.

Figure 9: Procedure Generate_Permutations().

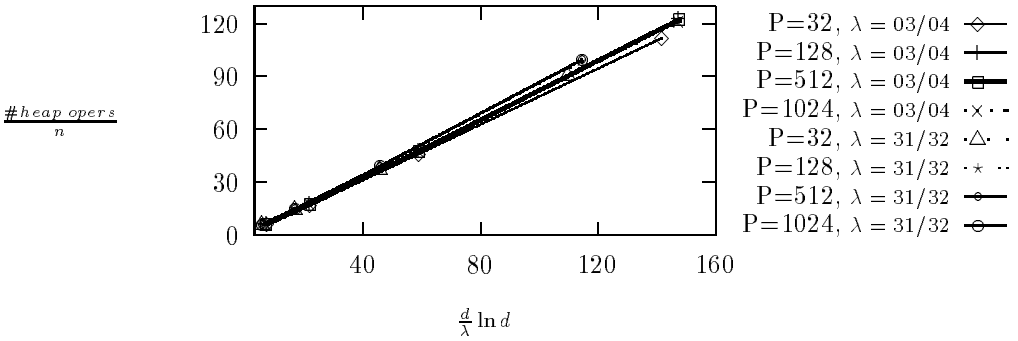


Figure 10: (*Number of heap operations / n*) versus $(\frac{d}{\lambda}\ln d)$.

most efficient message size at permutation pm_k. (We develop methods to choose the value of M^k_{thresh} in the next section.) Step 5 takes $O(n \log d)$ time.

The maximum message size allowed to be sent in one iteration is M_{thresh} (each iteration may have a different value of M_{thresh}, which is decided by the function *Decide_Size()*). Supposing the threshold is chosen so that only $(n-k)$ messages are greater than the threshold, we set $\lambda = \frac{k}{n}$.

The algorithm in Figure 9 can be decomposed into two stages. The first stage performs only the scheduling required for all communication phases. The second stage performs all necessary communication. For ease of explanation, we have combined these two stages. The worst case computational complexity of the algorithm is $O(Cdn)$, where C is the number of permutations generated by this algorithm. This assumes that all of the entries are searched in every iteration (Step 3 of Figure 8)

However, one would expect that on an average the algorithm should have much better behavior. The analysis is very difficult as it depends on several parameters (n, d, sizes of different messages, destinations of different messages). Further, the number of messages to be sent (and received by every processor) may be different at intermediate stages, even though this value may be the same for all nodes before the beginning of first stage.

The number of heap operations in Step 2 (Figure 9) was measured for different values of n and λ for randomly generated communication matrices with uniform message sizes. We have plotted *number of heap operations / n* against $\frac{d \ln d}{\lambda}$ in Figure 10. In this simulation, we arbitrarily picked up $n(1-\lambda)$ messages in each permutation (to simulate the $(n-k)$ messages that are greater than the threshold M_{thresh} in the permutation) and put them (entire messages) back in the heap. The results show that the number of heap operations in Step 2 is approximately $O(\frac{dn}{\lambda}\ln d)$. Thus, the time taken for this step could be approximated by $O(\frac{dn}{\lambda}\log^2 d)$. This shows that the expected behavior of this algorithm could be much better then the worst case. In Section 6, we propose several schemes to choose the value λ.

6 Approaches for Evaluating λ

When the message sizes in one permutation are non-uniform, communication time is bounded by the maximum message size in that permutation and processors with smaller message size may be idle. A suitable value of λ needs to be found to decide the threshold for message size to be sent in one permutation.

In function *Decide_Size()*, the first step is to sort all messages in one iteration by their size. There are several schemes that can be used to decide on an appropriate value of λ.

6.1 Fixed λ

The most straightforward approach is for λ to be fixed throughout the entire scheduling. This approach requires running the application program several times with different values of λ in order to find the best value. If the algorithm needs to be executed at runtime, each processor can begin with a different λ to schedule the communication. The processor with the minimum estimated communication time will send the schedule generated to other processors. This can then be used by all processors to carry out the communication.

6.2 λ Proportional to d

In this approach, the value λ is proportional to the value of d^*[3] at the current stage. For example, λ can be set as $0.8d^*$, where d^* is the average number of active entries in each row at the current stage. The implementation of this scheme is similar to "Fixed λ" approach.

6.3 Incremental Approach

In Figure 11, when value λ increases by $\triangle\lambda$, the message size will increase by $\triangle M$. This will affect the communication cost in the following ways:

- Since the maximum message size is increased by $\triangle M$, the cost of this extra communication $= \triangle M \times \varphi$.
- The additional utilization of bandwidth $= (1 - \lambda) \times \triangle M \times \varphi$.
- The approximate reduced cost due to decrease in set up cost $\approx \triangle\lambda \times \tau$.

Thus we should choose $\lambda + \triangle\lambda$ instead of λ if

$$(1 - \lambda) \times \triangle M \times \varphi \geq \triangle M \times \varphi - \triangle\lambda \times \tau$$

$$\Longrightarrow \triangle M \times \lambda\varphi \leq \triangle\lambda \times \tau$$

[3] We denote d^* as the expected average number of active entries in each row of $CCOM$ after one iteration of scheduling.

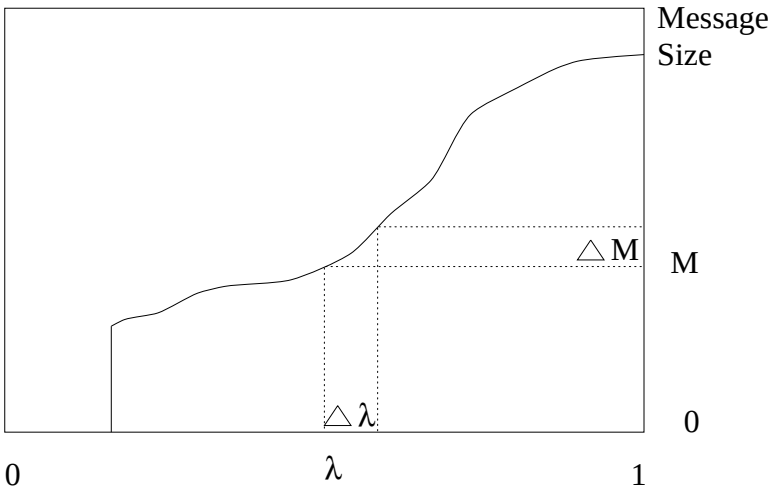


Figure 11: λ versus M graph.

$$\Longrightarrow \lambda \leq \frac{\triangle \lambda \tau}{\triangle M \varphi} . \tag{1}$$

The above analysis assumes that all permutations are completed synchronously. Clearly, this is not the case in the RS_NH algorithm given in Figure 9, in which some processors may begin the next permutation while other processors are still executing the current permutation.

7 Experimental Results

We have implemented our algorithms on a 32-node CM-5. In this section we describe the different versions of our algorithms tested and different data sets used for their evaluation.

Preliminary simulation results show that for schemes which use fixed value of λ, by the time the average number of messages left on nodes (after some iterations) is close to 1, the number of entries left in each row are uneven. Further, the degree of unevenness increases if one chooses a smaller value of λ. This effect is amplified for large values of n. Hence, we used a two-phase approach for choosing λ. In the first phase, we use one of the approaches presented in Section 6 until d^* is reduced to a small value (we use $\max\{2, \frac{d}{16}\}$ in this paper). Then, in the second phase (where d^* is small), λ is reset to 1, i.e., completely send out every message in one permutation.

7.1 Algorithms

In our experiments we used the following algorithms:

1. **AC** (the Asynchronous Communication algorithm).

2. **LP** (the Linear Permutation algorithm).

3. **RS_N**. This is essentially the same as the RS_NH algorithm, but the RS_N algorithm assumes that all the messages are of equal size and does not employ any heap operation.

4. **RS_NH**. The RS_NH algorithm with "Incremental" approach. Let $\lambda_k = \lambda_0 + k \times \frac{1}{n}$, where $\lambda_0 = 0.75$ and $0 \leq k \leq 0.25$. We define

$$Gain_k = \frac{\triangle\lambda}{\lambda_k} \cdot \frac{\tau}{\varphi} - \triangle M_k \ ,$$

   and λ is chosen to be λ_k such that

$$\sum_{i=0}^{k-1} Gain_i$$

   is maximized. The additional complexity of choosing λ by using this scheme is $O(n)$ per iteration.

5. **RS_NH+fixed**. The RS_NH algorithm with fixed value of λ. We experimented with the following λ values: $\frac{3}{4}, \frac{7}{8}, \frac{15}{16}, \frac{31}{32}$, and 1.0. In each instance we used the best performance among different values of λ to represent the performance (including number of permutations, scheduling cost, and communication cost) of this algorithm.

6. **RS_NH+($\lambda = 1$)**. This scheme is equivalent to the RS_NH+fixed with $\lambda = 1$ throughout the scheduling. We maintained the heap structures during the process, and let the messages in every permutation be completely sent out (i.e., there are no message splitting operations).

7. **RS_N+sort**. This algorithm is the same as RS_N except for the fact that we sort the active entries in each row of *CCOM* by message size at the beginning of the scheduling algorithm. We sort the rows only once, and do not make an effort to maintain the sort sequences during the scheduling. In contrast, RS_NHs maintain the sort sequences throughout the scheduling.

All the algorithms are executed in a loosely synchronous fashion. We did not explicitly use global synchronization to enforce synchronization between communication stages in any of the algorithms proposed above.

7.2 Data Sets

The data sets for our experiments can be classified into three categories:

1. This test set contains two subgroups, each of which has 50 different communication matrices with the same value of d. In each matrix, every row and every column have

approximately d active entries (d is equal to 8 and 16, respectively). The procedure we used to generate these test sets is described in [27].

The messages in one communication phase are non-uniform, where the size is equal to $COM(i,j)$ multiplied by msg_unit. The different values of msg_unit used in this test set are 2^k for $4 \leq k \leq 13$.

2. This test set (skewed distribution) contains samples with skewed size distribution. Three information arrays can be used to represent the characteristics of these samples: $dist[5] = \{1, 2, 4, 8, 17\}$, $dense[5] = \{1, 2, 4, 8, 16\}$, and $length[5] = \{16, 8, 4, 2, 1\}$. The rows of COM are grouped into five sets. Set k ($1 \leq k \leq 5$) can be characterized by $dist[k]$, $dense[k]$, and $length[k]$. $dist[i]$ = number of rows in the set i; $dense[i]$ = number of active entries in a row belonging to the set i; and $length[i]$ = length of each message in the set i.

   The motivation of this test set is to observe the case where a few processors have a small number of large messages, while other processors have a bulk of small messages. The total amount of data to be sent by every processor is equal. The different values of msg_unit used for our experiments are 2^k for $4 \leq k \leq 14$.

3. This test set contains communication matrices generated by graph partitioning algorithms [16]; the samples represent fluid dynamics simulations of a part of an airplane (Figure 12) with different granularities (2800-point and 53961-point). In order to observe the algorithm's performance with different message sizes, we multiplied the matrices in this test set by a variable msg_unit. The different values of msg_unit used for our experiments are 2^k for $4 \leq k \leq 12$.

In the test set 3, the number of messages sent (or received) by each node is uneven. For example, for the 2800-point sample we have the following parameters:

1. The maximum number of messages sent by any processor = 15.
2. The minimum number of messages sent by any processor = 3.
3. The average number of messages sent by any processor = 9.25.
4. The maximum length of all messages = 36 units.
5. The minimum length of all messages = 1 unit.
6. The average length of all messages = 14.2 units.

The corresponding values for the 53961-point sample are 18, 6, 10.81, 276, 1, and 93.21, respectively.

7.3 Results and Discussion

The scheduling costs of various algorithms do not include the time for the following operations:

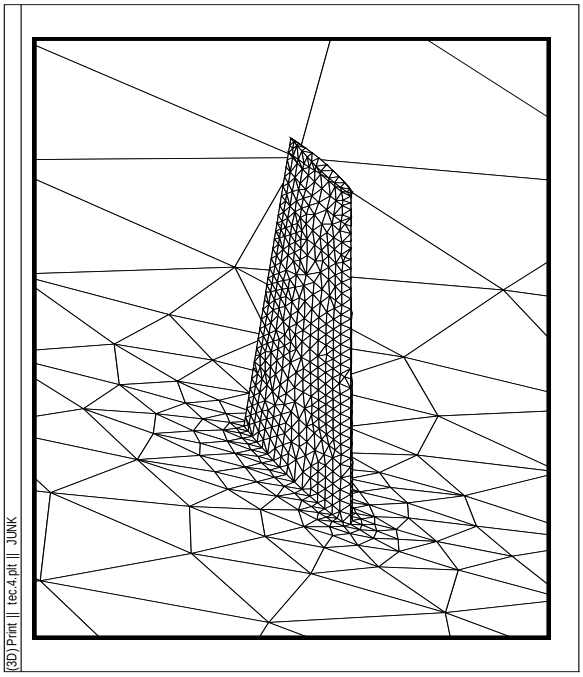


Figure 12: The unstructured grid used for our simulations.

1. Time to compress COM into $CCOM$ (RS_Ns and RS_NHs, which will take $O(n^2)$ time in the sequential mode and $O(dn + \tau \log n)$ time in the parallelized version).

2. Time to sort $CCOM$ at the beginning of scheduling for RS_N+sort, which will take $O(nd \log d)$ time in the sequential mode and $O(dn)$ time in the parallelized version.

3. Time to create heaps in $CCOM$ at the beginning of scheduling (RS_NHs), which will take $O(nd)$ time in the sequential mode as well as in the parallel version.

The main reasons for not including these timings are that they would be different in the static (sequential) and runtime (parallel) version. Although the time complexity of some of these operations looks very high, it is worth noting that these operations are executed only once during the scheduling. So the constant values before of the complexity terms are very small when compared with the constant before of the complexity terms of the scheduling cost.

Clearly, one could add these costs to the costs given in this section to get a more accurate estimate of the total cost. Table 4 shows that the exclusion of most of the above operations affects the total cost by only a small fraction. The sort portion of RS_N+sort is expensive; however, our experimental results (in the later sections) reveal that this method provides no

d	$\frac{compress}{comp}$	$\frac{heap}{comp}$	$\frac{sort}{comp}$
4	0.206	0.108	0.445
8	0.087	0.095	0.855
16	0.037	0.075	1.435
24	0.023	0.065	1.575

Table 4: Compress, heap, and sorting overhead in terms of corresponding scheduling cost for sequential execution.

improvement over RS_N in terms of the total cost of communication (RS_N has a significantly lower scheduling cost).

7.3.1 Uniform Distribution

Table 5 and Figure 13 show the results of $d = 8$ and $d = 16$. Results show that RS_N outperforms AC and LP by a big margin. RS_N+sort does not provide improvement over RS_N. The different variations of RS_NHs have very similar results, all of which provide a considerable improvement over RS_N. This clearly shows the usefulness of heap structures and thresholding to reduce the variance of messages in one permutation.

When $d = 16$, the performance difference between algorithms becomes prominent. Thus, when the density or message size increases, the RS_NH algorithms are the algorithms of choice.

Figure 14 shows that maintaining heaps (which are used in RS_NHs) is expensive. The overhead fraction of RS_N is less than 0.25 for messages of size 16K on a 32-node CM-5. The overhead of RS_NH remains high when the message size is less than 16K ($msg\_unit = 2^9$); it becomes negligible for larger messages. This overhead computation is based on the assumption that the same schedule is used only once. In most applications the same schedule is utilized many times, hence the fractional cost would be considerably lower (inversely proportional to the number of times the same schedule is used). In such cases, all our algorithms are also suitable for runtime scheduling.

7.3.2 Skewed Distribution

In test set 2, the total number of messages sent by every processor is same. This characteristic makes RS_NH+($\lambda = 1$) useless. This is because the heap structure will keep the active entries in each row in a similar order. This should, in general, make the probability of finding an entry in each row non-random and result in more permutations and larger communication cost. Our experimental results support this fact.

The rows with larger messages have a smaller number of messages, and the rows with the smallest messages have the largest number of messages, which in turn will dominate

the number of permutations needed. Thus, the splitting of large messages should even out the message sizes in one permutation without significantly increasing the number of permutations.

Table 6 and Figure 15 show the results of test set 2. As expected, the RS_NH+($\lambda = 1$) has a similar performance to that of RS_N. The results also show RS_NH and RS_NH+fixed have clear improvements over other approaches.

7.3.3 Airfoil Mesh

Table 7 and Figure 16 show the results for a 2800-point and 53961-point sample, respectively. The results for both samples have behavior similar to the first test set, which reveals that even if the number of messages in each row is non-uniform, our algorithms maintain their characteristics and performance. The RS_NHs are superior when the *msg_unit* becomes large, which in turn means that it is worth the extra effort (of using heap and message breaking) to reduce the variance of message sizes in each permutation. These results also show the comparison of fixed λ and variable λ (incremental approach). The observation reveals that the two methods have comparable performance. So for static applications (which can be pre-run to find the best value of λ), a fine-tuned fixed value of λ may be as good as (or even better than) the dynamic values of λ found during the scheduling. We can potentially run the algorithms for different values of λ in parallel and choose the best one; however, it is difficult to estimate the actual performance (with varying λ) and choose the best value of λ.

7.4 Discussion

It is hard to make generalizations on which algorithms are better, based on the limited number of experimental results presented above. In general, scheduling costs vary in the following manner:

$$S\_cost(\text{AC}) \leq S\_cost(\text{LP}) \leq S\_cost(\text{RS\_Ns}) \leq S\_cost(\text{RS\_NHs}) \ ;$$

while the communication costs vary in the following fashion:

$$C\_cost(\text{RS\_NHs}) \leq C\_cost(\text{RS\_Ns}) \leq C\_cost(\text{LP}) \leq C\_cost(\text{AC}) \ .$$

Clearly, depending on the structure of the communication matrix and the number of times a particular schedule is used, one method may be superior to another. However, if the number of times the same schedule is utilized is large, RS_NH seems to be a better approach.

8 Conclusions

In this paper we have developed several algorithms for scheduling all-to-many personalized communication with non-uniform message sizes. The performance of the asynchronous communication algorithm (AC) depends on network congestion. The memory requirements of

this algorithm are large. This algorithm is only suitable for small message sizes. The linear permutation algorithm (LP) is very straightforward and introduces little computation overhead, but it needs to go through the same number of communication phases ($n-1$) even if the density d is small.

The RS_NH algorithms are found to be very useful in handling non-uniform messages. The use of a heap structure to maintain the sort sequences so that the bigger messages will be scheduled earlier, and the decomposition of large messages into smaller messages, give a significant reduction of the total time required for communication.

We have proposed three approaches to decide the value λ (the number of complete messages sent out in every phase of communication). The first two require pre-running for several fixed values of λ, while the third chooses the value on-the-fly. Experimental results have shown that our algorithms perform well with artificially generated samples as well as with samples from an actual application.

Another advantage of our algorithms as compared to the other algorithms is that once the schedule is completed, communication can potentially be overlapped with computation, i.e., computation on a packet received in the previous phase can be carried out while the communication of the current phase is being performed. It is also worth noting that due to compaction, nearly all processors receive data packets (of nearly equal size). If any computation needs to be performed using incoming data and it is proportional to the size of the packet, it should lead to good load balance.

There is a large amount of literature on how to partition a task graph so as to minimize communication cost. A few methods that are iterative in nature can be found in [16]. After a particular threshold any improvement in partitioning is expensive. For problems requiring runtime partitioning, it is critical that this partitioning be completed extremely fast. For such problems, the gains provided by effective communication scheduling may far outweigh the gains obtained by spending the same amount of time on achieving better partitioning.

For different applications, different kinds of communication patterns are used. It is unclear which methods will be better than others for specific classes of communication patterns. However, we do believe that our methods can significantly reduce the total time of communication. Choosing the best method among the variety of algorithms presented in this paper will depend on the underlying architecture, the type of communication patterns, and on whether the scheduling has to be performed statically or at runtime.

One of the issues we have not addressed in this paper is link contention. On the CM-5, link contention does not significantly affect the communication cost of the schedules generated by our algorithms. We are currently developing algorithms for architectures on which link contention is an important issue.

Appendix A: Compressing Procedure

```
for i = 0 to n − 1 do
    k = 0
    for j = 0 to n − 1 do
        if COM(i, j) = 1 then
            CCOM(i, k) = j;
            k = k + 1;
        endif
    endfor
    prt(i) = k − 1;
    Randomly swap CCOM(i, 0..prt(i));
endfor
```

Acknowledgments

We wish to thank Geoffrey Fox and Joel Saltz for several discussions on topics related to this paper. we would also like to thank Raja Das, Joel Saltz, and Dimitri Mavriplis at ICASE and Nashat Mansour for the illustration in Figure 12 and for the corresponding communication matrices. Thanks are also due to the anonymous reviewers for their many insightful comments and suggestions for improvement.

This work was supported in part by NSF under CCR-9110812 and in part by DARPA under contract #DABT63-91-C-0028. The contents do not necessarily reflect the position or the policy of the United States government and no official endorsement should be inferred.

This work was supported in part by a contract between the Army Research Office and the University of Minnesota for the Army High Performance Computing Center.

References

[1] E. Anderson and Y. Saad. Solving Sparse Triangular Linear Systems on Parallel Computers. *International Journal of High Speed Computing*, 1(1):pp. 73–95, 1989.

[2] I. Angus, G. Fox, J. Kim, and D. Walker. *Solving Problems on Concurrent Processors*, volume 2. Prentice Hall, Englewood Cliffs, NJ, 1990.

[3] M. Barnett, D.G. Payne, and R. Geijn. Optimal Broadcasting in Mesh-Connected Architectures. Technical report, University of Texas at Austin, December 1991.

[4] D. Baxter, J. Saltz, M. Schultz, S. Eisentstat, and K. Crowley. An Experimental Study of Methods for Parallel Preconditioned Krylov Methods. In *Proceedings of the 1988*

Hypercube Multiprocessor Conference, pages pp. 1698–1711, Pasadena, CA, January 1988.

[5] Zeki Bozkus, Sanjay Ranka, and Geoffrey C. Fox. Benchmarking the CM-5 Multicomputer. In *Proceedings of the Frontiers of Massively Parallel Computation*, pages pp. 100–107, McLean, VA, October 19-21 1992.

[6] D. Callahan and K. Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *Journal of Supercomputing*, 2:pp. 151–169, October 1988.

[7] Alok Choudhary, Geoffrey C. Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Sanjay Ranka, and Chau-Wen Tseng. Compiling Fortran 77D and 90D for MIMD Distributed-Memory Machines. In *Proceedings of the Frontiers of Massively Parallel Computation*, pages pp. 4–11, McLean, VA, October 19-21 1992.

[8] Alok Choudhary, Geoffrey C. Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Sanjay Ranka, and Joel Saltz. Software Support for Irregular and Loosely Synchronous Problems. *Journal of Computing Systems in Engineering*, 3:pp. 43–52, 1993.

[9] 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.

[10] Geoffrey C. Fox. The Architecture of Problems and Portable Parallel Software Systems. Technical Report Revised SCCS-78b, Syracuse University, July 1991.

[11] E. Horowitz and S.Sahni. *Fundamentals of Data Structures in Pascal*. Computer Science Press, Maryland, second edition, 1987.

[12] S. Lennart Johnsson and Ching-Tien Ho. Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Trans. on Computers*, 38(9):pp. 1249–1268, September 1989.

[13] Charles Koelbel and Piyush Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):pp. 440–451, October 1991.

[14] Charles E. Leiserson, Zahi S. Abuhamdeh, and David C. Douglas et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Symposium on Paralle Algorithms and Architectures*, pages pp. 272–285, 1992.

[15] P.C. Liewer and V.K. Decyk. A General Concurrent Algorithm for Plasma Particle-in-Cell Simulation Codes. *Journal of Computational Physics*, 2:pp. 302–322, 1985.

[16] Nashat Mansour. *Parallel Genetic Algorithms with Application to Load Balancing for Parallel Computing*. PhD thesis, Syracuse University, Syracuse, NY 13244, 1992.

[17] D.J. Mavriplis. Three Dimensional Unstructured Multigrid for the Euler Equations. In *AIAA 10th Computational Fluid Dynamics Conference*, page pp. 91, June 1991.

[18] R. Mirchandaney, J.H. Saltz, R.M. Smith, D.M. Nicol, and Kay Crowley. Principles of Runtime Support for Parallel Processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages pp. 140–152, St. Malo, France, July 1988.

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

[20] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse. In *Proceedings of the Supercomputing '93*, pages pp. 361–370. IEEE Press, November 1993. Also available as University of Maryland Technical Report CS-T-3055 and UMIACS-TR-93-32.

[21] Sanjay Ranka and Sartaj Sahni. *Hypercube Algorithms with Applications to Image Processing and Pattern Recognition.* Springer-Verlag, 1990.

[22] Sanjay Ranka, Jhy-Chun Wang, and Geoffrey C. Fox. Static and Runtime Algorithms for All-to-Many Personalized Communications on Permutation Networks. In *Proceedings of the 1992 International Conference on Parallel and Distributed Systems*, pages pp. 211–218, HsinChu, Taiwan, December 1992.

[23] A. Rogers and K. Pingali. Process Decomposition Through Locality of Reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages pp. 69–80, Portland, OR, June 1989.

[24] Y. Saad. Communication Complexity of the Gaussian Elimination Algorithm on Multiprocessors. *Linear Algebra Application*, 77:pp. 315–340, 1986.

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

[26] D.W. Walker. Characterizing the Parallel Performance of a Large-Scale, Particle-in-Cell Plasma Simulation Code. *Concurrency: Practice and Experience*, pages pp. 257–288, 1990.

[27] Jhy-Chun Wang. *Load Balancing and Communication Support for Irregular Problems.* PhD thesis, Syracuse University, Syracuse, NY 13244, 1993.

[28] Hans P. Zima, Heinz-J. Bast, and Michael Gerndt. SUPERB: A Tool for Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, 6:pp. 1–18, 1988.

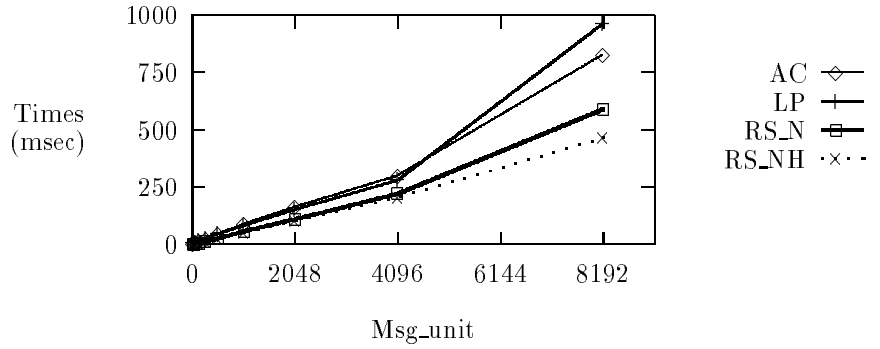
d	*msg_unit*	AC	LP	RS_N	RS_N +sort	RS_NH	RS_NH +($\lambda = 1$)
8	comm*						
	16	3.820	7.943	3.380	4.066	3.839	3.777
	64	8.124	11.463	5.455	6.370	5.879	5.901
	256	24.873	26.771	15.101	16.840	15.176	15.291
	1024	89.027	83.063	57.825	59.436	53.744	54.560
	4096	301.681	282.814	222.201	225.684	207.420	209.661
	8192	830.939	967.832	592.921	656.096	467.793	519.234
	comp†	0	0.091	3.211	3.245	16.872	10.1
	perm‡	0	31.0	10.1	10.22	11.2	10.14
16	comm						
	16	8.178	9.514	6.408	7.653	7.050	7.126
	64	17.780	16.112	10.494	12.152	10.959	11.212
	256	52.173	43.161	29.385	32.330	28.607	29.121
	1024	176.308	144.127	112.133	114.414	101.869	103.660
	4096	819.440	971.286	588.601	601.386	396.460	400.644
	8192	2916.056	2851.732	1609.473	1633.950	1309.655	1310.013
	comp	0	0.091	6.57	6.62	45.403	31.502
	perm	0	31.0	18.56	18.52	19.8	18.8

$*$: Communication cost, in milliseconds.

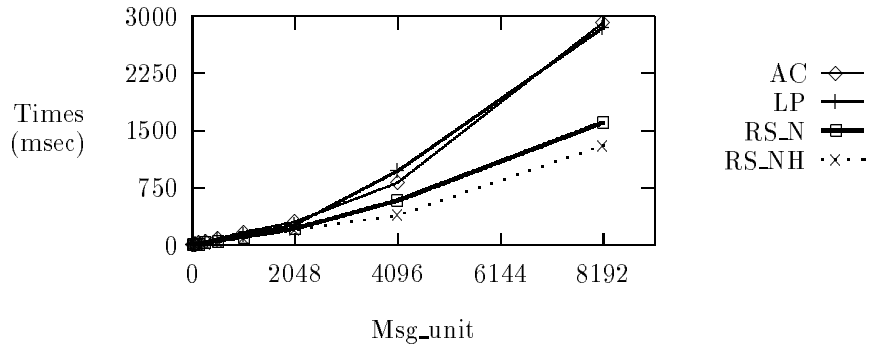
†: Scheduling cost, in milliseconds.

‡: Number of communication phases needed.

Table 5: Experimental results for non-uniform message sizes on a 32-node CM-5. The minimum message size in each level is *msg_unit* bytes, and the maximum size is $32 \times$ *msg_unit* bytes.



(density $d = 8$)



(density $d = 16$)

Figure 13: Communication cost for non-uniform message sizes on a 32-node CM-5.

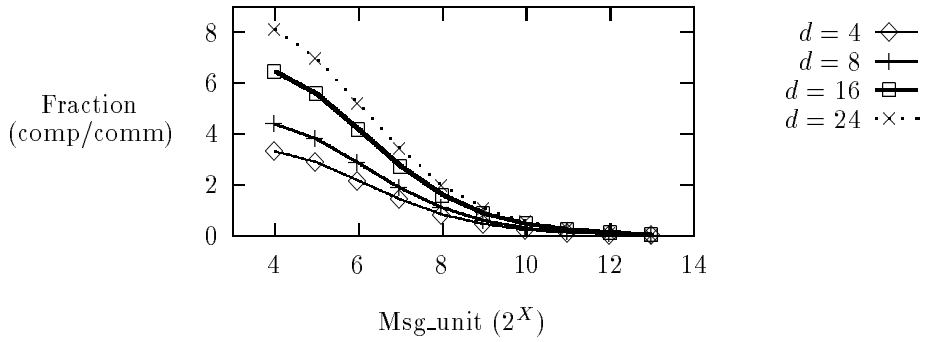


Figure 14: Computation overhead of RS_NH algorithm in terms of communication cost.

msg_unit	AC	LP	RS_N	RS_N +sort	RS_NH	RS_NH +($\lambda = 1$)	RS_NH +fixed
comm							
16	5.893	9.049	6.673	6.711	10.111	6.722	6.485
64	8.231	10.066	7.490	7.552	11.052	7.494	7.398
256	15.841	15.938	12.911	12.928	15.705	12.876	12.279
1024	44.761	40.159	36.977	36.741	36.655	36.513	32.722
4096	154.052	134.647	132.543	131.628	119.861	130.678	114.365
16384	813.610	904.941	949.817	1003.330	707.041	967.669	598.615
comp	0	0.097	7.678	8.84	43.41	21.77	34.451
perm	0	31.0	20.1	20.2	31.7	20.4	21.45

Table 6: Experimental results for skewed distribution pattern on a 32-node CM-5. The minimum message size in each level is *msg_unit* bytes, and the maximum size is $16 \times$ *msg_unit* bytes.

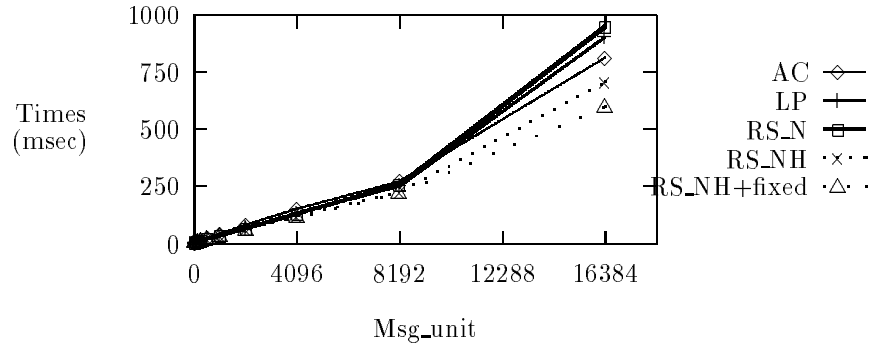


Figure 15: Communication cost for skewed distribution.

points	*msg_unit*	AC	LP	RS_N	RS_N +sort	RS_NH	RS_NH +($\lambda = 1$)	RS_NH +fixed
2800	comm							
	16	5.340	8.959	5.595	5.632	7.272	5.624	5.409
	32	6.710	9.762	6.258	6.264	7.886	6.264	6.122
	64	9.674	11.991	7.879	7.837	9.284	7.717	7.606
	128	15.323	16.861	11.478	11.359	12.226	10.805	10.875
	256	25.870	26.322	19.502	18.986	18.607	17.690	17.274
	512	47.209	44.454	35.147	34.045	31.365	31.247	30.076
	1024	86.679	79.324	65.342	63.657	57.582	57.224	55.537
	2048	165.237	146.995	125.460	119.634	108.972	110.711	104.951
	4096	297.637	283.917	232.721	225.080	208.906	209.687	197.226
	comp	0	0.097	5.052	5.03	29.38	14.523	22.137
	perm	0	31.0	15.15	15.2	19.65	15.45	15.55
53961	comm							
	16	16.103	17.941	12.907	12.718	14.920	11.700	12.253
	32	26.826	27.349	20.965	20.619	21.536	18.532	18.950
	64	48.367	46.552	37.662	36.642	35.513	32.599	32.771
	128	87.700	80.769	69.874	67.731	63.126	60.816	60.148
	256	163.598	149.746	135.387	129.456	118.149	115.609	113.558
	512	300.644	280.240	256.659	250.574	228.418	225.190	219.322
	comp	0	0.097	6.059	6.024	40.231	19.245	28.396
	perm	0	31.0	18.05	18.15	26.4	18.15	20.05

Table 7: Experimental Results for airfoil mesh simulations on a 32-node CM5. The minimum message size in each level is *msg_unit* bytes, and the maximum size is 36 (for grid of size 2800) and 276 (for grid of size 53961) $\times$*msg_unit* bytes.

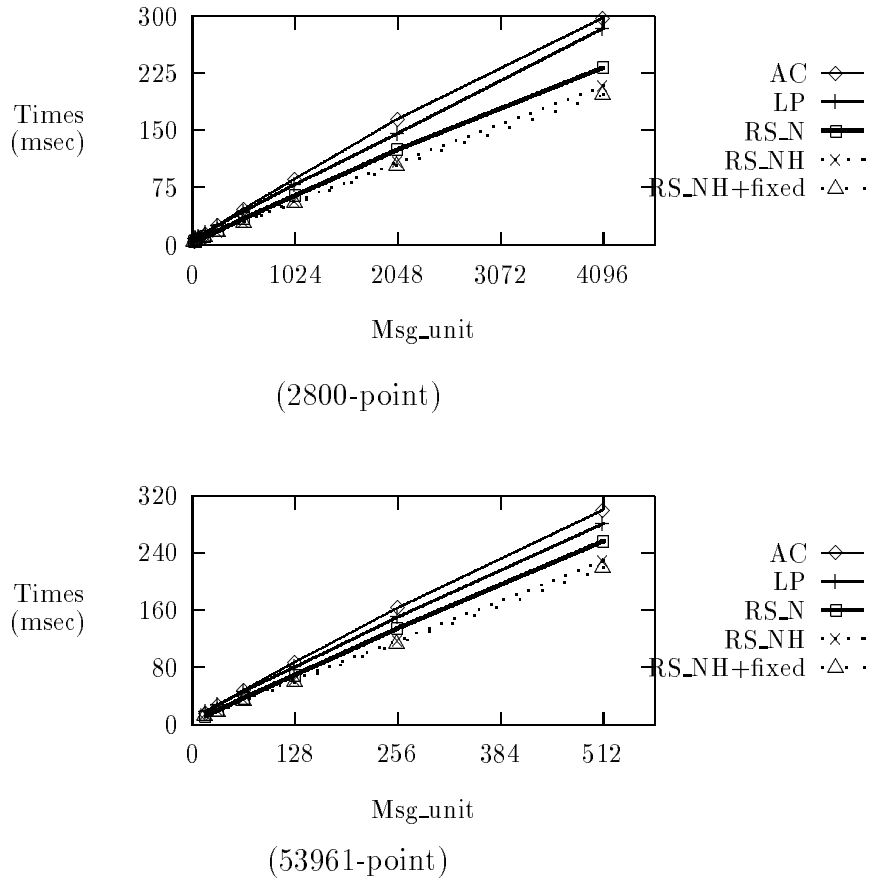


Figure 16: Communication cost for airfoil mesh simulation on a 32-node CM-5.