Irregular Personalized Communication on Distributed Memory Machines

Sanjay Ranka[1] and Jhy-Chun Wang[1]

4-116 Center for Science and Technology
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244

Manoj Kumar

IBM T.J. Watson Research Center
Hawthorne, NY

April 22, 1993

[1]Work was supported in part by NSF under CCR-9110812 and in part by DARPA under contract # DABT63-91-C-0028. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Abstract

In this paper, we present several algorithms for performing all-to-many personalized communication on distributed memory parallel machines. 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 as well as runtime scheduling.

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

1 Introduction

For distributed memory parallel computers, load balancing and reduction of communication are two important issues for achieving a good performance. It is important to map the program such that the total execution time is minimized; the mapping can be typically 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) [6]. This typically results in regular collective communication between processors. Many such primitives have been developed in [2, 19].

For a large class of scientific problems, which are irregular in nature, achieving a good mapping is considerably more difficult [7]. Further, the nature of this irregularity may not be known at the time of compilation, and can be derived only at runtime. The handling of irregular problems requires the use of runtime information to optimize communication and load balancing [9, 13, 17]. These packages derive the necessary communication information based on the nonlocal data required for performing the local computations.

Consider the parallelization of single concurrent computational phase of an explicit unstructured mesh fluids calculation (e.g.[25]). 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 (e.g.[21]). 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 (e.g. [16]), parallelized sparse triangular solver (e.g. [1, 4]), particle-in-cell codes (e.g. [14, 24]), and vortex blob calculations [3].

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 run time, this may not be done directly by the compiler; instead, calls to a run-time environment need be generated to do the partitioning. Several algorithms are available in the literature to perform this partitioning (see [15] for a detailed list of such references).

The partitioning described in Figure 2 generates a 8×8 communication matrix COM (Table 1). A "1" in the (i, j) entry represents processor P_i needs to communicate to processor P_j. Each message is of different size and each processor may send 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 out their outgoing messages, it may happen that at one stage processors P_0, P_1, P_3, P_4 and P_6 all try to send messages to processor P_2. Since the receiving processor can typically receive messages from one processor at a time,

	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: A 8×8 communication matrix (blank entries imply no communication)

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 the above situation. We will show that node contention has a deteriorating effect on the total time required for communication.

In this paper we develop and analyze 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.

Assuming a system with n processors, our algorithms take as input a communication matrix $COM(0..n-1, 0..n-1)$. $COM(i,j)$ 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, $pm^1, pm^2, \cdots, pm^l$, l is a positive integer, such that if $COM(i,j) = m$ then there exists a k, $1 \leq k \leq l$, that $pm_i^k = j$.

The communication matrix of Table 1 may be decomposed into 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 sends at most one message 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 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 [8, 18, 23], the distance to which a message

```
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

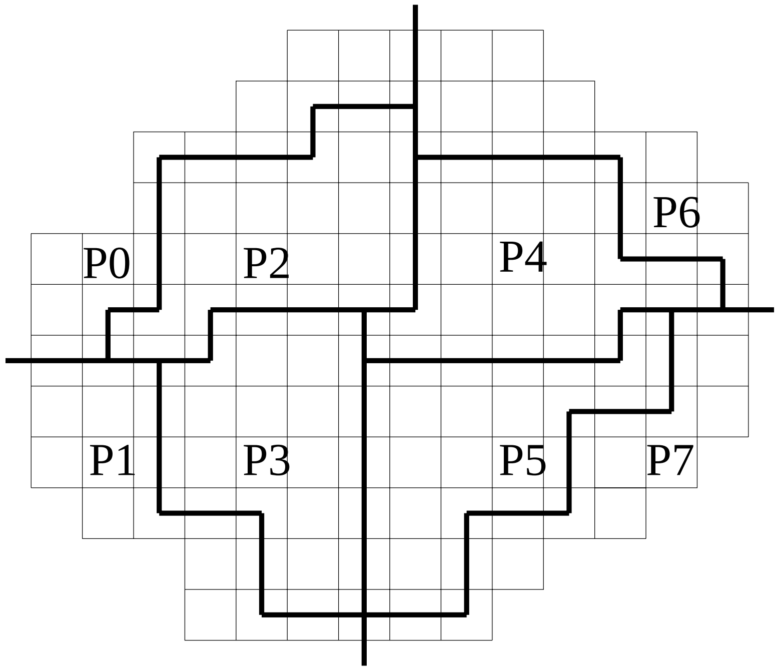

Figure 2: The partitioning of irregular mesh

is sent is becoming relatively less and less important. 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 would 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 (specially 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) one can build partial permutations which satisfy the property that no two messages interact. However, this would depend on the topology and the routing methodology and would increase the cost of obtaining a good schedule.

In this paper, we have not addressed link contention. One of the main reasons is that on CM-5 the routing is randomized. It is not easy to statically schedule messages in a fashion that link contention can be avoided (of course randomization alleviates the problem to some extent). On a 32 node CM-5, we generated 5000 random permutations in which each processor sends and receives a message of 1K bytes. Over 99.5% (4979 out of 5000) of the permutations were within 5% of the average cost (over 5000 random permutations) (Figure 4). Thus, 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. The algorithms developed in this paper can be extended to the architectures where link contention is an important issue by decomposing into partial permutation 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 [6]), the fractional cost of the scheduling algorithm is quite small. Further, compared to the naive algorithms, our algorithm can result in significant reduction in the total amount of communication.

The rest of the paper is organized as follows. Section 2 gives the notation and assumptions made for developing our message scheduling algorithms, and an overview of CM-5. Section 3 presents a simple asynchronous communication algorithm. Section 4 develops algorithms that will avoid node contention and discusses their time complexity. Section 5 proposes approaches to reduce the message size variance in each permutation. Section 6 presents experimental results for a 32 node CM-5. Finally, conclusions are given in Section 7.

2 Preliminaries

A $n \times n$ communication matrix COM can be decomposed into a set of communication phases, cp^k, $1 \leq k \leq l$, l is a positive integer, such that

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

We define the kth communication phase as:

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

if processor P_i need to send message to processor P_j at the kth phase, otherwise $cp_i^k = -1$.

Thus, *node contention* can be formally defined as:

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

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,

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

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

$pm_i^k = -1$ if P_i does not send out message at this permutation.

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

2.1 System Overview: CM-5

This section gives a brief overview of the CM-5 system which we used to conduct our experiments. The CM-5 is available in configuration of 32 to 1024 processing nodes, each node is 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. It 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.

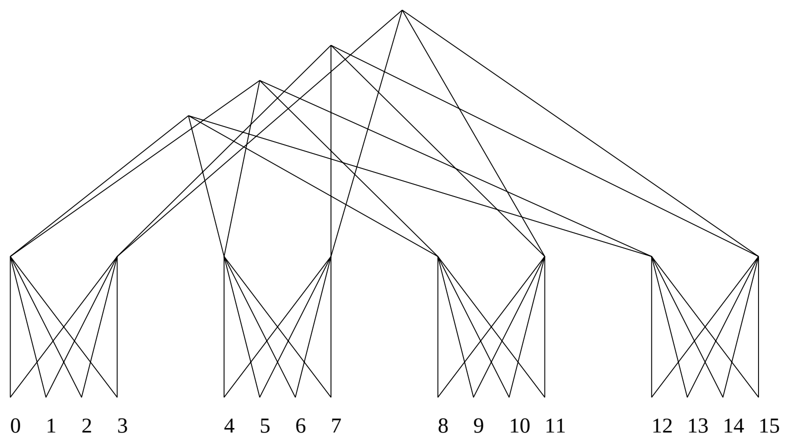

Figure 3: CM-5 data network with 16 nodes

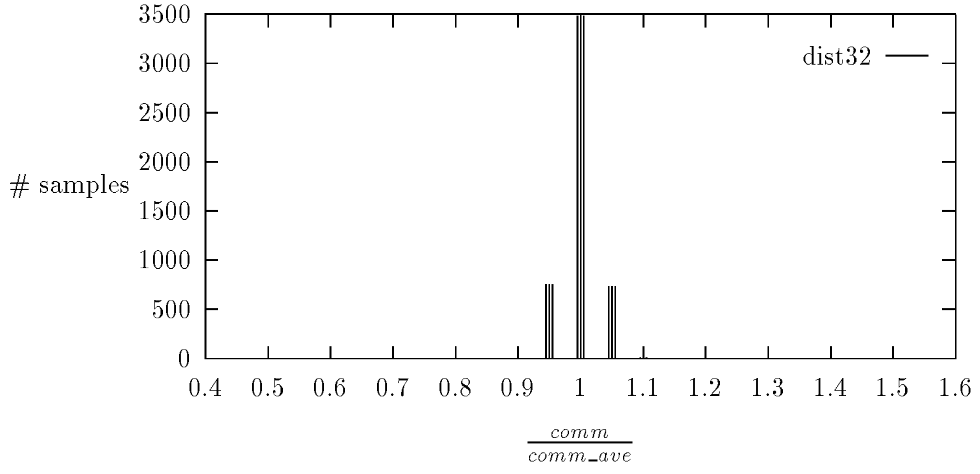

Figure 4: Communication cost distribution for 5000 permutation samples with message of length 1K bytes on a 32 node CM-5

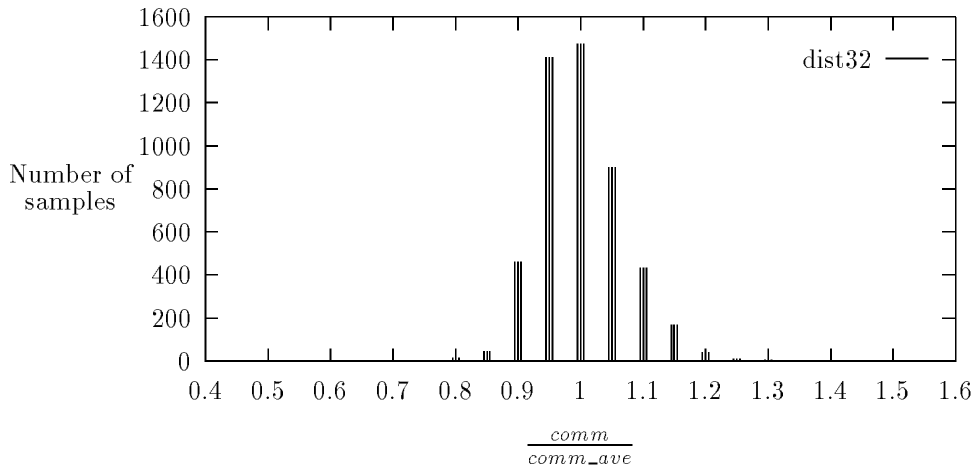

Figure 5: Communication cost distribution for 5000 permutation samples with message of length 256K bytes on a 32 node CM-5

2.2 Cost of Random Permutations

We randomly generate 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 Figure 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 performance 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 $P_{i \oplus dist}$[1], $0 \leq i < n$ and $dist = 1, 2, 4, 8, 16$. Each permutation represents a communication pattern where processors communicate with processors within cluster of 2, 4, 8, 16, and 32, respectively. The results for these permutations are given in Table 2. These results show that for 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. Our algorithms do not try to exploit these special permutations. However, we believe that our algorithms can be modified to exploit these special cases.

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

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

2.3 Notation and Assumptions

2.3.1 Notation

The communication matrix COM is a $n \times n$ 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 unit) to P_j, otherwise $COM(i,j) = 0$, $0 \leq i,j < n$. Thus, row i of COM represents the sending vector of processor P_i, which contains information about destination and size of different messages.

We categorize the scheduling algorithms into several different categories:

1. *Uniformity of the message* - All messages are of equal size or not. 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 [20] 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 few processors. Most of the algorithms developed in this paper assume that the latter is true. There are a number of algorithms for the totally dense cases [2, 12].

3. *Static or runtime scheduling* - The communication scheduling has to be performed statically or dynamically.

2.3.2 Assumptions

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

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

Figure 6: Asynchronous Communication Algorithm

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

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

3 Asynchronous Communication (AC)

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

1. each processor first post requests for incoming messages (this operation will pre-allocate buffers for those messages).

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

3. each processor checks and confirms incoming messages (some of them may already arrived at its receiving buffer(s)) from other processors.

During the send-receive process, the sender processor does not need to wait for a completion signal from the receiver processor, so it can keep sending outgoing messages till they are all done. This naive approach is expected to perform well when the 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 the network. Also, each processor may only have limited space of message buffers. In such cases, when the system buffer space is fully occupied by unconfirmed messages, further messages will be blocked at sender processors side. The overflow may block processors from doing further processing (include receiving messages) because processors are waiting for other processors to consume and empty their buffer to receive new incoming messages. This situation may never resolve and a dead lock 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 *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 be, in general, avoided. The experimental results described in this paper use the approach given in Figure 6.

4 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 a 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, 19]. In case that the communication matrix COM is sparse in nature, *i.e.* each processor will send and receive d messages (in a system with n processors, $d < n$), one 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 a $n \times d$ matrix such that each row is a sparse representation of the corresponding sending vector (we will discuss the details later).

4.1 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 message to processor P_j (but will receive 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 $P_{(i+k) \, mod \, n}$ and receives a message from $P_{(i-k) \, mod \, n}$, 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. One can easily extend this algorithm when n is not a power of 2.

4.2 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 a $n \times d$ 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 $O(dn)$.

```
**Linear_Permutation()**
For all processor $P_i$, $0 \le i \le n-1$, *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

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 a $n \times d$ 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 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 this entry doesn't contain active information. 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 smaller messages may be idle while waiting for processors with largest message to complete). In order to eliminate processors' idle time, we will introduce several approaches in next section to choose a reasonable message size in each communication phase such that processors with

```
RS_NH()

  1. Use matrix $COM$ to create a $n \times d$ 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

small messages will send their messages completely, while processors with large messages will only send part of their messages.

The RS_NH algorithm is described in Figure 8.

Step 1 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 3.1 takes $O(n)$ time. Step 3.3 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 3.4 is $O(\tau + \varphi M^k_{thresh})$ time (where M^k_{thresh} is the most efficient message size at permutation pm_k. We develop methods to choose the value of M^k_{thresh} in the next section). Step 3.5 takes $O(n \log d)$ time to complete.

We are interested in evaluating the average time complexity of step 3.2 and the average number of iterations to complete step 3. These steps reflect on the time spent on the scheduling algorithm. The algorithm in Figure 9 can be decomposed into two stages. The first stage only performs the scheduling required for all the communication phases. The second stage performs all the necessary communication. For ease of explanation, we have combined these two stages.

The analysis of the complexity of this step 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.

In the following we make simplifying assumptions to get an estimate for the complexity of our algorithms. Wherever possible, we support our assumptions based on the simulation results. This analysis assumes that all messages are of equal size. We also assume that at the beginning of each outer loop (of step 3.2), the value d (number of active entries) in each row is approximately equal and the destinations to which each node has to send data are random (between 1 and n).

```
Generate_Permutations()
For all processor $P_i$, $0 \le i \le n-1$, in parallel do
Repeat

  1. Set vectors $send = receive = -1$;

  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 $j \le$ 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()

With these assumptions, it can be shown that the number of iterations the *while* loop in step 3.2 is executed is proportional to $O(n \ln d + n)$ [20]. Each heap operation in step 3.2 will require $O(\log d)$ time. Thus the complexity of step 3.2 is $O(n \log^2 d)$.

To find out the number of times the outer loop is executed, we also need to find out the number of entries $CCOM(i,j)$ being consumed in one iteration, *i.e.* the number of entries $CCOM(i,j)$ being reset to -1 in one iteration. It can be shown that with the above assumptions, if every processor in one permutation sends a complete message, then the expected number of entries $CCOM(i,j)$ consumed in one iteration is at least $n - \frac{n}{d+1}$. The analysis is given in [20]. For completeness, we present the analysis in Appendix B.

The maximum message size allowed to be sent in one iteration is M_{thresh} (each iteration may have different value of M_{thresh}, which is decided by the function *Decide_Size()*). Suppose only k messages are smaller than M_{thresh}, then $(n-k)$ partial messages (with remaining message sizes) are put back in the heap. Thus the expected number of entries consumed is at least $k - \frac{n}{d+1}$. In following sections we use $\lambda = \frac{k}{n}$.

We denote d^* as the expected average number of active entries in each row after one iteration of scheduling. Assuming the original number of entries in each row be d, we have

$$d^* = \frac{1}{n}(nd - (k - \frac{n}{d+1}))$$

$$= d - \frac{k}{n} + \frac{1}{d+1}$$

$$= d - \lambda + \frac{1}{d+1} \tag{1}$$

It is difficult to analyze the number of messages in each row at the next step. We are interested in finding out the number of partial permutations generated by the algorithm. Clearly λ should be set such that $\lambda > \frac{1}{d+1}$. We use d^* as the new value of d at the next step. This assumption is made for all future steps. Assume Y_i be the expected number of useful entries remained at each row after the ith iteration. Then choosing a m such that

$$m = \frac{d}{2\lambda} + \frac{1}{\lambda^2} \tag{2}$$

would reduce Y_m to $\frac{d}{2}$ (analysis is given in Appendix C). The proof assumes $(1 + \frac{d}{2})\lambda \geq 1$.

We can calculate the expected number of iterations needed to complete the scheduling. According to Equation 2, the number of iterations is upper bounded by

$$(\frac{d}{2\lambda} + \frac{1}{\lambda^2}) + (\frac{d}{4\lambda} + \frac{1}{\lambda^2}) + \cdots + (\frac{1}{\lambda} + \frac{1}{\lambda^2})$$

$$= \frac{1}{\lambda}(\frac{d}{2} + \frac{d}{4} + \cdots + 1) + \frac{1}{\lambda^2}\log d$$

$$\approx \frac{d}{\lambda} + \frac{\log d}{\lambda^2} \tag{3}$$

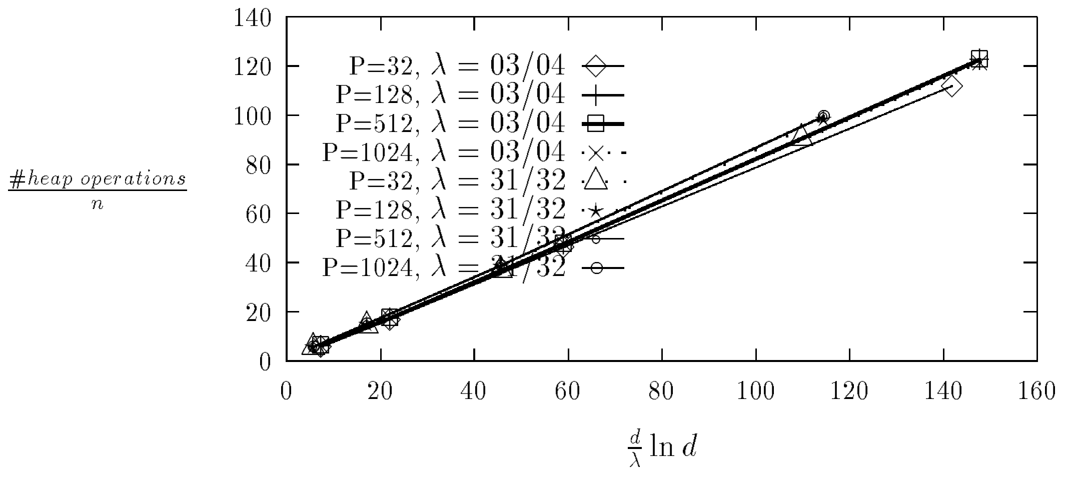

Figure 10: *Number of heap operations / n* versus $\frac{d}{\lambda}\ln d$

The above analysis is based on the assumption of equal number of messages in each row at the beginning and end of every round. With the analysis presented above, we conclude the following about the average time complexity of the RS_NH algorithm (assuming a fixed value λ):

- Time for compressing *COM* into *CCOM*: $O(n^2)$ in the sequential program and $O(dn + \tau \log n)$ in the parallelized version;
- Time for building heaps embedding in *CCOM*: $O(dn)$;
- Expected time for performing the scheduling: $O((\frac{d}{\lambda} + \frac{\log d}{\lambda^2}) \cdot (n \log^2 d))$, which is approximately $O(\frac{d}{\lambda}(n \log^2 d))$;
- Time for sorting one permutation by message sizes: $O(n \log n)$ for merge sort. Sorting can be approximated by histograming to reduce the complexity to $O(n)$.

The number of heap operations in Step 3.2 was measured for different values of n and λ for randomly generated communication matrices (Appendix D). We have plotted *number of heap operations / n* against $\frac{d \ln d}{\lambda}$ in Figure 10. The experimental results support our analysis. In this simulation we used uniform message sizes , in each permutation, we arbitrary picked up $n(1-\lambda)$ messages and put them (entire messages) back in the heap. In contrast, RS_NH is basically applied to non-uniform message sizes applications, messages in one permutation were sorted by size, and partial of larger messages (according to λ) were put back in the heap. Since the simulation is focused on the number of heap operations needed, the use of uniform message sizes does not affect the result.

Our simulation results show that by the time d is close to 1, the number of entries left in each row are uneven, and the degree of unevenness increases as λ is away from 1. This effect is amplified for large values of n. In order to reduce the impact of unevenness, one can propose a two-phase scheduling approach: using the original approach presented above during the process that d^* is reduced from original d to a small value (we use $\max\{2, \frac{d}{16}\}$ in this paper). The number of iterations, based on our analysis, is

$$\begin{aligned}&(\frac{d}{2\lambda} + \frac{1}{\lambda^2}) + (\frac{d}{4\lambda} + \frac{1}{\lambda^2}) + \cdots + (\frac{1}{16\lambda} + \frac{1}{\lambda^2}) \\ &= \frac{15}{16}\frac{d}{\lambda} + \frac{4}{\lambda^2}\ ;\end{aligned}$$

When d^* is small, it would be more suitable to reset λ to 1, *i.e.* completely sent out every message in one permutation, thus reducing d^* from small d to 0 will take $\frac{d}{16} + \log(\frac{d}{16})$ permutations [20]. Thus, the number of permutations to complete the scheduling, using the modified algorithm, is

$$(\frac{15}{16}\frac{d}{\lambda} + \frac{4}{\lambda^2}) + (\frac{d}{16} + \log(\frac{d}{16})) \tag{4}$$

Table 3 and 4 show the comparison of equations (3), (4), number of permutations generated by RS_NH with resetting λ, and number of permutations generated by RS_NH without resetting. These results reveal that without resetting λ to 1 when d^* is reduced to small value, for a system with 512 nodes, it may take as many as 15 extra iterations to complete the scheduling in comparison with algorithms which reset λ. In Table 4, for $\lambda = 0.75$ and $d = 512$, the difference between d_1 and *resetting* λ (or d_2 and *no resetting*) is approximate 30 permutations, this is because the calculation of d_1 and d_2 assumes that the average density in each row after one iteration is about the same, while in experiments the average density in each row after one iteration may vary in some range. In section 5, we propose several approaches to decide the value λ (we will use the resetting scheme in all of our approaches).

5 Approaches for Evaluating λ

When the message sizes in one permutation is non-uniform, the communication time is bounded by the maximum message size in that permutation (because the RS_NH is a loosely synchronous communication pattern), while other processors with smaller message size are 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 the sending message by their size. There are several schemes which can be used to decide on an appropriate value of λ.

5.1 Fixed λ

This is the most straightforward approach, λ is fixed throughout the entire scheduling. This approach requires running the application program several times with different value of λ

λ	d	$d_1^{\dagger}$	$d_2^{\ddagger}$	resetting λ	no resetting
0.75000	4	8.89	10.36	6.62	8.46
0.75000	8	16.00	16.61	13.52	15.34
0.75000	16	28.44	28.11	26.00	27.78
0.75000	32	51.56	50.11	48.86	50.84
0.93750	4	6.54	6.80	6.02	6.42
0.93750	8	11.95	12.05	11.02	11.42
0.93750	16	21.62	21.55	20.34	20.72
0.93750	32	39.82	39.55	37.76	38.00
0.96875	4	6.26	6.38	5.72	6.12
0.96875	8	11.45	11.50	10.54	10.96
0.96875	16	20.78	20.75	19.38	19.72
0.96875	32	38.36	38.23	35.90	36.24

$\dagger$: $d_1 = \frac{d}{\lambda} + \frac{\log d}{\lambda^2}$; $\ddagger$: $d_2 = (\frac{15}{16}\frac{d}{\lambda} + \frac{4}{\lambda^2}) + (\frac{d}{16} + \log(\frac{d}{16}))$.

Table 3: Scheduling on 32 nodes system

λ	d	d_1	d_2	resetting λ	no resetting
0.75000	64	96.00	93.11	99.86	103.22
0.75000	128	183.11	178.11	189.26	194.86
0.75000	256	355.56	347.11	364.42	374.20
0.75000	512	698.67	684.11	712.26	728.74
0.93750	64	75.09	74.55	77.10	77.88
0.93750	128	144.50	143.55	148.04	149.44
0.93750	256	282.17	280.55	288.12	290.58
0.93750	512	556.37	553.55	566.62	570.38
0.96875	64	72.46	72.20	73.84	74.16
0.96875	128	139.59	139.13	142.02	142.78
0.96875	256	272.78	272.00	277.08	277.88
0.96875	512	538.11	536.75	546.40	547.34

Table 4: Scheduling on 512 nodes system

in order to find out the best value. If this algorithm needs to be executed at runtime, each processor can begin with a different λ to schedule the communication. The processor with 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. The runtime approach will require an estimation function. We are currently developing methods for estimating the total communication cost for a given schedule.

5.2 λ Proportional to d

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

5.3 Incremental Approach

From Equation 1, we know that

$$d^* = d + \frac{1}{d+1} - \lambda, \lambda \geq \frac{1}{d+1}$$

which can be rewritten as

$$d(\lambda) = d + \frac{1}{d+1} - \lambda$$

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

- 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 $= (1-\lambda) \times \triangle M \times \varphi$;
- Additional cost due to increase in set up cost $= d'(\lambda) \times \triangle\lambda \times \tau$.

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

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

where $d'(\lambda) = -1$, we have

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

$$\lambda \leq \frac{\triangle\lambda\tau}{\triangle M\varphi} \tag{5}$$

The above analysis is under the assumption 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.

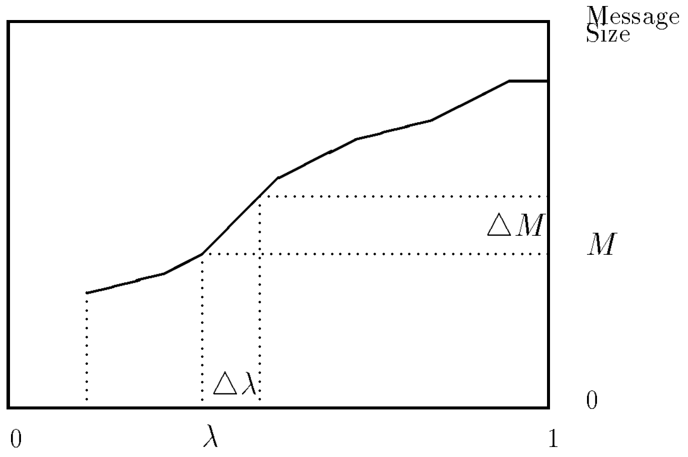

Figure 11: λ versus M graph

6 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.

6.1 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 RS_NH algorithm assuming that all the messages are of equal size (and hence there is no need to maintain any heaps).

4. **RS_NH+MS**: 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

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

   λ is chosen to be λ_k such that the following sum is maximized.

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

   The additional complexity of this step is $O(n)$ per iteration.

5. **RS_NH+BJ**: It is a variation of the algorithm RS_NH+MS. The value of λ is chosen to be λ_k such that

   $$Gain_k = \max_{0 \leq j \leq \frac{n}{4}} \{Gain_j\}$$

   The additional complexity of this step is $O(n)$ per iteration.

   We will use RS_NH to represent the better result of the two algorithms, RS_NH+MS or RS_NH+BJ, whenever there is no ambiguity.

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

7. **RS_N+sort**: This algorithm is same as RS_N except the fact that we sort the active entries in each row of *CCOM* by message size at the beginning of the scheduling algorithm (we only sort the rows once, and do not make an effort to maintain the sort sequences during the scheduling). This approach will tend to make the largest message in each row being scheduled in the earlier phases.

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

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.

6.2 Data Sets

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

1. The first test set contains two subgroups, each one has 50 different communication matrices. In each matrix, every row and every column have approximately d active entries (we select $d = 8$ and $d = 16$ in the two subgroups, respectively). The procedure we use to generate these test sets is described in Appendix D.

   The message lengths used in our test is $COM(i,j)$ multiplied by the variable msg_unit to study the effect of message size on each algorithm. The different values of msg_unit used for our experiments is 2^k for $4 \leq k \leq 13$.

2. The second test set (skewed distribution) contains samples with skewed size distribution. Three information arrays can be used to represent the samples' characteristics: $dist[5] = \{1, 2, 4, 8, 17\}$, $dense[5] = \{1, 2, 4, 8, 16\}$, 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 amount 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.

3. The third test set contains communication matrices generated by graph partitioning algorithms [15]; the samples represent fluid dynamics simulations of a part of a airplane (Figure 12) with different granularities (2800-point, 3681-point, 9428-point, and 53961-point). We will only present the results of 2800-point and 53961-point samples. In order to observe the algorithms' performance with different message sizes, we have multiplied the matrices in this test set by a variable msg_unit. The different values of msg_unit used for our experiments is 2^k for $4 \leq k \leq 13$.

In the third test set, 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 msg_units.
5. The minimum length of all messages = 1 msg_units.
6. The average length of all messages = 14.2 msg_units.

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

6.3 Results and Discussion

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

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.

Figure 12: The unstructured grid used for our simulations

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

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 ignored operations looks very high, we should point out that these operations are only executed once during the scheduling. So the constant values in front of the complexity terms are very small when compared with the complexity terms in front 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 5 suggests that the exclusion of most of the above operations affect the total cost only by a small fraction. The sort portion of RS_N+sort is expensive. However, our experimental results (in the later sections) reveal that this method does not provide any improvement over RS_N in terms of the total cost of communication (RS_N has a significantly lower scheduling cost).

6.3.1 Uniform Distribution

Table 6 and Figure 13 show the results of $d = 8$. 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 them providing 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.

Table 7 and Figure 14 show the results of $d = 16$. The results are similar to that of $d = 8$, but the differences between each different algorithms become prominent. Thus, when the density or message size increases, the RS_NH algorithms are the algorithms of choice.

Figure 15 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 [20]. 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 will be 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.

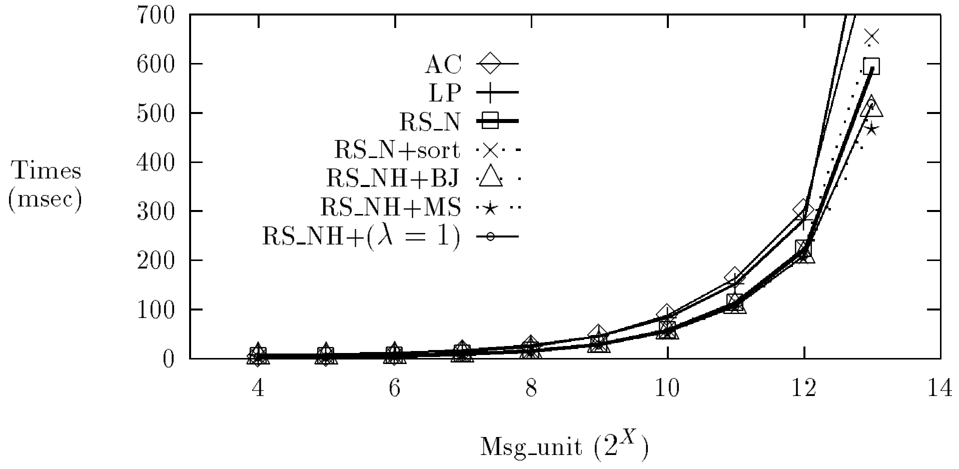

Figure 13: Communication cost for density $d = 8$

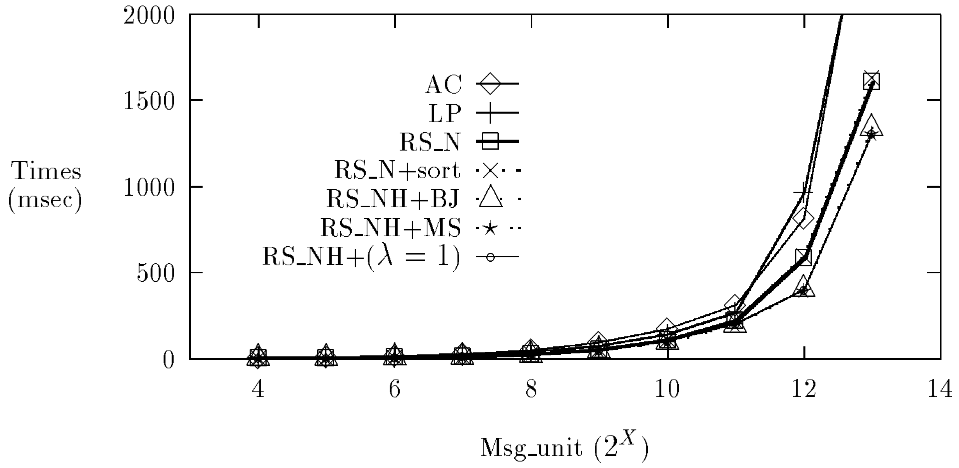

Figure 14: Communication cost for density $d = 16$

Msg_ unit	AC	LP	RS_N	RS_N +sort	RS_NH +BJ	RS_NH +MS	RS_NH +($\lambda = 1$)
comm*							
16	3.820	7.943	3.380	4.066	3.473	3.839	3.777
64	8.124	11.463	5.455	6.370	5.440	5.879	5.901
256	24.873	26.771	15.101	16.840	14.409	15.176	15.291
1024	89.027	83.063	57.825	59.436	54.020	53.744	54.560
2048	163.600	152.639	112.984	115.733	107.056	105.464	106.576
4096	301.681	282.814	222.201	225.684	209.728	207.420	209.661
8196	830.939	967.832	592.921	656.096	507.086	467.793	519.234
comp†	0	0.091	3.211	3.245	16.307	16.872	10.1
# iters‡	0	31.0	10.1	10.22	11.04	11.2	10.14

$*$: the total communication cost in milliseconds.
†: the scheduling cost in milliseconds.
‡: number of iterations (permutations) to complete the scheduling.

Table 6: Experimental results for density $d = 8$, the minimum message size in each level is *Msg_unit* bytes, and the maximum size is $32 \times$ *Msg_unit* bytes.

6.3.2 Skewed Distribution

In the second test set, all messages to be sent by one processor is same. This characteristics make RS_NH+($\lambda = 1$) useless. This is because the heap structure will keep the active entries in each row in a very similar order. This should, in general, make the probability to find a 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 smaller amount 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 the message sizes in one permutation without significantly increasing the number of permutations.

Table 8 and Figure 16 show the results of the second test set. As expected, the RS_NH+($\lambda = 1$) has similar performance as RS_N. While RS_NH+MS (which tends to select a small value of λ and splits a larger number of big messages into smaller ones as compared to RS_NH+BJ) and RS_NH+fixed have clear improvements over other approaches.

6.3.3 Airfoil Mesh

Table 9 and Figure 17 and Table 10 and Figure 18 show the results for a 2800-point and 53961-point sample respectively. The results for both samples have similar behavior as the first test set, which reveal 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

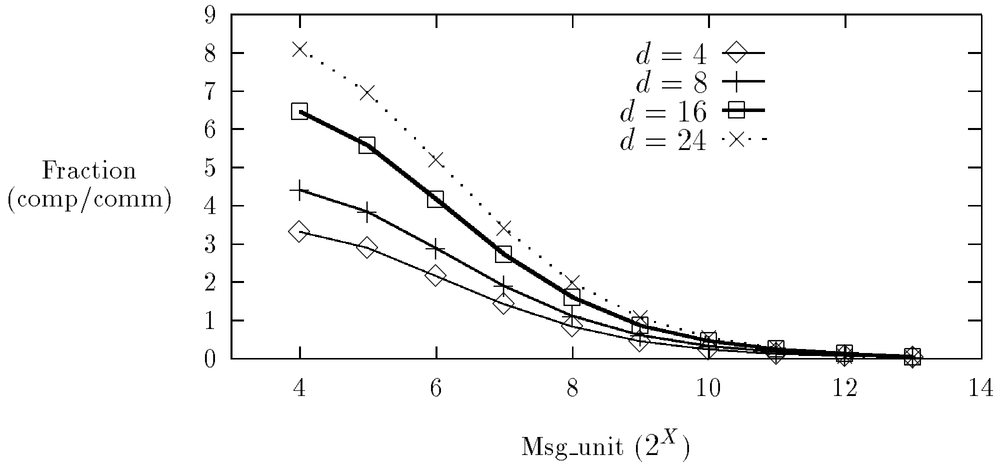

Figure 15: Computation overhead of RS_NH algorithm in terms of communication cost

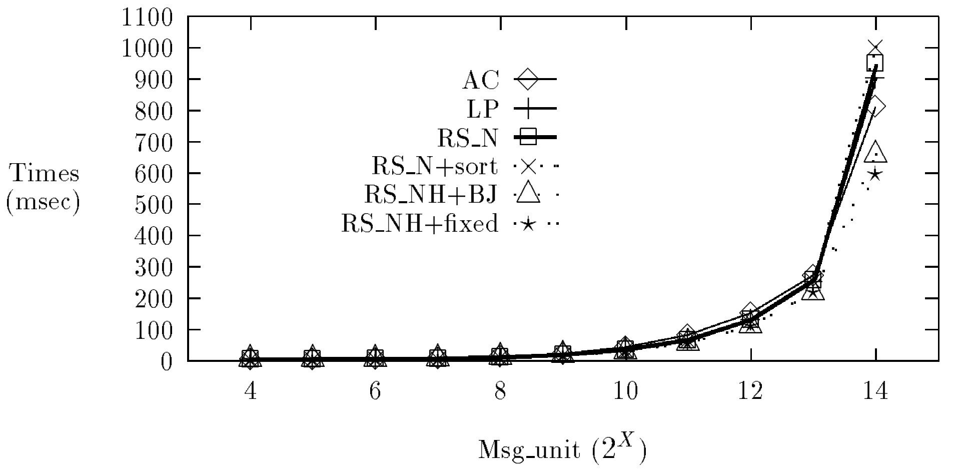

Figure 16: Communication cost for test set 2

Msg_ unit	AC	LP	RS_N	RS_N +sort	RS_NH +BJ	RS_NH +MS	RS_NH +($\lambda = 1$)
comm*							
16	8.178	9.514	6.408	7.653	6.404	7.050	7.126
64	17.780	16.112	10.494	12.152	10.184	10.959	11.212
256	52.173	43.161	29.385	32.330	27.304	28.607	29.121
1024	176.308	144.127	112.133	114.414	102.022	101.869	103.660
2048	311.054	270.140	222.023	224.392	201.459	199.639	202.816
4096	819.440	971.286	588.601	601.386	409.684	396.460	400.644
8196	2916.056	2851.732	1609.473	1633.950	1342.151	1309.655	1310.013
comp	0	0.091	6.57	6.62	43.943	45.403	31.502
# iters	0	31.0	18.56	18.52	19.66	19.8	18.8

Table 7: Experimental results for density $d = 16$, the minimum message size in each level is Msg_unit bytes, and the maximum size is $32 \times Msg\_unit$ bytes.

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 reveal that both methods have comparable performance. So for static applications (which can be pre-run to find the best value of λ), a fine tuned fixed λ may be as good as (or even better than) the dynamic λs found during the scheduling. One 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.

6.4 Discussion

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

$$cost(AC) \leq cost(LP) \leq cost(RS\_Ns) \leq\leq cost(RS\_NHs),$$

while the communication cost vary in the inverse fashion

$$cost(RS\_NHs) \leq cost(RS\_Ns) \leq cost(LP) \leq cost(AC)$$

Clearly, depending on the structure of 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 (with variable λ) seems to be a better approach (specially if the scheduling has to be performed at runtime).

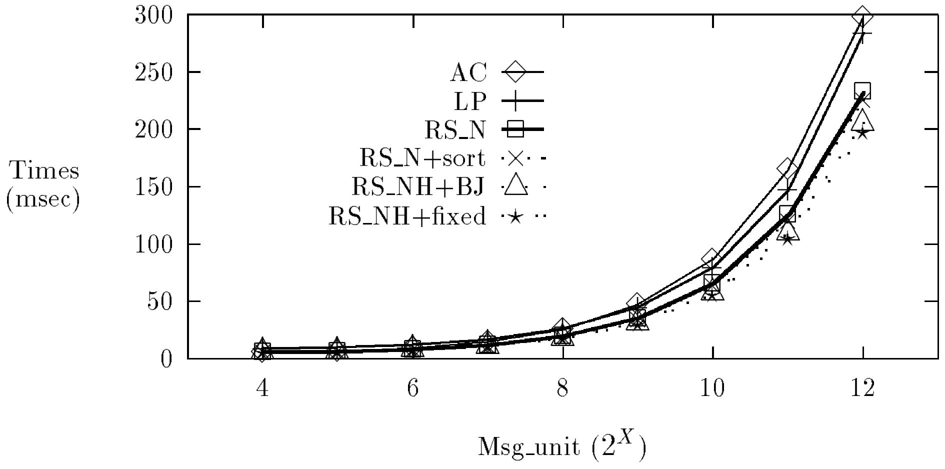

Figure 17: Communication cost for test set 3: 2800-point

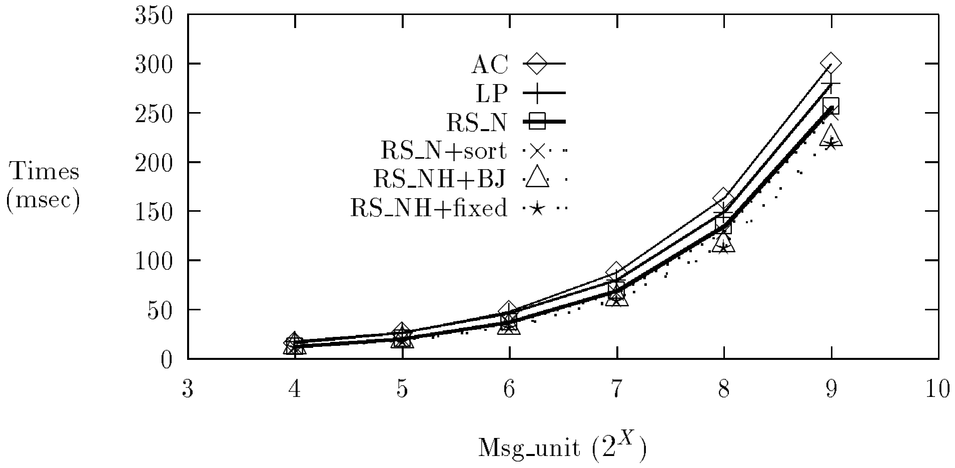

Figure 18: Communication cost for test set 3: 53961-point

Msg_ unit	AC	LP	RS_N	RS_N +sort	RS_NH +BJ	RS_NH +MS	RS_NH +($\lambda = 1$)	RS_NH +fixed
comm*								
16	5.893	9.049	6.673	6.711	8.127	10.111	6.722	6.485
64	8.231	10.066	7.490	7.552	9.002	11.052	7.494	7.398
256	15.841	15.938	12.911	12.928	13.416	15.705	12.876	12.279
1024	44.761	40.159	36.977	36.741	33.456	36.655	36.513	32.722
4096	154.052	134.647	132.543	131.628	114.109	119.861	130.678	114.365
8192	273.867	254.734	259.216	258.201	221.488	225.053	256.154	221.661
16384	813.610	904.941	949.817	1003.330	661.086	707.041	967.669	598.615
comp	0	0.097	7.678	8.84	39.561	43.41	21.77	34.451
# iters	0	31.0	20.1	20.2	24.0	31.7	20.4	21.45

Table 8: Experimental results for test set 2, the minimum message size in each level is Msg_unit bytes, and the maximum size is $16 \times Msg\_unit$ bytes.

7 Conclusion

In this paper, we have developed several algorithms for scheduling all-to-many personalized communication with non-uniform message sizes. The performance of asynchronous communication algorithm (AC) depends on the network congestion. The memory requirements of this algorithm is large. This algorithm is only suitable for small message sizes. The linear permutation algorithm is very straightforward, it introduces little computation overhead, but it needs to go through 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 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 one chooses the value on-the-fly. The experimental results have shown that our algorithms perform well with artificially generated samples as well as samples from an actual application.

Another advantage of our algorithms as compared to the other algorithms is the fact that once the schedule is completed, communication can potentially be overlapped with computation, *i.e.* computation on a packet received in previous phase can be carried out while the communication of the current phase is being carried. It is also worth noting that due to the 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.

Msg_ unit	AC	LP	RS_N	RS_N +sort	RS_NH +BJ	RS_NH +MS	RS_NH $+(\lambda = 1)$	RS_NH +fixed
comm*								
16	5.340	8.959	5.595	5.632	6.206	7.272	5.624	5.409
64	9.674	11.991	7.879	7.837	8.287	9.284	7.717	7.606
256	25.870	26.322	19.502	18.986	17.849	18.607	17.690	17.274
512	47.209	44.454	35.147	34.045	30.961	31.365	31.247	30.076
1024	86.679	79.324	65.342	63.657	57.431	57.582	57.224	55.537
2048	165.237	146.995	125.460	119.634	109.692	108.972	110.711	104.951
4096	297.637	283.917	232.721	225.080	205.231	208.906	209.687	197.226
comp	0	0.097	5.052	5.03	23.578	29.38	14.523	22.137
# iters	0	31.0	15.15	15.2	15.95	19.65	15.45	15.55

Table 9: Experimental results for test set 3: 2800-point, the minimum message size in each level is $2 \times Msg\_unit$ bytes, and the maximum size is $36 \times Msg\_unit$ bytes.

There is a large amount of literature on how to partition the task graph so as to minimize the communication cost. Many of these methods are iterative in nature, [15, 22] are a few of them (The reader is referred to [15] for a complete list). After a particular threshold any improvement in partitioning is expensive. For problems which require runtime partitioning, it is critical that this partitioning be completed extremely fast. For such problems, the gains provided by effective communication scheduling may far outperform the gains by spending the same amount of time on achieving a better partitioning.

For different applications, the kind of communication patterns used are different. It is unclear which methods will be better than others for specific class of communication patterns. However, we do believe the methods which avoid node contention and reduce the size variance in each permutation 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 whether the scheduling has to be performed statically or at runtime.

One of the issues which we have not addressed in this paper is link contention. On the CM-5, link contention does not 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.

Msg_ unit	AC	LP	RS_N	RS_N +sort	RS_NH +BJ	RS_NH +MS	RS_NH +($\lambda = 1$)	RS_NH +fixed
comm*								
16	16.103	17.941	12.907	12.718	12.895	14.920	11.700	12.253
32	26.826	27.349	20.965	20.619	19.512	21.536	18.532	18.950
64	48.367	46.552	37.662	36.642	33.479	35.513	32.599	32.771
128	87.700	80.769	69.874	67.731	61.605	63.126	60.816	60.148
256	163.598	149.746	135.387	129.456	116.832	118.149	115.609	113.558
512	300.644	280.240	256.659	250.574	224.406	228.418	225.190	219.322
comp	0	0.097	6.059	6.024	30.358	40.231	19.245	28.396
# iters	0	31.0	18.05	18.15	20.05	26.4	18.15	20.05

Table 10: Experimental results cost for test set 3: 53961-point, the minimum message size in each level is *Msg_unit* bytes, and the maximum size is 276 $\times$ *Msg_unit* bytes.

Appendix A: Procedure for Compressing *COM*

```
for i = 0 to n-1 do
    $k = -1$
    for j = 0 to n-1 do
        if COM(i,j) > 0 then
            $k = k + 1$;
            $CCOM(i, k) = j$;
        endif
    endfor
    $prt(i) = k$;
endfor
```

Appendix B

For a system with n nodes and the number of active entries in each row of *CCOM* is equal to d, in RS_NH step 3.2, the probability of success in finding an available entry in each row of *CCOM* is

$$S = 1^2 + 1 + \cdots + 1 + (1 - (\frac{d}{n})^d) + (1 - (\frac{d+1}{n})^d) + \cdots + (1 - (\frac{n-1}{n})^d)$$

$$= n - \frac{1}{n^d} \sum_{i=d}^{n-1} i^d$$

[2]there are d rows which would find an available entry with probability 1

$$\geq n - \frac{1}{n^d}\int_d^n x^d dx$$
$$= n - \frac{n}{d+1} + \frac{d}{d+1}(\frac{d}{n})^d$$
$$\geq n - \frac{n}{d+1}$$

Thus the expected number of entries in *CCOM* which can be sent in one iteration is at least $n - \frac{n}{d+1}$.

Appendix C

Assume Y_i be the expected value of the average number of useful entries remaining in each row after the *ith* iteration. Then

$$Y_0 = d$$
$$Y_1 = Y_0 - \lambda + \frac{1}{Y_0 + 1}$$
$$Y_2 = Y_1 - \lambda + \frac{1}{Y_1 + 1}$$
$$\vdots$$
$$Y_m = Y_{m-1} - \lambda + \frac{1}{Y_{m-1} + 1}$$

Summing all of these statements together, we have

$$Y_m = d - m\lambda + (\frac{1}{Y_0+1} + \frac{1}{Y_1+1} + \cdots + \frac{1}{Y_{m-1}+1})$$
$$Y_m \leq d - m\lambda + \frac{m}{Y_m + 1}$$

We are interested in finding the number of iterations needed to reduce Y_m to $\frac{d}{2}$.

$$\frac{d}{2} \leq d - m\lambda + \frac{m}{\frac{d}{2}+1}$$
$$m \leq \frac{d}{2\lambda}(\frac{1}{1 - \frac{1}{(1+\frac{d}{2})\lambda}})$$

Assuming that $(1 + \frac{d}{2})\lambda > 1$,

$$m \leq \frac{d}{2\lambda}(1 + \frac{1}{(1+\frac{d}{2})\lambda})$$
$$= \frac{d}{2\lambda} + \frac{d}{(d+2)\lambda^2}$$

If d is large, the second term at RHS can be reduced to $\frac{1}{\lambda^2}$, then choosing a m that

$$m = \frac{d}{2\lambda} + \frac{1}{\lambda^2}$$

would reduce Y_m to $\frac{d}{2}$.

Appendix D: Random *COM* Generator

```
for i = 0 to d-1 do
    k = i;
    for j = 0 to n-1 do
        COM(j,k) = 1;    k = (k + 1) mod n;
    endfor
endfor

for i = 0 to ManyTimes do
    loc1 = random() mod n;    loc2 = random() mod n;
    switch row loc1 with row loc2;
    (or switch column loc1 with column loc2);
endfor

Msg_Range = n
for i = 0 to n-1 do
    for j = 0 to n-1 do
        if (COM(i,j) = 1) then
            COM(i,j) = random() mod Msg_Range;
```

Acknowledgments

We wish to thank Raja Das, Joel Saltz, and Dimitri Mavriplis at ICASE and Nashat Mansour for the illustration depicted in Figure 12 and the corresponding communication matrices.

This work was supported in part by a contract between the Army Research office and 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] S. Baden. Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors. *SIAM Journal of Sci. and Stat. Computation*, 1991. to appear.

[4] D. Baxter, J. Saltz, M. Schultz, S. Eisentstat, and K. Crowley. A experimental study of methods for parallel preconditioned krylov methods. In *Proceedings of the 1988 Hypercube Multiprocessor Conference*, pages 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*, 1992. to appear.

[6] 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*, 1992. to appear.

[7] Alok Choudhary, Geoffrey C. Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Sanjay Ranka, and Joel Saltz. Software support for irregular and loosely synchronous problems. In *Proceedings of the Conference on High Performance Computing for Flight Vehicles*, 1992. to appear.

[8] Willian J. Dally and Chuck L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. on Computers*, 36(5):547, May 1987.

[9] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed memory compiler methods for irregular problems - data copy reuse and runtime partitioning. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier, Amsterdam, The Netherlands, 1991. to appear.

[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 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] 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.

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

[16] D.J. Mavriplis. Three dimensional unstructured multigrid for the euler equations. In *AIAA 10th Computational Fluid Dynamics Conference*, June 1991. paper 91-1549cp.

[17] 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 140–152, St. Malo, France, July 1988.

[18] 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.

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

[20] 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 211–218, HsinChu, Taiwan, December 1992.

[21] Y. Saad. Communication complexity of the gaussian elimination algorithm on multiprocessors. *Linear Algebra Application*, 77:pp.315–340, 1986.

[22] H. Simon. Partitioning of unstructured mesh problems for parallel processing. In *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications.* Permagon Press, 1991.

[23] Thinking Machines Corporation, Cambridge, MA. *The Connection Machine CM-5 Reference Manual*, 1992.

[24] D.W. Walker. Characterizing the parallel performance of a large-scale, particle-in-cell plasma simulation code. *Concurrency: Practice and Experience*, 1990.

[25] D.L. Whitaker, D.C. Slack, and R.W. Walters. Solution algorithms for the two-dimensional euler equations on unstructured meshes. In *Proceedings AIAA 28th Aerospace Sciences Meeting*, Reno, Nevada, January 1990.