Static and Runtime Scheduling of Unstructured Communication[1]

Sanjay Ranka and Jhy-Chun Wang

School of Computer and Information Science
Syracuse University
Syracuse, NY 13244

February 22, 1993

[1]This 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

With the advent of new routing methods, the distance to which a message is sent is becoming relatively less and less important. Thus assuming no link contention, permutation seems to be an efficient collective communication primitive. All-to-many communication is required for solving a large class of irregular and loosely synchronous problems on distributed memory MIMD machines. In this paper we present several algorithms for decomposing all-to-many personalized communication into a set of disjoint partial permutations. These partial permutations avoid node contention and/or link contention. We discuss several algorithms and study their effectiveness both from the view of static scheduling as well as runtime scheduling. Experimental results for our algorithms are presented on the iPSC/860.

Index Terms - Asynchronous communication, link and node contention, loosely synchronous communication, runtime and static scheduling.

1 Introduction

Experience with parallel computing has shown that a 'good' mapping is a critical part of executing a program on such computers. This mapping can be typically performed statically or dynamically. For most regular and synchronous problems, 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 [1, 16].

For a large class of scientific problems, which are irregular in nature, achieving a good mapping is considerably more difficult [7]. The nature of this irregularity may not be known at the time of compilation, and can be derived only at run time. Packages like PARTI [9, 12, 15] derive the necessary communication information based on the data required for performing the local computations and data partitioning. This tends to result in unstructured communication patterns. Each processor needs to send messages to some number of processors, with no obvious patterns. Further, for a large class of such problems, the same schedule is used a large number of times [6]. Thus, it may be feasible to perform the scheduling of communication at runtime if the effective gains from using such an schedule are greater than the cost of finding such a schedule.

In this paper we develop and analyze several simple methods of scheduling all-to-many personalized communication. The scheduling overhead of many of the methods developed in this paper is small enough that they can be used at runtime.

These methods can be classified into three categories:

1. Methods based on asynchronous communication
2. Methods which avoid node contention
3. Methods which avoid link contention

In the algorithms presented in this paper, we assume that the communication is represented by a matrix, COM, which is sparse in nature. $COM(i, j) = m, m > 0$ if

processor P_i needs to send a message of length m units to P_j, $0 \leq i, j \leq n-1$. Our algorithms decompose the communication matrix COM into a set of disjoint partial permutations, $pm_1, pm_2, \cdots, pm_l$, such that if $COM(i,j) > 0$ then there exists a k, $1 \leq k \leq l$, that $pm_k(i) = j$.

With the advent of new routing methods [8], the distance to which a message is sent is becoming relatively less and less important. Thus assuming no link contention, permutation seems to be an efficient collective communication primitive. Permutations have a useful property that each node receives at most one message and sends at most one message. If a particular node 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 number of messages.

There are some permutations (partial) which avoid link contention ($e.g.$ bit complement permutation on the hypercube [16]). We call such permutations a link contention avoiding permutation (LCAP). We also extend our methods to provide decompositions of the communication into a set of LCAPs. Our experimental results are presented on the iPSC/860. There are several idiosyncrasies of the iPSC/860 architectures. We also present modifications to our algorithms to exploit these idiosyncrasies.

The rest of this paper is organized as follows. Section 2 gives the assumptions of message routing algorithms and an overview of iPSC/860. Section 3 presents a simple asynchronous communication algorithm. Section 4 develops algorithms that will avoid node contention and discusses their time complexity. Section 5 describes an algorithm which avoids link contention. The algorithms given in section 4 and 5 assume that all messages are of equal size. In section 6, modifications are presented to the algorithms in the previous sections, in case the messages are of unequal size. Section 7 presents experimental results for a 64 node iPSC/860. Finally, conclusions are given in Section 8.

2 Preliminaries

2.1 System Overview: Intel iPSC/860

The experiments described in this paper are developed on a 64 node iPSC/860 at CalTech. The Intel iPSC/860 system consists of compute nodes, I/O nodes, and a host computer [11].

1. The *nodes* are i860-based processor boards.
2. The *I/O nodes* are Intel386-based processor boards through which the nodes have access to the Concurrent File System (CFS) and an Ethernet network.
3. The *host computer*, called the System Resource Manager (SRM), is an Intel386-based computer that runs UNIX[1]. Users logged into the SRM can allocate computer nodes and run node programs.

The iPSC/860 uses a circuit-switched communication via a hypercube interconnection network. When two nodes need to communicate, a dedicated path is set up between them. The communication path is determined by the *e-cube* routing algorithm. This algorithm chooses a fixed, shortest-path by changing the source node's address one bit at a time (from the least significant bit to the most significant bit) until the address of the destination node is achieved. Since the routing is deterministic, a message may encounter node or link contention during the communication.

Following are several observations about the communication network of iPSC/860 [3, 4, 13]:

1. Each node can support at most one send and one receive operation concurrently. A pairwise exchange is guaranteed to proceed concurrently if the two nodes involved first do a "pairwise synchronization" [3, 4]. However, if the two nodes do not start at the same time, the communication is essentially unidirectional. If a node P_i sends data to node P_j, and at same stage receives data from node P_k, where $j \neq k$, the send and receive operations rarely proceed concurrently.

[1]UNIX is a registered trademark of AT&T Bell Laboratories.

2. A communication circuit passing through a node has no discernible effect on other communication operations performed by that node.

3. Intersecting communication paths have no discernible effect on any of these paths.

4. For long messages, buffer copying is costly enough that the sender should wait until the receiver indicates that it is ready. This can typically be accomplished by the exchange of a dummy (*i.e.* 0 byte) message.

The detailed measurements of these observations are given in [2, 3, 19].

The iPSC/860 provides three kind of message types: the regular (unforced) message type, the forced message type, and the system message type (which is dedicated to system and should be avoided by general users) [10]. If a message with a regular message type arrives at a node before a receive buffer being posted for it, it will be put into a system buffer. The message is then copied into application buffer when the receiver node issues a receive command. In contrast, a message with a forced type is discarded upon arrival if no receive is posted for it.

The iPSC/860 uses different communication protocols for sending short and long messages. The long message protocol is used for messages of length greater than 100 bytes. It consists of three phases: (sender node) informs the receiver, (receiver node) requests the message, and then message transmission occurs [3]. The short message protocol is used for messages of length $\leq$ 100 bytes. It skips the first two phases and goes directly into data transmission.

For messages of length $\leq$ 100 bytes, it doesn't make difference for using forced or unforced message type. For larger messages, the unforced message type will require the additional *inform-request* overhead described above; If forced message type is used, it is user's responsibility to guarantee the availability of an application buffer when the message arrives. In this paper, we use regular (unforced) message type throughout our experiments.

Thus, in order to maximize the utilization of iPSC/860 interconnection network, care should be taken to avoid contention by efficient communication scheduling. The

communication scheduling should also exploit special features of the machine like concurrent bidirectional communication (by pairwise exchange).

2.2 Assumptions

We make the following assumptions for the development of our algorithms and 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, and φ represents the inverse of the data transmission rate.

2. In case the communication is sparse, all nodes send and receive approximately equal number of messages. If the density of sparseness is d, then at least d permutations are required for processors to complete sending their outgoing messages.

3. Each processor knows the destinations of its outgoing messages as well as the sources of its incoming messages. The latter restriction can be removed by an initial exchange of the local destination vectors.

3 Asynchronous Communication (AC)

The most straightforward approach is asynchronous communication. This scheme does not introduce any scheduling overhead. 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 confirm incoming messages (some of them may already arrived at its receiving buffer(s)) from other processors.

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 1: Asynchronous Communication Algorithm

During the send-receive process, the sender processor does not need to wait for a complete 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 1.

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 will 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. The 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. As mentioned in the previous section, buffer copying is very costly and should be avoided. The experimental results described in this paper use the approach given in Figure 1.

4 Methods Avoiding Node Contention

The input to the algorithms developed in this paper is a communication matrix COM, $COM(i,j)$ represents the amount of data which needs to be sent from node i to node j (for cases that all messages are assumed to be of equal length, we will use $COM(i,j) = 1$). The communication matrix COM is sparse in nature, *i.e.* each processor sends and receives at most d different messages (in a system with n processors, $d \leq n$). Our algorithms can be easily modified to be useful at runtime. Assuming that each processor knows its sending vector only at runtime, all processors can then participate in a concatenate operation [5] which will combine each processor's sending vector to form the communication matrix COM and leave a copy at every processor. This operation has efficient implementation on architectures like hypercubes and meshes.

We propose several algorithms that decompose the communication matrix COM into a set of disjoint partial permutations, $pm_1, pm_2, \cdots, pm_l$, such that if $COM(i,j) > 0$ then there exists a unique k, $1 \leq k \leq l$, that $pm_k(i) = j$. We present several scheduling algorithms, and the analysis of their time complexity in following subsections.

4.1 Linear Permutation (LP)

In this algorithm (Figure 2), each processor P_i sends a message to processor $P_{(i \oplus k)}$[2] and receives a message from $P_{(i \oplus k)}$, where $0 < k < n$ [3]. If $COM(i,j) = 0$, 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 exchanges.

The worst case time complexity of this algorithm is $O(n(\tau + \varphi M))$. One advantage of this algorithm is that it uses pairwise exchange throughout the entire communication. Further, the paths between different pairs in same phase do not have any link contention with each other. This feature of the algorithm can be used to exploit bidirectional communication on iPSC/860.

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

Linear_Permutation()

For all processor 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 2: Linear Permutation Algorithm

4.2 Random Scheduling Avoiding Node Contention (RS_N)

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 [17]. This procedure will improve the worst case time to access each active element (of $CCOM$) to $O(dn)$.

The vector prt is used as a pointer whose elements point to the maximum number of non-negative columns in each row. $CCOM$ also randomizes the list of destinations. This is necessary to reduce collisions and thus keep the expected number of collisions to be bounded. If we perform this compression statically, the time complexity is $O(n(n+d)) = O(n^2)$. This operation can be performed at runtime: each processor compacts one row, and then all processors participate in a concatenate operation which will combine all rows into a $n \times d$ matrix. The cost of this parallel scheme is $O((n+d) + (dn + \tau \log n)) = O(dn + \tau \log n)$ (assuming the concatenate operation can be completed in $O(dn + \tau \log n)$ time).

We set $CCOM(i,j) = -1$ if an entry doesn't contain active information. After the compressing procedure, the first d columns of each row may contain active entries. Two associated vectors, $send$ and $receive$, are introduced in this algorithm, where $send(i) = j$ denotes processor P_i need to send a message to processor

P_j, and $receive(j) = i$ denotes processor P_j will receive a message from processor P_i. 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 $receive(k) = i$. In order to prevent creating a hole in $CCOM$ (*i.e.*, $CCOM(x,y) = 0$, while $CCOM(x, y - k_1) = 1$ and $CCOM(x, y + k_2) = 1$, $k_1, k_2 > 0$), we move entry $CCOM(i,l)$ to $CCOM(i,j)$ and reset $CCOM(i,l) = -1$, where $l = prt(i)$. The worst case time complexity to form one partial permutations using this algorithm is $O(dn)$, as compared to $O(n^2)$ using the method without the compressing operation.

The RS_N algorithm is described in Figure 3.

The detailed complexity analysis of the RS_N algorithm is given in [18]. Assuming that each node is sending d messages to random destinations and receives d messages from different sources, we have the following results:

- The average time complexity for generating a permutation in one iteration is $O(n \ln d + n)$;
- The number of iterations needed to complete the entire message scheduling is upper bounded by $d + \log d$.

Thus,

- Time for compressing COM into $CCOM$ is $O(n^2)$ in sequential case and $O(dn + \tau \log n)$ in the parallelized version;
- Time for performing the scheduling: $O(d + \log d) \cdot O(n \ln d + n)$, which is approximately $O(dn \ln d)$;

5 Methods Avoiding Link Contention

For systems that use *circuit switched* message routing, the path between two processors is pre-claimed before the actual data is transferred. During the time data is transferred, no other communication paths are allowed to overlap with this path. The

Random_Scheduling_Node()

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

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

   (a) Set vectors $send = receive = -1$;

   (b) *x = random(1..n);*
   for y = 0 to n-1 do
   $i = (x + y) \ mod \ n$; $j = 0$;
   while (send(i) = -1 AND $j \leq prt(i)$*) do*
   $k = CCOM(i, j)$;
   if (receive(k) = -1) then
   $send(i) = k$; $receive(k) = i$;
   $CCOM(i, j) = CCOM(i, prt(i))$;
   $CCOM(i, prt(i)) = -1$;
   $prt(i) = prt(i) - 1$;
   endif
   $j = j + 1$;
   endwhile
   endfor

   (c) *if (send(i)* $\neq$ *-1) then* P_i sends a message to $P_{send(i)}$;
   if (receive(i) $\neq$ *-1) then* P_i receives a message from $P_{receive(i)}$;

   Until all messages are sent

Figure 3: RS_N Algorithm: Random Scheduling Avoiding Node Contention

scheduling algorithm proposed in this section modifies the previous algorithm (RS_N) to avoid any link contention. In this algorithm RS_NLP[3] (Figure 4), we introduce a $n \times n$ array *PATHS* which is used to record all claimed paths in one iteration (Obviously, for regular topologies like mesh and hypercube, the size of *PATHS* can be much smaller than the one we propose here). The function *Check_Path()* is used to verify that the path between P_i and P_j is not occupied by other communication pairs in the same iteration. The underlying assumption is that the hardware uses a deterministic routing algorithm. The function *Check_Path()* will return a TRUE if there is no contention, otherwise, the value returned is FALSE. Once a path is available, the procedure *Mark_Path()* is called to mark the path's corresponding entries in *PATHS* so no other communication can overlap this path in the same iteration.

For iPSC/860 which supports concurrent send and receive only under certain circumstances (specially pairwise exchange), locating all pairwise exchange within one iteration reduces the total communication time (which we prove to be correct in our experiment results). The function *Locate_Pair()* is inserted to find out the possible pairwise exchanges in each iteration and modify the corresponding entries in vectors *send* and *receive*. With the help of an associated array *PAIRS*, each pairwise exchange can be easily detected: if $PAIRS(i,j) = k_1$ and $PAIRS(j,i) = k_2$, then the entries $CCOM(i,k_1)$ and $CCOM(j,k_2)$ can form a pairwise exchange. Once a entry $CCOM(i,j)$ is selected, its corresponding entry $PAIRS(i,k)$ is reset to -1. Readers are referred to [17] for detailed description of *Locate_Pair()* and *PAIRS*.

6 Methods for Non-uniform Message Size

The RS_N and RS_NLP algorithms proposed in previous sections assume uniform message size. When the messages in one permutation are non-uniform, the largest message may dominate the communication cost (RS_N and related algorithms are

[3]The naming convention used in this paper is as following: RS implies random scheduling; N implies the algorithm avoids node contention; L implies the algorithm avoids link contention; P implies the algorithm uses *Locate_Pair()* to detect pairwise exchange; and H implies the algorithm uses heap structure (which will be addressed in next section).

RS_Node/Link_Pairwise()

1. Use matrix COM to create matrix $CCOM$ and matrix $PAIRS$;
2. Set matrix $PATHS$ to -1;
3. For all processor P_i, $0 \leq i \leq n-1$, *in parallel do*
   Repeat
   (a) Set vectors $send = receive = -1$;
   (b) $x = random(1..n)$;
   Locate_Pair(x);
   for y = 0 to n-1 do
   $i = (x + y) \ mod \ n$;
   $j = 0$; $found = FALSE$;
   while ((NOT found) AND (j ≤ prt(i))) do
   $k = CCOM(i, j)$;
   if (receive(k) = -1 AND Check_Path(i,k,PATHS)) then
   $send(i) = k$; $receive(k) = i$;
   Mark_Path(i,k,PATHS);
   $CCOM(i, j) = CCOM(i, prt(i))$; $CCOM(i, prt(i)) = -1$;
   $prt(i) = prt(i) - 1$; $found = TRUE$;
   endif
   $j = j + 1$;
   endwhile
   endfor
   (c) *if (send(i) ≠ -1) then* P_i sends a message to $P_{send(i)}$;
   if (receive(i) ≠ -1) then P_i receives a message from $P_{receive(i)}$;

   Until all messages are sent

Figure 4: RS_NLP Algorithm: RS Avoiding Node/Link Contention and Using Pairwise Exchange

loosely synchronous in nature, processors with smaller messages may be idle while waiting for processors with largest message to complete), thus we introduce methods to reduce the variance of message size within one permutation. The scheme we used in this section is to split large messages into smaller pieces such that the variance of message size within one permutation can be minimized, and each of these smaller piece is sent in different phases. The algorithms in this section decompose the COM into a set of partial permutations, $pm_1, pm_2, \cdots, pm_l$, such that if $COM(i,j) > 0$ then there exists at least a k, $1 \leq k \leq l$, that $pm_k(i) = j$.

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.

To reduce the variance of message size within one permutation, we propose a simple method to choose a reasonable message size in one permutation such that processors with smaller messages will send their messages completely, while processors with bigger messages only send part of their messages and restore the remaining message back to their proper location within heaps.

RS_NLPH is the new algorithm which includes a heap operation in the original RS_NLP algorithm (readers are referred to [17] for detailed description of the RS_NLPH algorithm). A simple method is used to evaluate the most efficient message size M_{thresh} in one permutation, where messages smaller than or equal to M_{thresh} are completely sent out, while only part of the message is sent out for messages larger than M_{thresh}. The partial messages are inserted in their new location (based on the remaining size) in the heap. The value of M_{thresh} is equal to the kth message size (in ascending order) in a particular permutation (if a node does not send a message in one permutation, its message size is assumed to be 0). We define $\lambda = \frac{k}{n}$.

The value λ is fixed throughout the entire scheduling. This approach requires running the application program several times with different value of λ 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 working on methods for estimating the total communication cost for a given schedule.

7 Experimental Results

We have implemented our algorithms on iPSC/860. The experiments are focused on evaluating three factors: (1) the number of permutations required to complete the communication; (2) the cost of executing the communication scheduling algorithms; and (3) the communication cost. The algorithms presented in section 4, 5, and 6 assume phase synchronization, *i.e.* phase $i+1$ should not start before phase i. This would require an expensive global synchronization at the end of every phase. To avoid global synchronization, we have modified the communication strategies in the following manner: whenever a node needs to receive data at one communication phase, it first posts its message buffer, then sends a signal (0 bytes message) to the sender node. Once the sender node receives the signal, it sends out the data. By using this strategy (we will call it S1 from now on), we can maintain a loose synchrony at a relatively lower cost. Another advantage of this method is that all the data will go directly into receiver node's application buffer, which will avoid extra buffer copying operations (from system buffer to application buffer).

We also experimented with other communication scheme: According to its communication scheduling table, every processor first posts all of its receiving requests (and allocates receiving buffers), then sends out all of its outgoing messages (without waiting for any kind inquire or completion signal), and finally verifies and confirms its incoming messages (we will call this scheme as S2). This scheme is essentially the scheme described in section 3, with the modification that the communication ordering is chosen so as to reduce node and/or link contention. Any of S1 or S2 can be performed in conjunction with the algorithms described in the previous sections. Our experimental results suggest that S1 performs better (in terms of communication cost) than S2 in most cases unless the density is small and/or the algorithm does not

exploit the pairwise bidirectional communication on iPSC/860.

The experimental results presented in this paper are thus for S1 in case the algorithm exploit pairwise bidirectional communication (LP, RS_NLP, RS_NLPH), and for S2 otherwise (AC, RS_N, RS_NLH).

To measure the time spent on communication, we perform the communication several times for each scheduling table generated by a particular algorithm. In each run, we take the maximum time spent by any processor. The average of this communication cost is the cost of a given schedule. Each test data set contains number of samples. We use the average communication cost of each sample to calculate the average communication cost of a given scheduling algorithm.

The experiments are divided into two categories, the first part assumes of equal message size, and the second part concentrates on non-uniform message size.

7.1 Uniform Message Size

Besides the algorithms we mentioned in previous sections, we also performed experiments on two other variations:

1. RS_NP: RS_N with *Locate_Pair()* to find out possible pairwise exchanges, but does not use *Check_Path()* and *Mark_Path()* to avoid link contention;

2. RS_NL: RS_N with *Check_Path()* and *Mark_Path()* to avoid link contention, but does not use *Locate_Pair()* to find out possible pairwise exchanges).

For each algorithm evaluated in this section, we use the same test data set. This set contains 50 random generated samples for each density d, the value of d ranges from 4 to 48. The number of nodes, n, is 64.

d	*msg_size*	AC	LP	RS_N	RS_NLP	RS_NP	RS_NL
4	comm						
	1K	5.908	34.313	6.514	6.507	6.484	6.463
	128K	579.249	1318.438	505.875	486.111	504.598	486.538
	# iters	-	63.0	5.92	7.1	5.78	7.04
	comp	-	0.056	1.727	8.157	5.026	4.539
8	comm						
	1K	14.002	40.238	13.462	13.157	13.333	13.235
	128K	1378.551	1898.211	1069.595	1008.682	1051.873	1019.104
	# iters	-	63.0	10.5	11.92	10.46	11.88
	comp	-	0.055	3.159	13.558	7.794	8.593
16	comm						
	1K	33.004	48.118	27.198	25.859	26.581	26.306
	128K	3211.788	2610.740	2186.585	2018.767	2114.707	2067.697
	# iters	-	63.0	19.16	20.74	19.14	20.62
	comp	-	0.054	6.365	24.531	13.274	17.064
32	comm						
	1K	75.267	57.415	54.384	49.52	51.683	52.127
	128K	7176.156	3271.964	4408.187	3854.764	4077.744	4155.519
	# iters	-	63.0	35.52	37.76	35.66	37.7
	comp	-	0.054	13.236	46.409	24.225	35.054
48	comm						
	1K	117.184	62.731	81.148	69.417	72.702	77.568
	128K	11188.302	3631.686	6610.211	5260.511	5554.96	6214.995
	# iters	-	63.0	51.58	53.74	51.6	53.84
	comp	-	0.056	20.26	65.434	34.095	53.6

Table 1: Experimental Results on a 64 node iPSC/860 for fixed message size (Timings are in milliseconds; # iters means number of phases of communication).

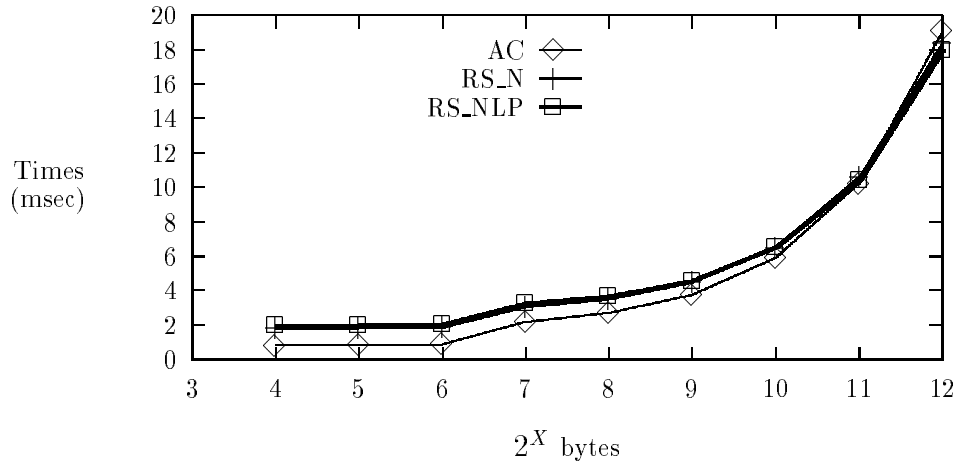

Figure 5: Communication cost for $d = 4$ and $n = 64$ for small message sizes. The cost of LP is at least 20 msec higher than others.

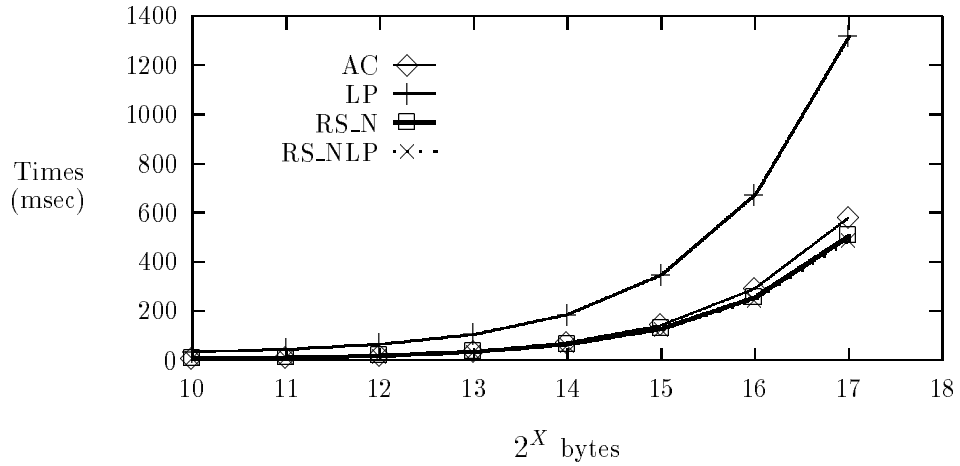

Figure 6: Communication cost for $d = 4$ and $n = 64$ for large message sizes

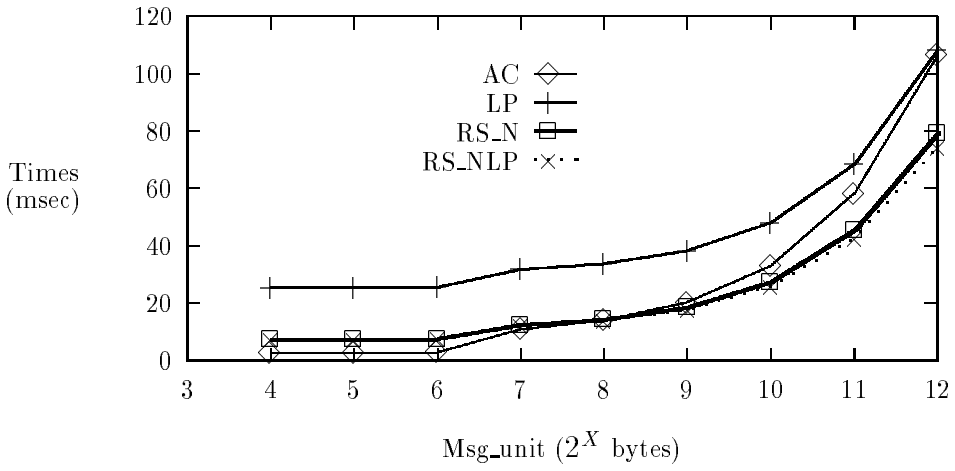

Figure 7: Communication cost for $d = 16$ and $n = 64$ for small message sizes

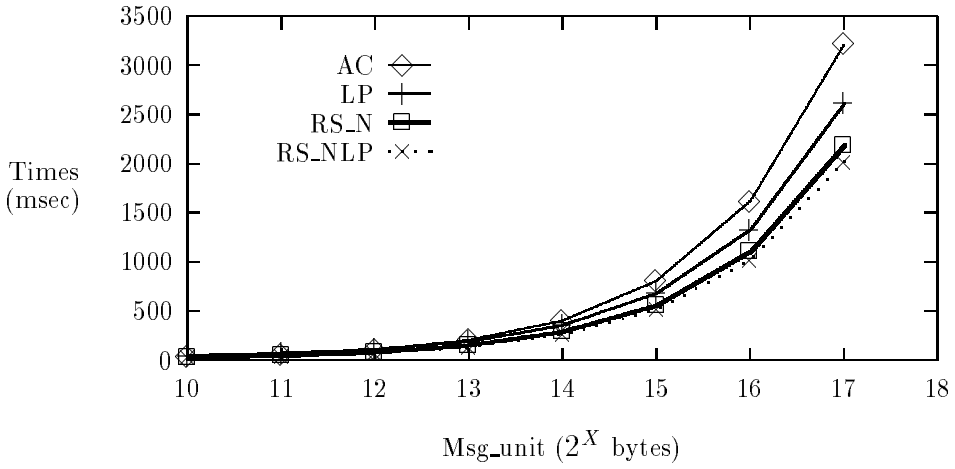

Figure 8: Communication cost for $d = 16$ and $n = 64$ for large message sizes

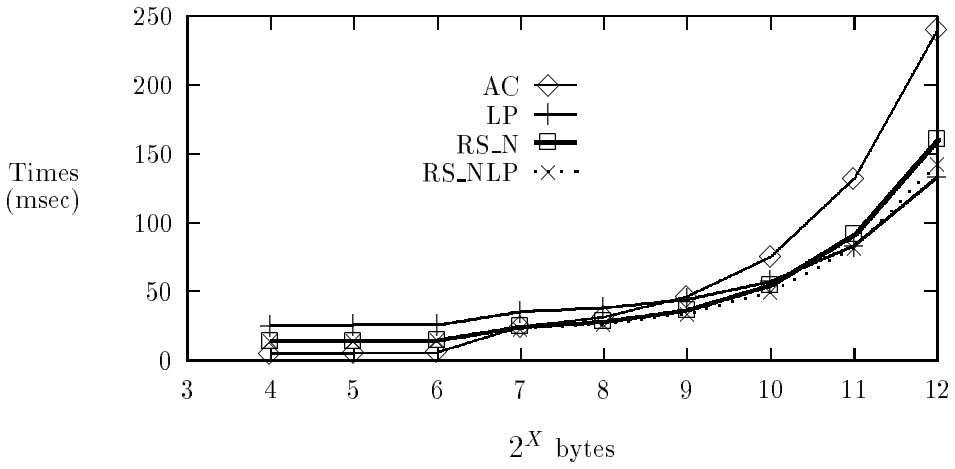

Figure 9: Communication cost for $d = 32$ and $n = 64$ for small message sizes

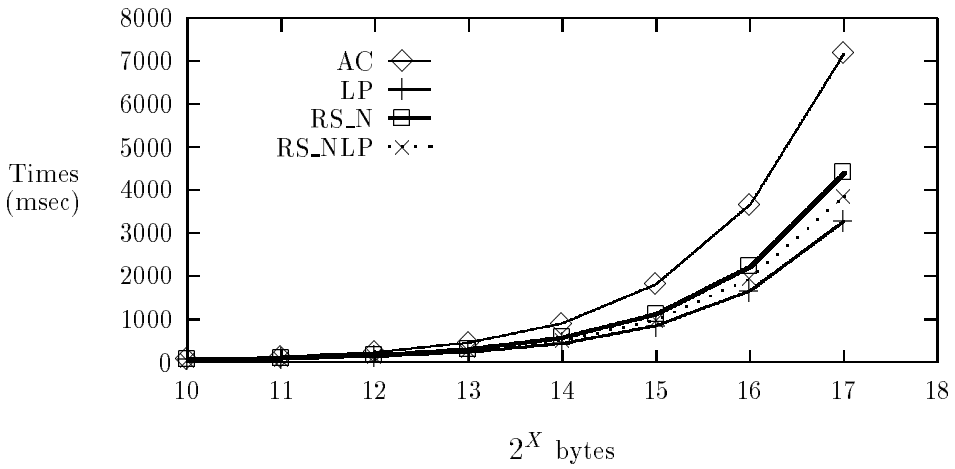

Figure 10: Communication cost for $d = 32$ and $n = 64$ for large message sizes

Table 1 and Figure 5 to 9 show the experimental results for message sizes of 1K bytes and 128K bytes. These results reveal the following:

1. AC performs better than all algorithms for small density ($d \leq 4$) and/or small messages ($\leq$ 1K bytes for $d = 4$ and $\leq$ 128 bytes for $d = 32$);

2. LP performs better than all algorithms for large density and large messages ($>$ 1K bytes and $d \geq 32$);

3. For most of the other cases RS_NLP has superior performance than all the other algorithms. This observation confirms the importance of exploiting node contention, link contention, and pairwise bidirectional communication.

The experiments demonstrate that each of the above algorithms is useful for certain (d, M) combinations.

In Figure 11 and 12, we present the scheduling overhead for a 64 node iPSC/860 using the RS_N algorithm and RS_NLP algorithm respectively for cases where each node has to send d messages. It depicts that this fraction decreases as the message size increases (assuming the same communication schedule is utilized only once). The fraction declines sharply when the message size is between 64 and 128 bytes; this behavior is caused by the change of the underlying iPSC/860 communication protocols. In such cases, we have shown (in Figure 9) that when message size is small, the AC algorithm is the better choice. For message size ranging from 128 bytes to 128K bytes, the cost of scheduling for RS_N algorithm is thus at most 0.6 the cost of communication and the cost is negligible for large messages (less than 0.25 for messages of size 2K bytes). For RS_NLP algorithm, the cost of scheduling is at most 2.5 the cost of communication for small messages and negligible for large messages (less than 0.25 for messages of size 8K bytes). (We did not spend much effort trying to improve the time spent on scheduling by optimizing the program to make it run faster. This can be done to reduce the scheduling overhead by a significant factor.) 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

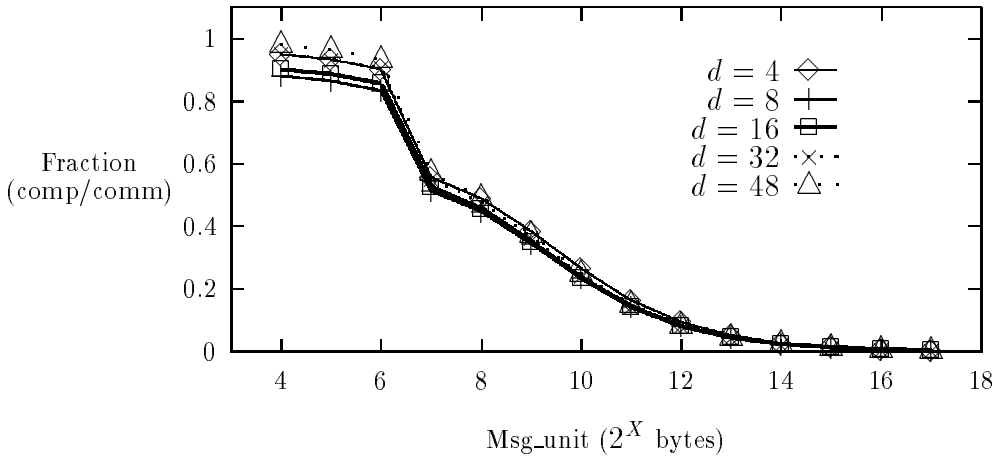

Figure 11: Computation overhead of RS_N algorithm in terms of communication cost

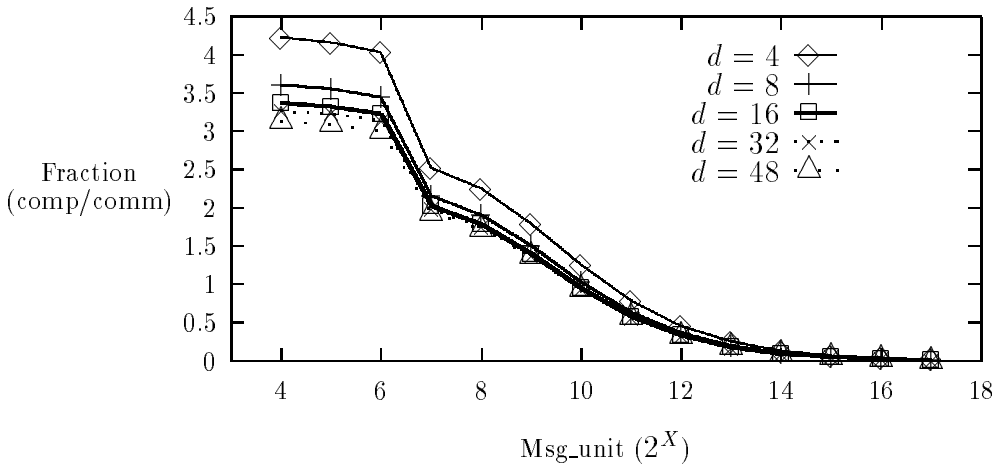

Figure 12: Computation overhead of RS_NLP algorithm in terms of communication cost

same schedule is used). In such cases, our algorithms are also suitable for runtime scheduling.

7.2 Non-uniform Message Size

In addition to the RS_NLPH algorithm we proposed in the previous section, we also experimented on another variation:

1. RS_NLH: same as RS_NLPH except that no extra effort is given to exploit possible pairwise bidirectional communication.

In the experiments, we test each sample with different values of λ ($\lambda = \frac{n-1}{n}$, $\frac{n-2^1}{n}$, $\frac{n-2^2}{n}$, ..., $\frac{3}{4}$). The best result (in terms of communication cost) is the cost of a given sample.

The test data sets used for non-uniform message size can be classified into two categories. In the following we present the test sets and corresponding results for these cases.

1. The first test set (uniform distribution) contains 50 random generated samples for each density value d ($d = 4, 8, 16, 32, 48$). The value of each entry $COM(i, j)$ is decided by following expression: $COM(i, j) = random()\ mod\ n$. The actual message size in each level is $COM(i, j) \times msg\_unit$, where msg_unit ranges from 2^4 to 2^{11}. Thus the minimum message size may be msg_unit bytes, while the the maximum size can be $n \times msg\_unit$ bytes.

   Table 2 and Figure 14 to 15 show the results of d from 4 to 16. Since the LP and RS_NLP do not use the heap structure, the message sizes in each permutation may vary in a large range such that the maximum message size in each permutation will affect the outcome. On the contrary, RS_NLPH (and RS_NLH) always begins with the largest message sizes possible. An efficient message size (M_{thresh}) is obtained in each permutation. The results show that RS_NLPH (and RS_NLH) have a significant improvement over RS_NLP. This observation reveals that when the variance in message size is large, it is worthwhile maintaining the heap structure.

Figure 13: The unstructured grid used for our simulations

2. The second test set (applications) contains communication matrices generated by load balancing algorithms [14] on some realistic data samples for a 32 node hypercube. The samples represent fluid dynamics simulations of a part of a airplane (Figure 13) with different granularities (2800-point, 3681-point, 9428-point, and 53961-point). We will only present the results of 53961-point samples. In order to observe the algorithms' performance with different message sizes, we have multiplied each entry of the matrix in this test set by a variable msg_unit which we mentioned above.

   In this test set, the number of messages sent (or received) by each node is uneven. For example, the 53961-point sample has: $max\_d = 18$, $min\_d = 6$, $ave\_d = 10.81$, and the length of each message sent is also variable: $max\_len = 276$, $min\_len = 1$, $ave\_len = 93.21$.

   Table 3 and Figure 16 show the results of test set 2 with 53961-point granularity. The results reveal that even when the number of messages sent by each processor is non-uniform, our algorithms still maintain their characteristics and performance. In this test set, RS_NLPH produces better results than RS_NLH.

d	*Msg_unit*	AC†	LP‡	RS_NLP‡	RS_NLPH‡	RS_NLH†
4	comm					
	256	56.807	99.659*	52.167	42.306*	42.286
	2048	437.516	741.833*	392.286	331.106*	328.542
	# iters	-	63	7.04	8.32	8.16
	comp	-	0.063	8.277	17.838	10.836
8	comm					
	256	123.849	147.657*	97.166	81.291*	80.186
	2048	957.764	1111.647*	740.803	630.406*	628.348
	# iters	-	63	11.86	13.5	13.62
	comp	-	0.063	13.775	35.677	24.004
16	comm					
	256	271.679	223.482*	180.348	155.767*	156.562
	2048	2101.211	1708.177*	1378.729	1210.089*	1213.994
	# iters	-	63	20.78	23.42	23.16
	comp	-	0.064	24.999	81.032	59.126
32	comm					
	256	591.937	304.078	335.545	314.442	310.258
	2048	4577.354	2251.429	2570.522	2394.474	2404.86
	# iters	-	63	37.74	42	42.3
	comp	-	0.065	47.335	198.336	153.925
48	comm					
	256	926.605	336.451	460.094	441.018	461.975
	2048	7128.014	2505.107	3513.423	3361.018	3585.96
	# iters	-	63	53.68	60.98	60.78
	comp	-	0.065	66.965	330.038	273.988

$\dagger$: Using S2; $\ddagger$: Using S1 unless otherwise mentioned.

$*$: For these entries S2 produces better results than S1.

Table 2: Experimental Results for non-uniform messages: test set 1 on a 64 node iPSC/860

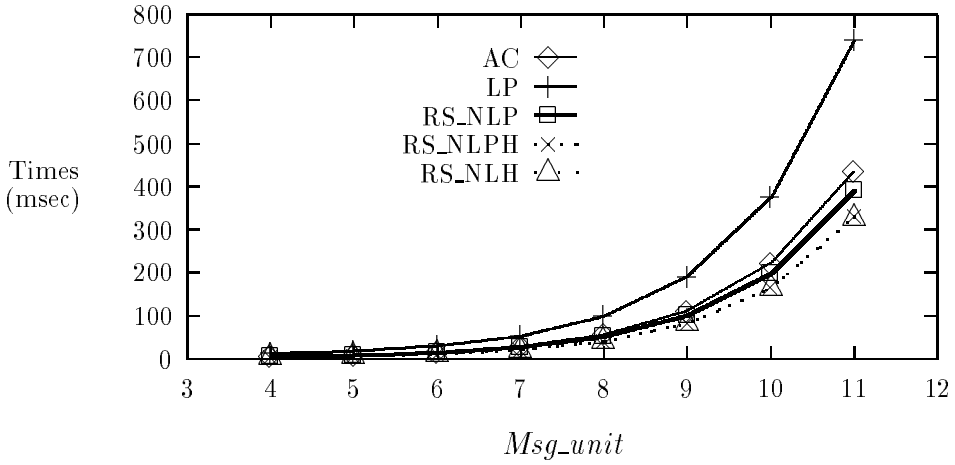

Figure 14: Communication cost for $d = 4$ and $n = 64$ for non-uniform messages: test set 1

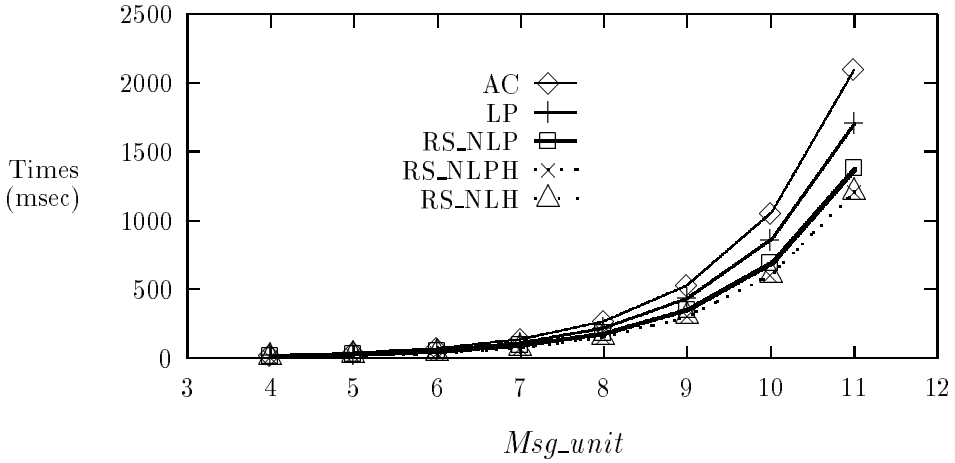

Figure 15: Communication cost for $d = 16$ and $n = 64$ for non-uniform messages: test set 1

Msg_unit	AC	LP	RS_NLP	RS_NLPH	RS_NLH
comm					
32	62.995	51.968	45.923	40.52	38.091
128	240.519	168.497	158.011	130.738	139.012
512	917.665	641.116	606.411	500.528	544.592
# iters	-	31	18.8	21.06	26.63
comp	-	0.034	7.904	21.15	19.226

Table 3: Experimental Results for non-uniform messages: test set 2 – 53961-point on a 32 node iPSC/860. The minimum message size in each level is *Msg_unit* bytes, and the maximum size is $276 \times Msg\_unit$ bytes.

8 Conclusions

This paper develops several algorithms for all-to-many communication on iPSC/860 and shows that using the above methods can significantly reduce the communication time over naive methods. For many cases the cost of scheduling is small enough that it can be performed at runtime.

The performance of these algorithms are presented for a 64 node iPSC/860 machine. The following conclusions are based on the limited experimental results for a fixed number of nodes.

1. The performance of asynchronous communication algorithm (AC) will depend on the network congestion and contention on the underlying architecture. The memory requirements of this algorithm is large. This algorithm is only suitable for small message sizes.

2. The linear permutation algorithm (LP) is very straightforward, it introduces very low computation overhead. One benefit of LP is its inherent property of pairwise exchange, which can be easily implemented to achieve concurrent send and receive for machines like iPSC/860. Further, there is no node or link

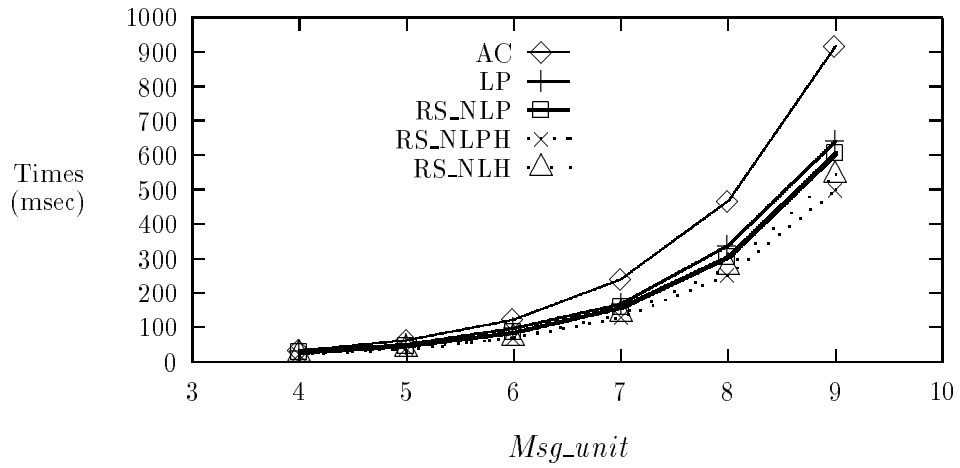

Figure 16: Communication cost for non-uniform messages: test set 2 – 53961-point

contention. This approach is not suitable for low values of d, because it needs to go through n iterations even when the value d is very small, but it performs very well for large value of d.

3. Avoiding node contention and link contention can significantly reduce the total time spent on the communication.

4. For iPSC/860, it is worthwhile exploiting pairwise bidirectional communication for machines which support concurrent send and receive.

5. Using a heap structure to keep all the messages approximately of the same size, in every phase, can pay reasonable dividend in terms of communication cost (although at a higher computation cost).

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 [14]. 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. In this paper, we provide schemes which can efficiently execute and achieve good performance in lowering communication cost.

The experimental results presented in this paper are for limited communication patterns which are randomly generated (except non-uniform test data set 2). For different applications, the kind of patterns used are different. It is unclear which methods will be better than others for specific class of communication patterns. However, we believe the methods which avoid node/link contention 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.

Acknowledgments

All the experiments conducted in this paper were performed on the CalTech's 64 node iPSC/860 machine. We would like to thank the support staff at CCSF for their help.

We wish to thank Raja Das, Joel Saltz, and Dimitri Mavriplis at ICASE; and Nashat Mansour for the illustration depicted in Figure 13 and the corresponding communication matrices.

References

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

[2] Shahid H. Bokhari. Communication overhead on the Intel iPSC/860 hypercube. Technical Report NASA Contractor Report: ICASE Interim Report No. 10, NASA Langley Research Center, May 1990.

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

[4] Shahid H. Bokhari. Multiphase complete exchange on a circuit switched hypercube. Technical Report NASA Contractor Report: ICASE Report No. 91-5, NASA Langley Research Center, January 1991.

[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] Intel Corporation, Beaverton, Oregon. *iPSC/860 System Calls Reference Manual*, 1992.

[11] Intel Corporation, Beaverton, Oregon. *iPSC/860 System User's Guide*, 1992.

[12] Charles Koelbel and Piyush Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[13] Ming-Horng Lee and Steven R. Seidel. Concurrent communication on the Intel iPSC/2. Technical Report CS-TR 9003, Michigan Technological University, July 1990.

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

[15] 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.

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

[17] Sanjay Ranka and Jhy-Chun Wang. Static and runtime scheduling of unstructured communication. Technical Report SU-CIS-92, Syracuse University, November 1992.

[18] 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*, HsinChu, Taiwan, December 1992. to appear.

[19] Steven R. Seidel and Thomas E. Schmiermund. Refining the communication model for the Intel iPSC/2. In *Proceedings of the 5th Distributed Memory Computing Conference*, pages 1334–1342, Charleston, SC, April 1990.