Syracuse University
SURFACE

College of Engineering and Computer Science - Former Departments, Centers, Institutes and Projects

College of Engineering and Computer Science

1994

# Scheduling of Unstructured Communication on the Intel iPSC/860

Jhy-Chun Wang
*Syracuse University, School of Computer and Information Science*

Sanjay Ranka
*Syracuse University, School of Computer and Information Science*

Follow this and additional works at: http://surface.syr.edu/lcsmith_other

Part of the Computer Sciences Commons, and the Systems and Communications Commons

Recommended Citation
Wang, Jhy-Chun and Ranka, Sanjay, "Scheduling of Unstructured Communication on the Intel iPSC/860" (1994). *College of Engineering and Computer Science - Former Departments, Centers, Institutes and Projects.* Paper 48.
http://surface.syr.edu/lcsmith_other/48

This Conference Document is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in College of Engineering and Computer Science - Former Departments, Centers, Institutes and Projects by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

# Scheduling of Unstructured Communication on the Intel iPSC/860*

Jhy-Chun Wang† Sanjay Ranka

School of Computer and Information Science
Syracuse University
Syracuse, NY 13244-4100

## Abstract

*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 as well as link contention. We discuss the theoretical complexity of these algorithms and study their effectiveness both from the view of static scheduling and from runtime scheduling. Experimental results for our algorithms are presented on the iPSC/860.*

## 1 Introduction

Experience with parallel computing has shown that a "good" mapping is a critical part of executing a program on massively parallel processing machines. The mapping typically can be 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) [5]. This typically results in regular collective communication between processors. Many such primitives have been developed in [1, 13].

For a large class of scientific problems, which are irregular in nature, achieving a good mapping is considerably more difficult [6]. 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 [8, 11] 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 [5]. Thus, it may be feasible to perform the scheduling of communication at runtime if the effective gains from using such a 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. The methods developed in this paper can be classified into three categories:

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

With the advent of new routing methods [7, 12], the distance to which a message is sent is becoming relatively less and less important. Permutations have a useful property that, in one phase, each node receives at most one message and sends at most one message, thus permutation seems to be a good candidate for collective communication primitive. 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. Sometimes processors also compete for communication path (that will result in link contention), the contention, if not well-managed, may severely degrade overall performance. There are some (partial)

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

†Jhy-Chun Wang's current address is Department of Computer Science, University of Illinois at Urbana-Champaign, email: *jcwang@cs.uiuc.edu* .

permutations which have the property of avoiding link contention (e.g., bit complement permutation on the hypercube [13]).

In general, assuming a system with $n$ processors, our algorithms take as input an $n \times n$ communication matrix $COM$. $COM(i,j)$ is equal to a positive integer $m$ if processor $P_i$ needs to send a message (of $m$ unit) to $P_j$, $0 \leq i,j \leq n-1$. Our algorithms decompose the communication matrix $COM$ into a set of partial permutations, $pm_1, pm_2, \cdots, pm_l$, where $l$ is a positive integer and $pm_k^i$ represents the $i^{\text{th}}$ entry in vector $pm_k$. The decomposition is made such that if $COM(i,j) \neq 0$, then there exists a $k$, $1 \leq k \leq l$, such that $pm_k^i = j$. These partial permutations are made to avoid node and/or link contention. Experimental results for these algorithms are presented on the iPSC/860.

The rest of this paper is organized as follows. Notations, definitions, general communication properties, and an overview of iPSC/860 are given in Section 2. It also discusses several idiosyncrasies of the iPSC/860 architecture which require modifications to the general strategies to achieve good performance. 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 both node and link contention. The algorithms given in Section 4 and 5 assume that all messages are of equal size. Section 6 presents experimental results for a 64-node iPSC/860. Finally, conclusions are given in Section 7.

## 2 Preliminaries

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

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

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

Thus, *node contention* can be formally defined as

$$\exists k,\ \ 1 \leq k \leq l,$$

$$cp_k^{i_1} = j_1 \ \ and \ \ cp_k^{i_2} = j_2 \Rightarrow i_1 \neq i_2 \ \ and \ \ j_1 = j_2 \ ,$$

where $0 \leq i_1, i_2, j_1, j_2 < n$.

A partial permutation $pm_k$ is a communication phase that

$$pm_k^{i_1} = j_1 \ \ and \ \ pm_k^{i_2} = j_2,$$

$$0 \leq i_1, i_2, j_1, j_2 < n \ ,$$

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

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

Since permutation has the useful property that every processor both sends and receives at most one message, it can significantly reduce node contention.

The methods developed to reduce link contention assume a static routing algorithm is used in message routing, i.e., based on the source and destination nodes, one can determine the path that will be used for routing. Let $edge_{ij}$ represent the direct communication link (if one exists) between processors $P_i$ and $P_j$. Let $path_k^{ij}$ represent the set of links that $P_i$ will use in the $k^{\text{th}}$ permutation in order to send a message to $P_j$,

$$path_k^{ij} = \{edge_{im_1}, edge_{m_1 m_2}, \ldots, edge_{m_x j}\} \ .$$

If $pm_k^i = j = -1$, then $path_k^{ij} = \phi$.

We define the term *link contention* as:

$$\exists k,\ \ 1 \leq k \leq l,$$

$$pm_k^{i_1} = j_1 \quad \text{and} \quad pm_k^{i_2} = j_2,\ \ 0 \leq i_1, i_2, j_1, j_2 < n \ ,$$

$$\Rightarrow \ \ i_1 \neq i_2 \quad \text{and} \quad path_k^{i_1 j_1} \cap path_k^{i_2 j_2} \neq \phi \ .$$

Thus, a communication scheduling that avoids node/link contention is a scheduling such that,

$$\forall k,\ \ 1 \leq k \leq l,$$

$$pm_k^{i_1} = j_1 \quad \text{and} \quad pm_k^{i_2} = j_2,\ \ 0 \leq i_1, i_2, j_1, j_2 < n \ ,$$

$$i_1 \neq i_2 \quad \Rightarrow \quad path_k^{i_1 j_1} \cap path_k^{i_2 j_2} = \phi \ .$$

### 2.1 Assumptions

We make the following assumptions for the development of our algorithms and complexity analysis.

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

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

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

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

### 2.2 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.

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 important observations about the communication network of iPSC/860 and its communication software [3, 9]:

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]. 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.

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, 14].

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).

## 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.

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

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

Figure 1: Asynchronous communication algorithm.

3. Each processor checks and confirm 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 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 that Avoid 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 $j$. 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, $d \leq n$). Assuming that each processor knows its sending vector only at runtime, all processors can participate in a concatenate operation [4] 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 [1, 13].

In following subsections, 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 $pm_k^i = j$.

```
Link_Contention_Avoiding_Permutation()
For all processors P_i, 0 ≤ i ≤ n − 1, in parallel do
  for k = 1 to n − 1 do
      j = i ⊕ 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: Scheduling with a special class of link contention–free permutations.

## 4.1 Scheduling using Special Class of Permutations (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 especially for symmetric communication matrices.

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

### 4.2 Randomized Scheduling Avoiding Node Contention (RS_N)

During the communication scheduling, the worst case time complexity to traverse through every entry of the $n \times n$ $COM$ is $O(n^2)$. In order to reduce this overhead, the first step of this algorithm is to compress $COM$ into a $n \times d$ matrix $CCOM$ by a simple compressing procedure which moves the $d$ active entries in each row to the first $d$ columns [15]. This procedure will improve the worst case time to access every active element (of $CCOM$) to $O(dn)$.

The vector $prt$ (in Figure 3) is used as a pointer whose elements point to the maximum number of non-negative columns in each row. The compressing procedure also randomly swaps the active entries in each row. This is necessary to reduce collisions and thus keep the expected number of collisions to be bounded. Without the randomization, the active entries in each row are in ascend order, that, during the first several communication phases, tends to result in node contention among processors with small IDs. 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, only the first $d$ columns of each row may contain active entries. The vectors $Tsend$ and $Trecv$ are used to record the destination of each outgoing message and the source of each incoming message in one permutation, respectively; $Tsend(i) = j$ denotes that processor $P_i$ needs to send a message to processor $P_j$, and $Trecv(j) = i$ denotes that processor $P_j$ will receive a message from processor $P_i$. These two vectors are initialized to $-1$ at the beginning of each iteration. When searching for an available entry along row $i$, the first column $j$ with $CCOM(i,j) = k \geq 0$ and $Trecv(k) = -1$ will be chosen. We then set $Tsend(i) = k$ and $Trecv(k) = i$, and the value of $CCOM(i, prt(i))$ is then assigned to entry $CCOM(i,j)$, while $prt(i)$ is decreased by 1. The worst case time complexity to form one partial permutation here is $O(dn)$, as compared to $O(n^2)$ 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 [15]. Assuming that each node

---

**Random_Scheduling_Node()**

1. Use the $n \times n$ matrix $COM$ to create an $n \times d$ matrix $CCOM$ and generate a vector $prt$;
2. For all processors $P_i$, $0 \leq i \leq n-1$, *in parallel do*
   *Repeat*
   (a) Set all entries of vectors $Tsend$ and $Trecv$ to $-1$;
   (b) $x = random(0..n-1)$;
   (c) *for* $k = 0$ *to* $n-1$ *do*
      i. Along row $x$ of $CCOM$, find an entry $CCOM(x,z) = y$ that satisfies $y \geq 0$ and $Trecv(y) = -1$;
      ii. If such a $z$ exists, then set
         $Tsend(x) = y$;
         $Trecv(y) = x$;
         $CCOM(x,z) = CCOM(x, prt(x))$;
         $CCOM(x, prt(x)) = -1$;
         $prt(x) = prt(x) - 1$;
      iii. $x = (x+1) \ mod \ n$;
      *endfor*
   (d) *if* ($Tsend(i) \neq -1$) *then*
         $P_i$ sends a message to $P_{Tsend(i)}$;
      *if* ($Trecv(i) \neq -1$) *then*
         $P_i$ receives a message from $P_{Trecv(i)}$;

   *Until* all messages are sent

---

Figure 3: RS_N algorithm: randomized scheduling that avoids node contention.

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 Scheduling that Avoid Link Contention

For systems that use *circuit switched* message routing (e.g., iPSC/860), the path between two processors is pre-claimed before the actual data is transferred. During the period that data is transferred, no other communication paths are allowed to overlap with this path. The scheduling algorithm proposed in this section modifies the RS_N algorithm to avoid any link contention. In this algorithm (RS_NL, Figure 4) we introduce an $n \times n$ array $PATHS$ which is used to record all claimed paths in one communication phase (Obviously, for regular topologies like mesh and hypercube, the size of $PATHS$ can be much smaller than the one proposed here).

The function $Check_Path()$ is used to verify that the path between nodes $P_i$ and $P_j$ is not occupied by other communication pairs in the same phase. The underlying assumption is that the hardware uses a deterministic routing algorithm. $Check_Path()$ will return a value TRUE if there is no link 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 $PATHS$ such that no other communication can overlap this path in the same phase.

Further, for iPSC/860 which supports concurrent send and receive only under certain circumstances (especially pairwise exchange), it is beneficial to locate (and use) as many pairwise exchange as possible. In Step 3(c)i (Figure 4), priority is given to entries that can result in pairwise exchange. Detail discussion of locating pairwise exchange in one communication phase can be found in [15].

---

**RS_Node_Link()**

1. Use the $n \times n$ matrix $COM$ to create an $n \times d$ matrix $CCOM$ and a vector $prt$;
2. Set all entries of matrix $PATHS$ to $-1$;
3. For all processors $P_i$, $0 \leq i \leq n-1$, *in parallel do* *Repeat*
   (a) Set all entries of vectors $Tsend$ and $Trecv$ to $-1$;
   (b) $x = random(0..n-1)$;
   (c) $for\ \ k = 0\ \ to\ \ n-1\ \ do$
      i. Along row $x$ of $CCOM$, find an entry $CCOM(x, z) = y$ that satisfies $y \geq 0$ ,$Trecv(y) = -1$, and $Check_Path(x, y) = TRUE$;
      ii. If such a $z$ exists, then set
      $Tsend(x) = y$;
      $Trecv(y) = x$;
      Call $Mark_Path(x, y)$;
      $CCOM(x, z) = CCOM(x, prt(x))$;
      $CCOM(x, prt(x)) = -1$;
      $prt(x) = prt(x) - 1$;
      iii. $x = (x + 1)\ \ mod\ \ n$;
   $endfor$
   (d) *if (Tsend(i)* $\neq -1$*) then*
      $P_i$ sends a message to $P_{Tsend(i)}$;
   *if (Trecv(i)* $\neq -1$*) then*
      $P_i$ receives a message from $P_{Trecv(i)}$;

   *Until* all messages are sent

---

Figure 4: RS_NL algorithm: RS that avoids node/link contention.

## 6 Experimental Results

We implemented our algorithms on a 64-node 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 previous sections assume phase synchronization, i.e., phase $i+1$ should not be started before phase $i$ is completed. 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 this paper. 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 and RS_NL), and for S2 otherwise (AC and RS_N).

To measure the time spent on communication, we perform the communication $k$ times for each scheduling table generated by a particular algorithm. In each run, we take the maximum time spent by any processor as the cost of this test run. The average of the (maximum) communication cost (over $k$ runs) 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 conducted here assuming equal message size, i.e., in one test, every processor sends and receives messages of equal size. The test set used in the experiments contains 50 random generated samples for each density $d$, the value of $d$ ranges from 4 to 48. The machine used in the experiments is a 64-node iPSC/860.

Table 1 and Figure 6 to 9 show the experimental results for message sizes of range 16 bytes to 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_NL 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. Figure 5 shows the different regions for which each of the algorithm is most useful on a 64-node iPSC/860. This diagram does not take scheduling cost into account (i.e., it assumes the scheduling is performed statically, or the scheduling is conducted at runtime and its cost can be amortized over repeated utilizations and become negligible).

In Figure 10 and 11, we present the scheduling overhead for a 64-node iPSC/860 using the RS_N algorithm and RS_NL 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 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_NL 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). 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, our algorithms are also suitable for runtime scheduling.

## 7 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 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 machine likes iPSC/860, it is worthwhile exploiting pairwise bidirectional communication to achieve concurrent send and receive.

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 [10]. 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. 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 do 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.

Because of space limitation, the algorithms and experiments discussed in this paper assuming uniform message size (i.e., every processor sends and receives messages of equal size). Readers are referred to [15] for a complete discussion of methods used in non-uniform message size problems.

## 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.

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

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

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

[7] Willian J. Dally and Chuck L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. on Computers*, 36(5):pp. 547–553, May 1987.

[8] 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.

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

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

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

[12] 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.

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

[14] 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 pp. 1334–1342, Charleston, SC, April 1990.

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


| *d* | *msg_size* | AC | LP | RS_N | RS_NL |
|---|---|---|---|---|---|
| | comm | | | | |
| | 256 | 2.70 | 28.13 | 3.54 | 3.63 |
| | 1K | 5.91 | 34.31 | 6.51 | 6.51 |
| 4 | 128K | 579.25 | 1318.44 | 505.88 | 486.11 |
| | # iters | - | 63.00 | 5.92 | 7.10 |
| | comp | - | 0.06 | 1.73 | 8.16 |
| | comm | | | | |
| | 256 | 6.05 | 30.48 | 7.05 | 7.10 |
| | 1K | 14.00 | 40.24 | 13.46 | 13.16 |
| 8 | 128K | 1378.55 | 1898.21 | 1069.60 | 1008.68 |
| | # iters | - | 63.00 | 10.50 | 11.92 |
| | comp | - | 0.06 | 3.16 | 13.56 |
| | comm | | | | |
| | 256 | 14.02 | 33.92 | 14.00 | 13.75 |
| | 1K | 33.00 | 48.12 | 27.20 | 25.86 |
| 16 | 128K | 3211.79 | 2610.74 | 2186.59 | 2018.77 |
| | # iters | - | 63.00 | 19.16 | 20.74 |
| | comp | - | 0.05 | 6.37 | 24.53 |
| | comm | | | | |
| | 256 | 31.60 | 38.67 | 27.74 | 26.38 |
| | 1K | 75.27 | 57.42 | 54.38 | 49.52 |
| 32 | 128K | 7176.16 | 3271.96 | 4408.19 | 3854.76 |
| | # iters | - | 63.00 | 35.52 | 37.76 |
| | comp | - | 0.05 | 13.24 | 46.41 |
| | comm | | | | |
| | 256 | 49.82 | 41.58 | 41.17 | 37.79 |
| | 1K | 117.18 | 62.73 | 81.15 | 69.42 |
| 48 | 128K | 11188.30 | 3631.69 | 6610.21 | 5260.51 |
| | # iters | - | 63.00 | 51.58 | 53.74 |
| | comp | - | 0.06 | 20.26 | 65.43 |

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

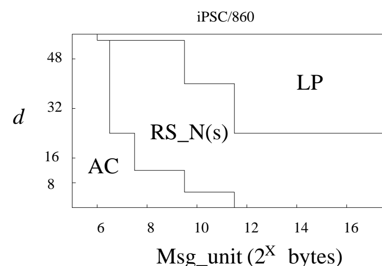


Figure 5: Regions for which the different algorithms outperform the others (on a 64-node iPSC/860)

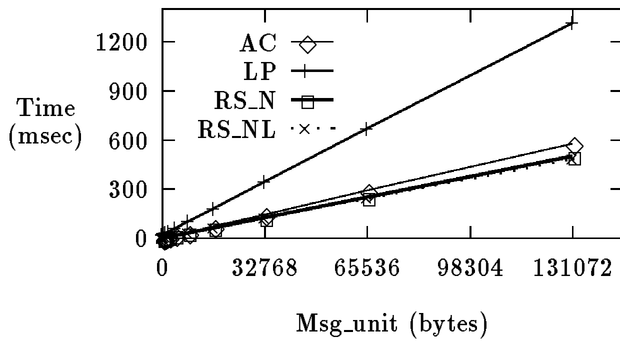


Figure 6: Communication cost for uniform messages with $d = 4$ on a 64-node iPSC/860.

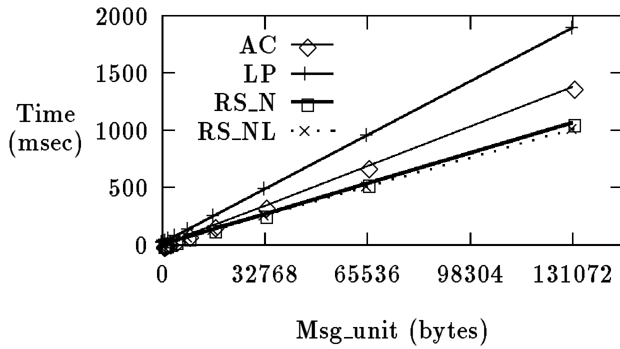


Figure 7: Communication cost for uniform messages with $d = 8$ on a 64-node iPSC/860.

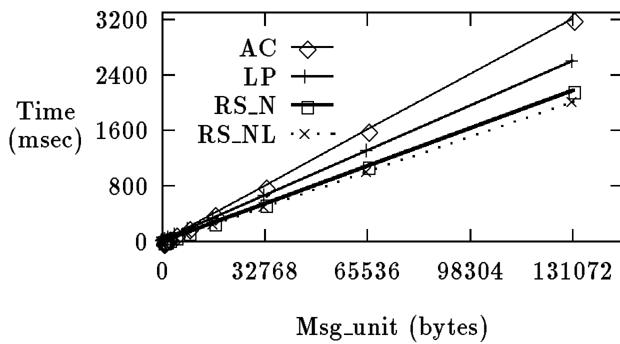


Figure 8: Communication cost for uniform messages with $d = 16$ on a 64-node iPSC/860.

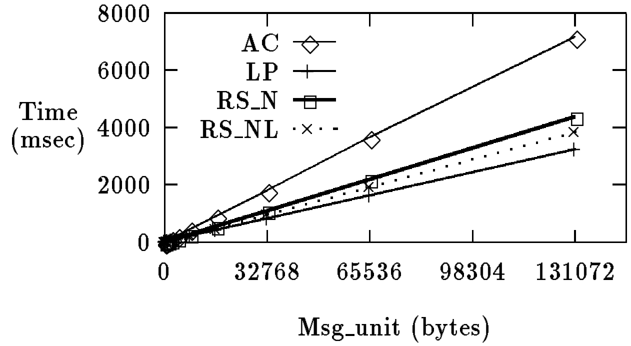


Figure 9: Communication cost for uniform messages with $d = 32$ on a 64-node iPSC/860.

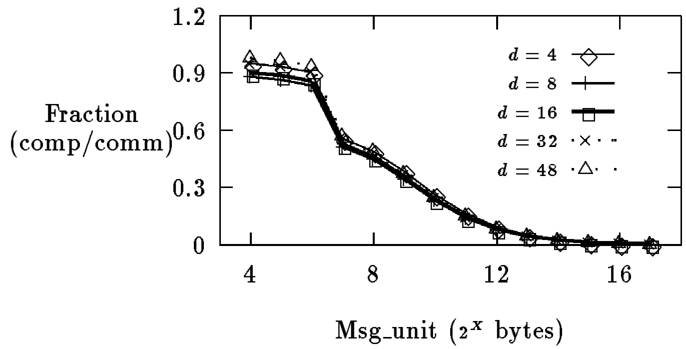


Figure 10: Computation overhead of RS_N algorithm in terms of communication cost

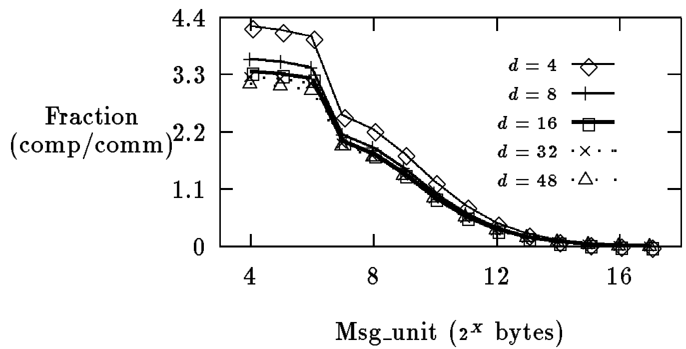


Figure 11: Computation overhead of RS_NL algorithm in terms of communication cost