NICE: Non-uniform Irregular Communication Exchange on Distributed Memory Systems[1]

Jhy-Chun Wang   Tseng-Hui Lin   Sanjay Ranka

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

Email: *jcwang/thlin/ranka@top.cis.syr.edu*

[1]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.

Abstract

A communication package, Non-uniform Irregular Communication Exchange (NICE), is designed to help users in scheduling message-passing requests on distributed-memory machines. This package schedules a batch of messages into a set of partial permutations and provides communication primitives to carry out the communication. The NICE primitives are focused on generating communication schedules to minimize node contention and also link contention.

Index Terms: Communication scheduling, distributed-memory systems, irregular communication, message-passing, runtime support.

1 Introduction

For distributed-memory parallel computers, load balancing and reduction of communication are two important operations for achieving a good performance. It is important to map a program such that the total execution time is minimized; 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 (such as High Performance Fortran) to decompose the data and its corresponding computations. This usually results in regular collective communication between processors. Many such primitives have been developed in the literature [1, 9].

Mapping of irregular problems (such as unstructured finite element grid) tends to result in unstructured communication patterns, such that each processor needs to send messages, with no obvious patterns, to some number of processors. Further, for a large class of such problems, the same schedule is used many of times. 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 it.

We have developed scheduling algorithms that decompose a communication matrix (representing an all-to-many communication in which each node sends different messages to a subset of processors) into a set of disjoint partial permutations, $pm_1, pm_2, \ldots, pm_l$, l, a positive integer, such that if processor P_i needs to communicate with processor P_j, then there exists a a, $1 \leq a \leq l$, so that $pm_a^i = j$. Permutations have the useful property that each node receives at most one message and sends at most one message (and hence there is no node contention). With the advent of new routing methods [5, 8], the distance to which a message is sent is becoming relatively less and less important. Thus, assuming no link contention, permutation can be an efficient communication primitive despite the fact that the number of hops each message needs to travel may be different.

Thus communication proceeds through a number of phases in a loosely synchronous fashion, and each communication phase is free of node contention and/or link contention. Another problem needs to be addressed: In one communication phase, processors with small messages may be idle while waiting for processors with large messages to complete their work. In this case, the largest message in one communication phase may dominate the communication cost.

We introduce methods to reduce the variance of message sizes within one permutation. Large messages can be split into smaller pieces such that the variance of message sizes within one permutation can be minimized, and each of these split pieces is sent in different phases. We studied several tradeoffs in "increase in number of phases" versus "cost of non-uniformity of message sizes in one communication phase."

We have carefully studied the node contention and link contention properties of several MPP machines (e.g., the CM-5, iPSC/860, and Intel Touchstone Delta) from the above perspective and, using the schemes we developed, performed extensive experiments on these machines, The results show that these methods can significantly reduce the communication cost over naive methods. For most cases the cost of scheduling is small enough that it can be performed at runtime.

The NICE (Non-uniform Irregular Communication Exchange) package is a set of primitives that takes an input user program's communication request and returns a scheduling table (each node program gets its own table) that contains information concerning incoming messages. It also creates a communication phase table (CPT), a list structure with each block containing information of one permutation. Thus, if the same batch of communication requests is required again, the schedule table can be reused. NICE also provides communication primitives that are used to take the CPT as input and carry out the required communication.

The NICE primitives are specifically aimed to deal with unstructured communication. Several scheduling schemes can be implemented in the NICE package. Depending on the cost of generating the communication schedule and the number of times a schedule needs to be used, one scheduling algorithm may be preferable to another.

In this paper, notations, definitions, and general communication properties used throughout are given in Section 2. Section 3 describes the scheduling schemes used in the NICE package. Section 4 presents an overview of the NICE package, and Section 5 describes the use of NICE primitives in one application. Experimental results on the CM-5 and iPSC/860 are given in Section 6, while Section 7 discusses the relative comparison of different scheduling methods. Finally, conclusions are given in Section 8.

2 Preliminaries

In order to minimize the cost of communication on distributed-memory machines, the following factors must be addressed:

1. *node contention*, where two or more nodes each try to send one message to the same node in one communication phase.

2. *link contention*, where two or more messages, in the same communication phase, interact with each other due to shared communication link(s).

Figure 1 shows examples of (a) node contention and (b) link contention. In (a), P_1, P_3, P_5, and P_7 all try to send messages to P_4 in the same communication phase. For machines that can only send/receive a limited number of messages, this situation will cause some processors to delay sending their messages. In (b), P_3 sends a message to P_8, while P_4 sends a message to P_5 in the same communication phase. As observed, both communication pairs use the communication link $edge_{45}$, which will cause communication delay.

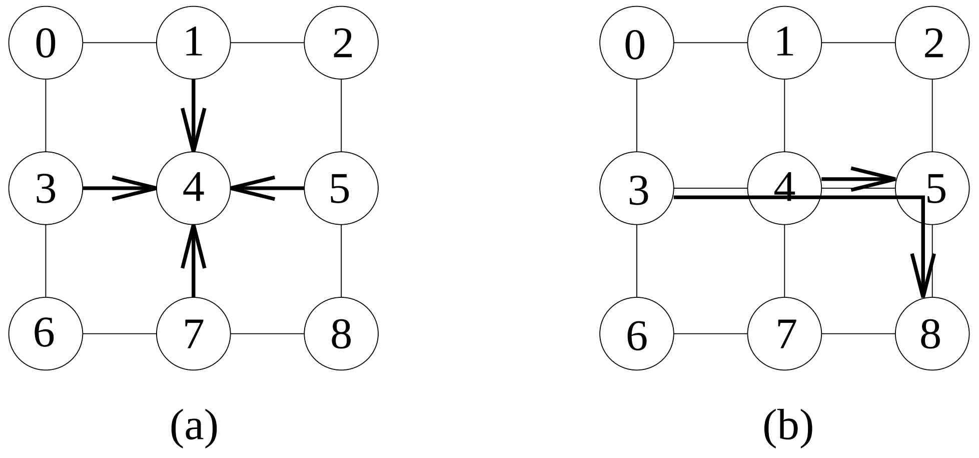


Figure 1: Examples of: (a) node contention; (b) link contention

For systems that can receive only one (or a limited number of) message(s) at a time, node contention may cause delay for some of the sending nodes. Link contention will cause a different degree of impact for different routing strategies [8]. (It will cause severe communication degradation in machines using *circuit-switching*, message-routing strategy [8, 10].) Bandwidth utilization, which is related to link contention to some degree, is another issue that should be taken care of in order to achieve efficient communication.

2.1 Notations and Definitions

We first give the formal definition of node contention and link contention. In this paper, the terms “node” and “processor” are used interchangeably.

On a system with n processors, we define the communication matrix COM as an $n \times n$ matrix, and $COM(i,j)$ is equal to a positive integer m if processor P_i needs to send a message (of size m unit) to P_j, $0 \leq i,j < n$. Otherwise, $COM(i,j) = 0$. Thus, row i of COM represents the destination vector of processor P_i, which contains information about the destination and size of different messages. The $n \times n$ communication matrix COM can be decomposed into a set of communication phases (cp_k, $1 \leq k \leq l$, l, a positive integer) such that

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

We define the k^{th} communication phase as

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

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

Thus, *node contention* can be formally defined as

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

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

A partial permutation pm_k is a communication phase that

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

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

$pm_k^i = -1$ 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 will not cause any node contention.

The methods developed to reduce link contention assume a static routing algorithm, i.e., based on the source and destination nodes, we 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^{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:

$$\exists k, \;\; 1 \le k \le l, \;\; pm_k^{i_1} = j_1 \quad \text{and} \quad pm_k^{i_2} = j_2, \;\; i_1, i_2 = 0, 1, \ldots, n-1 \quad \text{and} \quad 0 \le j_1, j_2 < n \; ,$$

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

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

$$\forall k, \;\; 1 \le k \le l, \;\; pm_k^{i_1} = j_1 \quad \text{and} \quad pm_k^{i_2} = j_2, \;\; i_1, i_2 = 0, 1, \ldots, n-1 \quad \text{and} \quad 0 \le j_1, j_2 < n \; ,$$

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

2.2 Random Permutation as a Collective Communication Primitive

On a 32-node CM-5, we generated 5,000 random permutations in which each processor sends and receives a message of 1K bytes. Over 99.5% (4,979 out of 5,000) of the permutations were within 5% of the average cost (Figure 2). Thus, the variation of time required for different random permutations (in which each node sends data to a random, but different node) is very small on a 32-node CM-5. With this observation, the performance of proposed approaches in NICE, which uses random permutation as the underlying communication scheme, are not significantly affected by a given sequence of permutation instances.

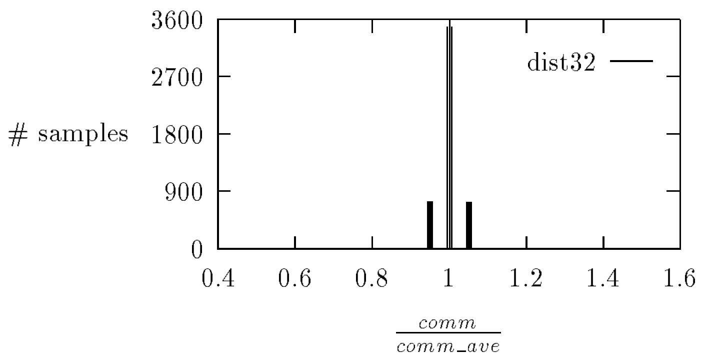


Figure 2: Communication cost distribution for 5,000 permutation samples with message of length 1K bytes on a 32-node CM-5

Figure 3 shows the results of 5,000 random permutations on a 32-node iPSC/860 and reveals that even under the influence of link contention—iPSC/860 uses a circuit-switching routing algorithm that will create some degree of link contention—random permutation can still provide reasonably good performance.

3 Proposed Approaches

The NICE provides several communication scheduling schemes to deal with a wide range of communication patterns. Those schemes have been intensively studied in [10, 11] where their ability to reduce communication cost is shown. Following is a summary of the scheduling schemes implemented in the NICE.

3.1 Asynchronous Communication (AC)

The most straightforward approach is to use asynchronous communication. This scheme does not introduce any scheduling overhead. The algorithm is divided into three phases:

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

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

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

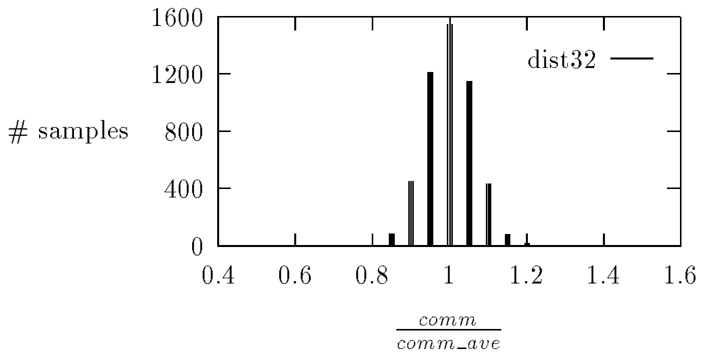


Figure 3: Communication cost distribution for 5,000 permutation samples with message of length 1K bytes on a 32-node iPSC/860

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

Figure 4: Asynchronous communication algorithm

During the send-receive process, the sender processor does not need to wait for a completion signal from the receiver processor in order to continue sending outgoing messages until they have all been sent. This naive approach is expected to perform well when the density, d, is small. The asynchronous algorithm is given in Figure 4.

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 have only limited space for message buffers. In such cases, when the system's buffer space is fully occupied by unconfirmed messages, further messages will be blocked at the sender processor's side. This overflow may block processors from doing further processing (including receiving messages), because processors are waiting for other processors to consume and empty their buffers in order to receive new incoming messages. The situation may never be resolved and a deadlock may occur among the processors.

This approach causes no scheduling overhead, and each processor sends out messages to their destinations without particular order. The performance of this scheme will depend

```
**Linear_Permutation()**
For all processors $P_i$, $0 \le i \le n-1$, *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 5: Linear permutation algorithm

on the congestion and contention on the nodes and network. It is basically suitable for situations where communication traffic is light and/or message sizes are small.

3.2 Linear Permutation (LP)

In this algorithm (Figure 5), each processor P_i sends a message to processor $P_{(i \oplus k)}$ and receives a message from $P_{(i \oplus k)}$, where $0 < k < n$. When $COM(i,j) = 0$, processor P_i will not send a message to processor P_j (but will receive a message from P_j if $COM(j,i) > 0$). The entire communication uses pairwise exchange ($j = i \oplus k \Leftrightarrow i = j \oplus k$).

The complexity of this algorithm is $O(n)$ regardless of the number of messages each processor actually sends/receives. This scheme is typically useful when every processor needs to communicate with all (or nearly all) other processors. The algorithm, in Figure 5, assumes that the number of processors, n, is a power of 2. However, it can be easily extended when n is not a power of 2.

3.3 Random Scheduling Avoiding Node Contention (RS_N)

During communication scheduling, the worst-case time complexity to access each entry of COM is $O(n^2)$. In order to reduce this overhead, the first step of this algorithm is to compress the COM into an $n \times d$ matrix $CCOM$ by a simple compressing procedure [10]. This procedure will improve the worst-case time to access each active element (of $CCOM$) to $O(dn)$.

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

The vector *prt* is used as a pointer whose elements point to the maximum number of positive columns in each row of *CCOM*. 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 that processor P_i needs to send a message to processor P_j, and $receive(j) = i$ 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. 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 an 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 $receive(k) = i$.

The RS_N algorithm [10] is described in Figure 6.

Assuming that each node sends d messages to random destinations and receives d messages from different sources, one can perform the following approximate analysis [10]:

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

The intent of this approach is to minimize the number of permutations needed to complete the communication while avoiding any node contention. The scheduling overhead of this scheme and its extensions is higher than LP, but the scheduling cost can be amortized over several utilizations, as the same schedule may be used repeatedly.

We also include three extended scheduling schemes that will avoid link contention and/or reduce the variance of message sizes in one permutation.

RS_NL: This approach avoids node contention as well as link contention and is extremely useful when the underlying message passing is done using *circuit-switching* [10]. It may also affect the utilization of bandwidth for machines using other message routing strategies.

RS_NH: When the variance of message sizes in one communication phase is large, if we allow every processor to completely send its message, then the communication time in each phase may be upper bounded by the maximum message size in each phase. We assume that each communication phase is executed in a loosely synchronous fashion. This alleviates the problem to some extent. However, if the variance of message size is large, processors with small messages may still be idle while waiting for processors with large messages to complete their execution. In order to reduce idle time for processors, this approach chooses a reasonable message size in each communication phase such that

Random_Scheduling_Node()

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

2. For all processors $P_i$, $0 \leq i \leq n-1$, *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 6: RS_N Algorithm: Random scheduling avoiding node contention

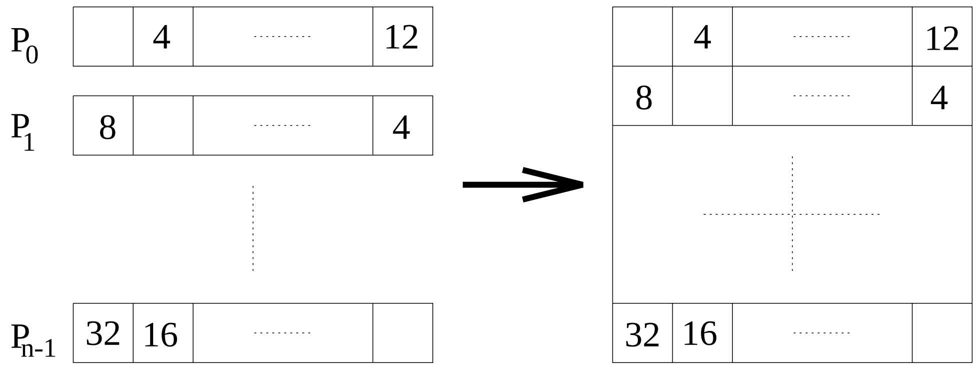


Figure 7: The concatenation of destination vectors

processors with small messages will send their messages completely, while processors with large messages will send only part of their messages. This scheme is shown to be useful in dealing with communication patterns with non-uniform message sizes [10].

RS_NLH: This scheme is similar to RS_NH, but it also takes link contention into consideration, which is important for machines using circuit-switching.

Each of the approaches described above have their corresponding primitives in the NICE, which we will introduce in the next section.

4 An Overview of NICE Package

The NICE primitives can be classified into two groups. The first group, *scheduling primitives*, works on concatenating destination vectors (of each processor) into a global communication matrix *COM*, and then decomposes the matrix into a set of permutations and returns a scheduling table to the user program. The second group, *communication primitives*, takes the returned scheduling table as input, then carries out the communication.

4.1 Scheduling Primitives

The scheduling primitives take as input the destination vectors of all processors and concatenates them into a communication matrix (Figure 7).

As mentioned in Section 3, each scheduling scheme may be suitable for a particular class of communication patterns. Besides the destination vector, the scheduling primitives will also take as input a parameter indicating which scheduling scheme should be applied. Users currently need to specify the scheduling scheme they prefer when they call the NICE package. We are working on developing an evaluation subsystem that will provide users with information about the relative performance and cost of these scheduling schemes. Users can

```
struct NICE_Comm_Phase {                              /* communication phase table */
      int s_partner, s_len, s_offset;             /* information of outgoing messages */
      int r_partner, r_len, r_offset;             /* information of incoming messages */
      NICE_comm_phase *next;                            /* pointer to next CPT block */
};
typedef struct NICE_Comm_Phase NICE_comm_phase;
```

Figure 8: The data structure of communication phase table (CPT)

use the information to decide upon the correct scheduling scheme. The final goal of this evaluation subsystem is to be able to automatically choose the most suitable scheduling scheme. Thus, making this decision process transparent to user programs.

The scheduling primitives will return to the user program a scheduling table that contains a (source) vector that stores information about the source and size of different incoming messages, as well as a pointer to a CPT that specifies the communication operations for each phase.

Since the same scheduling table may be used many times (which is the case in nested iterative computations), the scheduling table can be kept and reused as often as needed. The NICE package also provides a function for releasing the table space if the table is no longer needed [12].

4.2 Communication Primitives

The NICE assumes that the user program has packed the outgoing messages (of each processor) in such a way that each processor has a vector of pointers in which each element points to one outgoing message buffer. The length of each buffer is equal to its corresponding entry in the destination vector. The user program is also expected to pre-allocate incoming message buffers according to the source vector returned in the scheduling table (which also has a pointer to the communication phase table (CPT)). The data structure of the CPT is given in Figure 8.

In each communication phase, *s_partner* (*r_partner*) specifies the destination (source) of outgoing (incoming) message, *s_len* (*r_len*) specifies the length of the outgoing (incoming) message, and *s_offset* (*r_offset*) specifies the location of the outgoing (incoming) message at its corresponding message buffer. The NICE communication primitives use this information to conduct the communication.

The *s_len* and *s_offset* (*r_len* and *r_offset*) are particularly important when the scheduling

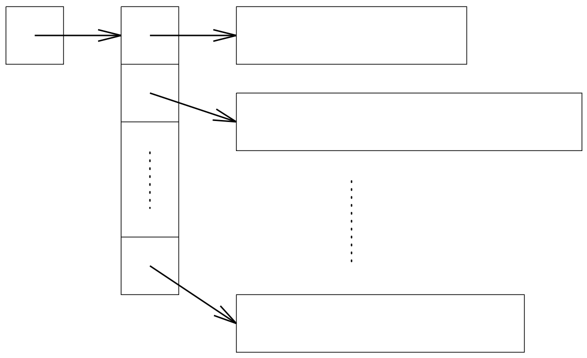


Figure 9: The data structure of send/receive vectors

scheme RS_NH (or RS_NLH) is used. As mentioned in Section 3.3, RS_NH and RS_NLH employ measures to reduce the variance of message size within one communication phase, thus some of the larger messages are split into pieces, and each piece is sent at a different phase. The *s_len* and *s_offset* (*r_len* and *r_offset*) are used to store these pieces of information, thus the communication primitives know the length of each message piece and its location in the buffer.

When the user program calls the communication primitives, it needs to pre-allocate the sending and receiving message buffers and pass them as parameters to NICE along with the CPT. The structure of these message buffers is given by a nested pointer structure as the one shown in Figure 9.

5 Using NICE Primitives in Unstructured Applications

In this section, we present a simple example (from examples discussed in [4]) to demonstrate the use of NICE primitives.

A static single-phase computation consists of a single concurrent computational phase, which may be executed repeatedly without change [4]. Examples of static single-phase computations, which are iterative solvers using sparse matrix-vector multiplications, can be found in [13]. Examples of explicit unstructured mesh fluids calculations can be found in [14].

Figure 10 depicts a schematic outline of a kernel from a fluid dynamics simulation that represents a loop that sweeps over the edges of a mesh. The kernel is based on an algorithm that maps a computational domain with irregular polygons. The area and shape of the polygons are determined by heuristic algorithms designed to ensure that the governing partial differential equation is solved with an approximately equal accuracy throughout the computational domain. The data structures used in solving this problem represent a

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

For $i = 1$ to $N$
    1. $v_1 = y\_old(node(i,1))$
       $v_2 = y\_old(node(i,2)$
    2. Calculating flux: $flux = F(x(v_1), x(v_2))$
    3. $y(node(i,1)) = y(node(i,1)) + flux$
       $y(node(i,2)) = y(node(i,2)) - flux$
Endfor
```

Figure 10: Outline of a fluid dynamic simulation sweeping over unstructured mesh

bidirectional graph where vertices represent polygons and edges represent adjacency of the polygons. Sweeping over the polygons is accomplished by traversing the edges of this graph.

In Figure 10, the indices of the two vertices connected by the i^{th} graph edge are denoted by $node(i,1)$ and $node(i,2)$. The computation of *flux* terms requires $y\_old(node(i,1))$ and $y\_old(node(i,2))$. In Step 3, *flux* is added to $y(node(i,1))$ and is subtracted from $y(node(i,2))$. No matter how we partition loop iterations and data, the structure of this problem requires accessing off-processor elements of y and y_old. On distributed-memory machines, it is inefficient to fetch individual off-processor data because of high communication startup latency. Several off-processor fetches can be combined together by using runtime inspectors [6]. NICE primitives can then be used to efficiently schedule the communication and to fetch off-processor data. Note that the scheduling tables can be reused as long as the same set of off-processor accesses is used, i.e., the array *node* is not changed.

6 Experimental Results

We have implemented the NICE package on the CM-5 as well as on the iPSC/860. In this section we show the experimental results of NICE primitives on a 32-node CM-5 and a 64-node iPSC/860. The data sets used in the experiment can be classified in two categories:

1. This test set contains several subgroups, each having 50 different communication matrices. In one matrix, every row and every column has approximately d entries with

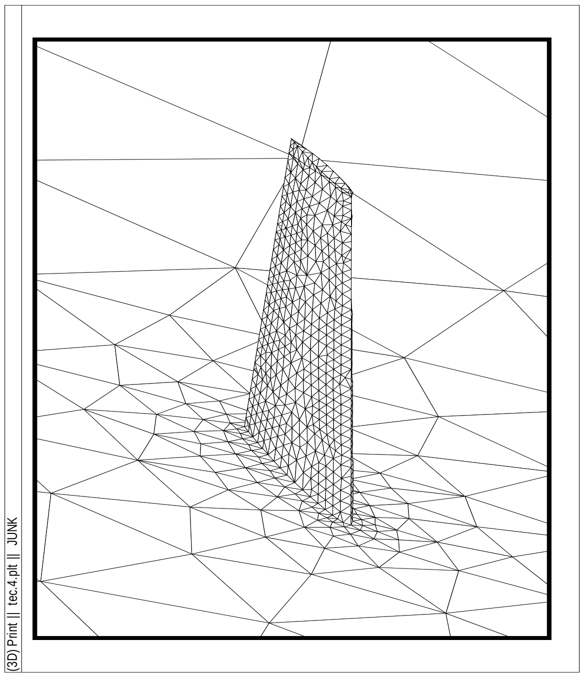


Figure 11: The unstructured grid used for our experiments

positive integer. The procedure we use to generate these test sets is described in [10]. The message lengths used in our test is $COM(i,j)$ multiplied by the variable msg_unit in order to study the effect of message size on each scheme.

2. This test set contains communication matrices generated by graph-partitioning algorithms [7]; the samples represent fluid dynamics simulations of a part of an airplane (Figure 11) with different granularities (2800-point, 9428-point, and 53961-point). In order to observe the NICE primitives' performances with different message sizes, we multiplied the matrices in this test set by a variable msg_unit. In contrast to test set 1, the number of messages sent (or received) by each node is uneven in this test set.

Figure 12 shows that the use of RS_NH (which splits large messages into small pieces to reduce the variance of message size in one communication phase) can significantly improve the performance. RS_NH and RS_NLH are useful only when the message sizes are non-uniform. When all messages in one communication phase are of equal size, other schemes should be used as they have smaller scheduling overhead. When the number of messages sent (and received) are large, LP performs better than other approaches (Figure 13). LP uses pairwise exchange in each communication phase. This can be exploited in machines like iPSC/860 [2, 3] in which a bidirectional communication can be achieved if the sender and receiver are synchronized. The synchronization typically can be achieved only if the communication occurs in pairs.

Figure 14 show the results of test set 2. In this test set, RS_NH performs better than other algorithms on a 32-node CM-5. Another important observation in this test set is that even in the cases where the number of messages sent/received by each processor varies in a wide range, our algorithms still maintain their characteristics and performance. Figure 15 provides similar results on a 64-node iPSC/860, which demonstrates that the schemes used in NICE can be suitable for a variety of architectures.

7 Choosing the Best Scheduling Algorithm

The following parameters are important in choosing the best scheduling algorithm for a given communication matrix:

1. Scheduling cost, S
2. Number of times the same schedule is utilized, R
3. Communication cost generated, C

The average time required for communication of one concurrent iteration of an irregular problem depends on the amortized cost of scheduling, A (S divided by R) and C. Minimization of the total time requires minimization of $A + C$.

If scheduling is done statically (not at runtime), then S can be assumed to be zero. Choosing the best algorithm thus requires estimates of the various costs. Each of these costs depends on the type of machine used, the topology, and the routing algorithm used.

Based on the experiments we conducted, the scheduling cost, in general, varies in the following manner:

$$S\_cost(AC) \leq S\_cost(LP) \leq S\_cost(RS\_N) \leq S\_cost(RS\_NL) \leq S\_cost(RS\_NH\text{s}) \; ,$$

while the communication cost varies in the inverse fashion:

$$C\_cost(RS\_NH\text{s}) \leq C\_cost(RS\_NL) \leq C\_cost(RS\_N) \leq C\_cost(LP) \leq C\_cost(AC) \; .$$

Figures 16 and 17 show the different regions for which each of the schemes is useful on a 32-node CM-5 and a 64-node iPSC/860. Assuming that the scheduling cost is negligible (i.e., the scheduling is performed statically or the scheduling is done at runtime and it is used a large number of times). Choosing between different RS_Ns depends on the underlying network [10, 11].

Currently, the choice of scheduling scheme has to be provided by user program. We are developing an expert system to choose the best approach automatically, making the decision process transparent to user programs.

8 Conclusions

There is a large class of scientific problems that require irregular collective communication. The minimization of the communication cost of such problems is an important issue in achieving good performance and scalability. We have designed the Non-uniform Irregular Communication Exchange (NICE) package, which includes several schemes for dealing with a variety of irregular communication patterns. This package is currently available for the CM-5 and the IPSC/860. We are working on extending the package to run on other distributed memory machines (e.g., the Intel Touchstone Delta). Most of our scheduling schemes are sequential in nature. We are currently investigating parallelization of these methods.

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. Complete exchange on the ipsc/860. Technical Report NASA Contractor Report: ICASE Report No. 91-4, NASA Langley Research Center, January 1991.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

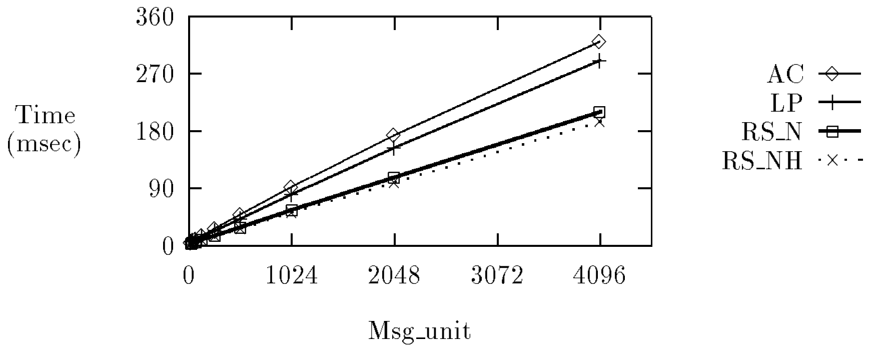
[10] Sanjay Ranka and Jhy-Chun Wang. Static and runtime scheduling of unstructured communication. In *Proceedings of the 2nd Symposium on Parallel Computational Methods for Large Scale Structure Analysis and Design*, Norfolk, VA, February 1993. To appear.

[11] Sanjay Ranka, Jhy-Chun Wang, and Manoj Kumar. All-to-many personalized communication on distributed memory machines. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993. To appear.

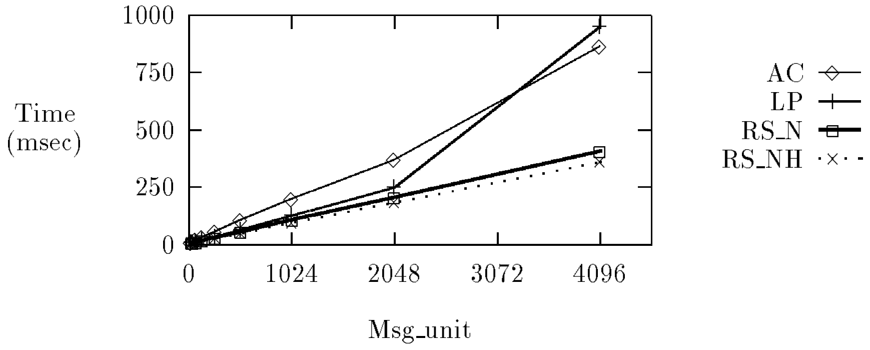
[12] Sanjay Ranka, Jhy-Chun Wang, and Tseng-Hui Lin. A manual for the nice runtime primitives, version 1.0. Technical Report SU-CIS-93, Syracuse University, March 1993.

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

[14] 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.



(density $d = 8$)



(density $d = 16$)

Figure 12: Communication cost for non-uniform message sizes on a 32-node CM-5

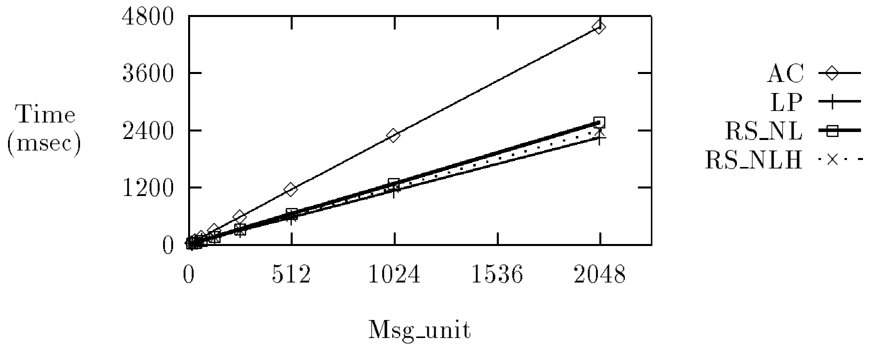
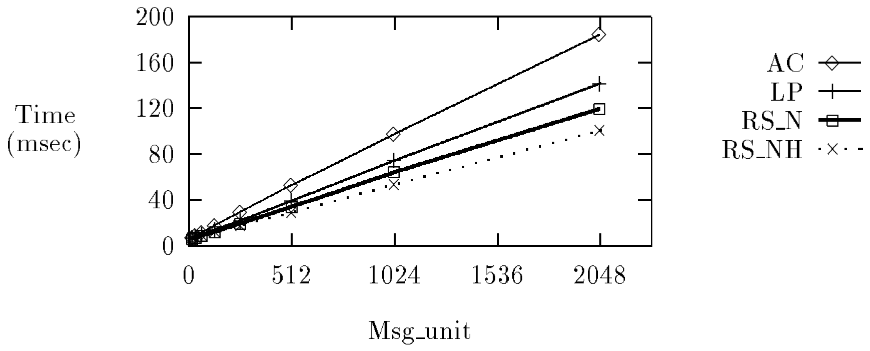
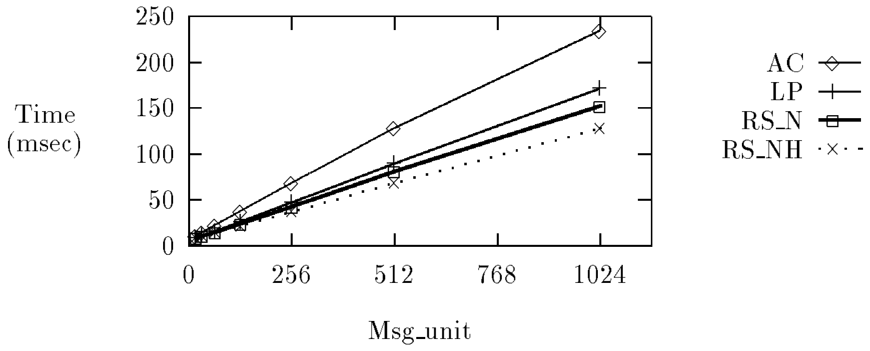


Figure 13: Communication cost for non-uniform message sizes with density $d = 32$ on a 64-node iPSC/860



(2800-point)



(9428-point)

Figure 14: Communication cost for airfoil mesh simulation on a 32-node CM-5

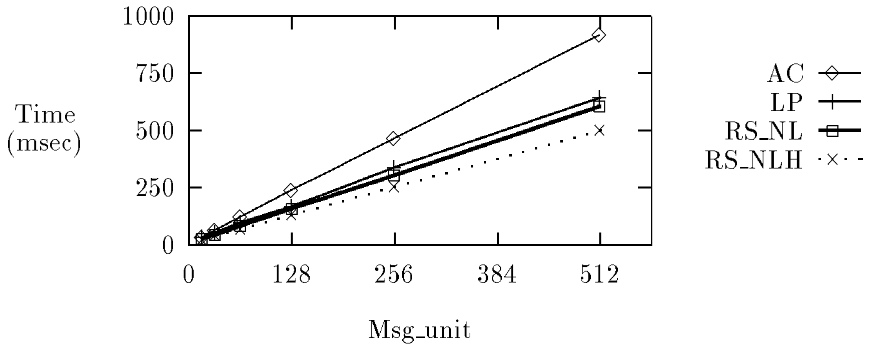


Figure 15: Communication cost for airfoil mesh simulation (53961-point) on a 64-node iPSC/860

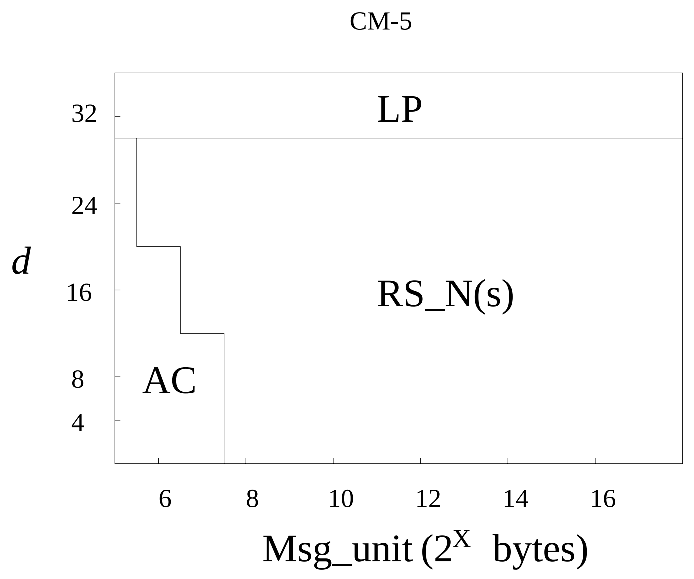


Figure 16: Regions for which the different algorithms outperform the others (on a 32-node CM-5)

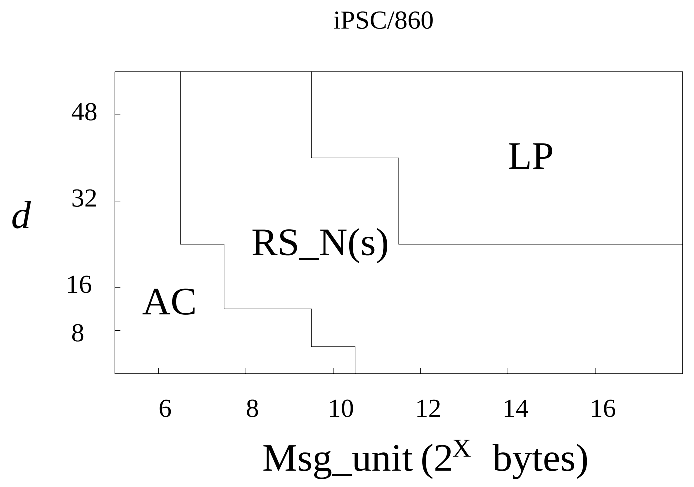


Figure 17: Regions for which the different algorithms outperform the others (on a 64-node iPSC/860)