Distributed Scheduling of Unstructured Collective Communication on the CM-5[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*

July 14, 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 contents do not necessarily reflect the position or the policy of the United States government and no official endorsement should be inferred.

Abstract

Parallelization of many irregular applications results in unstructured collective communication. In this paper we present a distributed algorithm for scheduling such communication on parallel machines. We describe the performance of this algorithm on the CM-5 and show that the scheduling algorithm has very small overhead and gives a significant improvement over naive methods.

Index Terms: Active Messages, communication latency, distributed scheduling, interrupt handler, node contention, personalized communication, unstructured communication.

1 Introduction

Parallelization of many irregular and loosely synchronous problems [1, 3, 7, 9, 14, 16, 17] result in all-to-many personalized communication. An example of all-to-many personalized communication is given in Table 1. A "1" in the (i, j) entry represents the fact that processor P_i needs to communicate to processor P_j. Each message is of different size and each processor may send a different number of messages. In general, assuming a system with n processors, Let COM represent the communication matrix. $COM(i, j)$ is equal to a positive integer m if processor P_i needs to send a message (of m units) to P_j, $0 \leq i, j \leq n-1$. In our example, P_0 sends only three messages while P_4 sends five messages. If we allow processors to arbitrarily send their outgoing messages, it may happen that at one stage processors P_0, P_1, P_3, P_4 and P_6 all try to send messages to processor P_2. Since the receiving processor can typically receive messages from only one processor at a time, one or more of the sending processors may have to wait for other processors to complete their communication. We use the term *node contention* to refer to this situation.

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

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

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

The results reveal that when the number of messages sent to the same node (at the same time) increases, the time each sending node needs to complete sending its message also increases (the same holds true for the maximum time and minimum[1] time among the

[1]One exception to the time increase is that when all 31 nodes send messages to processor P_{31}. In this

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

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

sending processors). Thus it is inefficient to send more than one message to a particular node at a given time. These observations suggest that node contention will result in overall performance degradation.

In this paper we propose a distributed communication scheduling scheme for reducing node contention. This scheme conducts the scheduling on the fly to reduce the node contention. Each processor maintains a status bit which describes whether the processor is busy receiving a message. Before sending a message a processor performs a test-and-set operation to find out if the receiving node is busy. The test-and-set operation requires hardware and software support for message interrupts at the receiving nodes. Further, for the method to be efficient, the cost of this operation should be small. In this paper we use Active Messages [6] for the test-and-set operation.

Our scheduling scheme is distributed in nature and hence is useful even in the cases that the same communication pattern is used only a few times (or once). In contrast, some of the algorithms we have developed [11, 12, 13] may be more suitable for the situations that the same schedule is used a large number of times so that the scheduling cost can be amortized.

We do not address link contention in this paper. A main reason being that the routing is randomized on the CM-5. It is not possible to statically schedule messages in such a fashion that link contention can be avoided, although randomization alleviates the problem to a large extent [13]. We show that compared to naive algorithms, our algorithm can result in a significant reduction in the total amount of communication cost.

The rest of this paper is organized as follows. Notations, definitions, and assumptions are given in Section 2. Section 3 briefly describes the different scheduling algorithms we have developed in other papers. Section 4 presents the distributed scheduling algorithm.

case, since nodes P_{28}, P_{29}, P_{30}, and P_{31} are in the same 4-node cluster, so the minimum time taken during this stage is decreased compared with the 16-node case.

Section 5 presents experimental results on a 32-node CM-5 and provides a comparison with other algorithms. Finally, conclusions are given in Section 6.

2 Preliminaries

2.1 Notation and Assumptions

The communication matrix COM is an $n \times n$ matrix where n is the number of processors. $COM(i, j)$ is equal to a positive integer m if processor P_i needs to send a message (of m units) to P_j, otherwise $COM(i, j) = 0$, $0 \leq i, j < n$. Thus, row i of COM represents the sending vector, $sendl_i$, of processor P_i, which contains information about the destination node and the size of outgoing message. Column i of COM represents the receiving vector, $recvl_i$, of processor P_i, which contains information about the source node and the size of incoming message. The entry $sendl_i^j$ ($recvl_i^j$) represents the j^{th} entry in the vector $sendl_i$ ($recvl_i$). Assuming $COM(i, j) = m$, then $sendl_i^j = recvl_j^i = m$. We will use $sendl$ and $recvl$ to represent each processor's sending vector and receiving vector. Several properties of the communication matrix are important in determining the best scheduling algorithm:

1. *Uniformity of message*—All messages are of equal size or not. When the messages are of non-uniform size, reducing the variance of message sizes in the same communication phase may lead to a reduction in overall communication cost [11].

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

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

For the reasons mentioned in the previous section, the algorithms described in this paper does not take link contention into account. We also assume that each processor can send only one message and receive only one message at a time.

2.2 Active Messages

Active Messages [6] is an asynchronous communication mechanism with the following underlying scheme: each message header contains the address of a user-level handler that is executed at the receiving node upon message arrival, with the message body as argument(s). The purpose of the handler is to get the message out of the network and into the current ongoing computation on the receiving node. The handler interrupts the computation immediately upon the arrival of the message and execute to completion. Active Messages are not buffered except as required for network transport, in such case the sending node is blocked until the message can be injected into the network and the handler executes immediately upon arrival receiving node.

2.3 CM-5 CMAML

Culler et al. [6] have shown that on the CM-5 (CMAM) sending a single-packet Active Messages (handler address and 16 bytes of arguments) takes 1.6 μs and receiving such a packet costs 1.7 μs. We have implemented our algorithms on CM-5 using CMMD and CMAML (the CMMD active messages layer)[15]. CMAML is the protocol-less transport layer upon which the higher level CMMD functions are built. CMAML represents an independent implementation of Active Messages developed by UC Berkeley (the functions of Berkeley CMAM and CMAML are not interchangeable).

3 Previous Approaches

We have proposed several algorithms in [11, 12, 13] to address the issues of scheduling unstructured communication on distributed memory machines. In this section we briefly describe these algorithms. We assume that each processor only knows its sending vector *sendl*. The scheduling algorithms we have developed can be classified into two groups.

1. Algorithms that require the global $n \times n$ communication matrix COM.

2. Algorithms that require the receiving vector *recvl*.

In deriving the $n \times n$ communication matrix *COM*, a concatenation operation [4] can be performed on the sending vector *sendl* (of length n) of each processor to derive this matrix at runtime. On an n-node CM-5, performing a concatenate operation with each node contributing a message of size n can be completed in $O(n^2 + \tau \log n)$ amount of time [4] (assuming that a communication can be completed in $(\tau + M\varphi)$ time, where τ is the communication latency, M is the message size, and φ represents the inverse of the data transmission rate).

If only the receiving vector *recvl* is required by each processor, it can either be derived from the COM—obtained from the concatenate operation. or be generated by the algorithm described in Figure 1.

Step 2 can be completed in $O(n)$ time on the CM-5. Step 3 is an all-to-many personalized communication using an asynchronous algorithm (to be described in the next subsection). Each of the messages is a few bytes long.

```
Generate_Recvl()
For all processors $P_i$, $0 \le i \le n-1$, in parallel do

  1. Set entry $sendl\_mask_i^j = 1$ if $sendl_i^j > 0$, otherwise $sendl\_mask_i^j = 0$;
     /* $sendl\_mask_i^j = 1$ means processor $P_i$ needs to send a message to $P_j$. */

  2. Parallel vector sum $sendl\_mask$ and store the results in vector $recvl\_cnt$;
     /* The parallel vector sum returns an identical vector $recvl\_cnt$ in each processor. The
     number of expected incoming messages for processor $P_i$, $recv\_cnt_i$, is equal to the value
     of the $i^{\text{th}}$ entry in vector $recvl\_cnt$. */

  3. Use Active Messages to send each active entry $sendl_i^j$, $0 \le j < n$ to $P_j$, and store the
     data in $P_j$'s $recvl_j^i$. Upon completion, reduce $P_j$'s counter $recv\_cnt_j$ by 1;

  4. Wait until the counter $recv\_cnt_i$ is equal to 0.
```

Figure 1: Procedure of generating receiving vector *recvl*

A comparison of the above two approaches for generating *recvl* for different number of nodes on the CM-5 is given in Table 3. The results show that the second approach, is more efficient than the global concatenate operation. The global concatenate also needs an $O(n^2)$ temporary buffer (*COM*) as compared to $O(n)$ space in the second approach.

3.1 Asynchronous Communication (AC)

The most straightforward approach is to use asynchronous communication. This scheme does not introduce any scheduling overhead. The asynchronous algorithm is given in Figure 2. This approach causes no scheduling overhead, and each processor sends messages to their destinations in a random order. The performance of this scheme will depend on the node contention. It is suitable for situations when density is small and/or message sizes are small.

3.2 Linear Permutation (LP)

In this algorithm (Figure 3), 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 each processor needs

d	32[⋆]		128		256	
	concat[†]	AM[‡]	concat	AM	concat	AM
4	6.624[§]	0.455	106.781	1.397	424.757	2.671
8	6.619	0.548	106.818	1.493	424.922	2.799
16	6.657	0.743	106.666	1.702	424.797	3.066
32	6.626	1.114	106.779	2.114	424.832	3.583
64	-	-	106.786	2.896	424.814	4.552
128	-	-	106.822	5.123	424.843	6.392
256	-	-	-	-	424.911	11.639

⋆: A 32-node (128, 256) partition of CM-5,

†: Using global concatenate to generate *recvl*,

‡: Approach based on Active Messages to generate *recvl*,

§: Cost in milliseconds.

Table 3: Performance comparison of two proposed *recvl* generating procedures

```
Asynchronous_Send_Receive()
For all processors $P_i$, $0 \le i \le n-1$, 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 2: Asynchronous communication algorithm

to send a message to a large subset of all the processors involved in the communication. The algorithm in Figure 3 assumes that the number of processors, n, is a power of 2; it can easily be extended to the case where n is not a power of 2.

3.3 Scheduling Algorithm that Avoiding Node Contention

This scheduling algorithm (RS_N [11]) decomposes the communication matrix into a set of disjoint partial permutations, $pm_1, pm_2, \ldots, pm_l$, where l is a positive integer, such that if processor P_i needs to communicate with processor P_j, then there exists a a, $1 \le a \le l$, such 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, 10], the distance to which a message is sent is becoming

```
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 $sendl_i^j > 0$ then $P_i$ sends a message to $P_j$;
      if $recvl_i^j > 0$ then $P_i$ receives a message from $P_j$;
  endfor
```

Figure 3: Linear permutation algorithm

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.

The communication proceeds through a number of phases in a loosely synchronous fashion, and each communication phase is free of node contention. The scheduling approach tries to minimize the number of permutations needed to complete the communication by using randomization in scheduling process. The RS_N algorithm is described in Figure 4, and a detailed description is in [11].

Assuming each node sends d messages to random destinations and receives d messages from different sources, we can perform the following approximate analysis [11]:

- The average time complexity for generating a permutation is $O(n \ln d + n)$.
- The number of permutations needed to complete the message-scheduling is bounded by $d + \log d$.

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. Although we assume the communication is executed in a loosely synchronous fashion, processors with small messages may be idle while waiting for processors with large messages to complete their execution.

In order to eliminate idle time for processors, this approach can be modified to use a cutoff message size in each communication phase such that processors with small messages will send their messages completely, while processors with large messages will send only part of their messages. This scheme uses a heap data structure to order the messages to be sent within each processor and is shown to be useful in dealing with non-uniform message sizes [11]. We use the term RS_NH to represent this algorithm.

```
Random_Scheduling_Node()

1. Use the $n \times n$ 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 all entries of vectors $Psendl$ and $Precvl$ to $-1$;
   (b) In each row $i$ of $CCOM$, try to find an active entry $CCOM(i,j) = k$ $(0 \leq k < n)$
       such that entry $COM(i,k)$ is the only entry picked along row $i$ and column $k$ of
       $COM$ in this iteration;          /* every processor executes the same program */
   (c) Set $Psendl(i) = k$ and $Precvl(k) = i$;
   (d) Reset $CCOM(i,j)$ to $-1$;
   (e) if (Psendl(i) $\neq -1$) then $P_i$ sends a message to $P_{Psendl(i)}$;
       if (Precvl(i) $\neq -1$) then $P_i$ receives a message from $P_{Precvl(i)}$;

   Until all messages are sent
```

Figure 4: RS_N Algorithm: Random scheduling avoiding node contention

4 Distributed Random Scheduling which Avoids Node Contention (DRS_N)

In contrast to RS_Ns algorithms described in previous section, the DRS_N approach does not create a schedule table. This scheme conducts the scheduling on the fly to reduce the node contention. Each processor maintains a status bit which describes whether the node is busy receiving a message. Before sending a message a node performs a test and set operation to find out if the receiving node is busy. If it is, the sending node will try another node using the same procedure. This approach guarantees that each processor will receive at most one message (excluding the test-and-set messages) at a time.

We use Active Messages to perform the test-and-set operation. Each processor has a local variable *busy_lock* initially set to FREE. When one processor's inquiry arrives, the receiving processor's computation is interrupted and the corresponding handler is executed. If the processor that sent the inquiry receives a FREE signal, it will send the required message; when the sending process is completed, the sending processor will send another Active Message with handler to reset the receiving processor's *busy_lock* to FREE so that it can receive messages from other processors. This process is continued on each processor

```
Random_Scheduling_ActiveMessages()
1. Generate_Recvl(sendl, recvl);
2. For all processor $P_i$, $0 \leq i \leq n-1$, in parallel do
    (a) Pre-allocate receiving buffers according to receiving vector recvl;
    (b) Repeat
        i. Select a destination node from sending vector sendl, use Active Messages to
           test-and-set destination node's busy_lock;
        ii. If the destination node is free to receive message,
            A. Send message to the destination node;
            B. Upon completion, reset destination node's busy_lock to free;
            C. Reset the corresponding entry in sending vector sendl;
        Until sending vector sendl is empty
    (c) Wait until all incoming messages arrive at their proper buffers.
```

Figure 5: DRS_N algorithm

until each processor has sent all its outgoing messages (and every processor has received all its incoming messages).

The DRS_N algorithm is given in Figure 5. In Step 2(b)i, a delay can be introduced so that a processor will wait a variable amount of time before it retries an inquiry on the same processor. This will, in general, reduce the number of inquiries.

5 Experimental Results

We have implemented our algorithms on a 32-node CM-5. In this section, we describe the test data sets used in the evaluation. The data sets for our experiments can be classified into the following categories:

1. This test set contains several subgroups, each of which has 50 different communication matrices with the same value of d. In each matrix, every row and every column have approximately d active entries (d is equal to $4, 8, 16, 24$, and 31, respectively). The procedure we use to generate these test sets is described in [11].

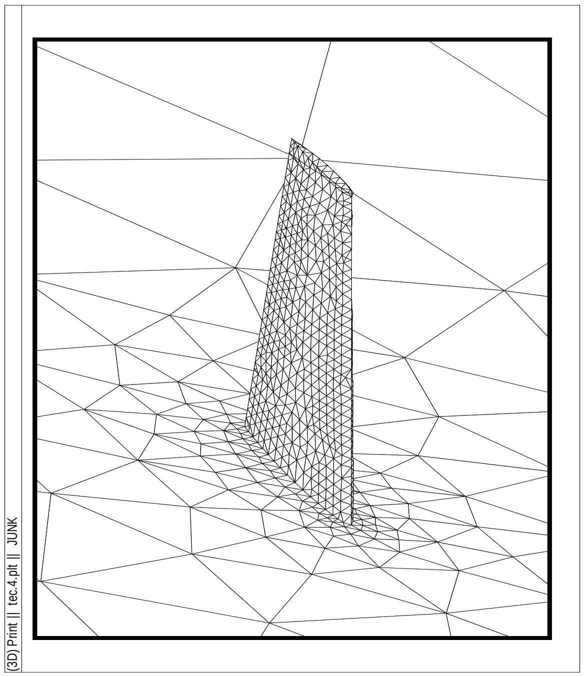


Figure 6: The unstructured grid used for our simulations

The messages in one communication phase are of equal size. The message length used in this test set is equal to msg_unit, which is ranged from 2^4 bytes to 2^{17} bytes.

2. This test set is similar to the previous one, except that the message sizes are non-uniform, where the size is equal to $COM(i,j)$ multiplied by msg_unit. The different values of msg_unit used in this test set are 2^k for $4 \leq k \leq 13$.

3. This test set contains communication matrices generated by graph partitioning algorithms [8]; the samples represent fluid dynamics simulations of a part of an airplane (Figure 6) with different granularities (2800-point, 9428-point, and 53961-point). In order to observe the algorithm's performance with different message sizes, we have multiplied the matrices in this test set by a variable msg_unit. The different values of msg_unit used for our experiments are 2^k for $4 \leq k \leq 11$.

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

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

The corresponding values for the 9428-point sample are 16, 3, 10.5, 99, 1, 32.04; and for the 53961-point sample they are 18, 6, 10.81, 276, 1, 93.21, respectively.

5.1 Results and Discussion

5.1.1 Uniform Distribution with Uniform Message Sizes

Table 4 and Figure 7 show the results of test set 1. If the same schedule is used a large number of times such that the scheduling cost can be amortized, RS_N is superior to other algorithms [11].

If the same schedule can be used only once, AC is the best algorithm for small sized messages, while RS_N is preferable for large sized messages. LP has good performance when each processor sends messages to a large subset of processors involved. For medium sized messages, DRS_N algorithm is the best.

5.1.2 Uniform Distribution with Non-Uniform Message Sizes

Table 5 and Figure 8 show the results of test set 2. With the non-uniform message sizes in this test set, the results of RS_NH show that it is worth the effort to reduce the variance of message sizes in one communication phase. However, that comes with a cost of maintaining heap structures in the communication matrix COM [11]. The relative performance of the algorithms is similar to the one described in the previous section. However, if the schedule is used only once, then DRS_N seems to be the best option for a large range of messages.

5.1.3 Airfoil Mesh

Table 6 and Figure 9 show the results of test set 3. In this test set, DRS_N performs better than RS_N and has results close to the performance of RS_NH. If the same schedule is used only once, DRS_N is the best choice for a large range of messages.

6 Conclusions

In this paper we have developed a distributed communication scheduling algorithm to reduce node contention. In contrast to centralized scheduling algorithms [11], the DRS_N has a small scheduling cost. This feature makes it useful in situations that the same communication pattern is used only a small number of times (or only once).

One issue we have not addressed in this paper is how to reduce the number of inquiries. Each processor must send one or more inquiries to another processor before it succeeds to send data. Each inquiry interrupts the receiving node's computing and forces the processor to execute the Active Messages handler. A good approach would reduce the number of inquiries and also reduce idle time for each processor between the reception of two messages from different processors. One solution is to insert a delay function that will wait for a certain amount of time (long enough for the receiving node to complete its current incoming message) before allowing a processor to send another inquiry to the same processor. This feature can be added to our algorithm. However, our experiments suggest that the improvement achieved is small and the optimal delay is dependent on the particular instant of the communication pattern.

Acknowledgments

We wish to thank Raja Das, Joel Saltz, and Dimitri Mavriplis at ICASE and Nashat Mansour for the illustration in Figure 6 and for the corresponding communication matrices.

References

[1] E. Anderson and Y. Saad. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing*, 1(1):pp. 73–95, 1989.

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

[3] D. Baxter, J. Saltz, M. Schultz, S. Eisentstat, and K. Crowley. An experimental study of methods for parallel preconditioned krylov methods. In *Proceedings of the 1988 Hypercube Multiprocessor Conference*, pages pp. 1698–1711, Pasadena, CA, January 1988.

[4] 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.

[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] T.V. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992. ACM Press.

[7] P.C. Liewer and V.K. Decyk. A general concurrent algorithm for plasma particle-in-cell simulation codes. *Journal of Computational Physics*, 2:pp. 302–322, 1985.

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

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

[10] 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.

[11] Sanjay Ranka and Jhy-Chun Wang. Static and runtime scheduling of unstructured communication. *International Journal of Computing Systems in Engineering*, 1993. To appear.

[12] Sanjay Ranka, Jhy-Chun Wang, and Geoffrey C. Fox. Static and runtime algorithms for all-to-many personalized communications on permutation networks. In *Proceedings of the 1992 International Conference on Parallel and Distributed Systems*, pages pp. 211–218, HsinChu, Taiwan, December 1992.

[13] Sanjay Ranka, Jhy-Chun Wang, and Manoj Kumar. Personalized communication avoiding node contention on distributed memory systems. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993. To appear.

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

[15] Thinking Machines Corporation, Cambridge, MA. *CMMD Reference Manual*, version 3.0 edition, December 1992.

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

[17] 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.

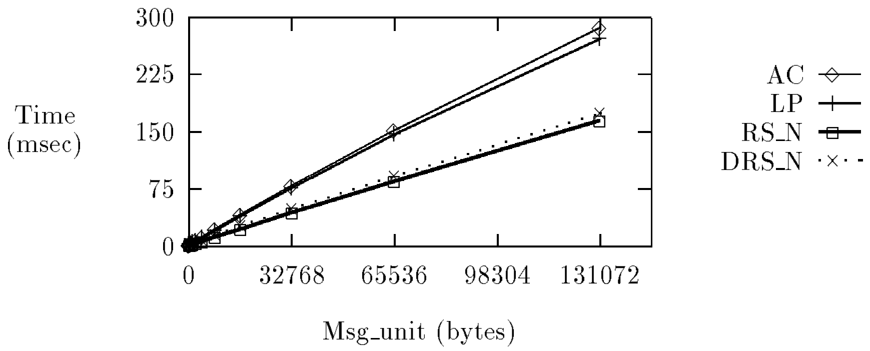
d	*msg_size*	AC	LP	RS_N	**DRS_N**
4	comm*				
	16	2.065	3.410	1.679	2.994
	512	3.539	4.415	2.480	3.743
	1024	4.819	5.547	3.192	4.505
	2048	7.388	7.738	4.365	5.911
	4096	12.368	12.118	6.911	8.764
	32768	79.343	77.439	44.317	49.391
	131072	286.952	272.649	164.951	174.290
	comp*	0	0.119	1.572	-
	perm†	-	31	5.54	-
16	comm				
	16	6.585	7.788	6.072	10.416
	256	11.317	9.136	7.728	11.750
	512	14.997	10.850	9.072	13.262
	1024	21.796	13.982	11.589	16.187
	2048	35.014	19.517	15.702	21.315
	4096	61.103	31.016	24.749	31.697
	131072	3236.886	1421.718	1105.936	1169.099
	comp	0	0.129	6.267	-
	perm	-	31	18.56	-
31	comm				
	16	16.558	8.934	11.335	23.563
	128	23.297	9.745	12.918	24.847
	256	28.944	11.111	14.678	26.226
	512	36.995	13.684	17.252	29.158
	1024	53.933	17.581	21.929	35.054
	16384	505.558	131.647	154.408	194.999
	131072	11409.862	1980.108	2388.292	2765.645
	comp	0	0.138	12.857	-
	perm	-	31	34.2	-

$\star$: Communication cost, in milliseconds;

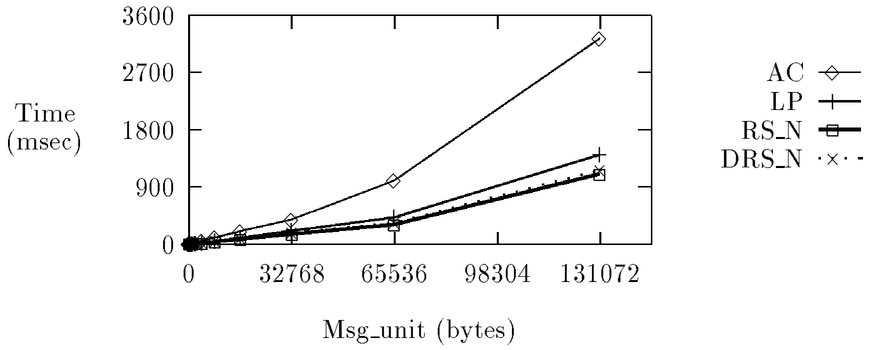
$*$: Scheduling cost, in milliseconds;

$\dagger$: Number of communication phases needed.

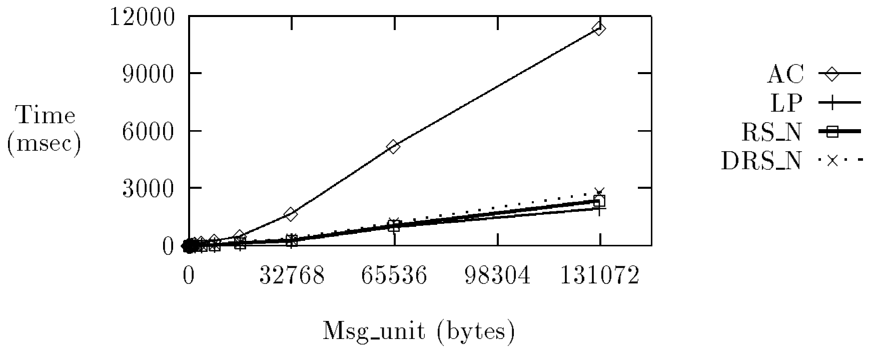
Table 4: Experimental Results for uniform message sizes on a 32-node CM5



(density $d = 4$)



(density $d = 16$)



(density $d = 31$)

Figure 7: Communication cost for uniform message sizes on a 32-node CM-5

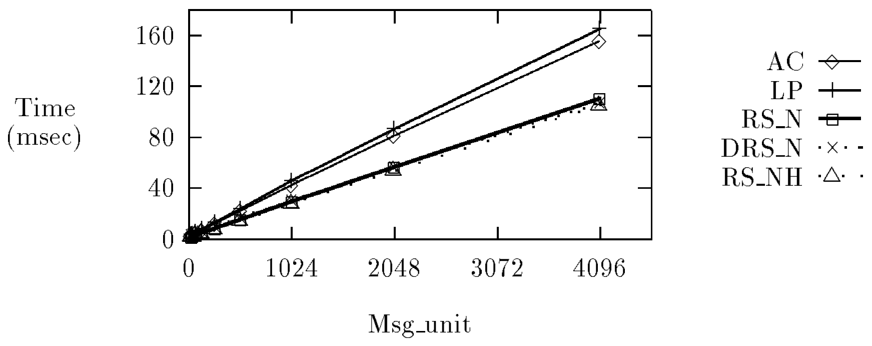
d	*msg_unit*	AC	LP	RS_N	**DRS_N**	RS_NH
4	comm*					
	32	3.619	4.577	2.613	3.827	2.571
	64	4.954	5.862	3.409	4.632	3.331
	256	12.689	13.477	8.392	9.891	8.113
	1024	42.846	46.208	29.550	30.601	28.209
	2048	81.316	87.027	56.625	56.750	53.990
	4096	155.872	165.550	110.546	106.924	105.026
	comp*	0	0.118	1.572	-	6.316
	perm†	-	31	5.54	-	5.66
16	comm					
	16	11.234	9.804	7.948	11.960	7.852
	32	14.727	11.603	9.473	13.523	9.136
	128	34.214	21.628	18.381	22.194	16.961
	512	108.618	66.060	55.899	58.828	49.703
	2048	371.240	250.002	207.397	196.709	183.184
	4096	865.381	953.548	406.768	468.583	359.516
	comp	0	0.128	6.325	-	43.358
	perm	-	31	18.56	-	19.1
31	comm					
	16	28.366	12.533	15.077	26.541	14.729
	64	50.627	20.128	23.576	35.462	22.100
	256	132.293	52.541	58.326	70.255	51.424
	1024	482.330	200.451	206.937	209.073	177.053
	2048	1668.837	396.685	400.478	579.559	342.341
	4096	4577.956	1298.740	1342.152	1481.374	1231.085
	comp	0	0.137	12.912	-	122.033
	perm	-	31	34.2	-	34.56

$\star$: Communication cost, in milliseconds;

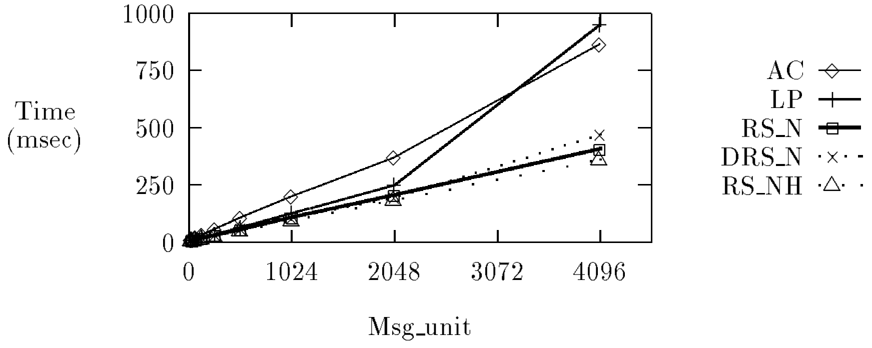
$*$: Scheduling cost, in milliseconds;

$\dagger$: Number of communication phases needed.

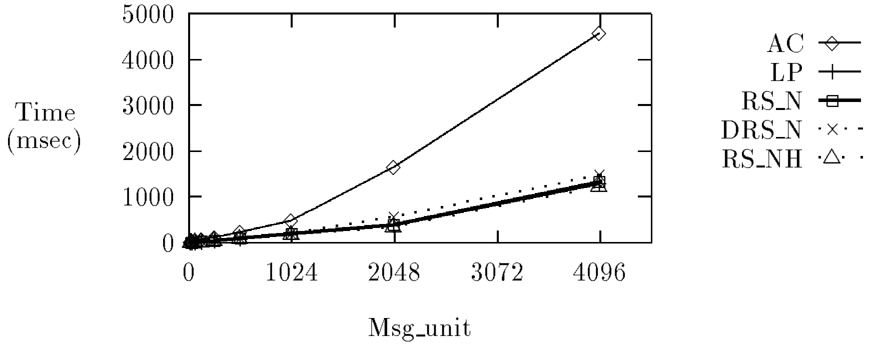
Table 5: Experimental Results for non-uniform message sizes on a 32-node CM5. The minimum message size in each level is *msg_unit* bytes, and the maximum size is $32 \times msg\_unit$ bytes.



(density $d = 4$)



(density $d = 16$)



(density $d = 31$)

Figure 8: Communication cost for non-uniform message sizes on a 32-node CM-5

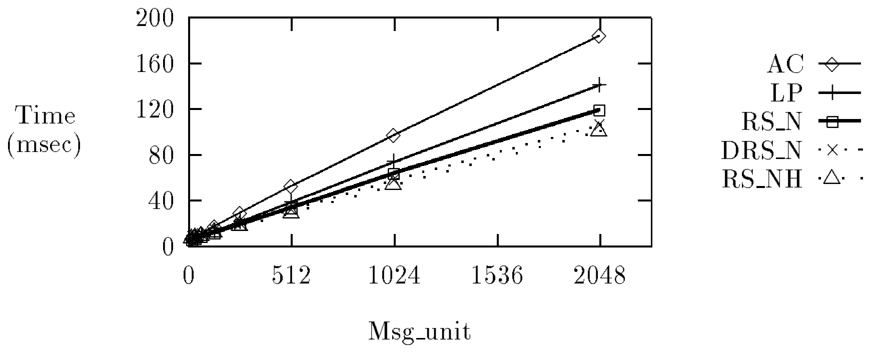
points	*msg_unit*	AC	LP	RS_N	**DRS_N**	RS_NH
2800	comm⋆					
	16	7.297	6.796	6.349	8.750	7.122
	32	8.825	8.007	7.204	9.628	7.803
	64	11.929	10.099	8.962	11.198	9.290
	128	18.047	13.810	12.456	14.302	12.143
	256	29.597	21.663	19.485	20.212	17.799
	512	53.111	38.904	34.011	32.315	29.253
	1024	97.933	74.094	64.044	58.028	53.717
	2048	184.354	141.512	119.301	106.379	100.708
	comp∗	0	0.148	7.954	-	27.211
	perm†	-	31	15.15	-	19.65
9428	comm					
	16	10.392	9.157	8.266	10.712	9.182
	32	14.597	11.631	10.414	12.781	10.991
	64	22.538	16.407	14.981	16.624	14.601
	128	37.867	26.516	24.067	24.243	21.710
	256	68.371	47.555	42.762	40.115	36.715
	512	128.381	90.137	81.079	70.539	68.232
	1024	234.905	171.527	152.494	131.186	127.600
	comp	0	0.149	8.945	-	34.946
	perm	-	31	16.45	-	22.5
53961	comm					
	16	19.104	15.419	14.109	16.791	14.965
	32	30.835	24.117	21.693	23.839	21.037
	64	53.607	42.560	37.050	37.310	33.671
	128	100.113	79.964	69.197	65.144	59.847
	256	179.254	153.044	130.772	118.193	110.744
	comp	0	0.149	9.754	-	37.272
	perm	-	31	18.05	-	26.4

⋆: Communication cost, in milliseconds;

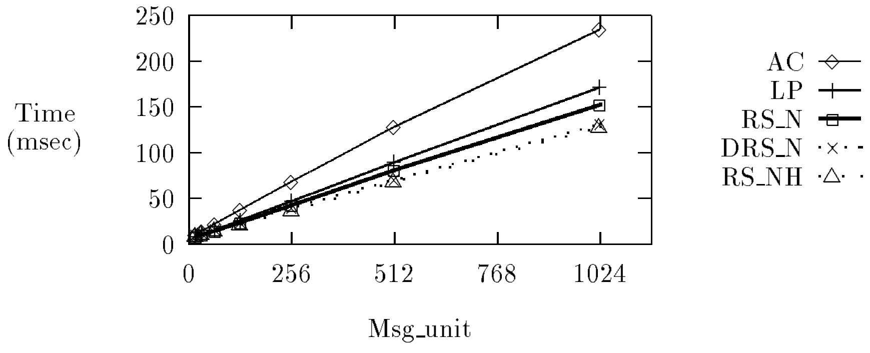
∗: Scheduling cost, in milliseconds;

†: Number of communication phases needed.

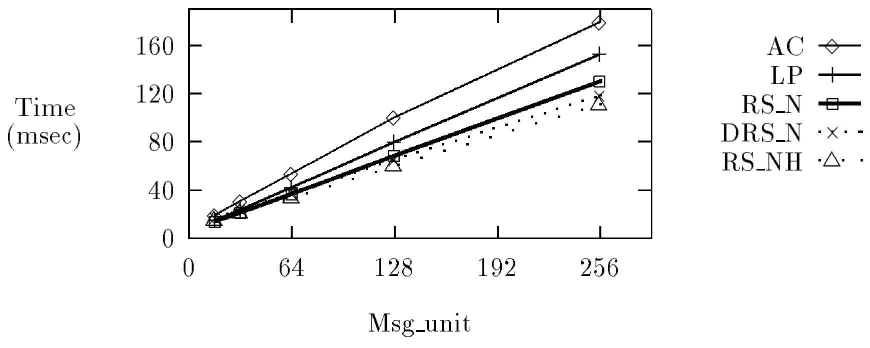
Table 6: Experimental Results for airfoil mesh simulations on a 32-node CM5. The minimum message size in each level is *msg_unit* bytes, and the maximum size is 36 (99, and 276 for each different points, respectively) $\times$ *msg_unit* bytes.



(2800-point)



(9428-point)



(53961-point)

Figure 9: Communication cost for airfoil mesh simulation on a 32-node CM-5