All-to-Many Communication Avoiding Node Contention

Sanjay Ranka
Syracuse University

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

SU-CIS-92-20

All-to-Many Communication Avoiding Node Contention

Sanjay Ranka and Jhy-Chun Wang

September 1992

School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, NY 13244-4100

All-to-Many Communication Avoiding Node Contention

Sanjay Ranka[1] and Jhy-Chun Wang[1]
4-116 Center for Science and Technology
School of Computer and Information Science
Syracuse University
Syracuse, NY 13210

Manoj Kumar
IBM T.J. Watson Research Center
Hawthorne, NY

[1]Work was supported in part by NSF under CCR-9110812.

Abstract

In this paper we present several algorithms for all-to-many personalized communications which avoid node contention. Our algorithms assume that the size of messages are non-uniform. We discuss these algorithms and study their effectiveness both from the view of static scheduling as well as runtime scheduling. The algorithms are based on decomposing the communication matrix into a set of partial permutations. We also propose measures to reduce the variance of message sizes in one permutation such that each processor's communication time in one permutation can be approximately equal. In a system with n processors and every processor sends and receives d unique messages, our algorithm can complete the scheduling in $O(\frac{d}{\lambda}(n\log^2 d))$ time on an average, and use an expected number of $\frac{d}{\lambda} + \frac{\log d}{\lambda^2}$ partial permutations to carry out the communication, where $\lambda = \frac{k}{n}$, k is the number of messages which are completely sent out in one permutation. We present experimental results of our algorithms on the CM-5.

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

1 Introduction

For distributed memory parallel computers, load balancing and reduction of communication are two important issues for achieving a good performance. it is important to map the program such that the total execution time is minimized, the mapping can be typically performed statically or dynamically. For most regular and synchronous problems [7], this mapping can be performed at the time of compilation. For applications with regular communication patterns, the communications can be easily explicitly specified using system primitives like *broadcast, send, receive, concatenate, etc*, and these regular communication patterns can be recognized by parallel compilers [3, 8, 12].

For some other class of problems, which are irregular in nature, achieving good mapping is considerably more difficult. The nature of this irregularity may not always be known, and can be derived only at runtime. The handling of irregular problems requires the use of runtime information to optimize communication and load balancing [6, 11, 14]. These packages derive the necessary communication information based on the data required for performing the local computations and data partitioning. Typically, the same schedule is used a large number of times. Thus communication optimization is very important and affects the performance of applications on a parallel machine.

In this paper we develop and analyze several simple methods of scheduling communication. These methods are efficient enough that they can be used statically as well as at runtime. Assuming a system with n processors, our algorithms take as input a communication matrix $COM(0..n-1, 0..n-1)$. $COM(i,j)$ is equal to 1 if processor P_j needs to send a message (of size $msg\_len(i,j)$) to P_i, $0 \leq i,j \leq n-1$. Because the message sizes in one permutation may vary in a very wide range, we propose schemes to reduce the variance of message size within one permutation by splitting large messages into smaller pieces, each of which is sent in different phases. Our algorithms decompose the communication matrix COM into a set of partial permutations, $pm_1, pm_2, \cdots, pm_l$, such that if $COM(i,j) = 1$ then there exists at least a k, $1 \leq k \leq l$, that $pm_k(j) = i$.

With the advent of new routing methods [5, 18], the distance to which a message is sent is becoming relatively less and less important. Thus assuming no link contention, permutation seems to be an efficient collective communication primitive. Permutations have a useful property that each node receives at most one message and sends at most one message. For an architecture like the CM-5, the data transfer rate seems to be bounded by the speed at which data can be sent or received by any processor [2]. Further, the routing is randomized, thus a random permutation should make effective use of the bandwidth of the fat-tree like architecture of the CM-5. Clearly, this is not going to be the case for all architectures. However, 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.

Assuming that each of the n processors sends and receives d messages, we perform a probabilistic analysis and show that the complexity of the algorithm is $O(\frac{d}{\lambda}(n \log^2 d))$ on an average. We show that our algorithms are inexpensive to be suitable for static as well as runtime scheduling. If the number of times the same communication schedule is used is large (which happens for a large class of problems [3]), the fractional cost of the scheduling algorithm is quite small. On an average the fraction of extra permutations generated are not very high. Further, compared to a naive algorithm which takes time proportional to n permutations, this algorithm has significant speedup.

The rest of the paper is organized as follows. Section 2 gives the notation and assumptions of message routing algorithms. Section 3 presents the algorithms and their time complexity analysis. Section 4 provides approaches to decide value λ. Section 5 presents the experimental results. Finally, conclusions are given in Section 6.

2 Notation and Assumptions

2.1 Category

We categorize the routing algorithms in several different categories:

1. *Uniformity of the message* - All messages are of equal size or not. In this paper we assume that messages are of non-uniform size. In case the messages are nearly of the same size, the Compact Global Masking (CGM) algorithm developed in [16] has considerably smaller scheduling overhead ($O(dn \ln d)$).

2. *Density of communication matrix* - If the communication matrix is nearly dense then all processors send data to all other processors. If the communication matrix is sparse then every processor sends to only a few processors. In this paper, we only discuss the latter case. There are a number of algorithms for the former case [1, 10].

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

2.2 Assumptions

We make the following assumptions for the complexity analysis.

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

2. In case the communication is sparse, all nodes send and receive approximately equal number of messages; if the density of sparseness is d then at least d permutations are required to send all the messages.

3 Scheduling Algorithms

In this paper, we assume that each processor has a identical communication matrix COM at compile time. The communication matrix COM is sparse in nature, *i.e.* each processor will send and receive d unique messages (in a system with n processors). Our algorithms can be easily modified to be useful at runtime : we assume each processor knows its sending vector only at runtime, all processors can then participate in a concatenate operation which will combine each processor's sending vector to form the communication matrix COM and leave a copy at every processor. For architectures like CM-5, performing a concatenate operation is efficient and can be completed in $O(dn)$ amount of time [2]. These operations have efficient implementation on other architectures like hypercubes and meshes [1, 15].

The communication patterns considered in this paper are *all-to-many personalized communication* (all-to-all personalized communication is a special case of all-to-many personalized communication). In personalized communication, one processor sends a different message to different processors [10]. We also assume that COM is a non-uniform communication pattern, *i.e.*, messages may not be of equal size. We have developed methods for the case when messages are uniform [16].

3.1 Linear Permutation (LP)

In this algorithm (Figure 1), each processor P_i sends a message to processor $P_{(i \oplus k)}$[1] and receives a message from $P_{(i \oplus k)}$, where $0 < k < n$. When $COM(i,j) = 0$, processor P_j will send a dummy message to processor P_i. In every iteration, there is a pairwise exchange: every processor sends messages to and receives messages from same processor. We also experimented the LP with following scheme: each processor P_i sends a message to processor $P_{(i+k) \, mod \, n}$ and receives a message from $P_{(i-k) \, mod \, n}$, where $0 < k < n$. The results show that, for the CM-5, the pairwise exchange approach performs slightly better than the latter scheme.

The time complexity of this algorithm is $O(n(\tau + \varphi M))$, where τ represents the

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

```
Linear_Permutation()
For all processor $P_k$, $0 \leq k \leq n-1$, parallel do
  for i = 1 to n-1 do
      $j = k \oplus i$;
      $P_k$ sends a message to $P_j$;
      $P_k$ receives a message from $P_j$;
  endfor
```

Figure 1: Linear Permutation Algorithm

communication set up cost, φ represents the inverse of the data transfer rate, and M represents the maximum message size.

3.2 Compact Global Masking using Heaps (CGMH)

In the sparse matrix $COM(i,j)$, there are holes ($COM(k,j) = 0$) along each column j. When we look for active entries (entry $COM(k,j) = 1$, $0 \leq k \leq n-1$) along column j, the time used to step through these holes is wasted and should be avoided to minimize unnecessary computation overhead. Keeping this in mind, we compress the $n \times n$ sparse matrix COM into a $d \times n$ totally dense matrix $CCOM$ (Appendix A) such that $CCOM(k,j) = i$ *if* $COM(i,j) = 1$, $0 \leq k \leq d$. Each column of $CCOM$ is sorted by message size in descending order and stored in heap data structure embedded in each column of COM.

The vector *prt* is used as pointers which point to the maximum number of row in each column that contains useful entry. If we perform this compression statically, the time complexity will be $O(n^2)$. This operation can be performed at runtime: each processor compacts one column, and then all processors participate in a concatenate operation which will combine all columns into a $d \times n$ matrix. The cost of this parallel scheme is $O(n+dn) = O(dn)$ assuming that a concatenate can be completed in $O(dn)$ time.

```
Compact_Global_Masking_with_Heap()

  1. Compress matrix $COM$ into a $d \times n$ matrix $CCOM$;

  2. Build a heap $heap_k$ in column $CCOM(*, k)$, $0 \leq k < n$, by each entry
     $CCOM(i, k)$'s, corresponding message size, where $0 \leq i < d$.

  3. Create_Permutation().
```

Figure 2: Compact Global Masking Algorithm with Heap

We assume that $CCOM(i, j) = -1$ if this entry doesn't contain active information. After the copy procedure, the first d rows of each column will contain active entries. When searching for a available entry along column j, the first row i with $CCOM(i, j) = k$ and $receive(k) = -1$ will be chosen. We then set $send(j) = k$ and $receive(k) = j$. Since the entries are sorted by message size, at each step, processors are trying to send out the biggest message size possible. However, at each step the message sizes may vary in a wide range, if we allow every processor to completely send its message, then the communication time complexity in each step is upper bounded by the maximum message size in each step (because CGMH is a loosely synchronous communication pattern, processors with smaller messages may be idle while waiting for processors with largest message to complete). In order to eliminate processors' idle time, we will introduce several approaches in next section to choose a reasonable message size in each step such that processors with smaller messages will send their messages completely, processors with bigger messages will only send part of their messages, and restore the remained messages back to their proper location within heaps.

The CGMH algorithm is described in Figure 2.

Step 1 takes $O(n^2)$ time to complete in sequential program, but we can parallelize this step: each processor create one column of $CCOM$, then all processors participate

```
Create_Permutation()
Repeat
  1. Set $send(0..n-1) = receive(0..n-1) = -1$;

  2. $x$ = random(1..n);
     for $y$ = 0 to n-1 do
        $j = (x+k)\ mod\ n$;  $i = 0$;  $S = \phi$;
        while (send(j) = -1 AND $i \leq$ prt(j)) do
           $k = CCOM(l,j)$, where $l = Heap\_Extract\_Max(heap_j)$;
           if (receive(k) = -1) then;
              $send(j) = k$; $receive(k) = j$;
           else
              $S = S \cup CCOM(l,j)$;  Heap_Remove($heap_j$,l);  $i = i+1$;
           endif
        endwhile
        For all entries, $CCOM(k,j)$, in $S$ (except the last one), Heap_Insert($heap_j$, $M_k^j$);
        /* $M_k^j$ is $CCOM(k,j)$'s corresponding message size */
     endfor

  3. $M_{eff} = Decide\_Size()$;

  4. For all processor $P_i$, $0 \leq i \leq n-1$, parallel do
        $P_i$ sends a message, no bigger than $M_{eff}$, to $P_{send(i)}$;
        $P_i$ receive a message from $P_{receive(i)}$;

  5. For all column $k$, $0 \leq k < n$, which sent a message at this iteration, decreases $prt(k)$
     by 1. If it only sent partial message, add the remainder of the message back to its
     proper location in $heap_k$ and increase $prt(k)$ by 1.

Until no more message to be sent
```

Figure 3: Procedure Create_Permutation()

in concatenating the result together. The time complexity of this parallel version is $O(n) + O(dn) = O(dn)$. Step 2 takes $O(d)$ to build a heap in one column [4], thus it takes $O(dn)$ to complete this step. Step 3.1 takes $O(n)$ time, step 3.3 requests a sort operation (we use merge sort in our paper, which has a time complexity of $O(n \log n)$, this sort operation can be further optimized by using more advanced approach like *radix sort*, which is only required $O(n)$ time) plus the λ approach (which will be discussed at next sections), and step 3.4 takes $O(\tau + \varphi M^k_{eff})$ time (where M^k_{eff} is the most efficient message size at permutation pm_k, which would be decided by the approaches proposed at next section) and step 3.5 takes $O(\log d)$ time to complete.

We are interested in evaluating the average time complexity of step 3.2 and the average number of iterations to complete step 3.

Theorem 1: The time complexity of step 3.2 is $O(n \log^2 d)$.

Proof: We assume that at the beginning of each iteration, the value d (number of active entries) in each column is approximately even and the destinations to which each node has to send data are random (between 1 and n). Then the number of iterations the *while* loop in 3.2 is executed is proportional to $O(n \ln d + n)$ [16]. Each heap operation in 3.2 will require $O(\log d)$ time. Thus the complexity of step 3.2 is upper bounded by $O(n \log^2 d)$. □

We are also interested in the number of entries $CCOM(i,j)$ being consumed in one iteration. *i.e.* the number of entries $CCOM(i,j)$ being reset to -1 in one iteration.

Theorem 2: If every processor in one permutation sends a complete message, then the expected number of entries $CCOM(i,j)$ consumed in one iteration is at least $n - \frac{n}{d+1}$.

Proof: The proof is given in [16]. For completeness, we present the proof in Appendix B. □

But as we mentioned in previous discussion, the maximum message size allowed to be sent in one iteration is M_{eff} instead of M_{max}, Assuming there are $n-k$ entries, in one permutation, with message size greater than M_{eff} (which will only send partial message and return the remaining message back to its heap). Thus the actual entries consumed in one iteration is $k - \frac{n}{d+1}$.

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

$$\begin{aligned} d^* &= \frac{1}{n}(nd - (k - \frac{n}{d+1})) \\ &= d - \frac{k}{n} + \frac{1}{d+1} \\ &= d - \lambda + \frac{1}{d+1} \end{aligned} \tag{1}$$

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

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

would reduce Y_m to $\frac{d}{2}$ (proof is given in Appendix C).

Now we can calculate the number of iterations needed to complete the scheduling. According to equation 2, the number of iterations is upper bounded by

$$\begin{aligned} &(\frac{d}{2\lambda} + \frac{1}{\lambda^2}) + (\frac{d}{4\lambda} + \frac{1}{\lambda^2}) + \cdots + (\frac{1}{\lambda} + \frac{1}{\lambda^2}) \\ &= \frac{1}{\lambda}(\frac{d}{2} + \frac{d}{4} + \cdots + 1) + \frac{1}{\lambda^2}\log d \\ &\approx \frac{d}{\lambda} + \frac{\log d}{\lambda^2} \end{aligned} \tag{3}$$

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

- Time for compressing *COM* into *CCOM*: $O(n^2)$ in the sequential program and $O(dn)$ in the parallelized version;

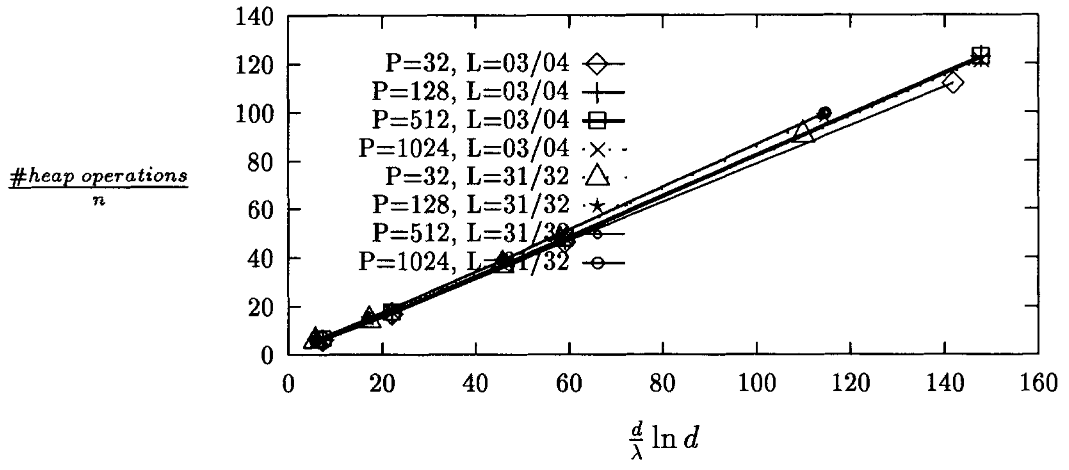

Figure 4: *Number of heap operations / n* versus $\frac{d}{\lambda}\ln d$

- Time for building heaps embedding in *CCOM*: $O(dn)$;
- Time for performing the scheduling: $O((\frac{d}{\lambda}+\frac{\log d}{\lambda^2})\cdot(n\log^2 d))$, which is approximately $O(\frac{d}{\lambda}(n\log^2 d))$;
- Time for sorting one permutation by message sizes: $O(n\log n)$ for merge sort, it may be reduced to take only $O(n)$ time by using histograming techniques.
- Time for performing the communication: $O((\frac{d}{\lambda}+\frac{\log d}{\lambda^2})\cdot(\tau+\varphi M_{eff}))$, which is approximately $O(\frac{d}{\lambda}(\tau+\varphi M_{eff}))$.

The number of heap operations in Step 3.2 was measured for different values of n and λ for randomly generated communication matrices (Appendix D). We have plotted *no. of heap operations / n* against $\frac{d\ln d}{\lambda}$ in Figure 4. The experimental results support our theoretical analysis.

However, our simulation results show that by the time d is close to 1, the number of entries left in each column are uneven, and the degree of unevenness increases as

λ is away from 1. This effect is amplified for large value of n. In order to reduce the impact of unevenness, we propose a two-phase scheduling approach: we use the original approach presented above during the process that d^* is reduced from original d to a small value (we use $\max\{2, \frac{d}{16}\}$ in this paper), at this stage, the number of iterations needed is bounded by

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

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

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

Table 1 and 2 show the comparison of equation (3), (4), number of permutations generated by CGMH with resetting λ, and number of permutations generated by CGMH without resetting. These results reveal that without resetting λ to 1 when d^* is reduced to small value, for a system with 512 nodes, it may take as many as 15 extra iterations to complete the scheduling in comparing with algorithms which employ resetting λ scheme. In section 4, we propose several approaches to decide the value λ (we will use the resetting scheme in all of our approaches).

3.3 Asynchronous Communication (AC)

The previous algorithms can be considered as a sequence of loosely synchronous communication patterns. In this section, we describe a asynchronous communication algorithm. The basic concept of this algorithm is: each processor first sends out all of its out going messages to other processors, then begins to receive in coming messages (some of them may already arrived its receiving buffer) from other processors. During the send-receive process, the sending processor does not need to wait until receiving

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

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

Table 1: Scheduling on 32 nodes system

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

Table 2: Scheduling on 512 nodes system

```
Asynchronous_Send_Receive()
For all processor $P_i$, $0 \leq i \leq n-1$, parallel do
  sends out all out going messages (one-by-one) to other processors;
  receives in coming messages (one-by-one from communication buffer(s)) from other
    processors.
```

Figure 5: Asynchronous Communication Algorithm

processor reply a signal, this is so called *non-blocking communication* that a processor can continue sending messages to different processors without any reply from those processors.

The asynchronous algorithm is given in Figure 5. Similar schemes were proposed in several parallel compiler projects [9, 11].

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. Unlike the LP algorithm discussed in previous section, this algorithm introduces no dummy messages (that will result in no useless communication). But each processor may only have limited space of message buffer, when the buffer is fully occupied by unconsumed messages, further messages will be blocked at sending 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 coming messages. This situation may never resolve and a dead lock may occur among processors. In order to avoid a deadlock, we need to monitor the *production/consumption* rate very carefully to guarantee the completion of communication. In case the message buffer is too small to hold all messages at one time, we need to introduce *strip mining* scheme [9] to perform sends and receives alternately such that there are smaller number of unreceived messages accumulated in the buffer and an overflow will not occur.

4 Approaches of Evaluating Value λ

When the message sizes in one permutation is non-uniform, the communication time is bounded by the maximum message size in that permutation, while other processors with smaller message size are idle and waste their CPU time. In CGMH step 3.3 (Figure 4), the function *Decide_Size()* is used to evaluate the most efficient message size at one permutation, which in turn is finding a value λ such that the communication cost can be reduced.

In function *Decide_Size()*, the first step is sorting all sending message by their size (Figure 6), then different approaches can be introduced to evaluate the best value of λ (that, in turn, will find the value of M_{eff}).

4.1 Fixed λ

This is the most straightforward approach, value λ is fixed throughout the entire scheduling excepts that when d^* is reduced to small value, where λ is reset to 1. This approach requires pre-run the applications several times with different value of λs in order to find out the best value. When the value of λ is found for a application, we can just apply this value at runtime without any extra overhead in finding λ. If we don't know the value of λ beforehand, each processor can begin with a different λ to schedule the communication. The processor with minimum estimated communication time will send its scheduling to other processors, which is then used by all processors to complete the communication.

4.2 λ Proportional to d

In this approach, the value λ is proportional to the value of d^* at current stage. For example, λ can be set as $0.8d^*$, where d^* is the average number of active entries in each column at current stage. The implementation of this scheme is similar to 'Fixed λ' approach that it will require pre-runs to find the best combination of (d, λ).

4.3 Differential Approach

From equation 1, we know that

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

which can be rewritten as

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

In Figure 6, when value λ increases by $\triangle\lambda$, the message size will increase by $\triangle M$, which will affect the communication cost at following categories:

- Since the maximum message size is increased by $\triangle M$, the cost of this extra communication = $\triangle M \times \varphi$;
- The additional utilization of bandwidth = $(1 - \lambda) \times \triangle M \times \varphi$;
- Additional cost due to increasing in set up cost = $d'(\lambda) \times \triangle\lambda \times \tau$.

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

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

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

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

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

The above analysis is under the assumption that all permutations are completed synchronously. Clearly this is not the case in the CGMH algorithm given in Figure 4, in which some processors may begin the next permutation while other processors are still executing the current permutation. Thus, this approach may only generate sub-optimal value of λ.

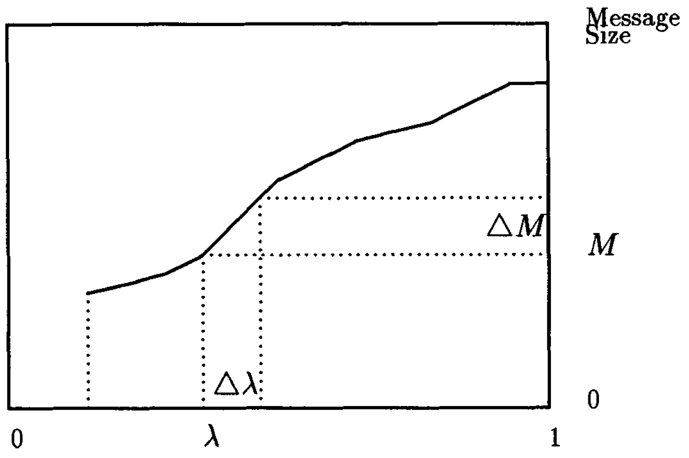

Figure 6: λ versus M graph (1)

5 Experimental Results

We have implemented our algorithms on a 32 nodes CM-5. The algorithms and data sets we use in our experiments will be described in the following sections. We will then discuss the results in detail.

5.1 Algorithms Used in the Experiments

In our experiments, we use following algorithms:

1. **LP**: The Linear Permutation algorithm.
2. **CGM**: The Compact Global Masking algorithm proposed in [16].
3. **CGMH+MS**: The CGMH algorithm with Differential Approach.
   In the process to find a best value of λ in this algorithm, we begin at $\lambda_0 = 0.75$, and increase λ by $\triangle\lambda = \frac{1}{n}$ each time, until it becomes 1. Thus $\lambda_k = \lambda_0 + k \times \frac{1}{n}$, we define

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

then the best value of $\lambda = \lambda_k$ such that k is chosen to maximize

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

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

4. **CGMH+BJ**: The CGMH algorithm with Differential Approach.
   In the process to find a best value of λ in this algorithm, we use the same definitions of CGMH+MS and choose the best value of $\lambda = \lambda_k$ such that

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

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

   Our experiments suggest that the CGMH+BJ algorithm tends to pick a higher value of λ than CGMH+MS, the reason is we always start the λ at $\frac{3}{4}$ and increase the value by $\frac{1}{n}$ each time, if $Gain_j$ is a negative value, it will becomes difficult for λ_{j+1} and above to make up the loss (this is specially true when message sizes are large). On the other hand, the CGMH+BJ is not affected by negative value of *Gain* and has a higher probability to choose a larger value of λ. Our experiments reveal that when the density d in each column of *COM* is approximately equal, the CGMH+BJ performs slightly better than CGMH+MS. But when the variance of density in columns becomes large, the CGMH+MS produces better result. From now on, we will use CGMH to represent the better result of CGMH+MS or CGMH+BJ whenever there is no ambiguity.

5. **CGMH+fixed:** The CGMH algorithm with fixed value of λ, we experiment following λ values: $\frac{3}{4}, \frac{7}{8}, \frac{15}{16}, \frac{31}{32}$, and 1.0. In each sample, at each message size level, we will use the best result among different values of λ to represent the performance (including number of permutations, scheduling cost, and communication cost) of this algorithm at that message size range.

6. **CGM+sort**: This algorithm is same as CGM except that we sort the active entries in each column of *CCOM* by message size at the start of the scheduling

algorithm (we only sort the columns once, and do not make an effort to maintain the sort sequences during the scheduling). This approach will tend to make the largest message in each column being scheduled at earlier permutations whenever possible.

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

5.2 Experimental Data Sets

The data sets we use in this paper are communication matrices *COM* with non-uniform message sizes. These can be classified into three categories:

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

The actual message length used in our test is $msg\_len(i,j)$ multiplied by the variable msg_unit, which is initially equal to 16, and is increased by 2 at every run, the largest value we used is 2^{15}, which in turn represents a possible message size of $2^5 \times 2^{15} = 1Mbytes$. The wide range of message sizes we tested allows us to observe the algorithms' performance at each message size level.

The second test set contains 10 communication matrices. We use three information arrays to represents these samples' characteristics: $dist[5] = \{1,2,4,8,17\}$, $dense[5] = \{1,2,4,8,16\}$, and $length[5] = \{16,8,4,2,1\}$. The columns in one *COM* are grouped into five sets, set k uses the information entries: $dist[k]$, $dense[k]$, and $length[k]$. For example, set 1 uses: $dist[1] = 2$ - meaning there are two columns in this set, $dense[1] = 2$ - meaning two active entries in each column, and $length[1] = 8$ - meaning each active entry's $Msg\_len = 8$. The motivation of this test set is to observe the case where a few processors have a small amount of large messages, while other processors have a bulk of small messages, however, the amount of data to be sent by every processor is equal. We specially want to study the effect of λ in CGMH under these circumstances.

The third test set contains communication matrices generated by load balancing algorithms [13]; the samples represent fluid dynamics simulations of a part of a airplane (Figure 7) with different granularities (2800-point, 3681-point, 9428-point, and 53961-point). We will only present the results of 2800-point and 53961-point samples. In order to observe the algorithms' performance with different message sizes, we have multiplied the matrices in this test set by a variable *msg_unit* which we mentioned above, the largest value we used in this test set is 2^{13}.

We should also mention that in the third test set the number of messages sends (or receives) by each node is uneven. For example, the 2800-point sample has: $max\_d = 15$, $min\_d = 3$, and $ave\_d = 9.25$, and the *msg_lens* are also vary: $max\_len = 36$, $min\_len = 2$, and $ave\_len = 14.12$; the 53961-point sample has: $max\_d = 18$, $min\_d = 6$, $ave\_d = 10.81$, and $max\_len = 276$, $min\_len = 1$, $ave\_len = 93.21$.

5.3 Results and Discussion

We conducted our experiments on a 32 nodes CM-5. The scheduling cost mentioned in this section does not include the time to compress COM into $CCOM$ (CGMs and CGMHs, which will take $O(n^2)$ time in the sequential mode and $O(dn)$ time in the parallelized version). Also, it does not include the time to sort $CCOM$ at the beginning of scheduling (CGM+sort, which will take $O(nd \log d)$ time in the sequential mode and $O(dn)$ time in the parallelized version) and it does not include the time to establish heaps in $CCOM$ at the beginning of scheduling (CGM+heap, which will take $O(nd)$ time in the sequential mode and $O(dn)$ time in the parallelized version). Although the time complexity of some of these ignored operations looks very high, we should point out that these operations are only executed once during the scheduling, so these complexities' constant values are very small when compared with the scheduling cost. Our preliminary experimental results suggested that the exclusion does not affect the final results.

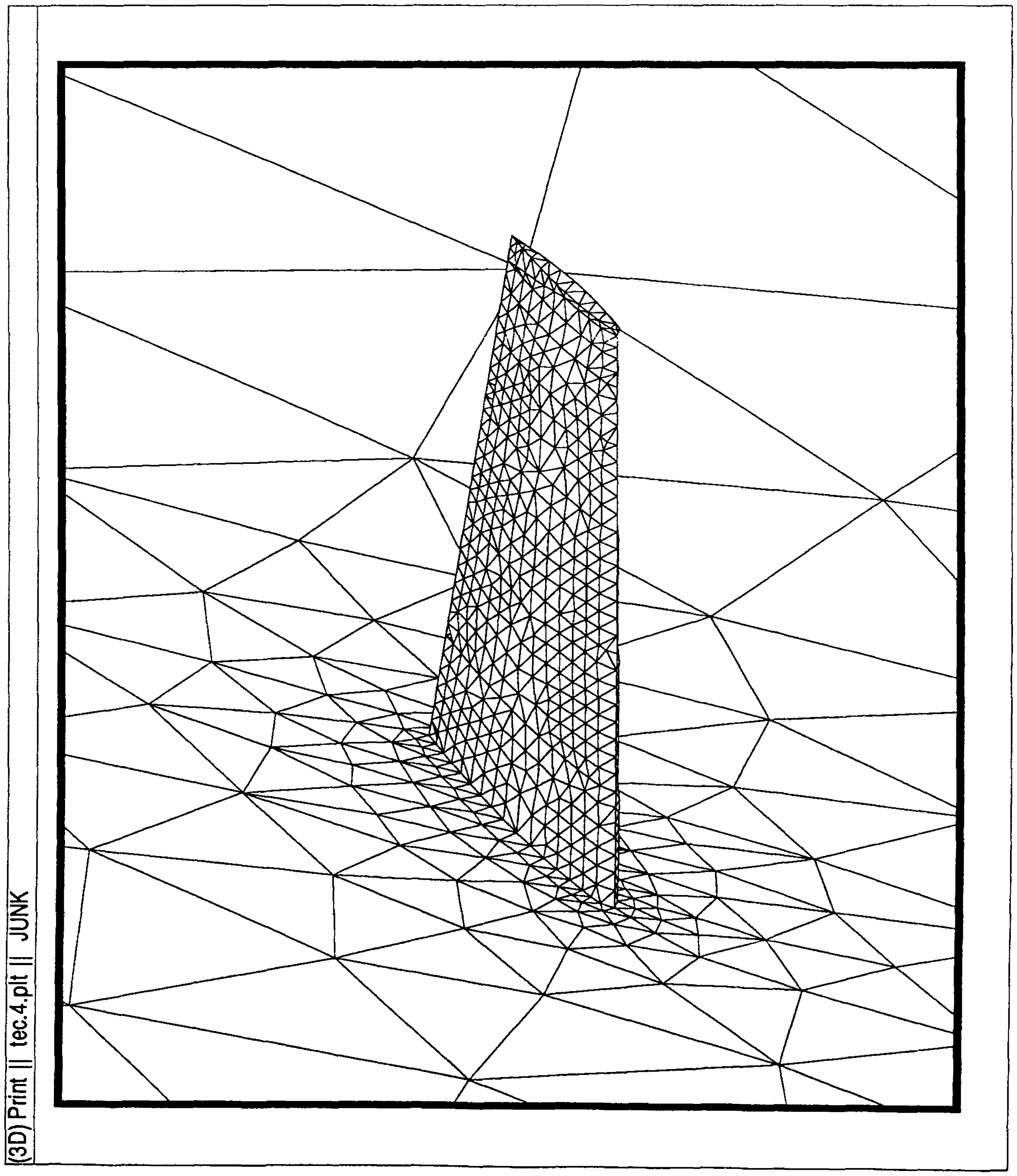

Figure 7: The unstructured grid used for our simulations

Msg_ unit	LP	CGM	CGMH +MS	CGMH +BJ	CGM +sort	CGM +heap
comm*						
16	9.271	3.696	3.7	3.711	3.663	3.608
64	16.838	7.862	7.987	7.933	7.674	7.437
256	46.056	24.007	23.83	23.599	24.177	23.23
1024	166.985	93.243	90.813	90.346	93.081	89.285
4096	659.505	461.421	376.511	375.833	418.1	361.665
16384	3912.66	2817.77	2486.44	2551.58	2588.58	2478.39
32768	8678.48	5505.83	5047.04	4993.06	5336.94	4996.32
comp†	0.331	4.86	23.66	21.038	4.824	14.406
# iters‡	31.0	10.18	12.8	11.6	10.1	10.2

$*$: the total communication cost in milliseconds.

†: the scheduling cost in milliseconds.

‡: number of iterations (permutations) to complete the scheduling.

Table 3: Communication and computation cost for density $d = 8$, the min message size at each level is *Msg_unit* bytes, and the max size is $32 \times$ *Msg_unit* bytes.

5.3.1 Uniform Distribution

Table 3 and Figure 8 show the results of $d = 8$. Since the LP and CGM do not use the heap structure, the message sizes in each permutation may vary in a large range such that the maximum message size in each permutation will affect the outcome. On the contrary, CGMHs always begins with the largest message sizes possible, and λ is employed to decide a efficient message size (M_{eff}) in each permutation. Results show that CGM outperforms LP by a big margin, CGM+sort has a slight improvement over CGM, while CGMHs have very similar results, which have a considerable improvement over CGM. The observations reveal that when the variance of message sizes is large, it is worth the effort to introduce heap structure.

Table 4 and Figure 9 show the results of $d = 16$, the outcomes are similar to the results of $d = 8$, but the differences between each groups become more clear, which

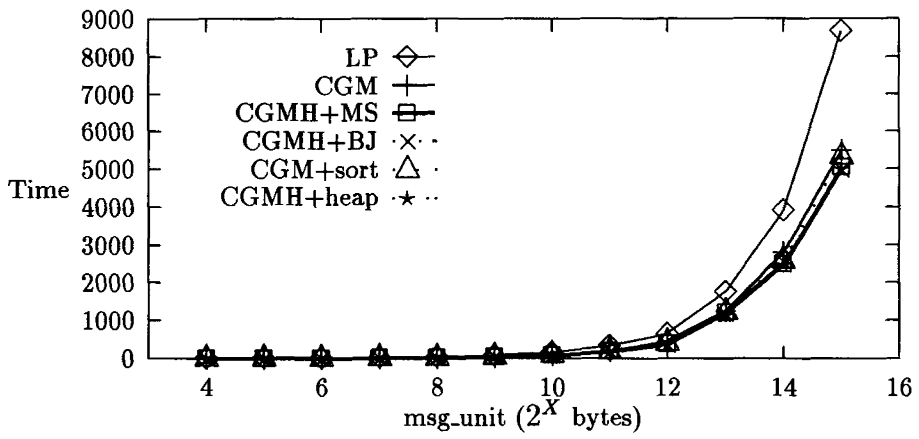

Figure 8: Comparison of communication performance for density $d = 8$

reveals that when the problem size becomes large (either message size or density), the CGMH algorithms are the better choice.

Figure 10 shows that maintaining heap (which are used in CGMHs) is expensive. While the overhead fraction of CGM will drop to less than 0.25 for messages of size 16K [16], the CGMHs' overhead remains high when the message size is less than 16K, but it becomes negligible when the size is larger. We should mention that this comparison is based on the assumption that the same schedule is used only once. In most applications the same schedule will be utilized many times, hence the fractional cost would be considerably lower (inversely proportional to the number of times the same schedule is used). Thus all our algorithms are also suitable for runtime scheduling.

5.3.2 Skewed Distribution

In the second test set, the message sizes in one column (which represent the messages being sent out by one processor) are always the same, this characteristics make

Msg_ unit	LP	CGM	CGMH +MS	CGMH +BJ	CGM +sort	CGM +heap
comm						
16	10.531	6.871	6.978	6.938	6.816	6.564
64	21.461	14.626	15.29	14.602	14.473	13.726
256	64.642	46.037	44.433	43.322	45.23	41.821
1024	246.212	177.323	165.46	163.812	174.903	163.695
4096	1689.99	1206.87	1080.36	661.314	750.824	1124.03
16384	6872.43	5458.28	4717.96	4514.93	4867.82	4793.25
32768	13866.9	10763.8	9321.5	9204.59	10158.6	9553.41
comp	0.363	9.729	72.764	60.87	9.759	43.108
# iters	31.0	18.54	27.4	22.0	18.6	18.88

Table 4: Communication and computation cost for density $d = 16$, the min message size at each level is *Msg_unit* bytes, and the max size is $32 \times$ *Msg_unit* bytes.

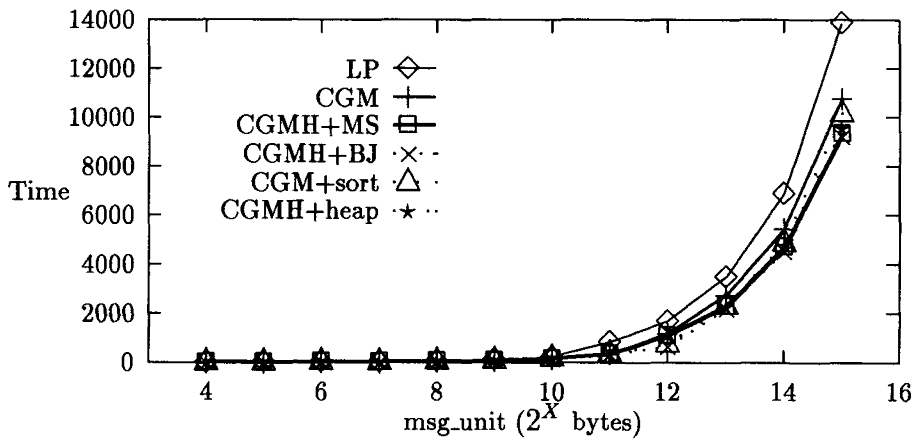

Figure 9: Comparison of communication performance for density $d = 16$

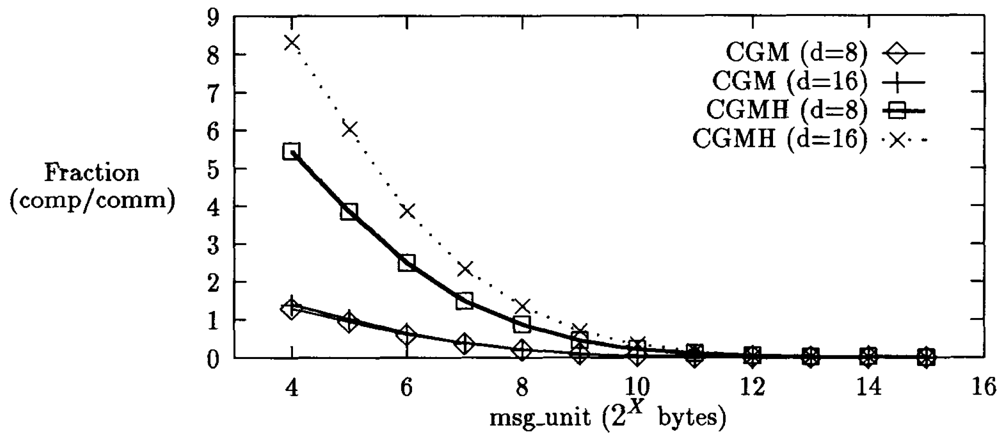

Figure 10: The scheduling algorithms' overhead fraction

CGM+heap useless, because the heap structure will keep the active entries in each column at a very similar order, which will make the probability to find a entry in each column become uneven, and result in more permutations and larger communication cost (in CGM, we randomly swap entries in one column to break the order in order to even the success probability in every column [16]). Also, the columns with larger messages have smaller amount of messages, and the columns with the smallest messages have the largest number of messages, which in turn will dominate the number of permutations needed. Because of this attribute, the splitting of large messages should even the message sizes in one permutation without significantly increasing the number of permutations.

Table 5 and Figure 11 show the results of the second test set. As anticipated, the CGM+heap does not improve the result while CGMH+MS (which tends to select a small value of λ and splits more larger messages into smaller ones) and CGMH+fixed have clear improvements over other approaches.

Msg_unit	LP	CGM	CGMH +MS	CGMH +BJ	CGM +sort	CGM +heap	CGMH +fixed
comm							
16	7.267	4.326	4.37	4.436	4.348	4.042	3.888
64	8.167	4.852	5.04	5	4.836	4.661	4.596
256	12.035	7.484	7.305	7.598	7.452	7.593	7.388
1024	27.077	18.48	16.389	17.238	18.982	18.406	15.918
4096	89.24	63.837	52.863	57.784	63.039	67.391	50.403
16384	343.345	252.116	197.247	226.384	258.203	258.204	200.155
32768	866.909	745.569	459.59	580.13	564.715	762.867	507.858
comp	0.323	7.797	31.944	29.827	8.81	19.177	35.53
# iters	31.0	17.6	19.1	18.9	18.1	19.0	20.8

Table 5: Communication and computation cost for test set 2, the min message size at each level is *Msg_unit* bytes, and the max size is $16 \times Msg\_unit$ bytes.

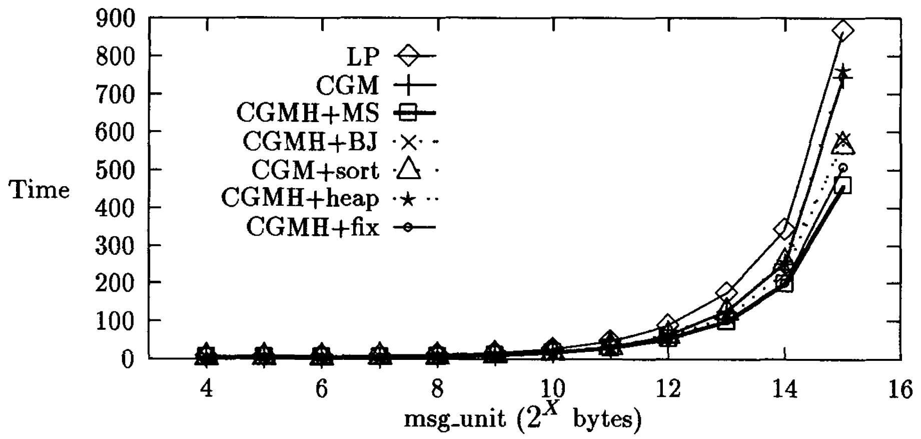

Figure 11: Comparison of communication performance for test set 2

5.3.3 Applications

Table 6 and Figure 12 show the results of 2800-point sample in the third test set, and Table 7 and Figure 13 show the results of 53961-point sample. The results of both samples have similar behaviors as the first test set, which reveal that even if the number of messages in each column is non-uniform, our algorithms maintain their characteristics and performance. The CGMHs will be superior when the *msg_unit* becomes large, which in turn means that it is worth the extra effort (of heap and λ) to reduce the variance of message sizes in each permutation. These results also show the comparison of fixed λ and variable λ (differential approach). The observation reveal that both methods have comparable performance. So for static applications (which can be pre-run to find the best value of λ), the fine tuned, fixed λ may be as good as (or even better than) the dynamic λs found during the scheduling. One can potentially run the algorithms for different values of λ in parallel and choose the best one. However, it is difficult to estimate the actual performance (with varying λ) and choose the best value of λ.

It is hard to make generalizations on which algorithms are better based on the limited number of experimental results presented above. In general, from the computation cost point of view, the comparison of these algorithms is

$$cost(LP) \leq cost(CGM) \leq cost(CGM{+}sort) \leq cost(CGM{+}heap) \leq cost(CGMHs)$$

and from the communication cost point of view, the comparison is

$$cost(CGMHs) \leq cost(CMG + sort) \leq cost(CGM) \leq cost(LP)$$

but, we should mention that when the communication matrix becomes total (or nearly total) density, the LP will have the same communication performance with much smaller amount of scheduling overhead.

Clearly, depending on the structure of communication matrix and the number of times a particular schedule is used, one method may be superior to another. However, in general if the number of times the same schedule is utilized is large, CGMH (with varied λ) seems to be a better approach (specially if the scheduling has to be performed at runtime).

*Msg_* *unit*	LP	CGM	CGMH +MS	CGMH +BJ	CGM +sort	CGM +heap	CGMH +fixed
comm							
16	9.366	5.146	4.843	4.956	5.195	4.785	4.808
64	17.669	9.797	9.127	9.099	9.843	9.091	9.049
256	49.685	29.524	25.151	25.475	29.133	25.673	24.936
1024	181.931	110.56	90.939	94.887	110.904	95.759	90.599
4096	1192.89	719.717	421.729	431.597	735.459	387.859	367.001
8192	2850.1	1459.54	1219.31	1247.11	1404.68	1254.11	1180.08
comp	0.308	6.918	33.327	27.996	6.785	19.922	34.263
# iters	31.0	15.4	17.4	15.6	15.3	15.9	18.0

Table 6: Communication and computation cost for test set 3: 2800-point, the min message size at each level is $2 \times Msg\_unit$ bytes, and the max size is $36 \times Msg\_unit$ bytes.

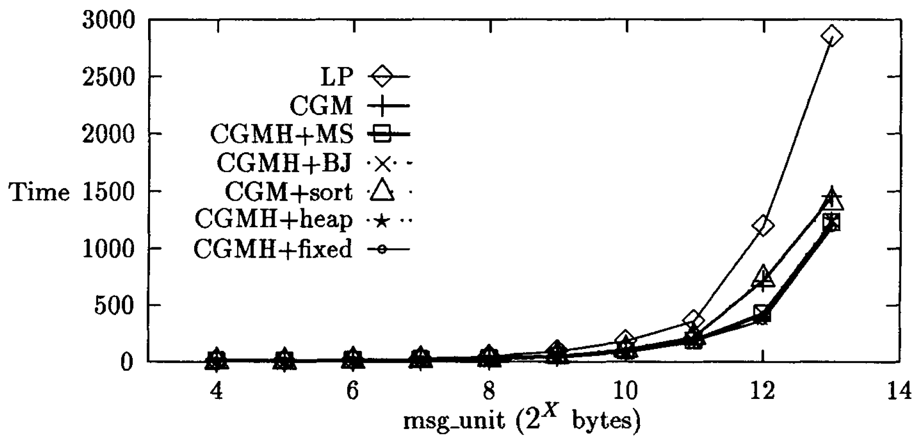

Figure 12: Comparison of communication performance for test set 3: 2800-point

Msg_ unit	LP	CGM	CGMH +MS	CGMH +BJ	CGM +sort	CGM +heap	CGMH +fixed
comm							
16	26.733	18.127	17.151	16.319	17.462	15.911	15.96
64	86.02	60.72	51.722	51.859	57.827	51.033	50.062
256	340.269	238.227	196.03	197.183	225.356	201.373	194.733
1024	1841.66	1675.2	1219.57	1262.86	1549.04	1243.48	1226.96
4096	9127.03	6844.15	5624.51	5683.95	6334.7	5687.45	5185.75
8192	16656.6	11787.9	10484.9	10223.6	11429	10215	10013.3
comp	0.333	8.48	51.574	37.633	8.492	27.325	34.738
# iters	31.0	18.2	25.3	19.4	18.2	18.2	18.0

Table 7: Communication and computation cost for test set 3: 53961-point, the min message size at each level is *Msg_unit* bytes, and the max size is 276 $\times$ *Msg_unit* bytes.

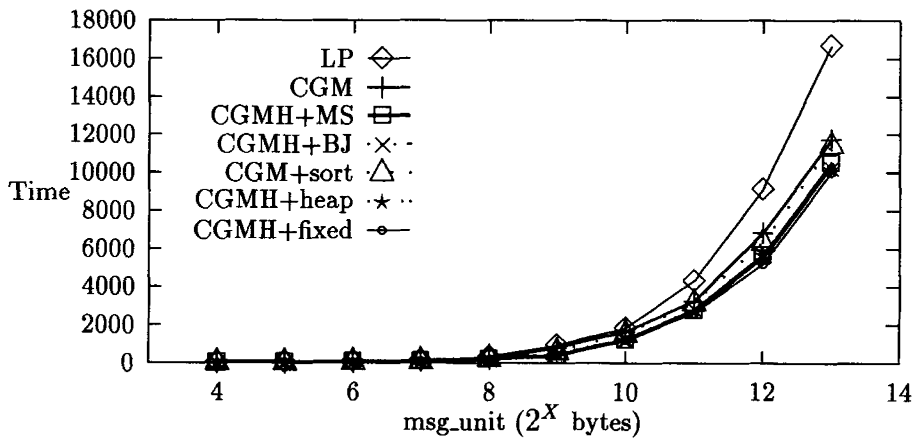

Figure 13: Comparison of communication performance for test set 3: 53961-point

6 Conclusion

In this paper, we developed algorithms to perform non-uniform message routing for all-to-many personalized communication. The linear permutation algorithm is very straightforward, it introduces little computation overhead, but this algorithm requires the use of dummy messages which wastes the communication bandwidth. The worst case complexity of this algorithm is $O(n(\tau + \varphi M))$ (The experimental results for the 32 nodes CM-5 show a complexity of $O(n\tau + C_1 dM\varphi)$, where every node sends d messages and $C_1 > 1$).

The asynchronous communication algorithm eliminates all dummy messages, but its performance will depend on the network congestion and contention on which it is performed. The complexity of this algorithm is machine dependent and may vary from machine to machine. Further, the memory requirements of this algorithm is large. Currently the asynchronous message passing is not available on the CM-5[2].

The CGMH algorithm is found to be very useful in handling non-uniform messages, the heap structure is used to maintain messages in each node in sorted order such that the bigger messages will be scheduled first when possible, and the λ concept is to decrease the variance of message size in one permutation. Because the biggest messages in one permutation has a major impact at the communication time, it is better if the messages in one permutation have approximately equal size. We propose three approaches to decide the value λ, the first two require prerunning of the applications to find the best value of λ, while the third one uses dynamic approach which is executed during the scheduling. The experimental results have shown that our algorithms perform well with samples which are randomly generated with density d and non-uniform message sizes. They also perform well with real applications that have non-even density d in each node.

Another advantage of our algorithms as compared to the other algorithms is the fact that once the schedule is completed, communication can potentially be overlapped with computation, *i.e.* computation on a packet received in previous phase can be

[2]We would add such a comparison in the final version of this paper if the asynchronous communication in CMMD is available at that time.

carried out while the communication of the current phase is being carried. It is also worth noting that due to the compaction, nearly all processors receive data packets (of nearly equal size). Thus, the load is nearly balanced on every node. Clearly, the number of computations phases would increase (from d to $\frac{d}{\lambda} + \frac{\log d}{\lambda^2}$). Thus, using overlap of communication and computation would only be useful if the overlap is more than the extra computation overhead (of the scheduling algorithm).

Our analysis is based on the assumption that each node sends d messages and receive d messages. These algorithms can be extended to the case when the number of messages to be sent by each processor are not equal. Clearly if d is the maximum number of messages to be sent, our CGMH algorithm should produce an expected number of $\frac{d}{\lambda} + \frac{\log d}{\lambda^2}$ permutations. In such case, we believe that our algorithm, on an average, would produce less number of permutations than the case when all processors need to send d messages. Since the number of permutations cannot be lower than $\frac{d}{\lambda}$, our algorithm would produce near optimal number of permutations.

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, [13, 17] are a few of them (The author is referred to [13] for a complete list). After a particular threshold any improvement in partitioning is expensive. For problems which require runtime partitioning, it is critical that this partitioning be completed extremely fast. For such problems, the gains provided by effective communication scheduling may far outperform the gains by spending the same amount of time on achieving a better partitioning.

This paper is restricted to the algorithms that will avoid node contention. We are currently investigating methods which are useful in avoiding link contention.

Appendix A: Procedure for Compressing *COM*

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

Appendix B: Proof of Theorem 2

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

$$\begin{aligned} S &= 1^3 + 1 + \cdots + 1 + (1 - (\frac{d}{n})^d) + (1 - (\frac{d+1}{n})^d) + \cdots + (1 - (\frac{n-1}{n})^d) \\ &= n - \frac{1}{n^d}\sum_{i=d}^{n-1} i^d \\ &\geq n - \frac{1}{n^d}\int_d^n x^d dx \\ &= n - \frac{n}{d+1} + \frac{d}{d+1}(\frac{d}{n})^d \\ &\geq n - \frac{n}{d+1} \end{aligned}$$

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

[3]there are d columns which would find an available entry with probability 1

Appendix C

Assume Y_i be the number of useful entries remained at each column after one iteration. Then

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

When sum all of these statements together, we have

$$\begin{aligned}
Y_m &= d - m\lambda + \left(\frac{1}{Y_0+1} + \frac{1}{Y_1+1} + \cdots + \frac{1}{Y_{m-1}+1}\right) \\
Y_m &\leq d - m\lambda + \frac{m}{Y_m+1}
\end{aligned}$$

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

$$\begin{aligned}
\frac{d}{2} &\leq d - m\lambda + \frac{m}{\frac{d}{2}+1} \\
m &\leq \frac{d}{2\lambda}\left(\frac{1}{1-\frac{1}{(1+\frac{d}{2})\lambda}}\right)
\end{aligned}$$

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

$$\begin{aligned}
m &\leq \frac{d}{2\lambda}\left(1 + \frac{1}{(1+\frac{d}{2})\lambda}\right) \\
&= \frac{d}{2\lambda} + \frac{d}{(d+2)\lambda^2}
\end{aligned}$$

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

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

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

Appendix D: *COM* Random Generator

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

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

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

Acknowledgments

We wish to thank Raja Das, Joel Saltz, and Dimitri Mavriplis at ICASE and Nashat Mansour for the illustration depicted in Figure 7 and the corresponding communication matrices.

References

[1] I. Angus, G. Fox, J. Kim, and D. Walker. *Solving Problems on Concurrent*

Processors, volume 2. Prentice Hall, Englewood Cliffs, NJ, 1990.

[2] Zeki Bozkus, Sanjay Ranka, and Geoffrey C. Fox. Benchmarking the cm-5 multicomputer. Technical Report SCCS-257, Syracuse University, March 1992.

[3] Alok Choudhary, Geoffrey C. Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Sanjay Ranka, and Chau-Wen Tseng. Compiling fortran 77d and 90d for mimd distributed-memory machines. In *Proceedings of the Frontiers of Massively Parallel Computation*, 1992.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 7. The MIT Press, Cambridge, MA, 1991.

[5] Willian J. Dally and Chuck L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. on Computers*, 36(5):547, 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] Geoffrey C. Fox. The architecture of problems and portable parallel software systems. Technical Report Revised SCCS-78b, Syracuse University, July 1991.

[8] Manish Gupta and Prithviraj Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[9] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler support for machine-independent parallel programming in fortran d. Technical Report Rice COMP TR91-149, Rice University, March 1991.

[10] S. Lennart Johnsson and Ching-Tien Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. on Computers*, 38(9):1249–1268, September 1989.

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

[12] Jingke Li and Marina Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[13] Nashat Mansour and Geoffrey C. Fox. Parallel genetic algorithms with application to load balancing for parallel computing. In *Supercomputing Symposium 1992*, Montreal, Québec, Canada, June 7-10 1992.

[14] R. Mirchandaney, J.H. Saltz, R.M. Smith, D.M. Nicol, and Kay Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 140–152, St. Malo, France, July 1988.

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

[16] Sanjay Ranka, Jhy-Chun Wang, and Geoffrey C. Fox. Static and runtime algorithms for all-to-many personalized communications on permutation networks. Technical Report SU-CIS-92, Syracuse University, June 1992.

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

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