Static and Runtime Algorithms for All-to-Many Personalized Communication on Permutation Networks

Sanjay Ranka
Jhy-Chun Wang
Geoffrey Fox

CRPC-TR94501
June, 1994

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Static and Runtime Algorithms for All-to-Many Personalized Communication on Permutation Networks[1]

Sanjay Ranka, Jhy-Chun Wang and Geoffrey Fox
4-116 Center for Science and Technology
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244

[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

With the advent of new routing methods, the distance to which a message is sent is becoming relatively less and less important. Thus, assuming no link contention, permutation seems to be an efficient collective communication primitive. In this paper we present several algorithms for decomposing all-to-many personalized communication into a set of disjoint partial permutations. We discuss several algorithms and study their effectiveness from the view of static scheduling as well as runtime scheduling. An approximate analysis shows that with n processors and assuming that every processor sends and receives d messages to random destinations, our algorithm can perform the scheduling in $O(dn \ln d)$ time on an average, and use an expected number of $d + \log d$ partial permutations to carry out the communication. We present experimental results of our algorithms on the CM-5.

Index Terms: Loosely synchronous communication, permutation networks, personalized communications, runtime scheduling, SPMD, static scheduling.

1 Introduction

In parallel computing, it is important to map the program such that the total execution time is minimized. Experience with parallel computing has shown that a "good" mapping is a critical part of executing a program on such computers. This mapping can typically be performed statically or dynamically. For most regular and synchronous problems [10], this mapping can be performed at the time of compilation by giving directives in the language to decompose the data and its corresponding computations (based on the owner computes rule where each processor computes only values of data it owns [5, 17, 21]). This ordinarily results in regular collective communication between processors. Many such primitives have been developed in [1, 16]. Load balancing and reduction of communication are two important issues for achieving a good mapping. The directives of Fortran D [6] can be used to provide such a mapping for a large class of regular and synchronous problems.

For some other classes of problems [3, 19, 20] that are irregular in nature, achieving a good mapping is considerably more difficult [7]. Further, the nature of this irregularity may not be known at the time of compilation and can be ascertained only at runtime. The handling of irregular problems requires the use of runtime information to optimize communication and load balancing [9, 13, 14]. These packages derive necessary communication information based on the data required for performing local computations and data partitioning. Typically, the same schedule is used a large number of times. Communication optimization is therefore 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 an $n \times n$ communication matrix COM. $COM(i,j)$ is equal to 1 if processor P_i needs to send a message to P_j, $0 \leq i,j \leq n-1$. Our algorithms decompose the communication matrix COM into a set of disjoint partial permutations, $pm_1, pm_2, \cdots, pm_l$, such that if $COM(i,j) = 1$, then there exists a unique k, $1 \leq k \leq l$, that $pm_k(i) = j$.

With the advent of new routing methods [8, 15, 18], the distance to which a message is sent is becoming relatively less and less important [2]. Thus, assuming no link contention, permutation is an efficient collective communication primitive. Permutation also has the useful property that every processor both sends and receives 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 [4]. Thus, if a particular processor receives more than one message or has to send out more than one message in one phase, then the

time will be lower bounded by the time required to remove messages from the network by the processor receiving the maximum amount of data.

Assuming that each of the n processors sends out at most d messages and receives at most d messages, we perform an approximate probabilistic analysis and show that the complexity of the algorithm is $O(nd \ln d)$ on an average. Assuming that the cost of completing one permutation is of $O(\tau + \varphi M)$, where τ is the communication set up time and φ is the transmission time per byte, the minimum time required for communication is of the $O(d(\tau + M\varphi))$. Thus the cost of the scheduling algorithm as compared to the cost of communication is negligible if $M \gg n \ln d$. If the number of times the same communication schedule is used is large (which happens for a large class of problems [6]), the fractional cost of the scheduling algorithm is quite small. Further, the average number of permutations generated is approximately $d + \log d$. Thus, on an average, the fraction of extra permutations generated is not very high. Compared to a naive algorithm for communication of messages for a sparse communication matrix that takes time proportional to n permutations, this algorithm has significant speedup. On a 32-node CM-5, our experimental results show that the cost of scheduling is no more than the cost of communication for small messages (16 bytes). For large messages (4K bytes or larger sizes), the cost is less than one-quarter of the total time for communication. For many applications, the same schedule is utilized repeatedly [6], thus our algorithms would also be useful for many applications for which the communication structure can be derived only at runtime.

The rest of this paper is organized as follows. Notations and assumptions are given in Section 2. Section 3 presents scheduling algorithms and their time complexity analysis. Section 4 provides an improved version of our algorithm and its time complexity analysis. Section 5 presents the experimental results. Finally, conclusions are given in Section 6.

2 Preliminary

The communication matrix COM is an $n \times n$ matrix where n is the number of processors. $COM(i, j)$ is equal to 1 if processor P_i needs to send a message 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 processors of outgoing messages. Column i of COM represents the receiving vector, $recvl_i$, of processor P_i, which contains information about the source processors of incoming messages. The entry $sendl_i(j)$ ($recvl_i(j)$) represents the j^{th} entry in the vector $sendl_i$ ($recvl_i$). Assuming $COM(i, j) = 1$, then $sendl_i(j) = recvl_j(i) = 1$. We will use $sendl$ and $recvl$ to represent each processor's sending vector and

receiving vector when there is no ambiguity.

2.1 Notations and Assumptions

We categorize the routing algorithms in several different categories:

1. *Uniformity of message*—Uniform messages mean all messages are of equal size. In this paper we assume that all messages are approximately of the same size.

2. *Density of communication matrix*—If the communication matrix is 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.

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

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, and φ represents the transmission time per byte (i.e., $1/\varphi$ is the bandwidth of the communication channels).

2. Each processor can send only one message and receive only one message at a time.

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

2.2 Cost of Random Permutations on CM-5

The algorithms described in this paper do not take link contention into account. Principally because the routing is randomized on the CM-5 and it is not possible to statically schedule messages in such a fashion that link contention can be avoided, although randomization alleviates that problem to a large extent. On a 32-node CM-5, we generated 5000 random permutations in which each processor sends and receives a message of 1K bytes. Over 99.5% (4979 out of 5000) of the permutations were within 5% of the average cost (the average communication cost over these 5000 random permutations is 0.543 milliseconds) (Figure 1). Thus, the variation of time required for different random permutations (in which each node sends a data to a random, but different node) is very small on a 32-node CM-5. Observations

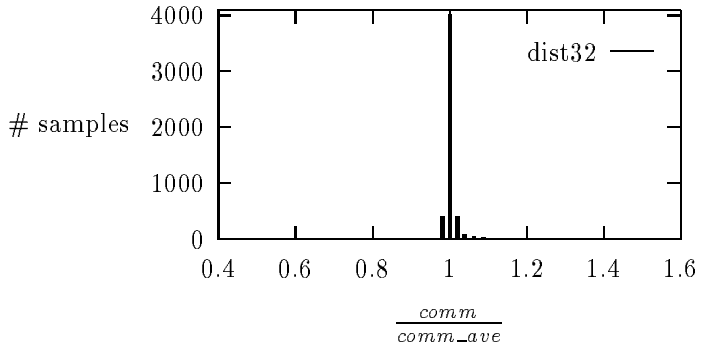


Figure 1: Communication cost distribution for 5000 permutation samples with message of length 1K bytes on a 32-node CM-5.

reveal that the performance of our algorithms, which use permutation as the underlying communication scheme, are not significantly affected by a given sequence of permutation instances. The bandwidth achieved for these permutations is approximately 4M bytes/sec, which is close to the peak bandwidth of 5M bytes/sec provided by the underlying hardware for long distance messages.

3 Scheduling Algorithms

In this paper we assume that each processor has an identical communication matrix COM. The communication matrix COM is a sparse matrix, i.e., each processor will send and receive d messages (in a system with n processors, $d \leq n$). In case only the vector *sendl* is available at every node, the communication matrix COM can be generated by using a concatenate operation. For architectures like the CM-5, performing a concatenate operation is efficient and can be completed in $O(dn)$ amount of time [4]. These operations have efficient implementation on other architectures such as hypercubes and meshes.

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 unique message to other processors [12]. We also assume that COM is a uniform communication pattern, i.e., all messages are of equal size. We are currently developing methods for the case when messages are non-uniform.

Asynchronous_Send_Receive()
For all processors P_i, $0 \leq i \leq n-1$, *in parallel do*

1. Allocate buffers and post requests for incoming messages;
2. Send out all outgoing messages to other processors;
3. Check and confirm incoming messages from other processors.

Figure 2: Asynchronous Communication Algorithm.

We propose several scheduling algorithms, and the analysis of their time complexity in following subsections. All the algorithms proposed in this paper are executed in SPMD (single-program multi-data) mode, i.e., every processor has the same copy of a program, but each processor runs its program in an asynchronous pattern.

3.1 Asynchronous Communication (AC)

The most straightforward approach is to use asynchronous communication. The algorithm is divided into three phases:

1. Each processor first posts requests for expected incoming messages (this operation will pre-allocate buffers for those messages).
2. Each processor sends all of its outgoing messages to other processors.
3. Each processor checks and confirms incoming messages (some of which may already have arrived at their receiving buffer(s)) from other processors.

During the send-receive process the sending processor need not wait for a completion signal from the receiving processor, but can keep sending outgoing messages until they are all done. This naive approach is expected to perform well when density d is small. The asynchronous algorithm is given in Figure 2. Similar schemes were proposed in several parallel compiler projects [11, 13].

In the worst case the time complexity of this algorithm is difficult to analyze, as it will depend on the network congestion and contention on which it is performed. Further, each processor may have only limited space of message buffer. When the buffer is fully occupied by unconsumed messages, further messages will be blocked at the sending processors'

Linear_Permutation()

For all processors P_i, $0 \leq i \leq n-1$, *in parallel do*

for $k = 1$ *to* $n-1$ *do*

$j = i \oplus k$;
if $COM(i,j) > 0$ *then* P_i sends a message to P_j;
if $COM(j,i) > 0$ *then* P_i receives a message from P_j;

endfor

Figure 3: Linear permutation algorithm.

side. The overflow will block processors from doing further processing (including receiving messages) because processors are waiting for other processors to consume and empty their buffers to receive new incoming messages. This situation may never resolve and a deadlock may occur among processors. In order to avoid a deadlock, one needs to monitor the *production/consumption* rate very carefully to guarantee the completion of communication. In case the system buffer is too small to hold all messages at one time, one needs to introduce a *strip mining* scheme [11] to perform sends and receives alternately such that there are a smaller number of unreceived messages accumulated in the buffer and an overflow will not occur.

3.2 Linear Permutation (LP)

In this algorithm (Figure 3), 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_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 overhead 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 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.

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

Global_Masking()

For all processors P_i, $0 \leq i \leq n-1$, *in parallel do*

Repeat

1. Set all entries of vectors $sendl$ and $recvl$ to -1;
2. $x = random(0..n-1)$;
3. *for* $k = 1$ *to* n *do*
   (a) Along row x of COM, try to find an entry $COM(x,y) = 1$ that satisfies $recvl(y) = -1$. If such a y exists, then set $sendl(x) = y$ and $recvl(y) = x$, also set $COM(x,y) = 0$.;
   (b) $x = (x+1)\ mod\ n$;

   endfor
4. *if* $sendl(i) \geq 0$ *then* P_i sends a message to $P_{sendl(i)}$;
   if $recvl(i) \geq 0$ *then* P_i receives a message from $P_{recvl(i)}$;

Until all messages are sent/received.

Figure 4: Global Masking Algorithm.

3.3 Global Masking (GM)

A high-level description of this algorithm is given in Figure 4. At each iteration we first set all entries of vectors $sendl$ and $recvl$ to -1. Then within each row x of COM, $0 \leq x \leq n-1$, we try to find a column y, $0 \leq y \leq n-1$, with $COM(x,y) = 1$ and $recvl(y) = -1$, if such a y exists, then set $sendl(x) = y$ and $recvl(y) = x$. Processors then send/receive messages according to vectors $sendl$ and $recvl$. This procedure is repeated until all messages are sent/received.

As mentioned in the previous section, we assume the communication matrix COM is a sparse matrix and each processor sends out d messages to d different processors. Further, we assume that each processor receives approximately d messages. Clearly, the number of permutations would be lower bounded by the maximum number of messages received by any processor. In this algorithm, the number of iterations, ξ, needed to complete the message routing

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

Figure 5: Compressing procedure.

is bounded by $d \leq \xi \leq U$, where $U = \max\{\text{the number of messages received by each processor}\}$. Because each iteration will take $O(n^2 + \tau + \varphi M)$ time to complete, the total time complexity of this algorithm is $O(\xi(n^2 + \tau + \varphi M))$.

As compared to the permutation algorithm presented in the previous subsection, the global masking algorithm takes fewer iterations to complete the message routing, but it takes extra time to schedule communication. If $n^2 \ll \tau + \varphi M$, i.e., the message size is large compared to the number of processors, the global masking algorithm may outperform the linear permutation algorithm.

4 Enhanced Scheduling Algorithm

In the global masking algorithm described in the previous section, when looking for an entry with $COM(i, j) = 1$ along row i, we may first visit several entries with $COM(i, k) = 0$, where $0 \leq k < j$, before reaching column j. The visits to useless entries should be avoided to minimize unnecessary computation overhead. Having this in mind, we present an enhanced version of the global masking algorithm—compact global masking algorithm (CGM). The scheme can be used to eliminate undesired computations by copying all useful COM entries to an $n \times d$ matrix $CCOM$ (Figure 5).

The vector prt is used as a pointer whose elements point to the maximum number of positive columns in each row of $CCOM$. Also, the reason for performing $Random\_Swap(CCOM)$

is to perturb the sorted order in each row so that the expected number of collisions (i.e., within one iteration, the entries along a column k are repeatedly chosen and tested, but eventually only one entry is selected and other tests are fruitless) can be reduced. If we perform this compression statically, the time complexity will be $O(n(n+d)) = O(n^2)$. Further, this operation can be performed at runtime: each processor compacts one row, and then all processors participate in a concatenate operation that will combine all rows into an $n \times d$ matrix. The cost of this parallel scheme is $O(n+d+dn) = O(dn)$, assuming that the concatenate can be completed in $O(dn)$ time, which was shown to be true for CM-5 [4].

We assume that $CCOM(i,j) = -1$ if this entry doesn't contain active information. After the copy procedure, the first d columns of each row will contain active entries. When searching for an available entry along row i, the first column j with $CCOM(i,j) = k$ and $recvl(k) = -1$ will be chosen. We then set $sendl(i) = k$ and $recvl(k) = i$. In order to avoid any unnecessary travel through useless holes (entries), we will move entry $CCOM(i,l)$ to $CCOM(i,j)$ and reset $CCOM(i,l) = -1$, where $l = prt(i)$. With this "compact" approach, the first several columns in each row contain no useless entries and one will eliminate any unnecessary visits to inactive entries in following iterations. The worst case time complexity to form a routing schedule in this algorithm is $O(dn)$, comparing to $O(n^2)$ in the GM algorithm. The compact global masking algorithm is described in Figure 6.

Step 1 takes $O(n^2)$ time to complete in a sequential program, but we can parallelize this step: each processor creates one row of $CCOM$, then all processors participate in concatenating the result together. The time complexity of this parallel version is $O(n) + O(dn) = O(dn)$. Steps 2a, 2b, and 2d take $O(n)$ time, $O(1)$ time, and $O(\tau + \varphi M)$ time, respectively. We are interested in evaluating the average time complexity of Step 2c and the average number of iterations to complete Step 2.

We make the following assumptions to get an insight of the average complexity of the CGM algorithm. Wherever possible, we support these assumptions by simulation results.

1. At the beginning of each outer loop (Step 2 of Figure 6), the number of active entries, d, in each row of $CCOM$ is approximately equal and the destinations to which each node will send data are random (between P_0 and P_{n-1}).

2. Different stages are assumed to act independently of each other. Each stage starts with the number of messages in each node equal to the average number of messages left in each node by the previous stage.

Assuming at Step 2c, the probability, $Prob_k$, of finding a available entry in row k is

$$Prob_0 = \frac{n}{n}$$

Compact_Global_Masking()

1. Use the $n \times n$ matrix COM to create an $n \times d$ matrix $CCOM$, also generate a vector prt;

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

   (a) Set all entries of vectors $sendl$ and $recvl$ to -1;

   (b) $x = random(0..n-1)$;

   (c) $for \;\; k = 1 \;\; to \;\; n \;\; do$

      i. Along row x of $CCOM$, try to find an entry $CCOM(x,z) = y$ that satisfies $y > -1$ and $recvl(y) = -1$.

      ii. If such a z exists, then set $sendl(x) = y$ and $recvl(y) = x$. Also set $CCOM(x,z) = CCOM(x, prt(x))$, $CCOM(x, prt(x)) = -1$, and $prt(x) = prt(x) - 1$.;

      iii. $x = (x+1) \;\; mod \;\; n$;

   $endfor$

   (d) $if \;\; sendl(i) \geq 0 \;\; then \;\; P_i$ sends a message to $P_{sendl(i)}$;
   $if \;\; recvl(i) \geq 0 \;\; then \;\; P_i$ receives a message from $P_{recvl(i)}$;

   Until all messages are sent/received.

Figure 6: Compact Global Masking Algorithm.

$$Prob_1 = \frac{n-1}{n}$$

$$\vdots$$

$$Prob_k = \frac{n-k}{n}$$

$$\vdots$$

$$Prob_{n-1} = \frac{1}{n}$$

and the expected tries to find a available entry in each row is: $T_0 = E(\frac{n}{n})$, $T_1 = E(\frac{n}{n-1})$, $\cdots$, $T_j = E(\frac{n}{n-j})$, $\cdots$, $T_{n-1} = E(\frac{n}{1})$.

Thus the total expected tries in one iteration are

$$\begin{aligned} T &= \min{}^2(T_0, d) + \min(T_1, d) + \cdots + \min(T_{n-1}, d) \\ &= 1 + \frac{n}{n-1} + \cdots + \frac{n}{k} + d + \cdots + d, \text{ where} \frac{n}{k} = d \\ &= n(\frac{1}{k+1} + \frac{1}{k+2} + \cdots + \frac{1}{n-1} + \frac{1}{n}) + kd \\ &= n(\sum_{i=1}^{n} \frac{1}{i} - \sum_{i=1}^{k} \frac{1}{i}) + n \ . \end{aligned}$$

Since

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma + O(\frac{1}{n}) \approx \ln n + \gamma$$

where γ is the Euler's constant. Thus

$$\begin{aligned} T &= n(H_n - H_k) + n \\ &\approx n(\ln n + \gamma - \ln k - \gamma) + n \\ &= n \ln \frac{n}{k} + n \\ &= n \ln d + n \ . \end{aligned} \tag{1}$$

Thus the expected computation cost of one iteration is $O(n \ln d + n)$. 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. In the case when each row has d active entries, the first d rows would always find an available entry, the probability of success

[2]The maximum number of tries in one row should be less than or equal to d, the number of messages that will be sent to other processors.

in finding an available entry in the $(d+1)^{\text{th}}$ row is $1-(\frac{d}{n})^d$ (there are d active entries in each row). The probability of success in finding an available entry in each row is

$$S = \underbrace{1+1+\cdots+1}_{d} + (1-(\frac{d}{n})^d) + (1-(\frac{d+1}{n})^d) + \cdots + (1-(\frac{n-1}{n})^d)$$

$$= n - \frac{1}{n^d}\sum_{i=d}^{n-1} i^d$$

$$\geq n - \frac{1}{n^d}\int_d^n x^d dx$$

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

$$\geq n - \frac{n}{d+1}\ .$$

Thus the expected number of entries $CCOM(i,j)$ consumed in one iteration is at least $n - \frac{n}{d+1}$.

If we denote d^* as the average number of active entries in each row after one iteration of scheduling (assume the original number of entries in each row be d), then

$$d^* = \frac{1}{n}(nd - (n - \frac{n}{d+1}))$$

$$= d - 1 + \frac{1}{d+1}\ . \tag{2}$$

It is difficult to analyze the number of messages in each row at the next step. We use d^* as the new value of d at the next step. This assumption is made for all future steps. Assuming Y_i is the number of useful entries remained at each row after one iteration. Then

$$Y_0 = d$$

$$Y_1 = Y_0 - 1 + \frac{1}{Y_0+1}$$

$$Y_2 = Y_1 - 1 + \frac{1}{Y_1+1}$$

$$\vdots$$

$$Y_m = Y_{m-1} - 1 + \frac{1}{Y_{m-1}+1}\ .$$

When we sum all of these statements together, we have

$$Y_m = d - m + (\frac{1}{Y_0+1} + \frac{1}{Y_1+1} + \cdots + \frac{1}{Y_{m-1}+1})$$

$$Y_m \leq d - m + \frac{m}{Y_m + 1}$$

$$Y_m^2 - (d - m - 1)Y_m - d \leq 0$$

$$Y_m \leq \frac{(d - m - 1) + \sqrt{(d - m - 1)^2 + 4d}}{2} .$$

Let m be the number of iterations required to reduce the average value of d to $\frac{d}{2}$ using the above equation:

$$Y_m \leq \frac{(d - m - 1) + \sqrt{(d - m - 1)^2 + 4d}}{2} \leq \frac{d}{2}$$

$$(d - m - 1) + \sqrt{(d - m - 1)^2 + 4d} \leq d$$

$$d - 2m + 2 \leq 0$$

$$m \geq \frac{d}{2} + 1 . \tag{3}$$

Thus the number of iterations used to reduce Y_m from d to $d/2$ is upper bounded by $\frac{d}{2} + 1$.

The number of iterations needed to complete the entire message routing is given by

$$(\frac{d}{2} + 1) + (\frac{d}{4} + 1) + \cdots + (1 + 1) + 1$$

$$= (\frac{d}{2} + \frac{d}{4} + \cdots + 1) + \log d + 1$$

$$= (d - 1) + \log d + 1$$

$$= d + \log d . \tag{4}$$

With the analysis presented above we find the following about the average time complexity of the compact global masking algorithm:

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

```
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;
    (and/or switch column loc1 with column loc2);
endfor
```

Figure 7: COM random generator.

5 Experimental Results

We have implemented our algorithms on the CM-5. The experiments are focused on evaluating three parameters: (1) the number of permutations to complete the communication; (2) the cost to execute the communication scheduling algorithms; and (3) the cost to carry out the communication. The first two parts have been implemented in a machine-independent fashion, so that the experiments are not restricted by the actual number of processors available. The third part is executed on a 32-node CM-5.

Most of the algorithms we present in this paper are executed in a loosely synchronous fashion. We did not explicitly use global synchronization to enforce synchronization between communication phases in any of the algorithms proposed in this paper.

In our experiments the number of processors, n, ranges from 32 to 1024, and every processor will send and receive d different messages, where $1 \leq d < n$. For each (d, n) combination, we sample 300 different communication matrices COM and record each category's maximum, minimum, and average values. In order to guarantee that in COM every row and every column has approximately d active entries, COM is generated by the algorithm given in Figure 7.

In order to prove that the communication cost on the CM-5 is not sensitive to different permutations, we randomly generate 1000 different permutations and record their communication cost (Table 1). The results show that the maximum and minimum values are within

	16*	64	256	1K	4K	16K	64K	256K
ave	0.211	0.220	0.258	0.422	1.046	3.608	14.013	55.833
max	0.223	0.231	0.265	0.448	1.116	3.951	15.565	62.792
max/ave	1.056	1.046	1.026	1.063	1.067	1.095	1.111	1.125
min	0.208	0.217	0.252	0.403	0.973	3.337	12.900	51.648
min/ave	0.983	0.983	0.977	0.955	0.930	0.925	0.921	0.925

$*$: message size, in bytes.

Table 1: Communication cost for one permutation on a 32-node CM-5.

$\pm 10\%$ of average value for most cases. Thus the performance of our algorithms is not significantly affected by a given permutation instance (i.e., the CM-5 can complete all permutations in nearly the same amount of time).

Tables 2 and 3 give the performance of our algorithms. The results reveal that the GM and CGM algorithms have a superior performance compared to other schemes (but GM employs a much higher scheduling cost). The comparisons in Figure 8 do not include the cost of scheduling, which is negligible compared to the total cost if the sizes of messages are large or the same schedule is used many times. The tables also show the number of permutations generated by each algorithm and their corresponding cost, and they reveal that the CGM algorithm generates the smallest number of permutations in most cases.

Figure 9 shows the fraction of scheduling overhead, *scheduling cost/communication cost*, of the LP, GM, and CGM algorithms. These observations reveal that the LP algorithm has a very small scheduling overhead (but its overall performance is not good enough, especially when d is small). The GM algorithm has a communication cost similar to that of the CGM algorithm, but it has a relatively high scheduling overhead. The CGM algorithm shows a moderate scheduling overhead, and the fraction decreases as the message size increases (assuming the same communication schedule is utilized only once). The cost of scheduling is thus at most equal to the cost of communication for small messages (16 bytes) and negligible for large messages (less than 0.25 for messages of size 4K). 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, our algorithm is also suitable for runtime scheduling.

Table 4 shows the performance of the CGM algorithm. The standard deviations of these results are small (in fact, the maximum and minimum values are within $\pm 10\%$ of the average

value in most cases), which indicates that this algorithm is very stable for a large class of communication patterns. Figure 10 shows the *scheduling time/n* versus $d \ln d$ (for $d \ln d$ less than 150). The experimental results confirm our theoretical analysis of scheduling time complexity (i.e., $O(dn \ln d)$).

5.1 Discussion

From the previous section it is clear that CGM is a better choice than GM. Thus, for the rest of this section, we will compare only the performances of LP and CGM, and discuss their use for different ranges of d and n. In Section 3 we showed that the time complexity for the LP algorithm is $O(n(\tau + M\varphi))$, but in this algorithm many permutations are in fact sending no message. Based on our experimental results a better modeling on the CM-5 is $n\tau + C_1 dM\varphi$, where C_1 is a constant. Also, the time complexity for CGM can be rewritten as $C_2 dn \ln d + C_3 d(\tau + M\varphi)$, where C_2 and C_3 are some constants. We are interested in finding the break-even points for different message sizes where CGM can outperform LP.

$$C_2 dn \ln d + C_3 d(\tau + M\varphi) \leq n\tau + C_1 dM\varphi$$

$$C_2 dn \ln d \leq (n - C_3 d)\tau + (C_1 - C_3) dM\varphi$$

$$d \ln d \leq \frac{(n - C_3 d)\tau}{C_2 n} + \frac{(C_1 - C_3) dM\varphi}{C_2 n} .$$

We first investigate the case where the message size M is small. When M is small, the second term in RHS can be eliminated. Also, the first term in RHS can be reduced to τ / C_2 when d is small. Thus CGM will outperform the LP algorithm when

$$d \ln d \leq \frac{\tau}{C_2} . \tag{5}$$

When the message size M is large, the effect of τ becomes less significant than $M\varphi$, thus

$$C_2 dn \ln d + C_3 d(\tau + M\varphi) \leq n\tau + C_1 dM\varphi$$

$$C_2 dn \ln d \leq (C_1 - C_3) dM\varphi$$

$$\ln d \leq \frac{C_1 - C_3}{C_2} \frac{M\varphi}{n} . \tag{6}$$

The above discussion is based on the assumption that the same schedule is used only once. When the number of times the same schedule is utilized increases, the CGM algorithm would be better for a large range of d.

6 Conclusions

In this paper we have developed algorithms to perform message routing for all-to-many personalized communication. The linear permutation algorithm is very straightforward. It introduces very small computation overhead. The worst case complexity of this algorithm is $O(n(\tau + \varphi M))$ (the experimental results for a 32-node CM-5 show a complexity of $O(n\tau + C_1 dM\varphi)$, where every node sends d messages). The second algorithm, GM, eliminates unnecessary communication at the cost of significant computation overhead. The complexity of this algorithm is $O(\xi(n^2 + \tau + \varphi M))$. When M is relatively large and n and d are small, this algorithm outperforms LP.

The performance of the asynchronous communication algorithm depends on the congestion and contention of the network on which it is performed. This algorithm is machine-dependent and its complexity may vary from machine to machine.

We also present an enhanced version of the GM algorithm—CGM algorithm. In this algorithm we use the information of $COM(i, j)$ to create an $n \times d$ matrix $CCOM$ such that all useful entries appear at the first several columns, and useless entries ($CCOM(i, j) = -1$) are moved to the bottom of each row. We show that with this approach, the time complexity to complete one iteration is $O((n \ln d + n) + (\tau + \varphi M))$, and we need only at most $d + \log d$ iterations to complete the whole message routing.

Another advantage of our algorithm as compared to the other algorithms is that once the schedule is completed, communication can potentially be overlapped with computation, i.e., computation on a packet received in a previous phase can be 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, and the load is nearly balanced on every node. Clearly, the number of computation phases would increase by $\log d$ (from d to $d + \log d$). Thus, using overlap of communication and computation would only be useful if the overlap is more than the extra computation overhead.

This paper assumes 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 CGM algorithm should produce no more than an expected number of $d + \log d$ permutations. In such case, we believe that our algorithm, on an average, would produce lower than $d + \log d$ permutations. Since the number of permutations cannot be lower than d, our algorithm would produce a near optimal number of permutations.

Our paper also assumes that all messages are approximately of the same size. For many

applications, this is not the case. We are currently investigating methods that are useful when the message sizes are not equal.

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] M. Barnett, D.G. Payne, and R. Geijn. Optimal Broadcasting in Mesh-Connected Architectures. Technical report, University of Texas at Austin, December 1991.

[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*, pages pp. 100–107, McLean, VA, October 19-21 1992.

[5] D. Callahan and K. Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *Journal of Supercomputing*, 2:pp. 151–169, October 1988.

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

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

[8] 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.

[9] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed Memory Compiler Methods for Irregular Problems—Data Copy Reuse and Runtime Partitioning. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier, Amsterdam, The Netherlands, 1991.

[10] Geoffrey C. Fox. The Architecture of Problems and Portable Parallel Software Systems. Technical Report Revised SCCS-78b, Syracuse University, July 1991.

[11] 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.

[12] S. Lennart Johnsson and Ching-Tien Ho. Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Trans. on Computers*, 38(9):pp. 1249–1268, September 1989.

[13] Charles Koelbel and Piyush Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):pp. 440–451, October 1991.

[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 pp. 140–152, St. Malo, France, July 1988.

[15] 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.

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

[17] A. Rogers and K. Pingali. Process Decomposition Through Locality of Reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages pp. 69–80, Portland, OR, June 1989.

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

[19] D.W. Walker. Characterizing the Parallel Performance of a Large-Scale, Particle-in-Cell Plasma Simulation Code. *Concurrency: Practice and Experience*, pages pp. 257–288, 1990.

[20] 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.

[21] Hans P. Zima, Heinz-J. Bast, and Michael Gerndt. SUPERB: A Tool for Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, 6:pp. 1–18, 1988.

d	*msg_size*	AC	LP	GM	CGM
4	comm*				
	16	2.110	3.593	1.836	1.855
	32	2.224	3.659	1.853	1.861
	128	2.364	3.829	1.970	1.989
	256	2.729	4.095	2.123	2.141
	1024	4.656	5.734	3.137	3.122
	2048	7.101	7.920	4.346	4.324
	8192	21.936	21.505	11.889	11.863
	16384	41.437	40.364	22.498	22.413
	32768	79.102	76.538	43.742	43.498
	65536	151.997	146.295	84.883	84.523
	comp†	0	0.116	14.608	1.570
	perm‡	-	31.000	5.640	5.540
8	comm				
	16	3.392	5.902	3.420	3.452
	32	3.577	5.989	3.502	3.495
	128	4.202	6.299	3.737	3.729
	256	5.165	6.733	4.041	4.068
	1024	9.573	9.613	5.949	5.897
	2048	15.379	13.275	8.199	8.182
	8192	50.294	36.758	22.377	22.337
	16384	95.294	69.690	42.342	42.106
	32768	179.563	133.827	82.534	82.035
	65536	324.347	260.924	160.129	159.560
	comp	0	0.121	22.062	3.050
	perm	-	31.000	10.260	10.100

$*$: Communication cost, in milliseconds.

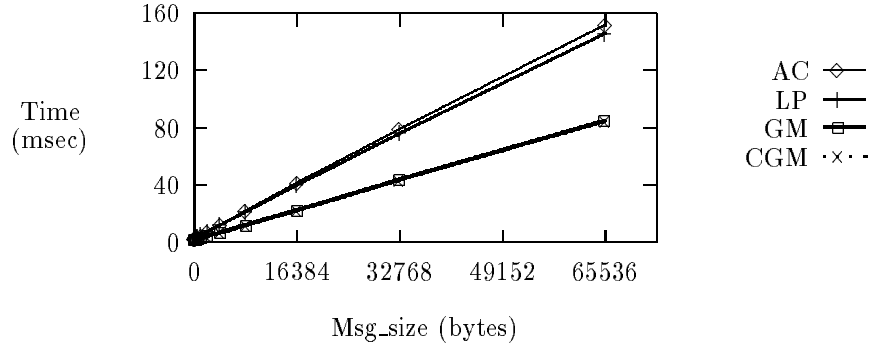
†: Scheduling cost, in milliseconds.

‡: Number of communication phases needed.

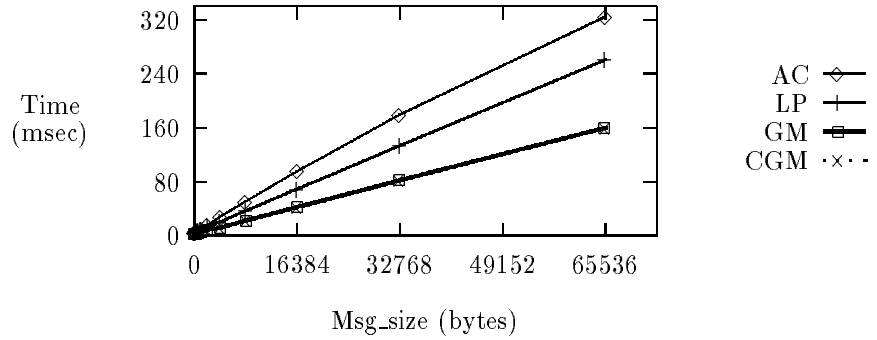
Table 2: Experimental results for different message sizes on a 32-node CM-5.

d	*msg_size*	AC	LP	GM	CGM
16	comm*				
	16	6.304	8.361	6.415	6.551
	32	6.813	8.526	6.540	6.636
	128	8.771	9.038	6.989	7.085
	256	10.927	9.662	7.591	7.720
	1024	21.427	14.181	11.153	11.197
	2048	34.634	19.641	15.404	15.504
	8192	111.244	56.812	42.092	42.301
	16384	205.605	109.885	79.431	79.733
	32768	402.905	214.635	155.073	155.541
	65536	1233.859	426.224	301.124	302.868
	comp	0	0.126	32.984	6.348
	perm	-	31.000	18.580	18.560
24	comm				
	16	10.201	9.617	9.289	9.465
	32	11.085	9.715	9.477	9.596
	128	14.929	10.331	10.163	10.300
	256	18.638	11.081	11.035	11.204
	1024	35.360	16.569	16.206	16.206
	2048	55.855	23.160	22.316	22.431
	8192	174.728	69.231	60.896	61.182
	16384	304.736	135.531	115.038	115.108
	32768	676.008	266.979	224.753	224.882
	65536	2362.268	842.655	438.658	435.817
	comp	0	0.131	42.007	9.547
	perm	-	31.000	26.560	26.600

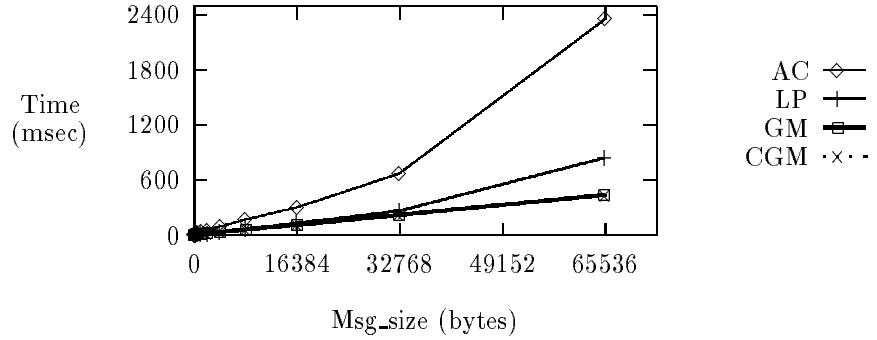
Table 3: Experimental results for different message sizes on a 32-node CM-5.



(density $d = 4$)

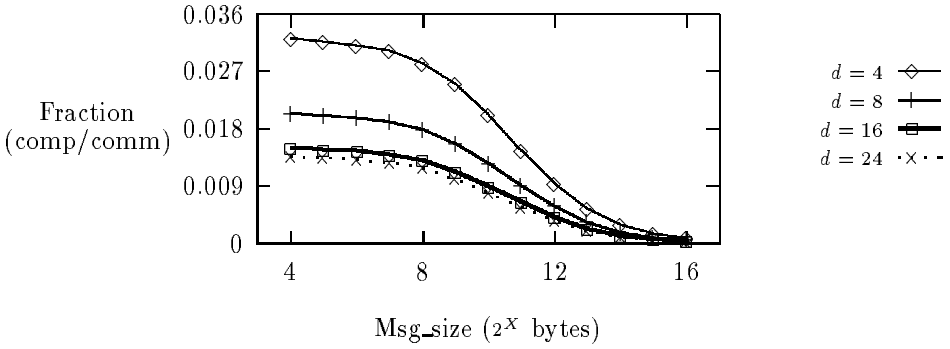


(density $d = 8$)

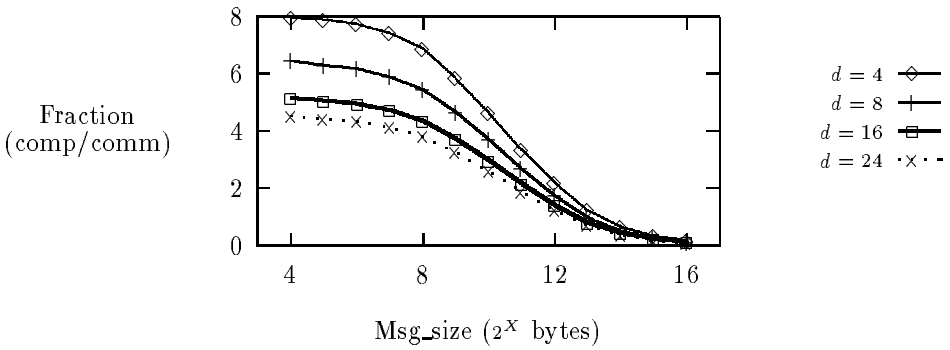


(density $d = 24$)

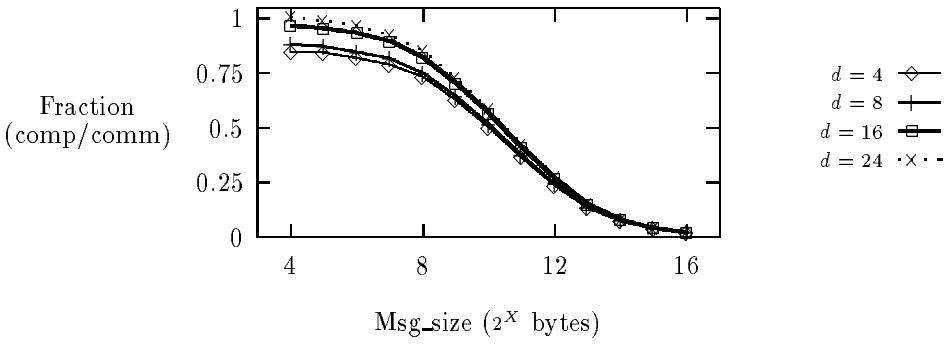
Figure 8: Communication cost for different message sizes on a 32-node CM-5.



(LP algorithm)



(GM algorithm)



(CGM algorithm)

Figure 9: Computation overhead of scheduling algorithms in terms of communication cost on a 32-node CM-5.

d	$d + \log d$	32 PEs		128 PEs		512 PEs	
		perm	comp	perm	comp	perm	comp
1	1	1.0	0.6	1.0	2.8	1.0	11.0
2	3	3.0	1.4	3.0	6.0	3.0	24.1
4	6	5.6	2.6	6.0	11.6	6.1	47.3
8	11	10.2	5.0	10.7	22.4	11.1	92.3
16	20	18.5	9.9	19.5	44.2	20.0	183.3
24	28.6	26.5	14.9	-	-	-	-
31	36	34.2	19.9	-	-	-	-
32	37	-	-	36.3	91.0	37.1	377.5
64	70	-	-	68.8	190.0	70.3	813.6
96	102.6	-	-	100.7	291.7	-	-
127	134	-	-	132.4	394.9	-	-
128	135	-	-	-	-	135.4	1786.0
256	264	-	-	-	-	263.7	3892.9
384	392.6	-	-	-	-	391.2	6068.9
511	520	-	-	-	-	519.0	8306.7

Table 4: The number of permutations generated and scheduling cost (on the CM-5) for different densities (d) and number of processors (n).

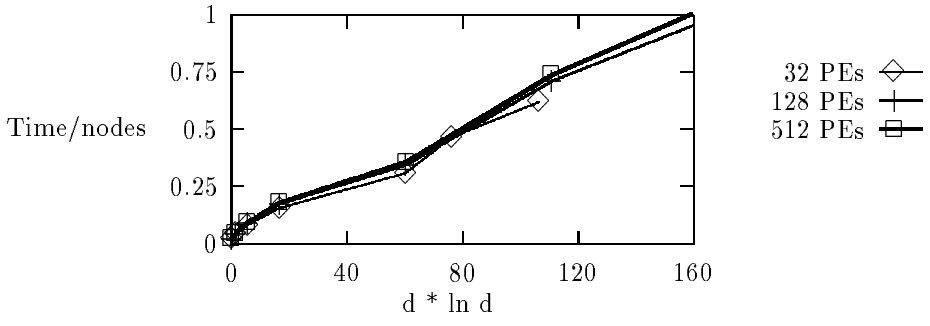


Figure 10: CGM scheduling cost divided by number of nodes versus $d \ln d$.