JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 20, 341–356 (1994)

Practical Algorithms for Online Routing on Fixed and Reconfigurable Meshes*

MARTIN C. HERBORDT,† JAMES C. CORBETT,‡ AND CHARLES C. WEEMS

Department of Computer Science, University of Massachusetts, Amherst, Massachusetts 01003

AND

JOHN SPALDING

Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan 48109

The critical problem in creating practical online SIMD mesh routing algorithms is to minimize both the number of communication steps and the size and complexity of the queues required at each PE (processing element). Currently, the best available algorithms for likely array sizes require $16n$ routing steps with queue size 1; if priority queues of size $2q - 1$ are allowed, the number of routing steps required is reduced to $14n/q + 2n$. We present an algorithm (the MGRA), based on wormhole routing, that has routed a large number of communication patterns (all patterns tried besides a synthetically constructed worst case) in $5n$ routing steps with a FIFO queue of size 2. We also show that the MGRA can be modified for meshes with broadcast buses and reconfigurable broadcast buses to route in a similar number of routing steps but with a queue size of 1. A second algorithm (the CGRA) uses reconfigurable broadcast buses in implementing cut-through routing. Using the CGRA, sparse patterns are routed in a small constant number of communication steps. We prove that the MGRA has bad worst case performance, but also show that a randomizing preprocessing step can improve the predictability of the original result. Finally, we show how performance scales with changing inter- and intra-PE path widths. © 1994 Academic Press, Inc.

1. INTRODUCTION

Mesh-connected array processors [7, 4, 10, 5, and many others] have been successfully applied to a number of domains, the foremost being image processing, matrix operations, and other areas where most of the computations either are local to each processing element (PE), or involve only regular or near-by communication. One fundamental problem with using these processors is the difficulty in routing data in communication patterns that are neither proximate nor regular, a situation that occurs during data dependent computation where the patterns are not known in advance. Such communication requires general online routing.

Much theoretical work has been done on the problem of routing on meshes but none of the results is completely satisfactory for practical online routing on SIMD meshes. The previous algorithms either assume unrealistic hardware (e.g., unit time priority queues) [25, 13]; require substantial preprocessing (offline algorithms) [2]; only route a small subset of possible patterns [20, 22]; or use sorting for preconditioning. These last algorithms are thus a factor of 8 removed from optimal for likely array sizes and queues of size 1, and a factor of 4.5 removed from optimal for priority queues of size 3 [12, 15]. Some recent processor designs such as the Connection Machine CM-2 [24] and the MasPar MP-1 [17] have addressed this problem by adding a dedicated router network; although these machines have been quite successful, they have certain disadvantages such as cost, a tradeoff of relatively slow nearest neighbor moves for improved support of general routing, or lack of support for intermediate combining. Also, communication networks with unbounded fanout such as Clos networks and hypercubes face scalability problems as array sizes increase.

Some of the results presented in this paper are as follows.

- A new online routing algorithm (the mesh greedy routing algorithm or MGRA) based on wormhole routing [6] that has the following characteristics: (1) it requires only hardware available on a SIMD mesh with no local indexing; (2) it has very low overhead as only queues of size 2 need to be emulated; (3) it routes random permutations optimally to within a factor of 1.5 and other permu-

* This work was supported in part by the Defense Advanced Research Projects Agency under Contract DACA76-89-C-0016, monitored by the U.S. Army Engineer Topographic Laboratory; under Contract DAAL02-91-K-0047, monitored by the U.S. Army Harry Diamond Laboratory; and by a CII grant from the National Science Foundation (CDA-8922572). The work of J. C. Corbett was supported by Office of Naval Research Grant N00014-89-J-1064.

† M. C. Herbordt was supported in part by an IBM Fellowship.

‡ J. C. Corbett was supported in part by a University of Massachusetts Graduate School Fellowship. His new address is Information and Computer Science Department, 2565 The Mall; University of Hawaii, Honolulu, Hawaii 96822.

0743-7315/94 $6.00

tations tried optimally to within a factor of 2.5; and (4) it supports intermediate combining.

- The MGRA is modified to take advantage of broadcast buses [10] by reducing the queue size to 1. The decrease in overhead more than offsets the increase in the number of communication steps.
- The MGRA is modified to take advantage of reconfigurable broadcast buses [16, 26]: the queue size is 1 *and* the number of communication steps required is similar to the version with size 2 queues.
- We present another new online routing algorithm for reconfigurable bus networks (the coterie greedy routing algorithm or CGRA), based on cut-through routing [11], that routes sparse permutations in a small constant number of routing steps.
- We prove matching upper and lower bounds for the worst case performance of the MGRA.
- We demonstrate that a randomization preprocessing step can improve the consistency of the MGRA routing performance.
- We use fine grained simulation to show how these algorithms scale as internal and external path widths are increased. We find that the speedup in machine cycles (over bit-wide paths) is asymptotic to 8.

This work is significant in two respects. First, it contains the most practical online routing algorithms (fastest for likely patterns and array sizes) for important classes of processors as represented by the MPP [4], the ICL DAP [10], and the Polymorphic Torus [16]. Second, these results are significant from the standpoint of matching architectures and applications. In some recent processor designs such as the Blitzen [5] and the CAAPP [26], the decision was made to forgo the general routing network: this is because the potential gain with respect to the target applications was thought not to be worth the cost, or because other hardware was considered to be more important. The question has been whether these machines must be restricted to running only regular and window-based operations, or whether effective means can be found to route data through nonuniform, data dependent communication patterns. While the answer is still dependent on individual processor implementations, this work should help users to decide on the most cost-effective machine for their applications, and architects on which features to add to their array processors for a given application.

The remainder of this paper is organized as follows: the models under consideration are presented in Section 2. We follow in Section 3 with a summary of previous work. In Section 4 we present the algorithms. The correctness, freedom from deadlock, and worst case bounds are proven in Sections 5 and 6. In Section 7 we present the experimental results.

2. MODELS UNDER EXAMINATION

In this paper we consider the class of architectures known as SIMD mesh connected arrays, or *meshes* for short. Many different machines of this type have actually been built, or are in the process of being built: among the more recent are the MPP [4], CLIP-4 [7], DAP [10], Polymorphic Torus [16], CAAPP [26], Blitzen [5], and MasPar MP-1 [17].

Our goal is to construct routing algorithms that run well on the existing and proposed machines in this class, but without making our work specific toward any one. However, we do not want to restrict our algorithms by using only features available to a "least common denominator," that is, features available on all machines. Rather, we want to take advantage of as many of the features available on subsets of machines as possible, but not to exclude any other subset. To do this we use a two part strategy.

1. Abstract those features common to all the machines into a *basic model*. Algorithms developed for this model will run on all the machines listed above with at worst a small constant slowdown, e.g., due to a difference in the set-up time in communication or some other minor variations.
2. Abstract those features not universally available to the class. Again, algorithms developed for the *enhanced models* will run on machines having the corresponding features with at worst a small constant slowdown.

The Basic Model

The prototypical SIMD mesh, as exemplified by the MPP, consists of two parts: the controller which broadcasts instructions, constants, and memory addresses, and the $n \times n$ array of N processing elements (PEs). The assumptions we use in the basic SIMD mesh model are as follows.

PE ALU and Memory. Each PE contains an ALU with the capability of performing the basic arithmetic and logical operations. Storage consists of a small number of registers and local memory, some of which is on-chip, some off-chip. The off-chip memory has substantially greater latency (up to a factor of 10). Branching takes place through the use of an activity register: when it is turned off, the PE is inhibited from writing the output of the instruction.

InterPE Communication. Communication takes place between neighboring PEs through a mesh connected network. The nearest-neighbor move instruction has as parameters a memory address (or register) and a direction. We assume (torus) wraparound connections at the edges of the mesh.

Feedback to the Controller. Feedback from array to controller is available through a global OR: a response register from each PE is ORed with the response registers from the other PEs in the array. This feature is critical for data dependent termination of loops and takes only a few cycles.

Enhanced Models

Recent SIMD processors have become more complex: as VLSI component sizes have gotten smaller, new features have been added. Some of those that can optionally be used in the routing algorithms are presented here, together with their costs.

Local Indexing. Because of the cost in VLSI area of implementing address circuitry between PE and on-chip memory, current implementations only allow local indexing when off-chip memory is accessed.

Broadcast Buses. The ability of PEs to transmit data through direct electrical connections over long distances has been included in many architectures, starting perhaps with the ILLIAC-III [18]. One version is row and column broadcast buses: PEs initiate a broadcast by writing to a communication register, and the signal then propagates along either the rows or the columns for some small, predetermined number of cycles. PEs then acquire the signal broadcast on their own row or column bus by reading the communication register. If multiple PEs write to the same bus simultaneously, the OR of those signals will be propagated. The cost of the broadcast instruction is dependent on the implementation and the size of the array. However, when we use broadcast buses for transmitting single bits of handshaking information, we assume a number of cycles less than that of an arithmetic instruction.

Reconfigurable Buses. In this variation, PEs also control switches that, when open, prevent a signal from propagating further down the bus. In this way the broadcast buses can be partitioned. Switches can be loaded like local storage, either from patterns stored in memory, or from data dependent calculations. The cost of broadcast in this model is linear with respect to the distance the signal propagates; however, the constant is very small and the size of each mesh dimension bounded. Experimental evidence on the CAAPP suggests that assuming a propagation of 50 PEs per machine cycle is more than adequate. For simplicity we always assume that the signal is propagating through the entire array: we therefore count a broadcast instruction as about 10 nearest neighbor moves.

3. REVIEW OF ROUTING ON A MESH

By *routing* we mean the selection of paths packets must travel in order to implement communication among PEs. If these paths are selected before the start of the packet transfer, then we are routing *offline*. In *online* routing, decisions of where to send packets next are made locally after the packet has been received by an intermediate PE; therefore the destination address must be carried along in the packet. When online routing algorithms are analyzed, the convention dictates that only packet transfer steps be counted, even though the packet size varies with the size of the network (the destination tag must have $\log N$ bits). The reasoning is that the tag is of similar size to the data portion of the packet for likely array sizes. This assumption is similar to that allowing a memory access to be counted as a unit time operation, even though the circuit depth required to decode an address is proportional to the log of the memory size.

Mesh routing algorithms have been developed for two models: one corresponds to our basic SIMD architecture, the other to a more complex MIMD model. In both models, at most one packet can be transferred from a PE to a neighbor on a communication step. There are two major differences: the first is that in the SIMD model, PEs can only transfer packets in the direction specified by the controller on that communication step, whereas in the MIMD model the direction can be determined by each PE. As a consequence, the trivial lower bound for mesh routing is $2n - 2$ communication steps on the MIMD model and $4n - 4$ communication steps on the SIMD model; these are the minimum numbers of routing steps needed for processors in opposite corners to exchange packets. When the model has wraparound connections, the lower bounds are halved. The second major difference between the SIMD and MIMD models is the complexity of queueing/dequeueing operations. In the MIMD model, unit time priority queues are assumed, while in the SIMD model these queues must be simulated at a cost proportional to the length of the longest queue.

Online MIMD Routing Algorithms

One way to route is to use a simple greedy algorithm: First send each packet along the column to the correct row, then along the row to the correct column. Packets arriving at the correct rows are ordered in the queues so that the ones that need to travel the furthest are given priority. This algorithm takes $2n - 2$ steps with no wraparound, but requires queues of size $\theta(n)$. Leighton has shown, however, that for random permutations the required queue size is no more than 4 with overwhelming probability [14]. The randomized routing algorithm of Valiant and Brebner [25] is an extension of greedy rout-

ing. The algorithm consists of three phases: randomize packets within the columns, send packets to the correct column along the row, and send packets to the correct row along the column. This algorithm results in routing in $\approx 3n$ steps, but the queue size has been reduced to $O(\log N)$ for all permutations with overwhelming probability. Kunde uses sorting as a preconditioner to routing [12]. His algorithm divides the array into $n/q \times n/q$ blocks, sorts those blocks into column major order, and then routes the packets to their destination using the basic greedy routing algorithm. The complexity is thus $2n - 2$ plus the time required to sort the $n/q \times n/q$ blocks. The queues required at each node are of size $2q - 1$. Leighton *et al.* [13] present a more complex algorithm that achieves $2n - 2$ step routing with constant size, though impractically large, queues.

Online SIMD Routing Algorithms

One way to perform online routing on the SIMD model is to emulate the MIMD model and use one of the MIMD algorithms. Recall the two differences between the models, the flexibility in the communication and the complexity of the queueing operation. The packet transfers in different directions on each time step can be emulated with only a factor of 4 slow-down. However, emulating queues is much more costly: either local indexing must be used (a feature that often is not available or is very costly if it is), or else the possible pointer positions must be successively broadcast by the controller at a cost proportional to the size of the longest queue. Clearly, we want to route with the smallest queues possible, eliminating all the MIMD model algorithms but that of Kunde with the minimal queue size.

Leighton has modified Kunde's algorithm slightly to route with queue size of 1 [15]. This algorithm consists of sorting the array into column-major order, routing each packet to the correct column, and finally routing the packets to their destinations. The complexity of this algorithm is $2n - 2$ on a mesh with wraparound ($4n - 4$ with no wraparound) plus the complexity of sorting the array into column-major order. The most practical sorting algorithm to use here (best performance for column-major sorting for likely values of n) is that of Nassimi and Sahni [19]. This algorithm requires $14n$ communication steps. Thus, the entire algorithm requires $16n$ communication steps on a mesh with wraparound, or a factor of 8 times the trivial lower bound.

Offline Routing Algorithms

Since optimal online MIMD routing algorithms exist, only offline SIMD routing algorithms are considered. Offline preprocessing can dramatically improve performance for certain communication patterns. For example, with $O(\log^2 N)$ preprocessing steps, optimal routes can be found for the class of permutations specifiable by permuting and complementing the bits in the PE ID [20] for meshes with no wraparound. Strong presents an efficient algorithm for image rotation that requires little preprocessing [22]. Raghavendra and Prasanna Kumar [21] give algorithms to route various permutations optimally in meshes with wraparound, and also prove that there exist offline algorithms to route any permutation in $3n$ steps. One such algorithm was developed by Annexstein and Baumslag [2]. This algorithm requires $O(N)$ preprocessing time, however.

Summary

The MIMD algorithms are optimal, but those assuming unit time priority queues are very costly to emulate on a SIMD model. The online SIMD algorithms are all at least as slow as sorting. The offline SIMD algorithms either are not general, or require $O(N)$ preprocessing.

4. PRACTICAL ROUTING ALGORITHMS

Before presenting the algorithms, we detail some requirements and constraints that go along with the domain of general online routing on SIMD meshes.

General Online Routing

Since it is impossible to precompute and store any reasonable fraction of the N^N possible communication patterns and too time consuming in general to compute them before the transfer, the data packet must contain a header with address information.

SIMD Control

A consequence of SIMD control is that communication is treated as a synchronous instruction (such as the Connection Machine SEND [24]) where the arguments are a pair of arrays mapped to the PE grid: the destination address, and the data. The data often consist of a single word. Another consequence of SIMD control is that PEs not involved in communication cannot perform unrelated instructions. Therefore, all PEs are occupied until the last packet reaches its destination. And finally, the complexity of emulating queues with no local indexing is proportional to the size of the queue; therefore the queue size must be very small.

Mesh Topology

In meshes, PEs need to process a substantial amount of address information in order to decide where to send packets. Unlike self-routing in a butterfly network where only a single address bit is needed to compute the next address, the PE must read the entire row or column index

to decide what to do. Since the log N bits needed to encode the address is much larger than the width of the communication links between processors, bit-serial routing as in [1] is not advantageous for this model. We show elsewhere that storing and forwarding entire packets is preferable to any kind of packet splitting for data sizes less than 4 log N and relatively sparse communication patterns [9].

Two of the above requirements are seemingly contradictory: we need to transfer entire packets from PE to PE as in store-and-forward routing, but can only support a very small queue size. The algorithms presented below are based on a combination of store and forward and wormhole routing [6]: because emulating queues is so costly, PEs are not allocated enough queue space to store all packets that could collide there. And although packets are not strung out in a series of flits, the overall behavior is similar to that in wormhole routing: trains of data proceed through the network until the head of the train is blocked; at that point the entire train must wait until the path is clear.

4.1. Outline of the Mesh Greedy Routing Algorithm

The basic algorithm, which we call the *mesh greedy routing algorithm* or MGRA, runs as follows. Every PE emulates a local section of two communication channels, X and Y, by using the nearest neighbor mesh and space allocated in on-chip memory. The X-channel and Y-channel are arbitrarily chosen to run in directions parallel to the rows and columns respectively. Conceptually, the algorithm runs as follows: PEs inject packets into the network, which are sent through the X-channel a distance of one PE per routing step until the correct X coordinate (column) is reached. At this point the packet is moved from the X-channel to the Y-channel. The packet then travels through the Y-channel until the destination is reached. The X- and Y-moves are interleaved so that each occurs during every iteration. Packets travel in only one direction in each channel and wraparound is used; because the packets have only unit length (are made up of single flits), having single X- and Y-channels does not cause deadlock. If the packet has reached the correct X coordinate but the section of the Y-channel being emulated at that PE is occupied, then the packet is "blocked," as are all the other packets contiguously behind that packet in the X-channel. Y-channels are never blocked, so overall progress is assured.

Initially, each iteration contains two data movement instructions, one in the X-channels and one in the Y-channels. After every iteration, however, the controller checks to see if there are still packets in the X-channels. If none remain, a second phase of the MGRA is begun where only Y-channel moves are executed. During this phase, the controller checks the Y-channels for packets: when none remain, the algorithm is terminated.

The critical problem in this algorithm is how to handle the collisions that occur when a packet needs to switch from an X- to a Y-channel but finds the Y-channel already occupied. If the X-channel packet is simply left in place, that packet runs the danger of being overwritten by a packet arriving in the next iteration. Naive approaches are to emulate queues within each PE as in the MIMD greedy routing algorithm, or to include a notification step. In the latter approach, each PE with a blocked packet sends a message to the packets contiguously behind it informing them that they too are blocked and should not proceed in the next iteration. However, both alternatives yield $\theta(N)$ algorithms: the former approach requires queues of length $\theta(n)$, while the notification step requires n data transfers. How we deal with collisions is the key difference among the implementations of the MGRA on the different models.

4.2. The MGRA on the Basic Model

In the basic model we handle collisions by emulating queues of size 2 in the X-channels. The details are as follows: we adapt the MGRA by adding another buffer to the X-channel; we call the two buffers X-head and X-tail. The algorithm now has some additional steps interposed: instead of transferring packets directly along the from X-channel buffer to X-channel buffer, a PE moves packets from its X-head to the X-tail of its neighbor, and then internally from its X-tail to its X-head. Of course in either case, PEs only send packets if the destination buffer is clear. We demonstrate the correctness of this algorithm later; intuitively the "blocked" information travels back down the train of contiguous packets at the same rate that incoming packets become compressed in the trailing queues. The pseudocode for the MGRA can be found in Fig. 1.

```
1.  WHILE OR(XHeadPacket.InUse) = TRUE                          ; While packets in X
2.      IF YPacket.InUse AND YPacket.YAddress = PE.YAddress     ; destination?
            THEN Output := YPacket.Data                         ;   then store packet
                 YPacket.InUse := False                         ;   and clear Y-channel
3.      YPacket := South(YPacket)                               ; Y-channel move
4.      Blocked := False                                        ; initialise flag
5.      IF XHeadPacket.InUse AND XHeadPacket.XAddress = PE.XAddress ; X destination?
            THEN IF ¬YPacket.InUse                              ;   Y not occupied?
                     THEN YPacket := XHeadPacket                ;   then transfer to Y
                          XHeadPacket.InUse := FALSE            ;   and clear XHead
                     ELSE Blocked := TRUE                       ;   else blocked
6.      IF ¬XTailPacket.InUse AND ¬East(Blocked)                ; If space and not blocked
            THEN XTailPacket := East(XHeadPacket)               ;   X-channel move
7.      IF ¬XHeadPacket.InUse                                   ; If space,
            THEN XHeadPacket := XTailPacket                     ;   align packet
                 XTailPacket.InUse := FALSE                     ;   and clear XTail
8.  WHILE OR(YHeadPacket.InUse) = TRUE                          ; While packets in Y
9.      IF YPacket.InUse AND YPacket.YAddress = PE.YAddress     ; destination?
            THEN Output := YPacket.Data                         ;   then store packet
                 YPacket.InUse := False                         ;   and clear Y-channel
10.     YPacket := South(YPacket)                               ; Y-channel move
```

FIG. 1. The Mesh Greedy Routing Algorithm on the basic model. Packet transfers automatically set InUse to TRUE.

```
1. IF YPacket.InUse AND YPacket.YAddress = PE.YAddress      ; destination?
     THEN Output := YPacket.Data                            ;   then store packet
          YPacket.InUse := False                            ;   and clear Y-channel
2. YPacket := South(YPacket)                                ; Y-channel move
3. Blocked := False                                         ; initialize flag
4. IF XPacket.1.InUse AND XPacket.1.XAddress = PE.XAddress  ; X destination?
     THEN IF ¬YPacket.InUse                                 ;   and Y not occupied?
             THEN YPacket := XPacket.1                      ;   then transfer to Y,
                  XPacket.1.InUse := FALSE                  ;   clear X
                  FROM i := 1 TO p - 1                      ;   and align
                       XPacket.i := XPacket.i+1             ;   X-channel packets
                  XPacket.p.InUse := FALSE
             ELSE Blocked := TRUE                           ;   else blocked
5. IF ¬XPacket.p.InUse AND ¬East(Blocked)                   ; If space and not blocked,
     THEN XPacket.p := East(XPacket.1)                      ;   X-channel move
6. FROM i := p-1 TO 1
     IF ¬XPacket.i.InUse                                    ; If space,
        THEN XPacket.i := XPacket.i+1                       ;   align packet
             XPacket.i+1.InUse := FALSE                     ;   and clear tail
```

FIG. 2. The body of one iteration of the Mesh Greedy Routing Algorithm simulating p-length queues on the basic model when both X- and Y-channels are in use.

4.3. The MGRA with Longer Queues

Since we later compare the performance of the MGRA with length 2 queues with the MGRA with FIFO queues of arbitrary length, we also show how this latter version is implemented. We do not attempt to implement priority queues, a structure that—although it would enable an optimal algorithm in terms of communication steps—would have unacceptable overhead.

Queues can be emulated on the basic model in several ways. However, when a queue size greater than 2 (all we needed above) must be supported, the algorithms are slightly more complicated. The intuitive method is to use a circular buffer and a bit vector. A somewhat simpler method using no bit vector is to simply keep the queue "justified" to one end of the buffer. Pseudocode for one iteration using this method can be found in Fig. 2.

If hardware support for local indexing is available, then FIFO queues can be implemented directly. As mentioned earlier, however, the queue and dequeue operations are often substantially slower (by about a factor of 10) than the PE to PE move operations. Therefore even if local indexing is available, longer queues should only be used if reduction in the number of communication operations is greater than the slowdown in memory access.

4.4. The MGRA on a Mesh with Broadcast Buses

Even queues of length 2 create unwanted overhead through internal move instructions. In this section we show how the MGRA can be modified to use broadcast buses to reduce the X-channel queue size to one. This has two advantages: (1) internal moves are eliminated and (2) nearest neighbor transfers are faster for some mesh architectures when the packet memory address in the source PE is identical to the memory address in the destination PE. The problem again is dealing with collisions: we must keep from overwriting packets that are blocked because of occupied Y-channels. We use the following solution: since a packet can only be overwritten by another packet in an X-channel, PEs with blocked packets broadcast that status to their rows. Therefore, if any packet in a row is blocked, then no packet in that row proceeds. Pseudocode for this algorithm can be found in Fig. 3.

```
1. IF YPacket.InUse AND YPacket.YAddress = PE.YAddress      ; destination?
     THEN Output := YPacket.Data                            ;   then store packet
          YPacket.InUse := False                            ;   and clear Y-channel
2. YPacket := South(YPacket)                                ; Y-channel move
3. Blocked := False                                         ; initialize flag
4. IF XPacket.InUse AND XPacket.XAddress = PE.XAddress      ; X destination?
     THEN IF ¬YPacket.InUse                                 ;   and Y not occupied?
             THEN YPacket := XPacket                        ;   then transfer to Y
                  XPacket.InUse := FALSE                    ;   and clear X
             ELSE Blocked := TRUE                           ;   else blocked
5. Blocked := RowBroadcast(Blocked)                         ; inform row that blocked
6. IF ¬Blocked                                              ; if not blocked,
     THEN XPacket := East(XPacket)                          ;   X-channel move
```

FIG. 3. The body of one iteration of the Mesh Greedy Routing Algorithm using broadcast buses when both X- and Y-channels are in use.

4.5. The MGRA on a Mesh with Reconfigurable Buses

The obvious disadvantage of the above method is that some packets are needlessly prevented from proceeding. When the broadcast buses are reconfigurable, however, it is possible to retain the queue size of one while blocking only those packets that could overwrite a blocked packet. The method is as follows. Each PE containing a packet in its X-channel closes its East and West switches, while all PEs open their North and South switches. If the PE contains a blocked packet then the West switch is opened. In this way circuits are formed along the horizontal buses that are made up of contiguous PEs containing packets in their X-channels; if there is a PE with a blocked packet within the circuit, it will be in the leftmost PE. See Fig. 4 for an illustration. Pseudocode can be found in Fig. 5.

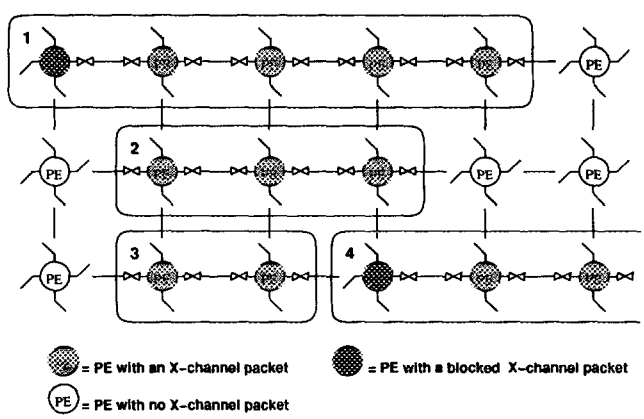

FIG. 4. Reconfigurable buses are used to form circuits containing exactly those PEs with blocked packets. After broadcast, the PEs in circuits 1 and 4 are blocked while the PEs in circuits 2 and 3 continue.

```
1. IF YPacket.InUse AND YPacket.YAddress = PE.YAddress      ; destination?
     THEN Output := YPacket.Data                            ;   then store packet
          YPacket.InUse := False                            ;   and clear Y-channel
2. YPacket = South(YPacket)                                 ; Y-channel move
3. Blocked := False                                         ; initialize flag
4. IF XPacket.InUse AND XPacket.XAddress = PE.XAddress      ; X destination?
     THEN IF ¬YPacket.InUse                                 ;   and Y not occupied?
             THEN YPacket := XPacket                        ;   then transfer to Y
                  XPacket.InUse := FALSE                    ;   and clear X
             ELSE Blocked := TRUE                           ;   else blocked
5. IF XPacket.InUse AND Blocked                             ; create circuits of adjacent
     THEN Switches(N,E,S,W) := (OPEN,CLOSED,OPEN,OPEN)      ;   PEs with X occupied
   ELSE IF XPacket.InUse AND ¬Blocked                       ;   partion if X packet
     THEN Switches(N,E,S,W) := (OPEN,CLOSED,OPEN,CLOSED);  ;   is blocked
   ELSE
     Switches(N,E,S,W) := (OPEN,OPEN,OPEN,OPEN)
6. Blocked := Broadcast(Blocked)                            ; inform circuit that blocked
7. IF ¬Blocked                                              ; if not blocked,
     THEN XPacket := East(XPacket)                          ;   X-channel move
```

FIG. 5. The body of one iteration of the Mesh Greedy Routing Algorithm using reconfigurable broadcast buses when both X- and Y-channels are in use.

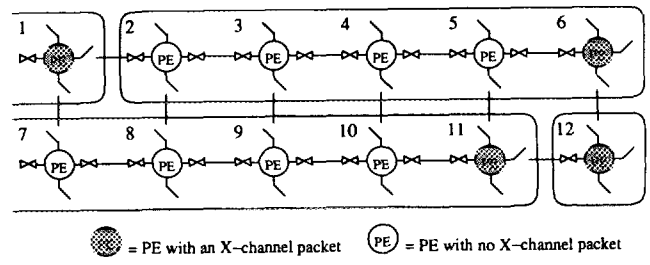

FIG. 6. In CGRA steps 1 and 3 (5 and 8) PEs broadcast packets directly to the destination PE, or to the furthest possible PE with an open channel, respectively.

4.6. The Four-Channel MGRA on the Basic Model

One way to cut down on the number of blocked packets is to double the number of channels emulated so that the scheme resembles more closely that in [6]; in this case the overhead per iteration is also roughly doubled. We call these new channels X2 and Y2. When a packet reaches the last row (column) of the torus, but has not yet reached its destination column (row), the packet switches channels to X2 (Y2) and wraps around. This scheme does indeed cut down on the congestion, but was not found to be worth the overhead.

A much more effective scheme is for the additional channels X2 and Y2 to route packets in the opposite directions to X1 and Y1, respectively. Packets are injected into the X1 or X2 (Y1 or Y2) channels so that they always travel by a shortest path from source to destination. We shall see later that this is advantageous when the maximum shortest path is significantly less than n (in Manhattan distance). The complexity of the channel emulation is significantly more than doubled, however, as there are now four ways that X- and Y-channels can interact rather than one. We call this variation the 4-channel MGRA. This algorithm again does not deadlock because the packets are transferred in their entirety.

4.7. The Coterie Greedy Routing Algorithm

Reconfigurable buses can also be used to emulate cut-through routing. Cut-through routing was developed by Kermani and Kleinrock [11] as a hybrid of store-and-forward and point-to-point circuit switched routing. Each packet is routed through the network to the furthest available PE toward its destination, where it is queued in its entirety. The advantage is that packets need not wait for the entire circuit between source and destination to be free before transfer, while also avoiding the need to be queued at every intermediate PE. The implementation of cut-through routing on reconfigurable buses involves substantially greater overhead per iteration than the MGRA, but as we see below, this is more than compensated by the decreased number of iterations when the communication pattern is sparse.

In this version of the greedy algorithm, which we call the Coterie Greedy Routing Algorithm (CGRA),[1] the X- and Y-channels are emulated not only by the nearest neighbor connections, but also by the reconfigurable broadcast buses. The major consequence is that rather than moving packets just one PE at a time, all of the open space between occupied PEs is traversed in a single iteration of the algorithm. The basic idea is to create circuits having the property that the rightmost PE (bottommost if these are Y-channels) contains a packet, while all other PEs in the circuit do not (see Fig. 6 for an illustration). The occupied PE then broadcasts its packet to the circuit, where it is read either by the destination or by the leftmost (topmost) PE. The pseudo-code with some implementation details removed can be found in Fig. 7.

4.8. Many-to-One Routing

A *combine* operation has been created by augmenting the routing algorithms as follows: Instead of simply moving the packets that have arrived at their destinations from the Y-channel(s) to the output buffer, a binary operator is interposed. For example, sum–combine adds the value in the packet to the value already in the output buffer. Many-to-one routing is implicit in the combine operation; more congestion is therefore likely to occur than in permutation routing. To deal with this situation, intermediate combining at the point of collision may optionally be executed. The cost is an increase in overhead of an extra compare and arithmetic operation for each iteration, but there are situations where intermediate combining is worthwhile. One example is the degenerate case where the entire array is combined at one destination: the complexity of that combine operation is reduced from $O(N)$ to $O(n)$.

[1] The *Coterie Network* is the name for the CAAPP reconfigurable broadcast mesh.

```
1.  IF YPacket.InUse                                    ; create circuit with one
      THEN Switches(N,E,S,W) = (CLOSED,OPEN,OPEN,OPEN)  ;   Y-channel occupied PE
    ELSE                                                ;   and adjacent empty PEs
        Switches(N,E,S,W) = (CLOSED,OPEN,CLOSED,OPEN)   ;   to north
2.  Broadcast(YPacket)                                  ; all PEs broadcast
3.  IF PE.YAddress = Broadcast(YPacket.YAddress)        ;   destination of broadcast?
      THEN Receive(YPacket)                             ;   then receive
4.  IF YPacket.InUse AND YPacket.YAddress = PE.YAddress ; destination?
      THEN Output := YPacket.Data                       ;   then store packet
        YPacket.InUse := False                          ;   and clear Y-channel
5.  Broadcast(YPacket)                                  ; all PEs broadcast
    IF North(YPacket.InUse) AND ¬YPacket.InUse          ;   furthest empty PE?
      THENReceive(YPacket)                              ;   then store packet
6.  YPacket = South(YPacket)                            ; guarantee some progress
7.  Blocked := False                                    ; initialize flag
8.  IF XPacket.InUse                                    ; create circuit with one
      THEN Switches(N,E,S,W) = (OPEN,OPEN,OPEN,CLOSED)  ;   X-channel occupied PE
    ELSE                                                ;   and adjacent empty PEs
        Switches(N,E,S,W) = (OPEN,CLOSED,OPEN,CLOSED)   ;   to west
9.  Broadcast(XPacket)                                  ;   all PEs broadcast
10. IF PE.XAddress = Broadcast(XPacket.XAddress)        ;   destination of broadcast?
      THEN Receive(XPacket)                             ;   then receive
11. IF XPacket.InUse AND XPacket.XAddress = PE.XAddress ; X destination?
      THEN IF ¬YPacket.InUse                            ;   and Y not occupied?
          THEN YPacket := XPacket                       ;   then transfer to Y
            XPacket.InUse := FALSE                      ;   and clear X
          ELSE Blocked := TRUE                          ;   else blocked
12. IF XPacket.InUse AND Blocked                        ; create circuits of adjacent
      THEN Switches(N,E,S,W) := (OPEN,CLOSED,OPEN,OPEN) ;   PEs with X occupied
    ELSE IF XPacket.InUse AND ¬Blocked                  ;   partion if X packet
      THEN Switches(N,E,S,W) := (OPEN,CLOSED,OPEN,CLOSED); is blocked
    ELSE
        Switches(N,E,S,W) := (OPEN,OPEN,OPEN,OPEN)
13. Blocked := Broadcast(Blocked)                       ; inform circuit that blocked
14. IF ¬Blocked                                         ; non-blocked PEs
      THEN Broadcast(XPacket)                           ;   broadcast
15. IF West(XPacket.InUse) AND ¬XPacket.InUse           ;   furthest empty PE?
      THEN Receive(XPacket)                             ;   then store packet
16. IF ¬Blocked                                         ; guarantee some progress
      THEN XPacket := East(XPacket)
```

FIG. 7. Reconfigurable broadcast buses used to transmit data. Partial code for the body of one iteration of the Coterie Greedy Routing Algorithm when both X- and Y-channels are in use.

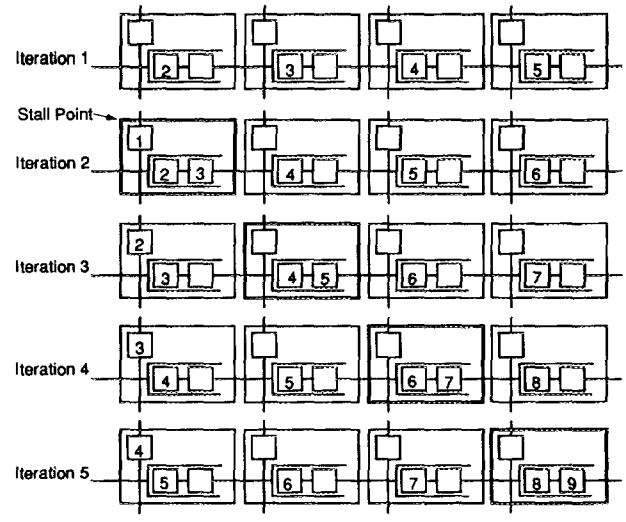

FIG. 8. Stall point progression.

5. CORRECTNESS AND FREEDOM FROM DEADLOCK

We present an informal argument for the correctness of the MGRA. Clearly if a packet does not get permanently blocked it proceeds across the row in which it starts to the column of its destination, switches from the X-channel to the Y-channel, and then proceeds up the column to the row of its destination. We must only show that no packet can be blocked forever. A packet is *blocked* on a given iteration when another packet occupies the buffer into which it must proceed. We define a *stall point* as a processor whose queue is full at the end of an iteration (i.e., both X-head and X-tail contain a packet). A packet in the Y-channel of a processor may create a stall point by blocking a packet in the X-channel wishing to enter the Y-channel at that processor. Assuming a contiguous stream of packets behind the blocked packet, this stall point would move right, against the flow of the packets, one processor per iteration, like a compression wave. The creation and propagation of a single stall point are shown in Fig. 8. Note that a packet will be delayed one iteration for every stall point that it encounters in the X-channel.

Suppose some packet is permanently blocked. Since packets in the Y-channel cannot be blocked, the permanently blocked packet must be in the X-channel. In order for the packet to remain in place, a continuous stream of stall points must pass over it. Each of these stall points must be created by a packet in the Y-channel blocking a packet in the X-channel of this row. Such a packet, once creating a stall point, must move out of this row on the next iteration since packets in the Y-channel cannot be blocked. Furthermore, since it must reach its destination before coming around to this row again, it can never create another stall point in this row. Therefore, each of the stall points passing over the permanently blocked packet must be created by a different packet, but since there are only a finite number of packets, this is impossible. Hence no packet is permanently blocked.

This proof depends on the packets being small enough to fit into one buffer. If the packets had to be strewn out in flits over several processors, as in wormhole routing, then four channels would be necessary to prevent deadlock [6].

6. WORST CASE

In the worst case, the MGRA can take time linear in the size N of the network, but never longer.

Upper Bound

THEOREM. *Let M be an $n \times n$ mesh. For all permutations π: $M \rightarrow M$ the routing algorithm takes at most $n^2 + o(n^2)$ iterations to route π.*

Proof. We augment the above proof of correctness by counting how many iterations a packet can be blocked. On each iteration, a packet either takes one step toward its destination or is delayed by a stall point. A packet can create stall points only when in the Y-channel and thus

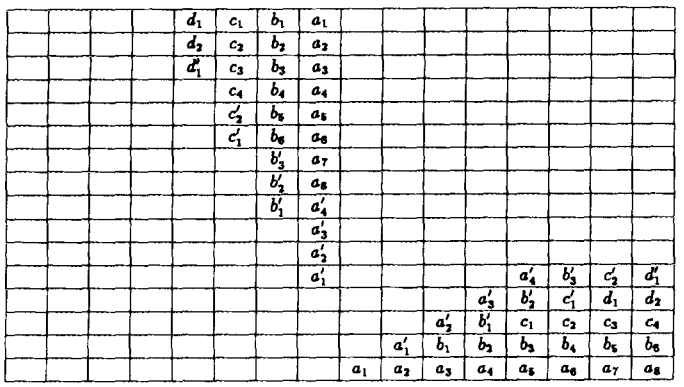

FIG. 9. Permutation causing a quadratic amount of blocking for $n = 16$.

can create only one per row since a packet in the Y-channel cannot be blocked and will only pass each row at most once. Therefore, at most n^2 stall points can be created in a given row, so packets in that row can be blocked for at most n^2 iterations. Since a packet can have at most $2n - 2$ units to travel, any packet must finish within $2n - 2 + n^2 = n^2 + o(n^2)$ iterations. ■

Lower Bound

THEOREM. *Let M be an $n \times n$ mesh. There exists a permutation π: $M \to M$ which takes $\Omega(n^2)$ iterations to route.*

Proof. We construct a permutation in which $\Omega(n^2)$ stall points pass through a packet; the result follows. The idea is to set up $\Omega(n)$ rows, each of which will block some packet for $\Omega(n)$ iterations. A sequence of rows is set up such that each row crosses in front of the remaining rows, creating stall points in them. The actual permutation for $n = 16$ is shown in Fig. 9; larger cases are analogous. Certain elements of the mesh are labeled. Each label appears exactly twice, so each labeled element has a unique "partner." Define π as the permutation which maps each labeled element to the location of the partner of that element, and maps all other elements to their current location.

We focus on the packets starting in the lower right quadrant, all of which are routed to the left half of the mesh. The letter in an element label determines the column in the left half of the mesh the packet is routed to. The permutation is constructed so that while elements with unprimed labels ascend a column, the remaining labeled rows are blocked by elements labeled with the same letter primed. During the route, the row of 8 a's creates 8 stall points in all the other rows as it ascends its column, since all the other rows begin with an element a_i' which also needs to enter that column. Similarly, the row of 6 b's then creates 5 stall points in the remaining rows (b_1 ascends the column one iteration too soon to create stall points in the rows above), the row of 4 c's creates 3 stall points (c_1 is similarly one iteration too soon), etc. Thus $8 + 5 + 3 + 1$ stall points pass through the element d_1', delaying it on 17 iterations.

In the general case, we set up $\lfloor n/4 \rfloor + 1$ rows in the lower right quadrant of the mesh and $\lfloor n/4 \rfloor$ columns in the left half of the mesh as follows. Number the rows 1, 2, 3, ..., $\lfloor n/4 \rfloor + 1$ starting from the bottom of the mesh; number the columns in the left half of the mesh 1, 2, 3, ..., $\lfloor n/4 \rfloor$ from the right (i.e., the column just to the left of center is column 1). The ith row ($i = 1, \ldots, \lfloor n/4 \rfloor + 1$) starts with $\lfloor n/2 \rfloor + i - 1$ unlabeled elements (to allow the labeled elements of row 1 to be in column 1 when the first labeled element of this row reaches column 1), followed by $i - 1$ labeled elements routed to columns 1, 2, 3, ..., $i - 1$, in that order (these correspond to the primed elements in the example above), followed by $\lfloor n/2 \rfloor - 2(i - 1)$ elements routed to column i. The labeled elements of row i create at least $\lfloor n/2 \rfloor - 2(i - 1) - 1$ stall points in all rows $j = i + 1, \ldots, \lfloor n/4 \rfloor + 1$, thus

$$\sum_{i=1}^{\lfloor n/4 \rfloor} \lfloor n/2 \rfloor - 2(i - 1) - 1 = \Omega(n^2)$$

stall points pass through the last element of row $\lfloor n/4 \rfloor + 1$. ■

Dealing with the Worst Case

Clearly the worst case performance is unacceptable in a practical routing scheme. In the next section we present experimental results which indicate that communication patterns that result in even double the optimal running time (much less anything approaching n times) are hard to find, and that random permutations have especially good and consistent performance.

Valiant and Brebner [25] addressed the problem of poor worst case routing performance by applying a randomization preprocessing step before greedy routing. The result is that routing on the MIMD model can be executed nearly optimally with extremely high probability. Although our algorithms differ significantly in how they deal with blocked packets, we can expect that similar preprocessing will yield a similar reduction in the amount of congestion. In the next section we demonstrate this result.

7. EXPERIMENTAL RESULTS

In previous sections we have presented online routing algorithms for a series of models based on classes of SIMD meshes that have low overhead per iteration. We have also proven correctness, freedom from deadlock,

and $O(N)$ worst case performance. In this section we demonstrate the practicality of those algorithms by showing that the typical performance is close to optimal, while any communication pattern approaching the worst case does not appear in practice. In particular, we seek to answer the following questions:

1. For all the algorithms described above, what is the typical and worst case performance in number of iterations and communication steps? What does this say about the likelihood of the theoretical worst case?
2. Which algorithms have advantages in routing certain types of communication patterns?
3. Does randomization preprocessing help? Is it worth the cost?
4. How do the algorithms compare when overhead is factored in?
5. What is the performance on a detailed model? How do these results scale?

Not all these questions are best answered by the same method; we use three types of simulation having increasingly complex models: (1) counting the number of communication steps and algorithm iterations, which is sufficient to answer the first three questions; (2) combining the number of iterations with the relative overhead of the different algorithms as obtained by comparing the number of identical instructions executed per iteration; and (3) accurate hardware models approximating actual machine execution. The last two models are needed to address the final two questions.

Once we have a sufficiently accurate simulation model for the task at hand, we must select the patterns with which to test the algorithms. Since we are interested in practical performance, the ideal set of patterns would be those extracted from execution traces of real programs. However, there are as yet very few applicable systems in operation and therefore few available traces. The alternative of using random permutations is inadequate, however. Though important (e.g., arising after a randomization preprocessing phase as in Valiant and Brebner [25]), random permutations are generally among the "easiest" patterns to route. And (as we shall show below) randomization preprocessing is not always cost-effective. Therefore, we also test our algorithms on many other patterns (and classes of patterns) found in the literature. This method is still not perfect, but the results we present in this section are in such consistent agreement so as to provide adequate support for our assertions.

We divide the communication patterns into two categories: (1) *particular* patterns such as transpose and shuffle, and (2) *classes* of patterns, which are defined by a rule for generating patterns, and over which a mean and standard deviation can be taken. Some particular patterns used are based on permuting and complementing the bits in the PE ID. These are known as bit permute/complement permutations and are described in [20]. The definitions of the classes of patterns are as follows:

- Bit Permute (BP): patterns derived from randomly permuting the bits of the PE ID
- Bit Permute and Complement (BPC): patterns derived from randomly permuting and flipping the bits of the PE ID
- P-ordered Vectors (see [23])
- Image Rotation: patterns derived using the standard matrix; rotation in general produces a many-to-one communication pattern.

All experiments in this section, unless otherwise specified, were run on an $n \times n$ array where $n = 256$. Results for the random, random BP, and random BPC classes of patterns are based on runs of at least 100 trials. For p-ordered vectors, a trial was run for each permutation generated by the relatively prime values of P less than 256. The image rotation was tested for all angles (in degrees) divisible by 5. For most of these patterns there exists a packet that must travel nearly the maximum distance and so the minimum number of neighbor moves (and therefore iterations of the MGRA) possible is $2n - 2$, or 510. We refer to this figure as the "trivial lower bound" or as "optimal performance." And finally a note about counting iterations and communication steps: recall that in the MGRA, iterations contain two communication steps in the first phase and one in the second. They can therefore be related as follows: $2 * \textit{iterations} - n \simeq \textit{communication steps}$.

7.1. Algorithm Performance in Communication Steps

Tables I and II contain the results of the basic two-channel MGRA on particular communication patterns and on classes of communication patterns respectively. The major results are that random permutations require only slightly more than the lower bound of iterations and 1.5 times the lower bound of communication steps, with very small variance, while the most costly patterns tried require only a factor of 1.5 times the lower bound of iterations and 2.5 times the lower bound of communication steps. Recall that the algorithms that use sorting as a preconditioner require a factor of 8 times the lower bound communication steps with queue size of 1 and 4.5 times the lower bound of communication steps with a priority queue of size 3. Also significant is that the standard deviations of the results from the classes of patterns were a fraction of n. This is an indication that encountering patterns from these classes that are much worse than those presented is very unlikely. Another interesting result is that for some permutations, no packets collide at all; these are so indicated in Table I. Additional experiments (not presented here) support these results for arrays of different diameters: trials were run for $n = 4, 8,$

TABLE I
Performance of the MGRA on a 256 × 256 Basic Architecture when Routing Particular Permutations

Pattern	# of iterations	# of comm. steps	Collisions?
Bit reverse (FFT)	498	754	No
Unshuffle	512	768	No
Shuffle	512	768	No
Transpose	258	514	No
Reflection in X-axis	257	513	No
Reflection in Y-axis	257	258	No
Vector reverse	512	768	No
Shuffled row-major	664	1101	Yes
Bit shuffle	758	1269	Yes
Snake-like row-major	257	258	No
Snake-like column-major	511	767	No
90° rotation	511	767	No
180° rotation	512	768	No
270° rotation	511	767	No

16, ..., 256 on all particular permutations with similar results. For random permutations, many thousands of trials were run for many additional n values up to 512.

Table III contains the results of the other versions of the MGRA and of the CGRA for the classes of communication patterns. The variances are similar to those in Table II and are not shown. For particular patterns that were routed by the MGRA with no collisions, performance is the same as above. The major result for each algorithm is as follows:

Long FIFO queues. Increasing the size of the queues from 2 to infinite does not significantly reduce the number of iterations required.

Reconfigurable Buses. Reducing the queue length from 2 to 1 and using reconfigurable buses to block only the necessary packets does not significantly increase the number of iterations required.

Broadcast Buses. When the nonreconfigurable buses are used to block all packets in a row where any packet is blocked, there was an increase in the number of iterations which varied from 5% to 20%.

TABLE II
Performance of the MGRA on a 256 × 256 Basic Architecture when Routing Classes of Permutations

Class of pattern	Mean # of iterations	Standard deviation	Worst case	Mean # of comm. steps
Random	524.65	3.76	539	807.03
Random BP	611.56	83.61	780	983.46
Random BPC	614.56	80.94	798	994.02
Rotation	631.57	96.84	827	1037.18
P-vector	511.10	19.12	761	774.29

Four Channels. Using four channels reduces the number of iterations by slightly more than a factor of 2 with respect to the original two-channel version. This factor is not precisely 2 because using extra channels also reduces congestion.

CGRA. The use of broadcast to transmit packets reduces the number of iterations by 45–55% with respect to the two-channel MGRA.

7.2. Algorithm Performance on Local and Sparse Communication

The Four-Channel MGRA routes packets via shortest paths and could therefore be assumed to perform better on communication patterns exhibiting locality. This is confirmed by the results in Table IV showing that the number of iterations depends almost entirely on the maximum distance a packet must travel, and is independent of the diameter of the network.

The CGRA transmits packets using broadcast; therefore performance should improve as the density of the communication pattern is decreased by enabling packets to propagate further on each communication step. As a test we used random permutations with a proportion of the packets removed at random. The results for the number of iterations as a function of the percentage of PEs sending packets are shown in Fig. 10. The minimum number of iterations required to route a packet (not starting at the correct row or column coordinates) using the CGRA

TABLE III
Performance of Different Algorithms on 256 × 256 Arrays when Routing Classes of Permutations

	Performance in iterations for different algorithms					
Class of pattern	MGRA	FIFO queues	Broadcast buses	Reconf. buses	CGRA	4C MGRA
Random	524.65	525.40	641.87	524.94	220.73	260.04
Random BP	611.56	606.08	650.02	615.70	339.35	303.14
Random BPC	614.56	613.94	650.30	618.90	345.40	306.77
Rotation	631.57	623.83	654.10	637.99	297.29	264.56

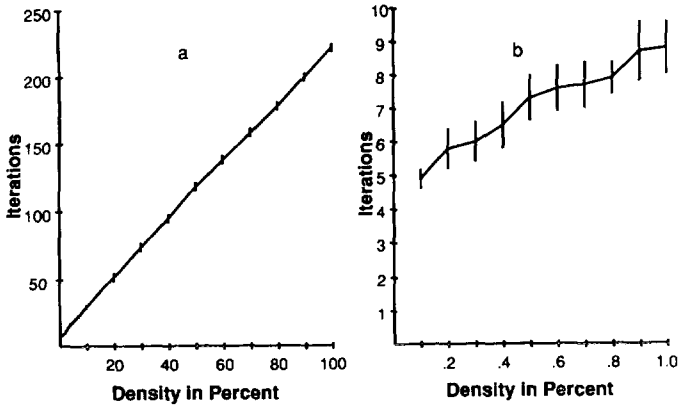

FIG. 10. CGRA results for random permutations on a 256 × 256 array: (a) The number of iterations decreases nearly linearly with respect to the density of the communication pattern; (b) the asymptotic limit of 4 is approached.

is 4. Therefore, when the density is very small, only very few iterations more than the minimum possible are required (as is shown in Fig. 10b).

7.3. Randomization Preprocessing

In Tables I and II we saw that the maximum number of iterations was nowhere near N and that the standard deviations were a fraction of n; still, the performance varied from slightly more than $2n$ with standard deviation of 4 for random permutations, to around $3n$ with a standard deviation of 84 for certain bit-permute permutations. Clearly there would be an advantage if the performance for all (nonregular) patterns were as good as that for random permutations. In particular, introducing a randomizing phase similar to that in [25] has two potential benefits: improvement of the performance (not including preprocessing) when routing nonrandom patterns, and reduction in the likelihood of poor performance.

We see in Tables V and VI that for the two- and the four-channel MGRA, randomization reduces the number of iterations and the standard deviations for the permutations tested. For the four-channel MGRA, randomization of the BP and BPC classes yields performance nearly identical to that of performance on random permutations. For the two-channel MGRA, the new performance of the BP and BPC classes is about halfway between that of the original performance and that for the random case.

Is randomization worth the cost? The preprocessing procedure requires n packet move steps, but since no comparisons need to be made, the overhead is less than for a comparable number of iterations of the MGRA. (Since the same randomization can be used for every communication pattern, a preprocessing route can be computed offline at compile time; the n-bit-long vector encoding of that route is brought into PE memory at program load time.) Even so, randomization does not increase performance enough (only 5% to 15%), for the communication patterns tried, to make it a standard feature. However, because the standard deviations were decreased substantially, especially for the four-channel

TABLE IV
The Performance of the Four-Channel MGRA when the Diameter and the Maximum Distance Are Varied

Diameter	Maximum distance	Mean # of iterations	Standard deviation
20	10	14.6	.89
40	10	15.7	.94
60	10	15.7	.95
80	10	16.3	.71
100	10	16.3	.84
120	10	16.6	.89
140	10	16.9	1.05
160	10	17.0	.45
180	10	17.0	.00
200	10	17.1	.54
220	10	17.4	.49
240	10	17.2	.55
256	10	17.4	.66
256	20	28.5	.67
256	40	48.9	.70
256	60	69.1	.94
256	80	89.4	.92
256	100	109.3	.78

TABLE V
Number of Iterations and Communication Steps Required for Classes of Permutations Using the MGRA in 256 × 256 Array: Comparison with Randomized Axis Version

	Two-channel MGRA			Two-channel MGRA with randomization		
Class of pattern	Iterations	Standard deviation	# of Comm. steps	Iterations	Standard deviation	Total # of comm. steps
Random BP	611.56	83.61	983.46	567.21	45.92	1152.01
Random BPC	614.56	80.94	994.02	570.90	48.43	1158.63
Rotation	631.57	96.84	1037.18	555.25	70.49	1135.94
Random	524.65	3.76	807.03			

TABLE VI
Number of Iterations and Communication Steps Required for Classes of Permutations Using the Four-Channel MGRA in 256 × 256 Array: Comparison with Randomized Axis Version

	Four-channel MGRA			Four-channel MGRA with randomization		
Class of pattern	Iterations	Standard deviation	# of Comm. steps	Iterations	Standard deviation	Total # of comm. steps
Random BP	303.14	42.66	975.18	260.53	9.43	1052.18
Random BPC	306.77	48.11	995.96	261.49	3.89	1057.80
Rotation	264.56	99.30	945.18	246.53	37.54	746.04
Random	260.04	1.57	792.46			

MGRA, randomization could be useful if more consistent performance were required.

7.4. Algorithm Performance Including Overhead

In order to compare the algorithms to each other we need to take into account the overhead per iteration. The problem is that the precise overhead of the various algorithms depends on feature implementations that change from machine to machine. However, certain comparisons among algorithms—such as counting identical instructions—can certainly be made. Also, instructions of similar complexity, such as internal moves and compares, can be related to instructions that take substantially longer, such as broadcast moves and off-chip memory references. Table VII contains the counts of instructions of different types for the different algorithms.

We make the following observations.

Basic MGRA. The MGRA requires an extra internal move (in comparison to the two bus algorithms) to simulate the size-two queue. Up to two additional internal moves are required depending on how nearest neighbor moves are implemented.

Four-Channel MGRA. The four-channel version has twice as many nearest-neighbor moves per iteration as the two-channel version. It also requires extra internal moves as a packet from either X-channel can be moved to either Y-channel.

Broadcast and Reconfigurable Buses. These versions have overhead similar to each other. The reconfigurable bus version *does* require a few extra cycles per iteration to deal with handshaking, but this does not significantly change the relative overhead.

FIFO Queues. The internal moves are replaced with queue and dequeue operations.

CGRA. The CGRA has about eight times the overhead of the MGRA.

We combine the overhead with the iteration counts of Table III to make the following algorithmic comparisons with respect to general routing.

Two- vs Four-Channel MGRA. The four-channel MGRA requires slightly fewer than half the iterations of the two-channel version. This difference, however, is not enough in general to offset the extra internal moves required for each iteration by the four channel MGRA.

TABLE VII
Overhead Comparisons: (1) Cost per Iteration in Terms of Numbers of Operations of Different Types, (2) Cost of Startup Overhead in Number of Operations

	Overhead per iteration					
Algorithm	Internal moves	Compares	Neighbor moves	Queue operations	Broadcast moves	Overhead at startup
MGRA	3–5	2	2	0	0	2
4 channel MGRA	8–12	4	4	0	0	15
FIFO queues	0	2	2	2	0	2
Broad. buses	2	2	2	0	0	2
Reconf. buses	2	2	2	0	0	2
MGRA + rand.	3–5	2	2	0	0	$2n + 2$
CGRA	4	6	2	0	4	2

TABLE VIII
Fine Simulation Comparisons for Routing 16 Bits of Data on a 256 × 256 Array

Parameters				Routing times			
Memory width	ALU width	Neighbor bus width	Broadcast bus width	MGRA	4C MGRA Max(d) = 10	CGRA: 500 packets	CGRA: 100 packets
1	1	1	1	80	8	11	5.5
8	1	1	1	50	5	10	5
8	8	1	1	40	4	9	4.5
8	8	8	1	20	2	9	4.5
8	8	8	8	20	2	2	1
Handshaking overhead				10	1	1	.5

Note. Times in 1000's of cycles. Width of features in bits.

Broadcast Buses vs Basic Model. Using broadcast buses improves performance: the 15–30% reduction in overhead more than compensates for the 5% to 20% increase in iterations.

Reconfigurable Buses vs Broadcast Buses. Using reconfigurable buses improves performance more than using broadcast buses does: the number of iterations is reduced by from 2.5% to 18% while the increase in overhead is negligible.

CGRA vs MGRA. All versions of the MGRA are superior to the CGRA for dense routing patterns since the factor of 2 reduction in iterations is offset by a factor of 10 increase in the overhead.

We have now seen that the MGRA, four-channel MGRA, and CGRA are all faster for certain communication patterns. Automating the online choice among these three algorithms is straightforward: The maximum distance any packet must travel can be obtained in a few microseconds using a global OR and an associative leader election algorithm. If hardware support for global count is available, the density of the communication pattern can be quickly found as well.

7.5. Performance and Scalability on a Detailed Model

In order to compare our approach with other available routing methods and to predict scalability with changing technology, we must use more detailed simulations. The problem is again that costs differ from machine to machine. Some important factors are the widths of the ALU, the memory paths, and the nearest neighbor connections; whether nearest-neighbor move instructions send data directly from PE memory to PE memory or whether they must first go through a transfer register; and whether memory is addressable on bit boundaries.

To try all the different combinations of possible features is of course impractical. Instead, we simulate a representative sample based on various configurations of the CAAPP [27]. The CAAPP has eight-bit internal memory paths and one-bit-wide ALU and communication paths, and can perform nearest-neighbor moves from on-chip memory to on-chip memory in a single cycle. We also simulate a model based on the CAAPP but less powerful (one-bit internal data paths), as well as a model that is more powerful (wider ALU and external data paths).

We test three algorithms: the MGRA using reconfigurable buses, the four-channel MGRA using reconfigurable buses, and the CGRA. The communication patterns tested are a complete random permutation, a random permutation with no packet distance greater than 10, and two sparse random permutations routing 100 and 500 packets respectively. These choices of architectures, algorithms, and communication patterns, though they only represent a small sample, yield some hard numbers from a representative set of design decisions.

We summarize here the fine simulation results appearing in Table VIII. The CAAPP cycle time has been conservatively estimated at 100 ns (in the first generation). Thus the execution times for a random permutation, a "nearby" random permutation, and a "sparse" random permutations would be 5 ms, 500 μs, and 500–1000 μs respectively. We also see in Table VIII that this performance can be increased by a factor of 4 when the path widths are all increased from 1 to 8. The speed-up is not directly proportional to the increase in width of the data paths because only single bits are exchanged during handshaking operations. Since the handshaking comprises 12% of the routing time when the data paths are all one bit wide, it follows that the possible improvement in performance due to wider data paths is limited to a factor of 8.

8. CONCLUSION

This work addresses the problem of general online routing on SIMD meshes having no dedicated hardware support for that purpose. Our goal in carrying out this

research was to find the best possible algorithms to be integrated into a system for use on real applications. Upon reviewing the literature, however, it was found that existing algorithms all make unacceptable tradeoffs. They are either slow (though completely general), or optimal but unsatisfactory in some other way: they either require adding impractical hardware to the model, are applicable to only a small subset of possible communication patterns, or require too much preprocessing. In this work we made the decision to create the fastest possible algorithms for patterns that are likely to occur in practice. The tradeoff is that there are routing patterns for which our algorithms do not work well, and thus that there exists a worst case which causes bad performance. However, we have also shown that this is unlikely: that communication patterns that require even 2.5 times the optimal number of communication steps occur very infrequently, and that randomization can be used to make them still more unlikely. This is done using only FIFO queues of size 2 or, if broadcast buses are available, queues of size 1.

We have thus shown that our algorithms have the twin benefits of having very low overhead and only rarely requiring significantly more iterations than the trivial lower bound. Hopefully a consequence of this paper will be that users who have only used their array processors for solving problems with regular and uniform communication patterns will be encouraged to use them to solve additional problems.

Proving that the expected complexity of the MGRA is close to $2n$ is closely related to similar problems involving wormhole routing in general. In [8], some of the authors outline a method for modeling the expected performance using differential equations. We believe that finding an analytical solution, although under investigation, remains an open problem at this time.

ACKNOWLEDGMENTS

We thank Arny Rosenberg for his comments on an earlier version of this paper, Seth Malitz for his suggestion to apply the basic algorithm to other models, and Jim Burrill for his help with the simulations.

REFERENCES

1. Aiello, B., Leighton, F. T., Maggs, B., and Newman, M. Fast algorithms for bit-serial routing on a hypercube. *Math. Systems Theory* **24,** 253–271.
2. Annexstein, F., and Baumslag, M. A unified approach to off-line permutation routing on parallel networks. *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures*. 1990, pp. 398–406.
3. Deleted in proof.
4. Batcher, K. E. Design of the massively parallel processor. *IEEE Trans. Comput.* **C-29,** 9 (Sept. 1980), 836–840.
5. Blevins, D. W., Davis, E. W., Heaton, R. A., and Reif, J. H. Blitzen: A highly integrated massively parallel machine. *J. Parallel Distrib. Comput.* **8,** 2 (Feb. 1990), 150–160.
6. Dally, W. J., and Seitz, C. L. Deadlock free routing in multiprocessor interconnection networks. *IEEE Trans. Comput.* **C-36,** 5 (May 1987), 547–553.
7. Duff, M. J. B. Review of the CLIP image processing system. *Proceedings of the National Computing Conference*. 1978, pp. 1055–1060.
8. Herbordt, M. C., Weems, C. C., and Corbett, J. C. Message passing algorithms for a SIMD torus with coteries. *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures*. 1990, pp. 11–20.
9. Herbordt, M. C., Corbett, J. C., Spalding, J., and Weems, C. C. Practical algorithms for online routing on SIMD meshes. COINS Technical Report 91-63, University of Massachusetts, Amherst, 1991.
10. Hunt, D. J. The ICL DAP and its application to image processing. In Duff, M. J. B., and Levialdi, S. (Eds.). *Language and Architectures for Image Processing*. Academic Press, London, 1981, pp. 275–282.
11. Kermani, P., and Kleinrock, L. Virtual cut-through: A new computer communication switching technique. *Comput. Networks* **3** (1979), 267–286.
12. Kunde, M. Packet routing on grids of processors. In *Lecture Notes in Computer Science*, Vol. 401. Springer-Verlag, New York, 1988, pp. 129–136.
13. Leighton, F. T., Makedon, F., and Tollis, I. A $2n - 2$ step algorithm for routing in an $n \times n$ array with constant size queues. *Proceedings of the 1st ACM Symposium on Parallel Algorithms and Architectures*. 1989, pp. 328–335.
14. Leighton, F. T. Average case analysis of greedy routing algorithms on arrays. *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures*. 1990, pp. 1–10.
15. Leighton, F. T. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufman, San Mateo, CA, 1992.
16. Lit, H., and Maresca, M. Polymorphic Torus network. *IEEE Trans. Comput.* **C-38,** 9 (Sept. 1989), 1345–1351.
17. MasPar Computer Corporation. *MasPar MP-1 Principles of Operation*. Document Part Number: 9300-5001, Revision: 1090, 1990.
18. McCormick, B. T. The Illinois pattern recognition computer—ILLIAC III. *IEEE Trans. Electron. Comput.* **C-12,** 12 (Dec. 1963), 791–813.
19. Nassimi, D., and Sahni, S. Bitonic sort on a mesh-connected parallel computer. *IEEE Trans. Comput.* **C-28,** 1 (Jan. 1979), 2–7.
20. Nassimi, D., and Sahni, S. An optimal routing algorithm for mesh-connected parallel computers. *J. Assoc. Comput. Mach.* **27,** 1 (Jan. 1980), 6–29.
21. Raghavendra, C. S., and Prasanna Kumar, V. K. Permutations on Illiac IV-Type networks. *IEEE Trans. Comput.* **C-35,** 7 (Jul. 1986), 662–669.
22. Strong, J. P. Basic image processing algorithms on the Massively Parallel Processor. In Preston, K., and Uhr, L. (Eds.). *Multicomputers and Image Processing: Algorithms and Programs*. Academic Press, New York, NY, 1982.
23. Swanson, R. C. Interconnections for parallel memories to unscramble p-ordered vectors. *IEEE Trans. Comput.* **C-23,** 11 (Nov. 1974), 1105–1115.
24. Thinking Machines Corporation. *Connection Machine Model CM-2 Technical Summary*. Thinking Machines Technical Report HA87-4, 1987.

25. Valiant, L. G., and Brebner, G. J. Universal schemes for parallel computation. *Proceedings of the 13th ACM Symposium on the Theory of Computing*. 1981, pp. 263–277.
26. Weems, C. C., Levitan, S. P., Hanson, A. R., Riseman, E. M. Nash, J. G., and Shu, D. B. The Image Understanding Architecture. *Internat. J. Comput. Vision*. **2**, 3 (Jan. 1989), 251–282.
27. Weems, C. C., and Burrill, J. R. The Image Understanding Architecture and its programming environment. In Prasanna Kumar, V. K. (Ed.). *Parallel Architectures and Algorithms for Image Understanding*. Academic Press, Orlando, FL, 1991.

MARTIN C. HERBORDT received the B.A. degree in physics and philosophy from the University of Pennsylvania in 1981 and the M.S. degree in computer science from the University of Massachusetts in 1990. He is currently working on his Ph.D. in computer science at the University of Massachusetts. His research interests include parallel algorithms, architectures, and languages, as well as machine vision and other applications requiring massive parallelism.

JAMES C. CORBETT received the B.S. degree in computer science from Rensselaer Polytechnic Institute in 1987 and the M.S. and Ph.D. degrees in computer science from the University of Massachusetts at Amherst in 1990 and 1992. He is currently an Assistant Professor in the Department of Information and Computer Science at the University of Hawaii at Manoa. His research is directed toward devising practical techniques and building automated tools for the analysis and verification of concurrent and real-time software.

CHARLES WEEMS received the B.S. and M.A. degrees from Oregon State University in 1977 and 1979, respectively, and the Ph.D. from the University of Massachusetts at Amherst in 1984. All degrees are in computer science. Since 1984 he has been director of the Parallel Image Understanding Architectures Research Group at the University of Massachusetts, where he is an associate professor. His research interests include parallel architectures to support low-, intermediate-, and high-level computer vision, benchmarks for vision, parallel programming languages, and parallel vision algorithms. He is also the coauthor of a widely used introductory computer science text.

JOHN SPALDING received the B.S. degree in physics from Carnegie–Mellon University in 1976 and the Ph.D. degree in experimental physics from the California Institute of Technology in 1983. Following his degree he worked in circuit, logic, and computer architectural design at Rockwell, Precision Monolithics, and Hughes Aircraft Co. He is currently a visiting assistant research scientist at the University of Michigan. His interests include SIMD machine architecture, hardware requirements for machine vision, and active vision sensor architecture.

Received September 19, 1992; accepted September 30, 1992