Practical Algorithms for Online Routing on SIMD Meshes*

Martin C. Herbordt James C. Corbett

Charles C. Weems

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

John Spalding

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

*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). M.C. Herbordt is also supported by an IBM Fellowship. The work of J.C. Corbett is supported by Office of Naval Research grant N00014-89-J-1064. J.C. Corbett is also supported by a Graduate School Fellowship.

Abstract:

The problem of online routing on SIMD meshes arises when these processors are used for real-time analysis of sensory input. Existing algorithms—ones not making unrealistic hardware assumptions—use sorting as a subroutine and so, while asymptotically optimal, have running times at least 3 times greater (for likely array sizes) than the lower bound of $4n - 4$ communication steps. Dedicated router networks address this problem, but are costly and not available on all processors. We present a greedy algorithm with very low overhead based on wormhole routing, which can be modified to use broadcast buses and reconfigurable broadcast buses to successively improve performance. While we prove a bad worst case, performance is usually at worst a small additive constant from optimal, and always within a factor of 2 for every communication pattern tried. We also show that preprocessing randomization can improve the reliabilty of this result. Other algorithms are presented: one takes advantage of communication where all packets must travel only a short distance; another uses broadcast buses to transmit packets, improving performance when communication is sparse. Finally, we present performance results of these algorithms with respect to increased path width and packet splitting.

1 Introduction

Mesh-connected array processors ([7, 5] and many others) have been found very useful in a variety of applications, the foremost being image processing, matrix operations, and other problems where most of the computations are either local to each processing element (PE), regular, or require exchange of information only within a small number of nearby PEs. One fundamental problem with these array processors, a problem that inhibits their use in a number of domains, is the difficulty in moving data efficiently among PEs that are neither proximate nor in a regular pattern, a situation that arises in analysis of sensory input [11]. Some recent designs such as the Connection Machine CM-II [12] and the MasPar MP-1 [24] 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 generality, or lack of support for intermediate combining. Also, router networks with unbounded fanout such as Benes networks and Hypercubes face scalability problems as array sizes increase.

In other recent machine designs such as the Blitzen [6] and the CAAPP [37], the decision

has been made to forgo the general routing network: this is because its 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. A very important question is thus 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 irregular, data dependent, communication patterns. In this article we introduce and evaluate numerous online routing algorithms for SIMD meshes with and without broadcast networks. We find that their performance is surprisingly good with respect to both the number of communications operations and overhead; and that while not always equal to a dedicated general routing network, often comparable.

Much theoretical work has been done on the problem of routing and sorting on meshes—sorting is applicable here because it is no more difficult than routing; routing is obtained by sorting packets and using their destinations as keys. However, much of the preceding work routes only a subset of possible permutations [27], assumes more complex hardware than can be expected on a SIMD PE (e.g. priority queues) [36, 17, 19, 20], or requires substantial offline computation [2]. Perhaps the best existing online SIMD routing algorithm is to route by sorting: the best sorting algorithms require a factor of 3.5 times the optimal number of data movement steps to complete [34]. These algorithms have more restrictive initial conditions than are necessary for routing, however. The method of [28] uses sorting as a subroutine. Our conclusion was that for the domain of online routing, the existing work was either slower than we would have liked, or too restrictive. We therefore took another approach: We tried to find whether simple algorithms—algorithms with low overhead—could be made to work, and then to evaluate their performance. What our investigation shows is the common situation where algorithms, although very slow in the worst case, can be expected to perform close to optimally in general.

We have developed routing algorithms, analogous to software versions of virtual cut-through and wormhole routing [14, 8], which use those and only those resources available on a variety of processors such as the Massively Parallel Processor (MPP) [5], the ICL DAP [13], the Blitzen [6], the Polymorphic Torus [21], and the Content Addressable Array Parallel Processor (CAAPP) [37]. The models under consideration are presented in section 2. We follow in sections 3 and 4 with a summary of previous work and the requirements and constraints of our investigation. In section 5 we present the algorithms: the basic mesh greedy routing algorithm (MGRA) which runs on a very simple SIMD model, and

modifications to take advantage of such features as local indexing, global count, broadcast buses, and reconfigurable buses. In section 6 we present the theoretical results of this paper, proving correctness, freedom from deadlock, and matching upper and lower bounds for the worst case. There follow the experimental results on two levels of simulation: the coarse level for use in comparing algorithms, and a fine level useful for evaluating the algorithms with respect to both other approaches and changing technology. Finally, in section 8, we tie up some loose ends, sketching extensions to the routing algorithms that improve performance for many-to-one and many-to-many routing, as well as for routing large packets.

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 [5], CLIP-4 [9], DAP [13], Polymorphic Torus [21], CAAPP [37], Blitzen [6], and the MasPar MP-1 [24]. The Mesh With Reconfigurable Buses [26] is primarily a theoretical model, but as it has many features in common with machines being built, we include it in our list as well. These machines all have their particular unique features and combinations of features; however, they also have many characteristics in common.

Our goal is to constuct routing algorithms that run well on the existing and proposed machines in this class, but without making our work specific towards 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 while not excluding 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* of computation. 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 variation.

2. Abstract those features not universally available to the class. Again, the implementation of these features is not identical across all designs possessing them, but we try to model them as generally as possible.

The result is a series of models, none of which matches any single machine precisely, but all of which consist of existing *features* or combinations of features. We first present algorithms for the basic model; later we add modifications that take advantage of the additional features, together with an analysis of their benefit towards routing. Note: Some SIMD arrays possess features that we have not found use for (other than, of course, a global router network, which we are trying to avoid using), e.g. the diagonal connections of the MP-1 X-network.

2.1 Basic Features of SIMD Meshes

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). In the SIMD regimen, PEs execute identical instructions, but operate on local data. Therefore PEs do not contain microsequencers or address decode logic. On the other hand, the resulting simplicity enables the user to apply maximum computing power to problems solvable with relatively simple procedures, but with very large amounts of data. Some standard features of the basic model are as follows.

The PE ALU and Memory

In the basic model, each PE contains a one bit wide ALU with the capability of performing the basic arithmetic and logical operations. Storage consists of some one bit registers and local memory, some of which is on-chip, some off-chip. The off-chip memory takes a factor of ten longer to access because multiplexing is usually required due to limited pin-out.

Branching Support

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. This is the only form of IF THEN/ELSE available on SIMD processors: All PEs that match a condition execute an instruction, those that do not "sit out." For the ELSE case, the activity bit can be inverted and the rest of the PEs execute the other clause. Arbitrarily complicated flow of control can be built up from this primitive.

InterPE Communication

Communication takes place between neighboring PEs through a mesh connected network. The move instructions have as parameters a memory address (or register) and a direction. On

execution, every PE shifts the data in the location specified in the direction specified. This operation can also be viewed as a sliding data plane. We assume wraparound connections.

Feedback to the Controller

Feedback to the controller is essential in any data dependent computation. In the basic model, this takes place through a global OR: a response register from each PE is output and OR'ed with the outputs of that register from all other PEs in the array. This feature is critical for data dependent termination of loops and typically takes only a few cycles.

Local Copy of PE ID

A seemingly minor but very important feature is for each PE to always have access in on-chip memory to a copy of its own ID, by convention in row and column coordinates.

2.2 Features Available on Some SIMD Meshes

Recent SIMD processors have become more complex: as VLSI component sizes have gotten smaller, architects have added new features. Some of the ones that will be used in the routing algorithms are presented here, together with their cost.

Global Count

This is a form of feedback like global-OR, but which returns an integer instead of a Boolean. Machines that have this feature built in hardware are the CLIP-4 and the CAAPP. In the CAAPP, the count operation takes 20 cycles [31], about half the number it takes to move packets from one PE to another.

Local Indexing

Local indexing gives a machine the capability to implement dynamic data structures such as queues and heaps. The Blitzen and MasPar have this capability, although only to a limited extent. In the Blitzen, the address used for indirection must be loaded with a shifter and thus is not useful for general queues. The MasPar has full indirect addressing, but can only use it to access off-chip memory which is ten times slower than on-chip.

Broadcast Buses

The ability to transmit data through direct electrical connections over long distances has been

included in many architectures, starting perhaps with the ILLIAC-III [25]. One manifestation is row and column broadcast buses: PEs initiate a broadcast by writing to a communication register, 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 communications register. (In the model used in this paper, writes onto the same bus by multiple PEs results in the OR of those signals being propagated.) The DAP and CLIP-4 have broadcast buses, other processors have a reconfigurable superset. The cost of the broadcast instruction is dependent on the implementation and the size of the array. However, since we will only be using broadcast buses for transmitting single bits of handshaking information, we can safely assume a number of cycles less than that of an arithmetic instruction.

Reconfigurable Buses

In this variation of the broadcast bus, PEs also control switches which can prevent a signal from propagating further down the bus by creating an open circuit. Thus the broadcast buses can be partitioned. Switches can be loaded like local storage, either from patterns stored in memory, or from data dependent calculations. Machines having this feature are the CAAPP, Polymorphic Torus, and the Mesh with Reconfigurable Buses. 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 ten nearest neighbor moves.

Wider Data Paths

Wider internal data paths are available in the MasPar MP-1, which has a four bit ALU, and the CAAPP, which supports eight bit internal data movement. No SIMD array currently has multi-bit nearest neighbor connections.

3 Requirements and Constraints

The premises are that our applications require on-line routing, and that the target architecture be a mesh-connected array with SIMD control. This domain provides requirements for, and constraints on, our possible routing algorithms.

3.1 Requirements For Applications

Online and Flexible

Since the computations are data dependent, we do not know the patterns in advance. We must also be able to deal with non-uniform communication patterns. We should be able to take advantage of the density and proximity of the communication patterns.

Packets are generally small and of uniform size

On SIMD processors communication is a synchronous instruction (such as the Connection Machine SEND [35]) where the arguments are two arrays mapped to the PE grid: the destination address, and the data. The data is usually a single word, for example a spectral value or a count. When vectors of data need to be transmitted, this usually requires multiple SENDs. In section 8 we present two algorithms to speed this process.

3.2 Constraints From Architecture

Communication is Atomic

Another consequence of SIMD control is that even PEs not involved in communication cannot perform unrelated instructions.

Packets move from PE to PE in their entirety

In meshes, PEs need to process a substantial amount of address information in order to decide where to send packets. Unlike the butterfly, where only the first bit of a packet need be stripped off, the PE must read the entire row or column index to decide what to do. Since the width of the communication links between processors is much smaller than the $\log n$ bits needed for the address, bit-serial routing as in [1], is not advantageous for this model. In section 8 we discuss some of these tradeoffs in more detail.

Queues must be small

The complexity of simulating queues in the SIMD environment is proportional to the size of the queue; therefore the queue size must be very small. (We *do* examine the use of local indexing to implement queues: in that model the queue size is irrelevant, however the memory latency dominates.)

Packet moves are uniform

The fundamental operation in any mesh routing algorithm is moving packets from PE to neighboring PE via the mesh communication network. Since we are moving packets in their entirety, the number of mesh network instructions executed during a packet move will equal the size of the largest packet.

4 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 by a global controller before the start of the packet transfer, then this is called *offline* routing. In *online* routing, decisions of where to send packets next are made locally after the packet has been received by an intermediate PE. In online routing, destination address information must be carried along in the packet.

Inter-PE communication can take many forms: permutations, where each PE sends and receives precisely one message; partial or *sparse* permutations, where some subset of PEs send and receive at most one message; $k - k$ routing, where each PE sends and receives k packets; one-to-many routing (multicasting) where, some small number of PEs send packets to a larger number of possibly overlapping PEs; and many-to-one routing (reduction), where a larger number of PEs sends packets to a smaller number. Most of the published results have concerned themselves with routing permutations, as this is commonly used and the most ammenable to theoretical analysis. One way to route permutations is to sort the packets by destination IDs: thus the complexity of sorting is no more than the complexity of permutation routing.

Much work has been done on the problem of sorting and routing on a mesh. Two models of computation have been used: the MPP model [5], which assumes SIMD processing, and the more general MIMD model. In the former no queues are used; in the latter, queue size becomes a variable to be minimized. In the following discussion, N refers to the total number of PEs, while n is the number of processors in a row or column: $n = \sqrt{N}$ for a two dimensional mesh or torus (i.e. a mesh with wraparound connections). The lower bound on mesh routing is $2n - 2$ on the MIMD model and $4n - 4$ on the SIMD model as this is the minimum number of routing steps needed for processors in opposite corners to exchange packets. The difference occurs because, in the MIMD model, different sets of processors can send packets in different directions on the same time step, while, in the SIMD model, the

direction must be the same for every packet. When the model has wraparound connections, the lower bounds are halved.

SIMD Sorting

The algorithms for sorting on the SIMD model can be divided into two categories, the asymptotically optimal, and the practical. The latter category includes the algorithms with the best performance for $n \leq 512$. Thompson and Kung developed a sorting algorithm called the s^2-way merge that sorts in $6n + o(n)$ communication steps [34], a result that was shown to be optimal by Kunde [16]. However, the low order terms still dominate for $n < 512$. There are many candidates for the most practical algorithm [34, 15, 18, 23], the choice depends on what operations the user is counting. For example, if *compare-and-exchange* is counted as a unit operation, then the algorithm of Ma, Sen, and Scherson [23] with $5.5n$ is best. But each compare-and-exhange is really two nearest neighbor shifts, a compare, and an internal exchange, which itself can be three move operations. Kumar and Hirschberg [15] use $11n$ routing steps, but only $2\log^2 n$ compares and $2.5n$ internal exchanges. The algorithm of Lang et al. [18] uses $7n$ compare-and-exhange operations, but is simple enough to be mapped onto a systolic array. And finally, Thompson and Kung [34] also developed an algorithm based on Batcher's bitonic sort [4] requiring $14n$ routing and $2\log^2$ compare and internal-exchange steps. This last algorithm is a factor of 3.5 from optimal for permutation routing for any n, with no hidden costs. Most of these "practical" SIMD sorting algorithms are based on recursive merging; Kunde has conjectured [16] that the lower bound on this approach is $4.5n$ interchanges, or $9n$ routing steps, bounding the approach to a factor of 2.25 from optimal.

MIMD Sorting

Schnorr and Schamir [32] have developed a $3n + o(n)$ MIMD sorting algorithm, together with a matching lower bound.

MIMD Routing

Since optimal online MIMD routing algorithms exist, offline algorithms are not considered. In the MIMD model, PEs are assumed to be very powerful: usually at least one priority queue operating in a single cycle per PE is assumed. One way to route using this model 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 four with overwhelming probability [20]. The randomized routing algorithm of Valiant and Brebner [36] is an extension of greedy routing. The algorithm consists of three phases: randomize packets within the columns, send packets to correct column along the row, and send packets to correct row along the column. This algorithm results in routing in $\simeq 3n$ steps, but the queue size has been reduced to $O(\log N)$ for *all* permutations with overwhelming probability. Kunde [17] has developed an algorithm for the MIMD model which will route a permutation in $2n + O(\frac{n}{f(n)})$ routing steps, where the queue size $= f(n) < n$, and $n + O(\frac{n}{f(n)})$ when wraparound is allowed. This algorithm has been improved by Leighton, Makedon, and Tollis [19] to achieve $2n - 2$ step routing with constant size queues.

Online SIMD Routing

Unlike the MIMD case, we are not aware of any online SIMD routing algorithm with complexity less than sorting. Nassimi and Sahni used mesh sorting as a subroutine in performing random access read and random access write on distributed memory computers [28]. The significance of these algorithms is that they address one-to-many and many-to-one communication, and are asymptotically optimal.

Another way to perform online routing on the SIMD model is to simulate one of the MIMD algorithms. The multiple communications per time step can be time-sliced with little slowdown, the real difficulty is in simulating queues. The procedures themselves are straightforward, but since local indexing is either not supported or very costly, and all PEs in the array execute the same instructions with *the same operand addresses*, the cost is very large. At every time-step during which a queuing operation takes place, the queue for each PE may be of any size up to the maximum that can occur during the running of the algorithm. For each of these possible pointer positions, the controller must broadcast a separate sequence of instructions. Therefore, if the maximum queue size is, say, $\log n$ for an algorithm requiring n queuing operations, then the complexity *just in servicing the queue* becomes $O(n \log n)$.

To summarize, the on-line SIMD routing algorithms of choice for permutations are probably the most practical sorting algorithms. These require no queues but are at least 3.5 times removed from the lower bound.

Offline Routing

When offline calculation is allowed, better results can be achieved. For example, with $O(\log^2 N)$ preprocessing time, optimal routes can be found for the class of permutations that can be specified by permuting and complementing the bits in the PE ID [27] for meshes with no wraparound. Raghavendra and Prassana Kumar [30] 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].

Cut-through and Worm-hole Routing

In the next section we present methods for routing on the basic SIMD mesh array model. Since these algorithms are in some ways analogous to the cut-through routing technique used in message-passing multicomputers [3], we present some background in that area.

Virtual cut-through was developed by Kermani and Kleinrock [14] as an alternative to routing via circuit switching or store-and-forward packet switching. "When a message arrives at an intermediate node and its selected outgoing channel is free, then the message is sent out to the adjacent node towards its destination before it is received completely at the node; only if the message is blocked due to a busy output channel is a message buffered in an intermediate node." The great advantage of this method is that the overhead of buffering the message at every node is eliminated. Dally and Seitz [8] modify cut-through routing (and rename it wormhole routing): packets are divided into a series of "flits." When the head of a packet is blocked, the rest of the packet is not queued in that intermediate node, rather the trailing "flits" remain where they are, occupying their current channel until they are allowed to continue.

Cut-through routing has the same deadlock properties as store-and-forward routing [8], but this is not the case for wormhole routing. The network can, however, be constructed to prevent deadlock. Take, for example, the two-dimensional torus. Let each processor have four channels, two going left down the rows and two going up the columns, and call these channels X1, X2, Y1, and Y2, respectively. When a packet is sent out, it traverses channel X1 until it either wraps around, or reaches the correct column. In the first case the packet switches to channel X2, in the second to channel Y1. It then continues in the same manner until the correct row is reached. Since each packet proceeds monotonically, this forms a total ordering of the packets in the system, thus preventing the occurrence of a cycle, and thus

preventing deadlock.

5 Greedy Routing Algorithms

Two requirements presented earlier are seemingly contradictory: the need to transfer entire packets from PE to PE as in store-and-forward routing, and the very small queue size of wormhole routing. We have developed new routing algorithms for SIMD meshes based on a combination of these methods: Because simulating 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 wormhole routing: Trains of data proceed through the network until the head of the train is blocked; at that point the entire train waits until the path is clear.

5.1 The Basic Algorithm on the Basic Model

The basic algorithm is called the "mesh greedy routing algorithm" or MGRA. Every PE simulates two channels, X and Y, by using the nearest neighbor mesh. Physically, the channels consist of a number of bits 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. The algorithm runs as follows: A PE sends a packet along the X-channel a distance of one PE per routing step, until the correct X coordinate (column) is reached. At this point the PE moves the packet from the X-channel to the Y-channel. The packet then continues along the Y channel until the destination is reached. The X- and Y-moves are interleaved so that each occurs on every time step. 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 permit deadlock. If the packet has reached the correct X coordinate but the Y-channel 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.

The critical question in this algorithm is how to inform those packets contiguous to and behind a blocked packet that they too are blocked. Using the most naive method in the basic model, this notification step requires n steps, the maximum possible number of blocked packets in a channel, and would yield a $\theta(N)$ algorithm. How we solve this problem

is the key difference between the implementations of the MGRA in the different models.

In the basic model we modify the MGRA as follows: we add another buffer to the X-channel, making it possible to simulate a queue of size two. 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 X-channels by sliding memory planes, we move packets from the X-head to the X-tail of the neighbor, and then from X-tail to X-head internally. Of course only PEs where the X-tail of the neighbor or X-head internally are clear send the packets. We demonstrate the correctness of this algorithm later, intuitively the "blocked" information travels back down the train at the same rate that incoming packets become compressed in the succeeding queues. We now present in detail one iteration of the MGRA. Matching pseudo-code can be found in Figure 1.

1. If there is a packet in the Y-channel whose address matches the address of the PE, then the packet has arrived. Move the data in the packet to the output buffer and clear the Y-channel.

2. Get the packet in the Y-channel from the south neighbor. If there is none, then a null packet will be received.

3. If there is a packet in the X-head having the same X-address as the PE, and if the Y-channel is clear, then shift the packet from the X-channel to the Y-channel and clear the X-head. If the Y-channel is not clear, then set the "blocked bit."

4. If X-tail is empty, and the east PE is not blocked, the PE reads the X-head of the east neighbor into X-tail.

5. Each PE with an empty X-head moves the packet in X-tail into X-head.

After each iteration, the controller checks to see if there are packets remaining in the X- and Y-channels. The controller stops executing steps 3-5 when no packets remain in the X-channel. The entire algorithm terminates when the controller detects no packets remaining.

5.2 Using Four Channels

One way to cut down on the number of packets that are blocked is to double the number of channels simulated so that the scheme resembles more closely that in [8]; in this case the

```
1. IF YPacket.InUse AND YPacket.YAddress = PE.YAddress
        THEN Output := YPacket.Data
             YPacket.InUse := False

2. YPacket := South(YPacket)

3. IF XHeadPacket.InUse AND XHeadPacket.XAddress = PE.XAddress
        THEN IF ¬YPacket.InUse
                THEN YPacket := XHeadPacket
                     XHeadPacket.InUse := FALSE
                ELSE Blocked := TRUE

4. IF ¬XTailPacket.InUse AND ¬East(Blocked)
        THEN XTailPacket := East(XHeadPacket)

5. IF ¬XHeadPacket.InUse
        THEN XHeadPacket := XTailPacket
```

Figure 1: The Mesh Greedy Routing Algorithm on the basic model

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. In the next section data will be presented showing that the MGRA has never, in practice, required more than $3n$ iterations, less than the $4n$ required to make doubling the channels in this manner worthwhile.

If, however, the additional channels are used to route packets in the opposite directions of X1 and Y1, respectively, then an algorithm has been created that is bounded by the minimum routing distance from source to destination. The SIMD simulation of the channels is slightly more than doubled, though, as there are now 4 ways that X- and Y-channels can interact rather than 1. We call this variation the 4-channel MGRA, and find that it is useful in routing permutations where the maximum distance is somewhat less than $n/2$. Again, this algorithm does not deadlock because the packets are transferred in their entirety.

5.3 Using Queues

It is possible to implement the MGRA with FIFO queues of arbitrary length, depending only on the memory capacity of each PE. 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.

Basic model

Queues can be simulated in several ways. However, when a queue size greater than two (all we needed above) must be supported, the algorithms are slightly more complicated.

With no indirect addressing, the data structures used are an array with an array with p slots where the information is stored, and a bit vector indicating the position of the current head and tail. Ignoring the details of implementing a circular buffer, queue (and dequeue) operations require p steps as each possible pointer position must be operated upon in turn. Some reduction in complexity is possible by using global OR circuitry: only those slots where the corresponding bit-vector bit is set need to be tried. The operations are then: For all p, if corresponding head bit is set, move the packets from the queue to an intermediate buffer. Transfer the packet to the neighboring PE. Then, for all p, if the corresponding tail bit is set, move the packet to that queue slot. The bit vectors must be updated accordingly.

A somewhat simpler method using no bit vector is to simply keep the queue "justified" to one end of the buffer. Assume that the X-channel has slots $X_1, \ldots, X_p$. In this mode, instructions 3-5 of the MGRA look like this (matching pseudo-code can be found in Figure 2):

3. If there is a packet in the X_1 having the same X-address as the PE, and if the Y-channel is clear, then shift the packet from X_1 to the Y-channel and clear X_1. If the Y-channel is not clear, then set the "blocked bit." If a packet switched channels, the remaining packets within the X-channel are rejustified.

4. Each PE with an empty X_p, and whose east neighbor is not blocked, reads X_1 of the east neighbor into X_p.

5. Move the newly arrived tail packet to the lowest empty queue slot.

Local Indexing

If local indexing is available, then use the standard technique. As mentioned earlier, however,

```
1. IF YPacket.InUse AND YPacket.YAddress = PE.YAddress
        THEN Output := YPacket.Data
                YPacket.InUse := False

2. YPacket := South(YPacket)

3. IF XPacket.1.InUse AND XPacket.1.XAddress = PE.XAddress
        THEN IF ¬YPacket.InUse
                THEN YPacket := XPacket.1
                        XPacket.1.InUse := FALSE
                        FROM i := 1 TO p - 1
                                XPacket.i := XPacket.i+1
                        XPacket.p.InUse := FALSE
                ELSE Blocked := TRUE

4. IF ¬XPacket.p.InUse AND ¬East(Blocked)
        THEN XPacket.p := East(XPacket.1)

5. FROM i := p-1 TO 1
        IF ¬XPacket.i.InUse
                THEN XPacket.i := XPacket.i+1
```

Figure 2: The Mesh Greedy Routing Algorithm simulating p length queues on the basic model

the queue and dequeue operations are substantially slower (about a factor of 10) than the PE to PE move operations, so even if available, this method should only be used if the queue size is greater than the relative slowdown in memory access.

5.4 Using Broadcast Buses

In this model, the X-channels only contain one queue slot. This has two advantages: one is that there is less overhead in that internal moves are saved, the other is that it is faster in some models to transfer data using nearest neighbor moves when the source and destination addresses are the same. The question is again how to keep from overwriting packets that are blocked because of occupied Y-channels. Since a packet can only be overwritten by another packet in an X-channel, packets broadcast their blocked status to their rows. If any packet in a row is blocked, then no packet in that row proceeds. We replace steps 3-5 of the original algorithm as follows, matching pseudo-code can be found in Figure 3.

3. If there is a packet in the X-channel having the same X-address as the PE, and if the Y-channel is clear, then shift the packet from the X-channel to the Y-channel and clear the X-channel. If the Y-channel is not clear, then set the "blocked bit".

4. Each PE broadcasts the blocked bit to the bus, and then reads the bus and sets its own blocked bit accordingly.

5. The PEs in X-channels that are not blocked perform a circular shift.

5.5 Using Reconfigurable Broadcast Buses

The obvious disadvantage of the above method is that some packets are needlessly prevented from proceeding. By adding reconfigurability to the broadcast bus model, we block 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 is blocked, then the West switch is opened. In this way circuits are formed along 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

```
1. IF YPacket.InUse AND YPacket.YAddress = PE.YAddress
        THEN Output := YPacket.Data
             YPacket.InUse := False

2. YPacket := South(YPacket)

3. IF XPacket.InUse AND XPacket.XAddress = PE.XAddress
        THEN IF ¬YPacket.InUse
                THEN YPacket := XPacket
                     XPacket.InUse := FALSE
                ELSE Blocked := TRUE

4. Switches(North,East,South,West) := (OPEN,CLOSED,OPEN,CLOSED)
   Blocked := Broadcast(Blocked)

5. IF ¬Blocked
        THEN XPacket := East(XPacket)
```

Figure 3: The Mesh Greedy Routing Algorithm using broadcast buses

be in the leftmost PE. See Figure 4 for an illustration. Retaining step 3 from above, steps 4 and 5 are replaced as follows. Matching pseudo-code can be found in Figure 5.

4. Each PE containing a packet in its X-channel closes its left and right switches, while all PEs open their up and down switches. If the PE is blocked, then the left switch is opened. In this way coteries are formed along horizontal buses of contiguous PEs containing packets in their X-channels; if there is a blocked packet within the coterie, it will be in the leftmost PE. Each PE broadcasts its "blocked" bit; on the next instruction it reads the wired-OR of this broadcast and sets its own "blocked bit" accordingly.

5. Each PE not blocked reads a packet from the X-channel of its east neighbor.

5.6 Transmitting Packets via Broadcast

If you have reconfigurable buses, you can also simulate the aspect of the cut-through/wormhole routing that involves sending each packet to the furthest available PE in each time step. When the communication pattern is sparse, the larger overhead is more than compensated by the decreased number of iterations.

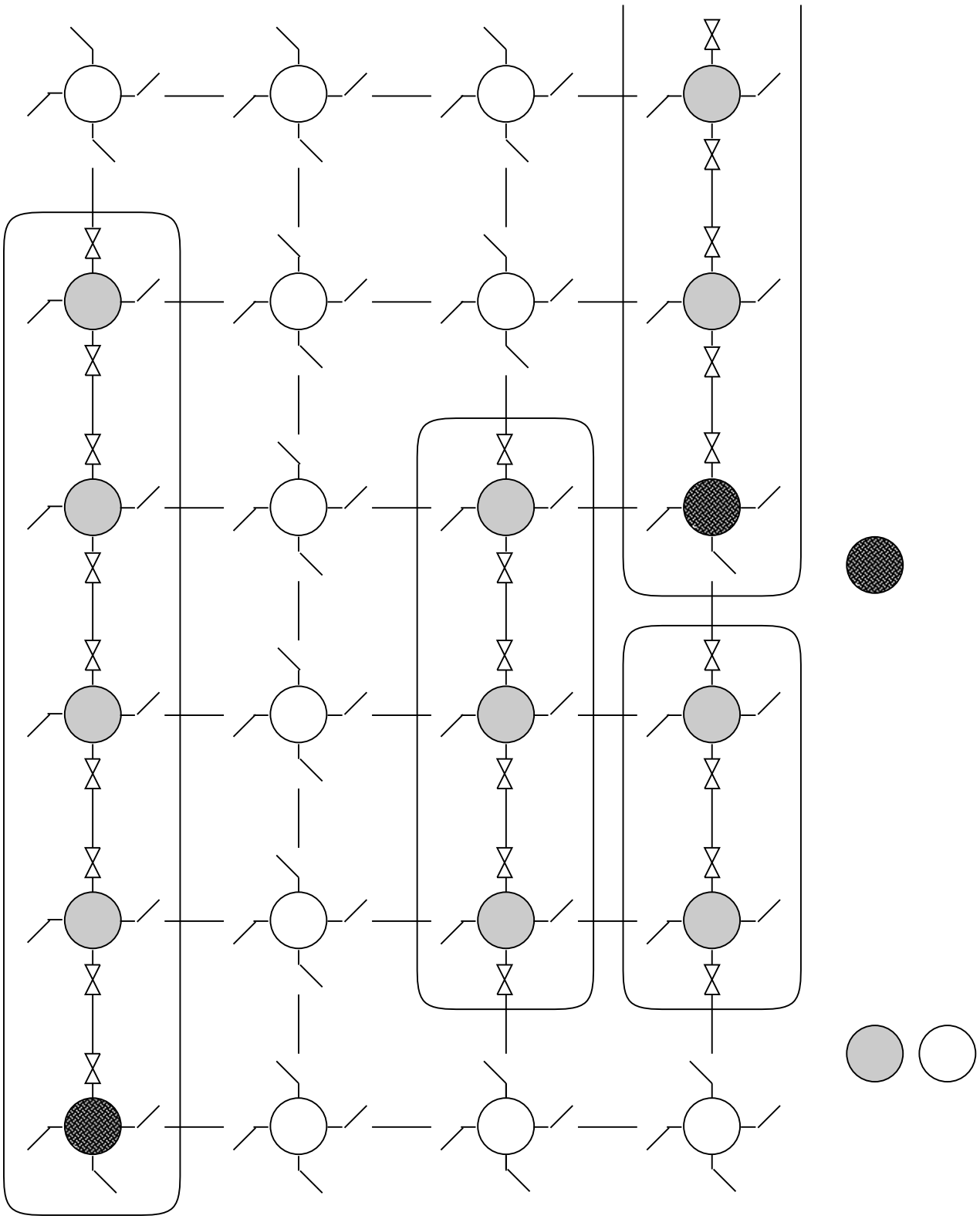

Figure 4: Reconfigurable buses are used to form circuits containing exactly those PEs with blocked packets. After broadcast, the PEs in circuits 1 and 4 will be blocked while the PEs in circuits 2 and 3 will contiue.

```
1. IF YPacket.InUse AND YPacket.YAddress = PE.YAddress
        THEN Output := YPacket.Data
                YPacket.InUse := False

2. YPacket = South(YPacket)

3. IF XPacket.InUse AND XPacket.XAddress = PE.XAddress
        THEN IF ¬YPacket.InUse
                THEN YPacket := XPacket
                        XHeadPacket.InUse := FALSE
                ELSE Blocked := TRUE

4. IF XPacket.InUse AND Blocked
        THEN Switches(N,E,S,W) := (OPEN,CLOSED,OPEN,OPEN)
   ELSE IF XPacket.InUse AND ¬Blocked
        THEN Switches(N,E,S,W) := (OPEN,CLOSED,OPEN,CLOSED)
   ELSE
                Switches(N,E,S,W) := (OPEN,OPEN,OPEN,OPEN)
   Blocked := Broadcast(Blocked)

5. IF ¬Blocked
        THEN XPacket := East(XPacket)
```

Figure 5: The Mesh Greedy Routing Algorithm using reconfigurable broadcast buses

In this version of the greedy algorithm, we call the Coterie Greedy Routing Algorithm (CGRA)[1], the X- and Y-channels are simulated 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 (bottom-most if these are Y-channels) contains a packet, while all other PEs in the circuit do not (see Figure 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 details of the algorithm follow, matching pseudo-code can be found in Figure 7.

1. If there is no packet in the Y-channel between a packet and its destination, then send the packet directly there. This is accomplished as follows: Each PE opens its "down" link if the Y-channel is occupied. This creates coteries consisting of the PE with the packet and the open space up to the next packet. The PEs with the packets then broadcast the Y destinations to their coteries. If the destination PE recognizes its address, it prepares itself to receive the packet. Packets are then broadcast to their destinations. The sending PE must be notified that its packet was received so that it will clear its Y-channel. Therefore, each destination PE sends a bit back down the coterie with this information, which is received by the sending PE. Also, so that packets do not overshoot their destinations, step 1 is now repeated.

2. If there is a packet in the Y-channel whose address matches the address of the PE, then the packet has arrived. Move it to the output buffer and clear the Y-channel.

3. Send the Y-channel packet as far as possible, that is, north to the empty PE just "south" of the next occupied PE. The procedure here is similar to that used in step 2, the difference being how the receiving PE is determined. In this step each PE examines its north neighbor to see if it is occupied. If yes, then the PE knows that it is at the end of a coterie and that broadcast packets are meant for it. All sending PEs, which are not also receiving PEs, update their Y-channels.

4. Move the Y-channel packet one step as in step number 2 in the MGRA. This guarantees

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

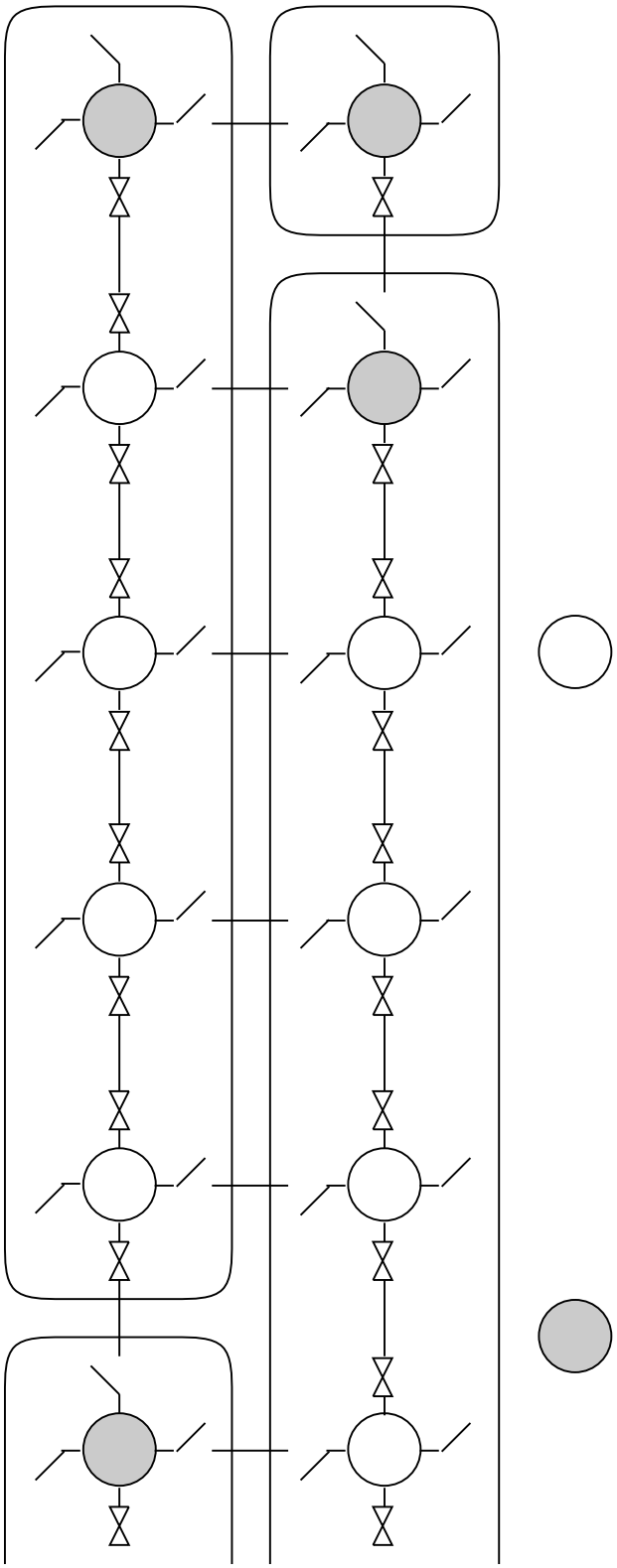

Figure 6: In CGRA steps 1 and 3 (5 and 8) PEs broadcast packets directly to the destination PE, or to the furthest possible PE with an open channel, respectively.

progress if the route is dense.

5. Repeat step 1 of this algorithm for packets in X.

6. If there is a packet in the X-channel that has the same X-address as the PE, and if the Y-channel is clear, shift the packet from the X-channel to the Y-channel and clear the X-channel. If the Y-channel is not clear, then set the "blocked bit".

7. Broadcast "blocked bit" in same way as in step 4 in the reconfigurable bus version of the MGRA.

8. Repeat step 3 for packets in X that are not blocked.

9. Repeat step 4 for packets in X that are not blocked.

6 Theoretical Results

6.1 Correctness and Freedom from Deadlock

We present an informal argument for the correctness of MGRA. Clearly if a packet does not get permanently blocked it will proceed across the row in which it starts to the column of its destination, switch from the X channel to the Y channel, and then proceed 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 propogation of a single stall point are shown in Figure 8. Notice 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

```
1. IF YPacket.InUse
        THEN Switches(N,E,S,W) = (CLOSED,OPEN,OPEN,OPEN)
   ELSE
                Switches(N,E,S,W) = (CLOSED,OPEN,CLOSED,OPEN)
   IF PE.YAddress = Broadcast(YPacket.YAddress)
        THEN YPacket := Broadcast(YPacket)

2. IF YPacket.InUse AND YPacket.YAddress = PE.YAddress
        THEN Output := YPacket.Data
                YPacket.InUse := False

3. IF North(YPacket.InUse) AND ¬YPacket.InUse
        THEN YPacket := Broadcast(YPacket)

4. YPacket = South(YPacket)

5. IF XPacket.InUse
        THEN Switches(N,E,S,W) = (OPEN,OPEN,OPEN,CLOSED)
   ELSE
                Switches(N,E,S,W) = (OPEN,CLOSED,OPEN,CLOSED)
   IF PE.XAddress = Broadcast(XPacket.XAddress)
        THEN XPacket := Broadcast(XPacket)

6. IF XPacket.InUse AND XPacket.XAddress = PE.XAddress
        THEN IF ¬YPacket.InUse
                THEN YPacket := XPacket
                        XPacket.InUse := FALSE
                ELSE Blocked := TRUE

7. IF XPacket.InUse AND Blocked
        THEN Switches(N,E,S,W) := (OPEN,CLOSED,OPEN,OPEN)
   ELSE IF XPacket.InUse AND ¬Blocked
        THEN Switches(N,E,S,W) := (OPEN,CLOSED,OPEN,CLOSED)
   ELSE
                Switches(N,E,S,W) := (OPEN,OPEN,OPEN,OPEN)
   Blocked := Broadcast(Blocked)

8. IF West(XPacket.InUse) AND ¬XPacket.InUse AND ¬Blocked
        THEN XPacket := Broadcast(XPacket)

9. IF ¬Blocked
        THEN XPacket := East(XPacket)
```

Figure 7: The Coterie Greedy Routing Algorithm: using reconfigurable broadcast uses to transmit data

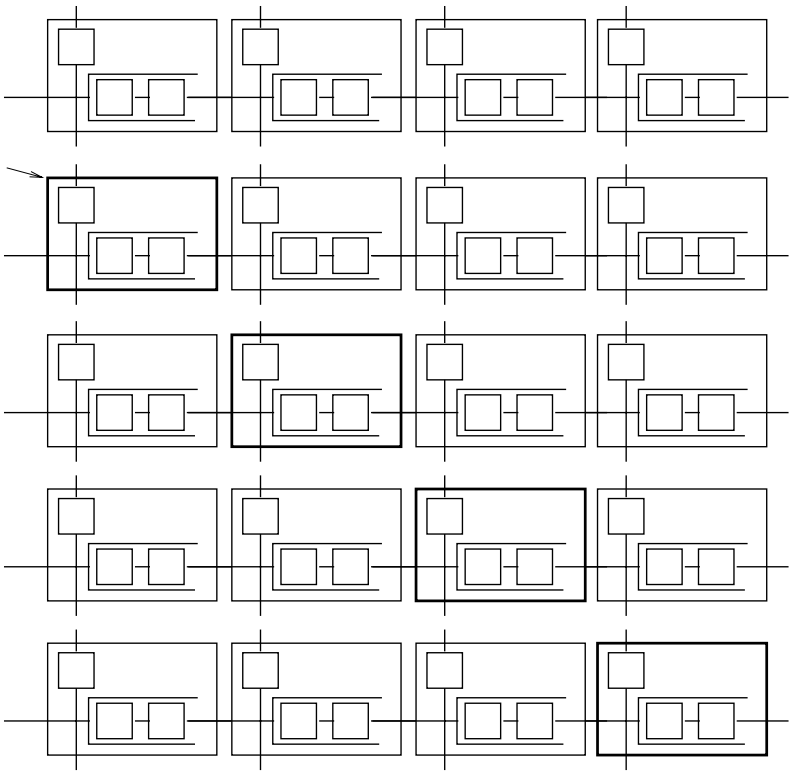

Figure 8: Stall point progression

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 in to 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 [8].

6.2 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 $\pi : M \rightarrow M$ 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 only create stall points when in the Y channel thus 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 $\pi : M \rightarrow M$ 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 Figure 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.

				d_1	c_1	b_1	a_1								
				d_2	c_2	b_2	a_2								
				d'_1	c_3	b_3	a_3								
					c_4	b_4	a_4								
					c'_2	b_5	a_5								
					c'_1	b_6	a_6								
						b'_3	a_7								
						b'_2	a_8								
						b'_1	a'_4								
							a'_3								
							a'_2								
							a'_1					a'_4	b'_3	c'_2	d'_1
											a'_3	b'_2	c'_1	d_1	d_2
										a'_2	b'_1	c_1	c_2	c_3	c_4
									a'_1	b_1	b_2	b_3	b_4	b_5	b_6
								a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8

Figure 9: Permutation causing a quadratic amount of blocking for $n = 16$

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 $\left\lfloor \frac{n}{4} \right\rfloor + 1$ rows in the lower right quadrant of the mesh and $\left\lfloor \frac{n}{4} \right\rfloor$ columns in the left half of the mesh as follows. Number the rows $1, 2, 3, \ldots, \left\lfloor \frac{n}{4} \right\rfloor + 1$

starting from from the bottom of the mesh; number the columns in the left half of the mesh $1, 2, 3, \ldots, \left\lfloor \frac{n}{4} \right\rfloor$ from the right (i.e., the column just to the left of center is column 1). The i^{th} row ($i = 1, \ldots, \left\lfloor \frac{n}{4} \right\rfloor + 1$) starts with $\left\lfloor \frac{n}{2} \right\rfloor + i - 1$ 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 $1, 2, 3, \ldots, i-1$, in that order (these correspond to the primed elements in the example above), followed by $\left\lfloor \frac{n}{2} \right\rfloor - 2(i-1)$ elements routed to column i. The labeled elements of row i will create at least $\left\lfloor \frac{n}{2} \right\rfloor - 2(i-1) - 1$ stall points in all rows $j = i+1, \ldots, \left\lfloor \frac{n}{4} \right\rfloor + 1$, thus

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

stall points will pass through the last element of row $\left\lfloor \frac{n}{4} \right\rfloor + 1$. □

Dealing with the Worst Case

Clearly the worst case performance in unacceptable in a practical routing scheme. In the next section we present experimental results that indicate that finding 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 reliable performance.

Valiant and Brebner [36] 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.

6.3 Expected Case

Lower Bound

To obtain a lower bound for the MGRA in the expected case, we must characterize the

packet that takes the longest to complete. Since the fastest that a packet can traverse the distance to its destination is one PE per time-step, this problem is the same as finding the expected greatest distance any packet must travel. Looking at one dimension at a time, a packet starts at a random distance between 0 and $n-1$ from its destination. If n is assumed to be large, say greater than 100, then the probability P_s that at least one element starts at least $n-s$ away from its destination is independent of n. Further, P_s can be found using the Poisson distribution. P_s approaches 1 rapidly: For example P_5 is less than 1% away from 1, and P_{10} is less than .005% away from 1. In other words, the chances are extremely good that at least one packet will need to travel nearly the entire distance n. Similar reasoning extends the result to two dimensions, yielding a lower bound of the MGRA expected value equal to the diameter of the mesh, or $2n$.

Upper Bound

The difficulty with finding an upper bound for the MGRA in the expected case is in characterizing the interaction between row and column, making the use of standard combinatoric arguments problematic. In [10] we outline an alternate approach, modeling the process using differential equations; that result supports the experiments presented in the next section.

7 Experimental Results

In this section we present the practical results of this paper and thereby determine the relative merits of the various algorithm/architecture combinations.

7.1 Methodology

Types of Simulation

Two levels of simulation were used: the *coarse* simulation counts only the number of iterations an algorithm needs to run to completion for a given communication pattern; the *fine* simulation counts actual machine cycles and thus provides an execution time. Both methods are necessary: fine simulation is very sensitive to architectural implementation details and so is only used to provide a mechanism for evaluating the entire MGRA/CGRA approach; coarse simulation—plus a count of the number of standard operations making up each iteration—provides the comparison of architectural features and algorithms within the

approach.

Choice of Communication Patterns

We wish to test our algorithms on the most realistic patterns possible. However, selecting those likely to occur in online computing is difficult because there are as yet so few systems in operation and therefore few available traces. On the other hand, common performance evaluation techniques usually assume a steady-state system and asynchronous communication, whereas we are interested in synchronous, atomic communication.

In evaluating routing algorithms, random permutations are often used. But even though random permutations are important (e.g. arising after the randomization preprocessing phase), they are generally among the "easiest" patterns to route. This would indicate that we should always include a randomizing preprocessing phase (as in Valiant and Brebner [36]). However, this incurs overhead (a substantial fraction of the running time of the algorithm), and is also not helpful in many important cases, such as when the maximum distance any packet must travel is much smaller than the size of the array. We therefore test our algorithms not only on random permutations, but on many other patterns (and classes of patterns) found in the literature. The advantages of this approach are that it allows us to test the algorithms on patterns that are both more realistic and that cause more congestion. An example of the latter is restricting the permutations to subsets of permutations on the ID bits, an example of the former is to route patterns where only a small number of PEs are transmitting data. This solution is obviously not perfect, but we hope to show in this section that the results obtained are so consistent as to provide adequate support for the conclusions following.

Pattern Definitions

We divide patterns into two categories: (1) *particular* patterns such as transpose and shuffle, and (2) *classes* of patterns defined as a rule for generating patterns over which a mean and standard deviation are taken. Some particular patterns used are based on permutations and complement of bits in the PE ID. These are known as Bit Permute/Complement permutations, or BPC's, and are described in [27]; we use the same notation (see Figure 10).

The definitions of some classes of patterns are as follows:

- Bit Permute: patterns derived from randomly permuting the bits of the PE ID

Name	Formulation
Bit Reverse	[0,1,...,p-1]
Unshuffle	[p-2,p-3,...,0,p-1]
Shuffle	[0,p-1,...,1]
Transpose	[p/2-1,...,0,p-1,...,p/2]
Shuffled Row-Major	[p-1,p/2-1,...,p/2,0]
Bit Shuffle	[p-1,p-3,...,1,p-2,...,0]
Vector Reverse	[-(p-1),-(p-2),...,-0]

Figure 10: Definitions of some BPC permutations

- Bit Permute and Complement: patterns derived from randomly permuting and flipping the bits of the PE ID
- P-ordered Vectors(see [33]): we use permutations for all relatively prime values of P less than 256
- Image rotation: patterns derived using the standard matrix; rotation in general produces a many-to-one communication pattern

Results for classes of patterns refer to runs of at least 100 trials, except for rotation which was done for every 5 degrees (72 trials).

7.2 Iteration-Level Simulation

All experiments in this section unless otherwise specified were run on an $n \times n$ array where $n = 256$. For most patterns there exists a packet that must travel nearly the maximum distance and so usually the minimum number of nearest neighbor moves (and therefore iterations of the MGRA) is $2n$ or 512.

Experiment 1
How does the basic two channel MGRA perform on various communication patterns? See results in Figures 11 and 12.

Note 1: All particular patterns except Shuffled Row Major and Bit Shuffle do not block at all.

Pattern	# of Iterations	blocks?
Bit Reverse (FFT)	498	No
Unshuffle	512	No
Shuffle	512	No
Transpose	258	No
Reflection in X-axis	257	No
Reflection in Y-axis	257	No
Vector Reverse	512	No
Shuffled Row Major	664	Yes
Bit Shuffle	758	Yes
Snake-like Row-major	257	No
Snake-like Column-major	511	No
$90^\circ * k$ rotation	511	No
$180^\circ * k$ rotation	512	No
$270^\circ * k$ rotation	511	No

Figure 11: Number of iterations required for particular permutations in 256×256 array: MGRA on the basic architecture

	Mean # of Iterations	Standard Deviation	Worst Case
Random	524.65	3.76	539
Random BP	611.56	83.61	780
Random BPC	614.56	80.94	798
Rotation	631.57	96.84	827
P-Vector	511.10	19.12	761

Figure 12: Number of iterations, variance, and worst case for classes of permutations in a 256×256 array: MGRA on the basic architecture

Note 2: Random permutations require a only slightly greater than optimal number of iterations, with very small variance.
Note 3: The "hardest" patterns tried require only a factor of 1.5 from optimal number of iterations.
Note 4: Additional experiments (not presented here) support these results: Trials were run for $n = 4, 8, 16, \ldots, 256$ on all particular permutations with similar results. For random permutations, many thousands of trials were run for many additional n values up to 512.
Note 5: The factors of 45° are the worst cases of all rotations tested ($0° \ldots 360°$ in increments of 5°).

Experiment 2
How do the other algorithms perform on those same communications patterns? Since the number of iterations is the same for permutations that do not block, only the classes of patterns were used. See Figure 13 for results.

	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

Figure 13: Number of iterations required for classes of permutations in 256×256 array. σs similar to table 3.

Note 1: Increasing the length of the FIFO queue in each PE from two to infinite does not significantly reduce the number of iterations required.
Note 2: Reducing the queue length from two to one and using reconfigurable buses to block only the necessary packets does not significantly increase the number of iterations required.
Note 3: When the non-reconfigurable buses are used to block all packets in a row where any packet is blocked, the increase in the number of iterations varies from 5% to 20%.
Note 4: The use of broadcast to transmit packets (CGRA) reduces the number of iterations required by 45%-55%.
Note 5: The four channel version of the MGRA reduces the number of iterations by slightly more than 50%. This is because the maximum number of times any packet is blocked is

reduced.

Experiment 3

The Four Channel MGRA routes packets via shortest paths. How does the number of iterations relate to the longest of these paths? The Four Channel MGRA was run on random permutations where the maximum manhattan distance any packet must travel has been restricted to a specified value. See Figure 14 for results.

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

Figure 14: The performance of the Four Channel MGRA depends almost entirely on the maximum Manhattan distance any packet must travel. First the diameter and then the maximum distance are varied

Note: The number of iterations depends almost entirely on the maximum distance a packet must travel, and is (in this case) independent of the diameter of the network.

Experiment 4

How does the CGRA perform when the density of the communication pattern is decreased? We use random permutations with a selected proportion of the packets removed at random. See graphs in Figures 15 and 16 for the results.

Note 1: The number of iterations required decreases steadily with the density of communi-

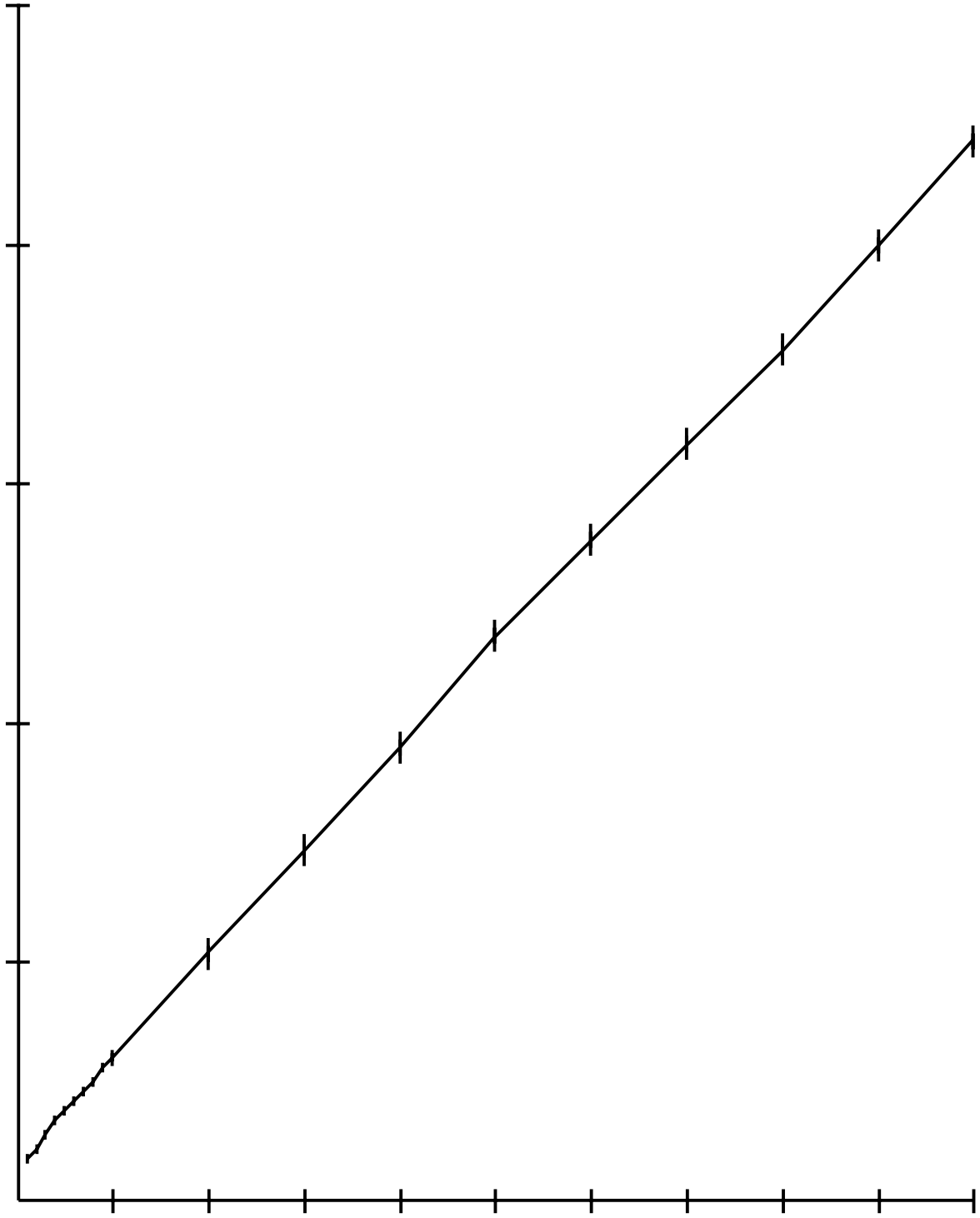

Figure 15: The number of iterations required for the CGRA to complete decreases nearly linearly with respect to the density of the communications pattern. Results for random permutations on a 256×256 array.

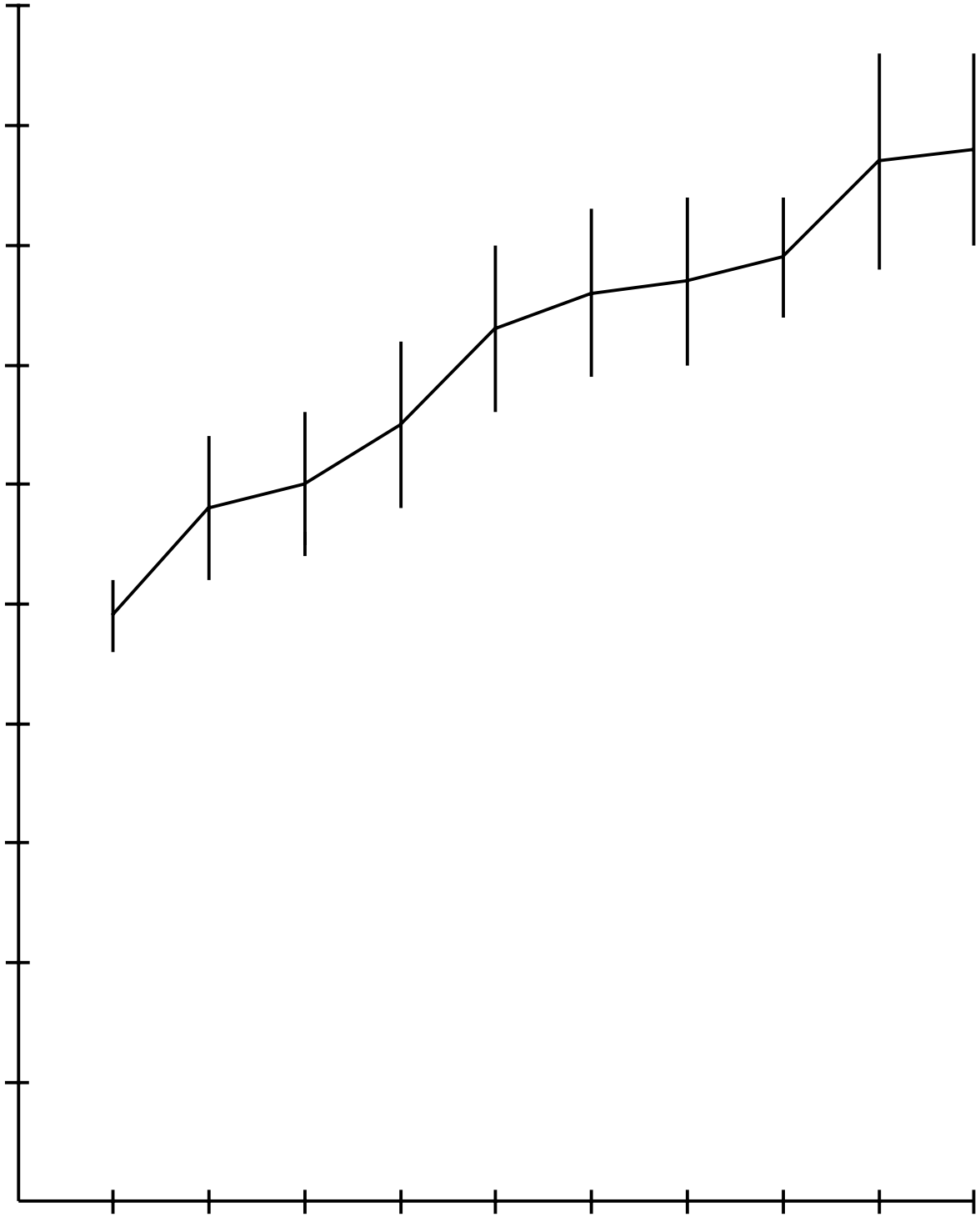

Figure 16: Due to the pipeline nature of the CGRA, the minimum number of iterations possible is four. This value is approached as the density is decreased further.

cation.

Note 2: The minimum number of iterations required to route a packet (not starting at the correct row or column coordinates) using the CGRA is four. Therefore, when the density is very small, only very few iterations more than the minimum possible are required.

Experiment 5

Certain patterns require significantly more iterations than others, the difference varying from slightly more than $2n$ for random permutations, to around $3n$ for certain BPCs. Although randomization requires n moves, these can be calculated offline and so have significantly less overhead (no compares) than an iteration of the MGRA. Therefore, a significant reduction in iterations could make randomization in one axis worthwhile. See Figures 17 and 18 for results.

	MGRA	Standard Deviation	MGRA + Randomization	Standard Deviation
Random BP	611.56	83.61	567.21	45.92
Random BPC	614.56	80.94	570.90	48.43
Rotation	631.57	96.84	555.25	70.49
Random	524.65	3.76		

Figure 17: Number of iterations required for classes of permutations using the MGRA in 256×256 array: Comparison with randomized axis version

	4C MGRA	Std. Deviation	4C MGRA + Randomization	Std. Deviation
Random BP	303.14	42.66	260.53	9.43
Random BPC	306.77	48.11	261.49	3.89
Rotation	264.56	99.30	246.53	37.54
Random	260.04	1.57		

Figure 18: Number of iterations required for classes of permutations using four channel MGRA in 256×256 array: Comparison with randomized axis version

Note 1: For the two and the four channel MGRA, randomization reduces the number of iterations and the standard deviations for the permutations tested.

Note 2: For the four channel MGRA, randomization yields performance nearly identical to that of performance on random permutations. For the two channel MGRA, the performance is about half way between that of the original and the random.

7.3 Overhead per Iteration per Algorithm

In this section we estimate the overhead of the various algorithms. Although the overhead is dependent on design features that differ from machine to machine and that are changing rapidly with technology, finding it is essential in order to carry out meaningful comparisons among the algorithms. Our method is to make assumptions similar to those made in analyzing serial algorithms, that is, to compare numbers of instructions of various kinds that occur in each iteration of each algorithm. Within this constraint, differences in hardware implementation such as number of memory accesses per cycle, and width of internal memory paths cause a uniform consequence across the algorithms. Once the overhead has been found, comparison is simply a matter of combining those counts with the results of the previous section. We make the following assumptions:

- Compare, Internal Move, and Nearest-Neighbor Move are all similar (within a factor of two of each other) in execution time.
- A Broadcast Move takes about five to ten times longer than a move.
- Queue operations take about five to ten times longer than a move when indirect addressing is restriced to referencing only off-chip memory. For PEs allowing indirect addressing of on-chip memory, queue operations will have similar execution time to a local memory reference.

See Figure 19 for a rough comparison of overhead of the various algorithms.

Note 1: 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.
Note 2: The four channel version of the MGRA obviously needs twice the overhead of the two channel version. In addition, it requires extra internal moves to deal with the fact that a packet from either X-channel can be moved to either Y-channel.
Note 3: The broadcast and reconfigurable broadcast bus algorithms have similar overhead. The reconfigurable bus version does require a few extra cycles per iteration, but this does not significantly change the relative overhead.

	Internal Moves	Compares	Neighbor Moves	Queue Operations	Broadcast Moves	Overhead at Startup
MGRA	3-5	2	2	0	0	2
4 Chan 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

Figure 19: Overhead comparisons: (1) cost per iteration in terms on numbers of operations of different types, (2) cost of startup overhead in number of operations

Note 4: FIFO queues replace the internal moves with queue and dequeue operations. Since the rest of the algorithms are similar, the difference in execution times with the other algorithms will be due to the difference in the times between these and the internal move operations.
Note 5: The CGRA has about eight times the overhead as the MGRA.

7.4 Conclusions from Coarse Simulation

1. The greedy routing algorithms are all nearly optimal in terms of number of iterations with respect to different types of communication patterns.

2. The four channel version is nearly optimal with respect to the maximum distance any packet must travel. Therefore the number of iterations is small when the maximum distance is small. In general, however, the four channel MGRA requires slightly fewer than half the iterations of the two channel version. The difference, however, is not quite enough to make up for the additional overhead making the two channel version the default algorithm.

3. FIFO queues of size greater than 2 do not help.

4. Broadcast buses do help: the reduction in overhead compensates for the increase in iterations.

5. Reconfigurable buses help even more: the reduction in iterations with respect to broadcast buses compensates for the very slight increase in overhead, while the slight increase in number of iterations over the MGRA on the basic model is compensated by the lower overhead.

6. Using broadcast to transmit packets in the reconfigurable bus model helps when the density of the communication pattern is small. When the pattern is very sparse, the number of iterations is very small.

7. Randomization does not increase performance enough for the communication patterns tried to make it a standard feature. Although the number of iterations is reduced, the difference is not enough to make up for the overhead of the preprocessing phase. However, the standard deviations were reduced significantly and so randomization should be used if more predictable performance is required.

8. The MGRA, 4 Channel MGRA, CGRA, and Randomized MGRA are all faster for certain communication patterns. Automating the online choice among the first three algorithms is straightforward if a global count is available to obtain the density of the communication pattern and the maximum distance any packet must travel. Determining when a pattern will cause extreme congestion is much more difficult. Therefore use of the randomized MGRA should be determined *a priori*.

7.5 Fine Simulation Results

In order to show the practicality of our approach, to compare it with other available methods, requires simulation in more detail. Of course fine simulation is problematic because costs will vary from machine to machine and will quickly be dated because of rapidly changing technology. Some of the critical factors are:

- Width of ALU/internal memory paths
- Width of nearest neighbor connections
- Number of memory accesses per cycle

- Is data sent from memory to memory in a single cycle in nearest neighbor moves, or must it first go through an intermediate register?
- Does data move from queue to queue in one operation, or must it go from queue to memory and then memory to queue?
- Memory (not)addressable on bit boundaries
- Number of activity bits

To try all the different permutations of possible characteristics is obviously futile. Instead, we present sample results from experiments run on the 256×256 CAAPP simulator [38] assuming different levels of hardware: the current configuration, a configuration less powerful than the current one, and versions with several potential enhancements. We use 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 ten, and two sparse random permutations routing 100 and 500 packets respectively. This gives some hard numbers from a representative set of design decisions on some typical communications patterns. With these results, together with those of the previous section, the reader should be able to fairly simply extend the results to any other combination of algorithms, communication patterns, and architectural features discussed in this article. See Figure 20 for the results.

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	NA	NA	2	1
Overhead				10	1	1	.5

Figure 20: Fine simulation comparisons for routing 16 bits of data on a 256×256 array. Times in 1000's of cycles. Width of features in bits. 256×256 array, 16 bits of data. Four channel MGRA has max distance of 10. CGRA routes 500 and 100 packets.

Note 1: The CAAPP has the second configuration (eight bit internal memory paths, one bit wide ALU and communication paths. It also processes one memory address per cycle, can perform nearest neighbor moves from on chip memory to on chip memory in a single cycle, and has a single activity register. The cycle time has been conservatively estimated at 100 nS (in the first generation). Thus the timing of a random permutation, a "nearby" random permuation, and a "sparse" random permutations would be 5 mS, 500 μS and 500 - 1000 μS respectively.
Note 2: The fact that only a four-fold speed-up was achieved with complete 8 bit wide processing is obviously due to the overhead.
Note 3: The code has not been minimized, nor have we analyzed the many other simple architectural features that could reduce the overhead.

8 More Routing Algorithms

8.1 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 certainly situations where intermediate combining is worthwhile. One example is the degenerate case where the entire array is combined at one destination: the complexity of the combine operation is reduced from $O(N)$ to $O(n)$.

8.2 Many-to-Many routing

In the case where many PEs must send a number of packets to different destinations (the $k - k$ routing problem) there are at least two alternatives. The first is to use multiple SEND operations. If, however, a small number of PEs is distributing information to a

number of different PEs, a preferable approach is to pipeline the MGRA. A simple mechanism (conceptually) is to replace the input and output buffers with input and output queues, features that must be simulated in the basic model. The actual implementation on a SIMD array must use global feedback to achieve good performance: the simulation of the input queues can then stop when they are empty, and the cost of simulating the output queues will only be proportional to the current number of different pointer positions.

The cost of the PEs to simulate the input queues is thus dependent on the density of the communication: if all of the packets can be injected quickly into the X-channels, that portion of the procedure can cease execution a short way into the route. The cost of simulating the output queues is related to the largest number of packets any PE has so far received.

8.3 Routing Large Packets: True Wormhole Routing

When routing large packets, splitting the packet into flits can be worthwhile. The major benefit is that since the amount of data resident in the buffers of any PE is only a fraction of the original packet, each PE can execute the nearest neighbor and internal move instructions in that fraction of the time required for those same operations on the entire packet. Another benefit is that only a small penalty is paid for sending variable size messages. One drawback is that the congestion is increased as the number of packets is multiplied by the number of flits per packet. Another drawback is that the overhead is at least twice that of the MGRA on the basic model. A detailed analysis of packet splitting is beyond the scope of this paper, but we will present just enough to compare the approach taken in this paper to that of other systems.

Splitting Packets Increases Overhead

Extension of the packets into multiple flits makes deadlock possible. The way to avoid this is to simulate two additional channels (X2 and Y2) that send packets in a direction *parallel* to X1 and Y1 [8]. Unlike the four channel MGRA discussed in this paper, the doubling of the overhead is not recovered by a halving of the number of iterations. Further, the Y-channel packets no longer have absolute priority over packets entering from the X-channels: the problem is that packets can now take many iterations to fully switch from the X- to the Y-channels, during which time a Y-packet can overtake the new entry. As a consequence, queues must be simulated in one of the Y-channels, increasing the overhead further.

A second problem is that the input and output buffers must now be queues. We can assume that the input queues are rapidly emptied as the packets are injected into the system, but the output queues must be active for the duration of the procedure. With no local indexing, the cost of simulating the output queues is equal to the size (in number of flits) of the information component of the packet.

Choosing the Flit Size

The maximum performance gain—in terms of decreased overhead of the move operations—is achieved when the flit size is equal to the path width. However, if this size is smaller than the size of the destination address (in the current dimension), then a single flit will not give a PE enough information to route the packet. In this situation, the PE must queue incoming flits until it has acquired the entire address, a costly operation in a SIMD environment. The minimum flit size can be reduced in two situations, however, though both are beyond the scope of this paper: (1) the network has a small number of destinations per dimension (as in a hypercube), or (2) there is support for asynchrounous routing operations.

When Should Packet Splitting Be Used?

A simple computation using the overhead calculations of the previous section yields the result that, ignoring increased congestion, a packet must be divisible into at least five flits of size $\log N/2$ in order to benefit from splitting. Preliminary analysis indicates that this algorithm is most likely the online routing scheme of choice under the following conditions:

1. the pattern is sparse (because of the increased congestion)

2. the amount of information being sent is relatively large—at least 32 bits to break even—(to make up for the increased overhead of simulating extra channels)

3. the pattern is not a "nearby" communication, that is, there is no advantage to routing via shortest paths as in the four channel MGRA.

   Creating a shortest path version of true wormhole routing increases the breakeven point still further: the smaller number of iterations means we must account for the substantial overhead of injecting the packets into the system. Also, if the pattern is very sparse and the model contains broadcast buses, then the CGRA will still have the advantage.

9 Summary and Conclusion

Summary

Computations using SIMD arrays often require online routing of irregular, non-local, communication patterns. Many current SIMD designs have nearest neighbor connections but no dedicated general routing support, making such applications difficult to implement. Currently, the best online SIMD routing techniques use sorting as a subroutine and so are at least a factor of 3.5 from optimal for plausible array sizes. We address this problem by (1) creating models with features abstracted from the MPP, DAP, Blitzen, Polymorphic Torus, Mesh with Reconfigurable Buses, and CAAPP, and then (2) creating routing algorithms, using only available the hardware, that are nearly optimal in practice with low overhead. In particular, the accomplishments presented here are as follows:

1. Created greedy algorithms based on wormhole routing that use the nearest neighbor connections for data movement (MGRA). These algorithms can run efficiently using only the extremely simple hardware available on the basic processor model, but can also take advantage of the hardware of the successively more complex processors to improve performance.

2. Created greedy algorithms based on cut-through routing that use the reconfigurable broadcast buses to transmit data over long distances at electrical speeds (CGRA).

3. Proved matching upper and lower bounds for the worst case of the MGRA.

4. Demonstrated experimentally the near optimal performance (within a factor of 2) of the MGRA on a large variety of communication patterns. Also demonstrated that the CGRA performs in a small constant number of iterations when the communication pattern is sparse.

5. Showed that adding features to the model had the following impact: lengthening the queues did not help, broadcast buses did help, and reconfigurable buses helped still more.

6. Showed that randomization can be used to make performance more predictable, if not necessarily better.

7. Used global count to choose among on-line algorithms on the basis of communication density and proximity.

8. Enhanced these algorithms to support routing with intermediate combining, and to improve performance in routing many-to-many and routing large packets.

Conclusion and Future Work

Proving that the expected complexity of the MGRA is close to $2n$ is closely related to similar problems involving wormhole routing in general. Although under investigation, we believe this remains an open problem at this time.

It is beyond the scope of this paper to analyze all the uses to which SIMD processors are put, and how much of a resource users are willing to apply to which problems. But without a doubt, a large number of users has been satisfied with solving problems using only nearest-neighbor connections and broadcast. These users now have the capability to solve additional problems. We have also presented some sample results in evaluating how these algorithms will scale, enabling the architect to make decisions about how best to use the silicon area that is rapidly becoming available.

Comparing our approach to that of using a dedicated general router network requires running a large number of applications on many machines. We have found that in low- and intermediate-level vision work, local communication dominates. It is here that meshes (with their ability to perform nearest neighbor communication in time equal to on-chip memory access), and meshes with broadcast (with their ability to send information at electrical speeds), perform extremely well.

Acknowledgments

We would like to 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, Jim Burrill for his help with the simulations, and Deepak Rana, Mike Rudenko, and Mike Scudder for useful discussions.

References

[1] Aiello, B., Leighton, F.T., Maggs, B., and Newman, M. Fast Algorithms for Bit-Serial Routing on a Hypercube. *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures.* 1990, pp. 55-64.

[2] Annexstein, F., and Baumslag, M. A Unified Approach to Off-line Permutation Routing on Parallel Networks. *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures.* 1990, pp. 398-406.

[3] Athas, W.C., and Seitz, C.L. Multicomputers: Message Passing Concurrent Computers. *IEEE Computer.* **21**, 8 (Aug. 1988), 9-24.

[4] Batcher, K.E. Sorting networks and their applications. *Spring Joint Computer Conf.* 1968, pp. 307-314.

[5] Batcher, K.E. Design of the Massively Parallel Processor. *IEEE Trans. on Comp.* **C-29**, 9 (Sep. 1980), 836-840.

[6] Blevins, D.W., Davis, E.W., Heaton, R.A., and Reif, J.H. BLITZEN: A Highly Integrated Massively Parallel Machine. *J. of Parallel and Distributed Computing.* **8**, 2 (Feb. 1990), 150-160.

[7] Bouknight, W.J., Denenberg, S.A., McIntyre, D.E., Randall, J.M., Sameh, A.H., and Slotnick, D.L. The Illiac IV System. *Proc. IEEE.* **60**, 4 (Apr. 1972), 369-388.

[8] Dally, W.J., and Seitz, C.L. Deadlock Free Routing in Multiprocessor Interconnection Networks *IEEE Trans. on Comp.* **C-36**, 5 (May 1987), 547-553.

[9] Duff, M.J.B. Review of the CLIP image processing system. *Proc. National Computing Conf.* AFIPS, 1978, pp. 1055-1060.

[10] Herbordt, M.C., Weems, C.C., and Corbett, J.C. Message Passing Algorithms for a SIMD Torus with Coteries. *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures.* 1990, pp. 11-20.

[11] Herbordt, M.C., Weems, C.C., and Scudder, M. Non-Uniform Region Processing on SIMD Arrays Using the Coterie Network. To appear in *Machine Vision and Applications.* (1991).

[12] Hillis, W.D. *The Connection Machine.* The MIT Press. Cambridge, MA, 1985.

[13] Hunt, D.J. The ICL DAP and its application to image processing. In Duff, M.J.B, and Levialdi, S. (Eds.). *Languages and Architectures for Image Processing.* Academic Press, London, 1981, pp. 275-282.

[14] Kermani, P., and Kleinrock, L. Virtual Cut-Through: A New Computer Communication Switching Technique. *Computer Networks.* **3**, (1979) 267-286.

[15] Kumar, M., and Hirschberg, D.S. An Efficient Implementation of Batcher's Odd-Even Merge Algorithm and Its Application to Parallel Sorting Schemes. *IEEE Trans. on*

Comp. **C-32**, 3 (Mar. 1983), 254-264.

[16] Kunde, M. Lower Bounds for Sorting on Mesh-Connected Architectures. *Acta Informatica.* **24**, 2 (Apr. 1987), 121-130.

[17] Kunde, M. Parallel Routing and Sorting on Mesh-Connected Processor Arrays. *VLSI Algorithms and Architectures: Proc. Aegean Workshop on Computing, AWOC '88.* LNCS #319, Springer Verlag, 1988, pp. 129-136.

[18] Lang, H.-W., Schimmler, M., Schmeck, H., and Schroeder, H. Systolic Sorting on a Mesh-Connected Network. *IEEE Trans. on Comp.* **C-34**, 7 (Jul. 1985), 652-658.

[19] 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. *Proc. 1st ACM Symp. on Parallel Algorithms and Architectures.* 1989, pp. 328-335.

[20] Leighton, F.T. Average Case Analysis of Greedy Routing Algorithms on Arrays. *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures.* 1990, pp. 1-10.

[21] Li H., and Maresca, M. Polymorphic Torus Network. *IEEE Trans. on Comp.* **C-38**, 9 (Sep. 1989), 1345-1351.

[22] Little, J.J., Blelloch, G.E., and Cass T.A. Algorithmic Techniques for Computer Vision on a Fine-Grained Parallel Machine. *IEEE Trans. on PAMI.* **PAMI-11**, 3 (Mar. 1989), 244-257.

[23] Ma, Y., Sen, S., and Scherson, I.D. The Distance Bound for Sorting on Mesh-Connected Processor Arrays is Tight. *Proc. 27th IEEE Symp. on the Foundations of Computer Science.* 1986, pp. 255-263.

[24] MasPar Computer Corporation. *MasPar MP-1 Principles of Operation.* Document Part Number: 9300-5001, Revision: 1090, 1990.

[25] McCormick, B.T. The Illinois Pattern Recognition Computer – ILLIAC III. *IEEE Trans. on Elec. Comp.* **C-12**, 12 (Dec. 1963), 791-813.

[26] Miller, R., Prasanna Kumar, V.K., Reisis, D., and Stout, Q.F. Meshes With Reconfigurable Buses. *Proc. MIT Conf. on Advanced Research in VLSI.* 1988, pp. 163-178.

[27] Nassimi, D., and Sahni, S. An Optimal Routing Algorithm for Mesh-Connected Parallel Computers. *J. of the ACM.* **27**, 1 (Jan. 1980), 6-29.

[28] Nassimi, D., and Sahni, S. Data Broadcasting in SIMD Computers. *IEEE Trans. on Comp.* **C-30**, 2 (Feb. 1981), 101-107.

[29] Orcutt, S.E. Implementation of Permutation Functions in Illiac IV-Type Computers.

IEEE Trans. on Comp. **C-25**, 9 (Sep. 1976), 929-936.

[30] Raghavendra, C.S., and Prasanna Kumar, V.K., Permutations on Illiac IV-Type Networks. *IEEE Trans. on Comp.* **C-35**, 7 (Jul. 1986), 662-669.

[31] Rana, D., and Weems, C.C. A Feedback Concentrator for the Image Understanding Architecture. *Proc. International Conf. on Application Specific Array Processors.* 1990, pp. 579-590.

[32] Schnorr, C.P., and Shamir, A. An Optimal Sorting Algorithm for Mesh Connected Computers. *Proc. 18th ACM Symp. on the Theory of Computation.* 1986, pp. 255-263.

[33] Swanson, R.C. Interconnections for Parallel Memories to Unscramble p-Ordered Vectors. *IEEE Trans. on Comp.* **C-23**, 11 (Nov. 1974), 1105-1115.

[34] Thompson, C.D., and Kung, H.T. Sorting on a Mesh Connected Computer. *C. of the ACM.* **20**, 4 (Apr. 1977), 263-271.

[35] Thinking Machines Corporation. *Connection Machine Model CM-2 Technical Summary.* Thinking Machines T.R. HA87-4, 1987.

[36] Valiant, L.G., and Brebner, G.J. Universal Schemes for Parallel Computation. *Proc. 13th ACM Symp. on the Theory of Computing.* 1981, pp. 263-277.

[37] Weems, C.C., Levitan, S.P., Hanson, A.R., Riseman, E.M., Nash, J.G., and Shu, D.B. The Image Understanding Architecture. *Int. J. of Computer Vision.* **2**, 3 (Jan. 1989) 251-282.

[38] Weems, C.C., and Burrill, J.H. 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.