Theoretical Computer Science 128 (1994) 31–62
Elsevier

Methods for message routing in parallel machines*

Tom Leighton

Mathematics Department and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

Abstract

Leighton, T., Methods for message routing in parallel machines, Theoretical Computer Science 128 (1994) 31–62.

In this paper, we survey many of the approaches that have been proposed for solving communication problems in parallel machines. The material is presented from a theoretician's perspective, although the paper was written for a general audience.

1. Introduction

The problem of getting the right data to the right place within a reasonable amount of time is one of the most challenging and important tasks facing the designer (and, in some cases, the user) of any large-scale general-purpose parallel machine. This is because the processors comprising a parallel machine need to communicate with each other (or with a common shared memory) in a tightly constrained fashion in order to solve most problems of interest in a timely fashion. Supporting this communication is often an expensive task, both in terms of hardware and time. In fact, most parallel machines devote a significant portion of their resources to handling communication between the processors and the memory.

In this paper, we will survey many of the ideas and approaches that have been proposed for solving communication problems in parallel machines. We will cover techniques that have been developed by theoreticians as well as practitioners, and we

Correspondence to: T. Leighton, Mathematics Department and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA.

* Portions of this survey were taken from [50, 54]. The paper appeared in the Proc. ACM-STOC, 1992. The work was supported in part by DARPA Contracts N00014-91-J-1698 and N00014-92-J-1799 and Air Force Contract AFOSR F49620-92-J-0125.

SSDI 0304-3975(93)E0173-2

will pay particular attention to methods that have been (or are currently being) implemented in real machines. The material will be presented from a theoretician's perspective, although the paper is intended for a general audience. Research questions and areas of current research will also be mentioned where appropriate.

The paper is divided into two parts. In Section 2, we review the rather large vocabulary associated with parallel computation and message routing in parallel machines. Included in this section is a review of the many models and types of parallel machines, as well as various methods for configuring a shared memory. In Section 3, we review a wide variety of ideas, techniques, and approaches for dealing with message routing problems in parallel machines. Although our coverage of this material is certainly not exhaustive, we have tried to mention and/or provide pointers to most of the major contributions to the field, both in theory and in practice.

As a consequence of our desire to cover a wide range of techniques, we will not be able to present any of the techniques at any significant level of depth. For those who want to learn more about this material, we have included a substantial bibliography. As a general rule of thumb, the topics covered in this paper are covered in much greater detail in the recent text by Leighton [50]. Those topics that are not covered in [50] (such as the material on randomly wired networks and area-universal networks) will be covered at length in a forthcoming text by Leighton and Maggs [55], portions of which can be obtained in draft form from the authors. For coverage of this material from a more practical perspective, we refer the reader to the survey paper by Dally [19]. For more information on specific parallel machine architectures, see the text by Almasi and Gottlieb [6] (for general coverage of machines built before 1990) and the paper by Leiserson et al. [62] (for coverage of the CM-5). We also refer the reader to the survey by Pippenger on communication networks [74] and to the survey by Valiant on general purpose parallel architectures [89] for more information on these subjects. Lastly, we refer the reader to the annual *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures* (*SPAA*) for extended abstracts of recent research results.

2. Terminology and models

We begin our survey by reviewing the many models and types of parallel machines that are used by theoreticians, algorithm designers, programmers and architects. Unfortunately, there are large differences between the models, and the parallel machine that is imagined by the theoretician or programmer (i.e., the user) is often quite different than the machine that was built by the architect. In fact, one of the key applications of message routing is to bridge the gap between different models so that the machine built by the architect (say a Connection Machine) can be made to appear like the machine imagined by the user (say a *parallel random access machine* (PRAM)).

Unfortunately, the coverage of machine models is lengthy (mostly because there are so many different views on what parallel machines are or should be) and it constitutes a significant digression from the main topic of message routing. The material is

important, however, because the motivation for many routing problems comes from trying to make one machine model emulate another. Hence, the material on models of parallel machines will provide an excellent background and perspective for our review of message routing.

2.1. Models of parallel machines

A wide variety of models have been proposed as the "right" abstraction of a parallel machine. In fact, there are probably more models of parallel machines than there are real parallel machines. In what follows, we will define many of the most important models and we will point out the basic differences among them. We have partitioned the models broadly according to the components which they use for communication. We begin with the most abstract (or wireless) models in Subsection 2.1.1.

2.1.1. Abstract models of parallel machines

The most popular abstraction of a parallel machine is, by far, the PRAM. A PRAM consists of a collection of independent processors and a single shared memory. It is assumed that each processor can read or write to any location in the memory in a single step.

There are many different types of PRAMs depending on whether or not multiple processors are allowed to read or write to the same memory location at the same time and on what arbitration rule is used if multiple processors try to write to the same location at the same time. For example, in an *exclusive-read, exclusive-write* (EREW) *PRAM*, only one processor is allowed to read from or to write to any location in memory at any given time. In a *concurrent-read, exclusive-write* (CREW) *PRAM*, any number of processors are allowed to read from the same location in memory at the same time, but only one processor can write to any location at any time. In a *CRCW PRAM*, multiple processors are allowed to concurrently write to the same memory location, although a protocol must be specified to arbitrate among them. Many different arbitration protocols have been proposed (e.g., nothing is written, an adversary chooses what to write, the sum of the competing values is written, the lowest-index value is written, etc.), and the choice of the protocol can affect the computational power of the machine. For example, the logical OR of N bits can be trivially computed in one step on an N-processor CRCW PRAM if concurrent writes are handled by writing only if every competing processor tries to write the same thing (and otherwise writing nothing) – this is known as the *COMMON CRCW PRAM* model – but the same computation requires $\Omega(\log N)$ steps on PRAMs with more limited concurrent writing capabilities [26]. (For more information on the various concurrent write models and their relationships to other models of computation, see [25, 27, 32].)

Unfortunately, the PRAM is not a particularly accurate model for most parallel machines. In fact, no PRAM has ever been built per se, although most real parallel machines can emulate a PRAM with some degree of efficiency by appropriate use of

message routing. It has been argued by some that optical technology will eventually make the PRAM a reality. In fact, tremendous progress has been made in optical communication technology during the past decade, but optical switching technology is still no match for traditional electronic switching.

The main advantage of the PRAM is that it provides a very simple and natural architecture-independent model for the parallel algorithm designer. In an effort to make the model more realistic, several variations of the PRAM model have been proposed and analyzed in the literature. Some of these variations are described in the following paragraphs.

A *distributed-memory PRAM* (also known as a *module parallel computer* (MPC)) is the same as a PRAM except that the global memory is partitioned into M blocks and access to the memory is restricted so that at most one memory location within each block can be accessed at any time. Concurrent reads and writes may or may not be allowed depending on the model. The distributed-memory PRAM more closely models large-scale parallel machines (such as the BBN Butterfly, IBM RP3 and GF11, Ametek 2010, Intel iPSC and Touchstone, NASA MPP, NCUBE, and Thinking Machine's Connection Machine Series) for which the memory is partitioned into blocks, each with their own sequential access.[1] Of course, the distributed-memory PRAM is harder to program than the generic shared-memory PRAM, but the partitioning of memory is a fact of life that must be confronted with large-scale parallelism. Techniques for simulating a generic shared-memory PRAM on more realistic distributed-memory models are covered in Section 3 of this paper.

The *distributed random access machine* (DRAM) model of computation is similar to the distributed-memory PRAM model except that each processor/memory pair is assigned a location in an area A square (or a volume V cube) and memory accesses are further restricted so that at most $O(p)$ accesses can "cross" any region in the square with perimeter p in one step. (An access is said to *cross* a boundary if its origin and destination are on opposite sides of the boundary.) The DRAM model attempts to capture the notion of physical locality in parallel machines, and it reflects the limitations imposed by the area and volume constraints inherent in two and three dimensional wiring technology. For example, if a parallel machine occupies volume V, then it has a cross section with area $V^{2/3}$, which means that at most $O(V^{2/3})$ wires can pass through the cross section and thus at most $O(V^{2/3})$ memory accesses can pass through the cross section at any time. This model is particularly appropriate for parallel machines based on arrays (such as the MPP, Intel Touchstone, and Ametek

[1] Recently, a variation of the distributed-memory PRAM model which includes parameters for bandwith and latency was proposed. This model is known as the *Log P model*, and it is described in "P. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian and T. von Eicken, Log P: Towards a realistic model of parallel computation, in: *4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming* (1993)".

2010) or fat trees (such as the CM-5). Further information on the DRAM model can be found in [61] as well as in Section 3 of this paper.

In the *block PRAM* (BPRAM) model, the cost of accessing data from memory is altered to reflect the fact that in many parallel machines, the time to access one byte of data is often little more than the time needed to access 100 bytes of data. In particular, the time needed to access b bytes of data is defined to be $\alpha + b\beta$ in the BPRAM model where α and β are predetermined constants such that $\alpha \gg \beta$. For more information on this model, see [2, 14]. The BPRAM model has also been generalized to models (such as the *local memory PRAM* (LPRAM) and the *hierarchical BPRAM*) where the cost of a memory access varies with the distance of the access in the memory hierarchy (e.g., see [3, 7]).

The *scan model* is similar to the generic PRAM model except that we are allowed to perform a prefix computation in a single step. In particular, if x_i is contained in the ith processor for $1 \leqslant i \leqslant N$ then the ith prefix $y_i = x_1 \otimes x_2 \otimes \cdots \otimes x_i$ can be returned to the ith processor for $1 \leqslant i \leqslant N$ as a primitive 1-step operation in the scan model, where $\otimes$ is an arbitrary associative operator. Since many parallel machines (such as a Connection Machine) can run a prefix computation in less time than it takes for each processor to access the global memory once, the scan model is more realistic than the PRAM model for many parallel machines. Allowing prefix as a primitive also allows the algorithm designer to design substantially faster algorithms for many problems (e.g., see [12]). Other variations of the PRAM that allow even more powerful primitives (such as multiprefix [80], Fourier transforms, sorting, etc.) have also been proposed.

PRAMs and related models are usually considered to be synchronous, multiple-instruction-multiple-data (MIMD) machines. In practice, MIMD machines (those for which different processors can be executing different instructions at the same time) are usually *asynchronous* (that is, different processors can start tasks independently of one another), although many such machines (such as the BBN Butterfly and the CM-5) have built-in mechanisms (such as a message router or a control network) that can be used to enforce some degree of synchronization. (For more information on methods for handling asynchrony in PRAMs, we refer the reader to [17, 18, 28, 71].) In addition, most PRAM algorithms can really be run in *SIMD* fashion (where every processor executes the same instruction at the same step, but on different data) without much loss in efficiency.

2.1.2. Network-based machines

Most real parallel machines use some form of network to interconnect the processors with each other and/or to interconnect the processors with the memory. Network-based machines can be broken into two broad classes depending on whether or not a processor is located at every node of the network.

In a *direct-network* parallel machine, there is a processor and a block of memory at each node of the network. When a processor wants to access some data that is not stored in its local memory, it sends a message through the network to the appropriate

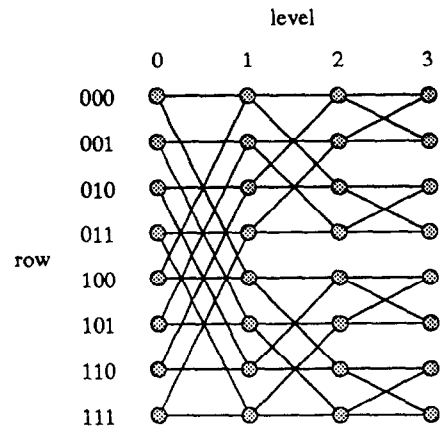

Fig. 1. An 8-input butterfly.

node requesting the data. Such machines are also sometimes referred to as *message-passing* and *distributed-memory* machines. Typical choices for the network include trees, arrays of varying dimensions, hypercubes, butterflies, and other hypercubic networks. (A *k-dimensional N-sided array* has N^k nodes $\{(i_1, i_2, \ldots, i_k) \mid 1 \leqslant i_1, \ldots, i_k \leqslant N\}$. Two nodes are connected if they differ in precisely one coordinate and they differ by precisely 1 in that coordinate. The *N-node hypercube* is a $\log N$-dimensional 2-sided array. The *N-input butterfly* is a graph with $\log N + 1$ levels, each with N nodes. The jth node $(0 \leqslant j < N)$ on level i $(0 \leqslant i \leqslant \log N)$ is denoted by $\langle j_1 j_2 \cdots j_{\log N}, i \rangle$ where $j_1 j_2 \cdots j_{\log N} = \text{bin}(j)$ and it is connected to nodes $\langle j_1 j_2 \cdots j_{\log N}, i+1 \rangle$ and $\langle j_1 \cdots j_i \overline{j_{i+1}} j_{i+2} \cdots j_{\log N}, i+1 \rangle$ on level $i+1$. The nodes on level 0 are called *inputs* and the nodes on level $\log N$ are called *outputs*. For example, see Fig. 1. (For definitions of other networks, see [50].) A key property of all networks used for large-scale machines is that they have low node degrees, which rules out the *complete network* (in which every pair of nodes is connected by an edge).

In an *indirect* or *multistage* network-based machine, the processors and blocks of memory are interconnected by a network of switches (which are not associated with any processor). Typically, the switches are grouped in levels (or stages) so that all the processors and memory are in a single stage and so that switches in one stage are linked only to a small number of switches in the adjacent stages. Each processor is usually equally distant from all the others, and thus there is usually no notion of locality in such machines. (A counterexample is the CM-5 which uses a fat tree and a butterfly in an indirect fashion that preserves some locality [62].) As a consequence, such machines are often referred to as *shared-memory* machines even though the memory is distributed into blocks across the network. Typical choices for an indirect network include crossbars (for small-scale machines), the butterfly, and other low-degree hypercubic networks.

By routing messages through the network, any network-based machine can emulate any PRAM model. The efficiency of the simulation depends on how much work is required for message routing. This notion is captured in the *bulk synchronous model* of parallel computation. In the bulk synchronous model, the parallel machine consists of N processors and memory modules, a message router, and a synchronizer. Time is partitioned into blocks of L steps each (called *supersteps*) by the synchronizer. At the end of each superstep, the synchronizer checks to see if each processor has completed its tasks. If so, the processors are told to begin the next set of tasks. If not, then the processors that are finished wait for those that are not yet done. Data can only be used if it is available locally at the beginning of the superstep. Communication is handled by the router, which is capable of routing any h-relation in $\alpha h + \beta$ steps where α and β are parameters that depend on the nature of the router that is used. (An h-relation is a routing problem where at most h packets start and finish at each node. Broadcasting and combining are not allowed.) For example, we will see in Section 3 that $\beta = \log N$ and $\alpha = O(1)$ for a randomized router that is based on an N-node hypercube.[2] (For more information on the bulk synchronous model, see [88].)

2.1.3. Bus-based machines

In addition to point-to-point wires, many parallel machines also use buses to interconnect processors. In fact, some parallel machines (such as the Encore, Sequent, Alliant, and Ardent) rely on a single high-speed bus as the primary means to interconnect the processors and the memory. In such machines, the access to main memory is sequential in nature, but if the number of processors is small and the bus is fast, then the accesses can appear to be parallel. Although the single-bus approach to parallel architecture does not scale well, it has been successfully utilized to obtain efficient speedups for several applications using relatively small numbers of processors. From the message routing point of view, there is not a lot to be said about such machines (since the routing is done by broadcast on the bus), although there are interesting problems involving cache coherency and bus access protocols (e.g., see [34, 38]).

Multiple bus and reconfigurable bus architectures have also been proposed for parallel machines, but these approaches are still in the experimental stages. One particularly popular abstract model using buses configures the processors into a 2-dimensional array with a bus in every row and column. In some variations of the model, the user is allowed to locally partition the buses into sub-buses, thereby partitioning the processors into regions of the plane that are connected by buses. (For more information on reconfigurable networks, we refer the reader to the recent survey by Li and Stout [63].)

Modeling the performance of multiple-bus architectures can be a tricky business since the cost of communicating across a bus increases with its size and with the number of points where it can be severed, a fact often overlooked in theoretical work

[2] In this paper, we use the notation $\log N$ to denote $\log_2 N$.

on this subject. In addition, message routing algorithms on such networks suffer from the same bisection-width constraints that plague array-based interconnection networks. As a consequence, message routing on a reconfigurable 2-dimensional bus architecture (for many problems) is not a lot faster than on a standard 2-dimensional array network, and so we will not devote much additional attention to bus-based architectures in this paper.

2.1.4. The vector and dataflow models

The preceding models of parallel computation are all architecture-based. In other words, we described the setup of the machine and did not worry about what it could do. In what follows, we will describe two language-based models of parallelism. This time, the model will be defined by what it does and not so much by how it does it.

In a *vector machine*, we specify the computation to be performed as a sequence of operations involving vectors of data. For example, one operation might be to compute $C(i) = A(i) \cdot B(i)$ for $1 \leqslant i \leqslant 100$. The vector operations are performed sequentially, but there are at least two ways to speed up the computation of a single vector operation. If we have a single processor, then we can gain a speedup by pipelining the various elementary instructions that make up each multiply. This technique is used widely in practice and can speed up computation by a reasonable constant factor (for one processor) if the array is sufficiently large (i.e., large enough to fill the pipeline). Given more than one processor, we can also speed up computation of a vector operation by spreading the work among the processors and performing the operation in a parallel SIMD fashion. For more information on vector processors, we refer the reader to [6, 12].

In a *dataflow model*, the computation to be performed is specified as a *task graph*. Nodes in the graph represent operations to be performed, and edges represent inputs and outputs for each operation. In order to carry out the computation, we need to map the task graph onto a machine. Whenever the inputs for a computation are ready and available at the same processor, the computation takes place and the outputs are sent to the appropriate locations.

The efficiency of the dataflow approach depends on several factors, including the amount of parallelism inherent in the task graph, the efficiency with which the task graph is embedded into the machine that will execute it, and the performance capabilities of the underlying machine. A variety of dataflow machines have been constructed, and most use architectures that (at a high level) are similar to those described earlier in this paper. The problem of embedding a task graph in a parallel machine involves message routing as well as scheduling and placement issues. For more information on the dataflow approach to parallelism, we refer the reader to [6] and the references contained therein.

2.1.5. Some comments on the meaning of shared memory

The observant reader will notice that we have used the terms *shared memory*, *distributed memory*, and *global memory* often in the paper without really defining them.

The reason that we have omitted the definition is because the terms mean different things to different people. For example, Almasi and Gottlieb [6] define a memory to be a shared memory if each processor has equal access to each location in memory. This definition includes multistage network machines such as the BBN Butterfly even though the memory in such machines is clearly distributed into distinct blocks. Other definitions of a shared memory include any memory with a global address space, independent of equal access. Still others (e.g., [40]) restrict the definition to include only machines (such as PRAMs or single-bus machines) where the global memory is not partitioned in any discernible way.

We will not attempt to resolve the confusion in terminology here. Rather, we will use the terms loosely, using the term *distributed memory* to refer to a memory that is physically broken into chunks, and using the terms *shared memory* and *global memory* to refer to the union of globally accessible addresses in any parallel machine. (Hence, in this interpretation, a memory can be both shared and distributed.)

2.2. Types of routing algorithms

There are many different models for message routing in a parallel machine depending on how each message moves through the machine and how it is buffered along the way. In this paper, we will focus on message routing in fixed-connection networks since virtually all of the parallel machines that have a need for message routing (such as the BBN Butterfly, IBM RP3 and GF11, Ametek 2010, Intel iPSC and Touchstone, NCUBE, MPP, and Thinking Machines CM-1, CM-2, and CM-5) use a fixed-connection network for communication.

In the *store-and-forward* (or *packet-switching*) model of routing, each message consists of a single entity (called a *packet*) that moves through the network, one node or switch at a time. (A packet may or may not be transmitted from one node to another in a bit-serial fashion, but the entire packet is transmitted before any part of the packet advances to the next node.) A *step* is defined as the amount of time it takes for one packet to cross one wire. Generally, at most one packet can cross each wire at each step in the store-and-forward model. (Machines that use store-and-forward routing include the MPP, Intel iPSC, and early NCUBEs, as well as any machine composed of Transputers.)

Packets may or may not be allowed to pile up in *queues* in the store-and-forward model. In *hot-potato routing* (also called *chaos routing*), packets cannot be queued. In other words, every packet must move at every step until it arrives at its destination. (This approach is being used in the Tera computer [82].) In some cases, this may mean that packets are *misrouted* since they might be sent in the wrong direction. (Approaches that allow misrouting are also referred to as *deflection* and *indirect routing* algorithms. Such approaches are used in several machines, including the CM-1 and CM-2.)

In routing with *combining*, two or more packets with the same destination can be combined into a single packet by any switch that contains both packets at the same time. Several different kinds of combining operations are possible, including just

about any binary operation. Combining has been used in several parallel machines, both in hardware (such as in the NYU Ultracomputer [29] and in some experimental machines [1]) as well as in software (such as in the Connection Machine series).

In the *circuit-switching* (or *path-lockdown*) model of routing, the entire path from the origin to the destination of a message is established and dedicated so that the data in the message can be transmitted as an uninterrupted stream of pipelined bits from origin to destination. Of course, it may not always be possible to establish disjoint paths through the network for all messages that need them, and sometimes messages will be *blocked* or *dropped*. Such messages will need to be resent later. In some cases, a given wire may represent several channels that are used in a multiplexed fashion so that the paths need not be completely disjoint. The BBN Butterfly and IBM GF11 are examples of machines that use the circuit-switching approach to message routing.

The BBN Butterfly and IBM GF11 differ in how the paths are set up, however. In particular, the GF11 computes routing paths in an *off-line* fashion (i.e., the communication patterns that will be used are known ahead of time, and disjoint message paths are precomputed by a global sequential processor), whereas the BBN Butterfly attempts to set up the paths in an *on-line* fashion (i.e., using only local control and local information while the parallel program is running). Off-line solutions to routing problems are usually superior to on-line solutions, but they require advance knowledge of the pattern of communication that will be used. Such advance knowledge is often not available, which limits the capabilities of machines such as the GF11.

The *wormhole* and *cut-through* models of routing form a compromise between the store-and-forward and circuit-switching approaches to routing. In wormhole and cut-through routing, each message consists of a collection of *flits* (usually about one byte of data each) and the message snakes its way through the network in a worm-like fashion. Each wire can transmit one flit per step and the message can occupy as many switches as it has flits at any time. If buffering is allowed, then the message can compress (like an accordion) into a smaller number of switches (possibly one). A key feature in wormhole and cut-through routing is that no message can ever be "cut". In other words, consecutive flits in a message must always reside in neighboring (or the same) processors in the network.

The precise differences between wormhole and cut-through routing depend on what sort of buffering is allowed. In [19], Dally states that in wormhole routing, no buffering is allowed, whereas buffering is allowed in cut-through routing, although other interpretations of the terms are common.

2.3. Types of routing problems

In addition to the myriad of routing models, there are also a wide variety of routing problems. The simplest kind of routing problem is a *static* problem in which there is at most one message to be sent from each processor and at most one message to be sent to each processor. Such routing problems are called *one-to-one* (or *partial permutation*) routing problems. They are *static* in the sense that all the messages are ready to

be sent at the first step. (This is as opposed to *dynamic* routing problems where messages enter the system continually over time and the routing of messages is never completed per se.) Machines equipped to handle static routing problems include the BBN Butterfly, IBM GF11, and Thinking Machines CM-2. Machines equipped for dynamic routing problems include the Intel Touchstone, Ametek 2010, and Thinking Machines CM-5.

In more complicated routing problems, a single message may have multiple destinations (as in a *one-to-many* routing problem or a *multibroadcast* problem), or many messages may be destined for the same place. If more than one packet is headed for some block of memory, then the routing problem is said to be *many-to-one*. If each packet is headed for a distinct address with in each block, however, then the routing problem is also considered to be *locally one-to-one*. Locally one-to-one routing problems arise in the simulation of EREW PRAMs, whereas one-to-one routing problems arise in the simulation of distributed-memory PRAMs.

Destinations for which there are an above average number of packets are often called *hot spots*. If p packets start and end at every processor in a static problem, then the problem is said to have *slackness* or *load p*. (Such routing problems are also called *p-relations*.)

In many-to-one routing problems, we may or may not allow packets heading for the same destination to be *combined* into a single packet before arriving at the destination. Combining is necessary if concurrent writes are to be emulated in an efficient manner in the network. Without combining, it might take N steps on an N-processor network to simulate just one step on an N-processor CRCW PRAM. By using combining, the simulation time can be reduced dramatically, as we will see in Section 3 of the paper.

Depending on the machine, combining can also be useful for simulating EREW PRAM algorithms. This is because of hot spot problems that can arise when several processors want to access different memory locations within the same block of memory. If the cost of routing a long message is not much different from the cost of routing a short message (as is the case with many machines – because it can take a long time to initiate the sending of a message but very little additional time to send each bit of the message), then it can be worthwhile to combine messages headed for the same block of memory (but possibly different addresses). Notice that this issue does not arise when emulating distributed-memory PRAM algorithms since the burden of preventing memory contention is shifted to the algorithm designer in this model.

2.4. Performance measures

Generally speaking, a routing algorithm performs well if it routes every message to its destination as quickly as possible using as small an amount of the network resources as possible. The degree to which an algorithm achieves this level performance can be measured in several ways, as is described in what follows.

In some static routing scenarios (such as on a CM-2) we wish to minimize the *total time* it takes to route all messages to their destinations. In static scenarios where messages are blocked or dropped (such as on a BBN Butterfly), we want to maximize the number of messages that successfully reach their destinations within a fixed amount of time (i.e., we want to maximize the *throughput*).

In a dynamic setting, we often wish to minimize the *latency* (or delay) in sending each message to its destination. (This means that we will want algorithms that exploit *locality* when it is present.) We will also want to avoid the possibility that the network becomes *deadlocked* (a state where no messages can move) or *livelocked* (a state where messages can move, but no message can make progress toward its destination). In applications where hot spots can arise, we will desire algorithms that minimize the effect of the *hot spots* on packets with other destinations.

In models that allow queueing, we often desire to minimize the *queuesize* at each switch. In scenarios where network bandwidth is limited (which is often the case), we want to maximize the *bandwidth utilization* (i.e., we want to maximize the fraction of wires in a critical bisection of the network that are being used productively at any given time). We also want to minimize the *VLSI area* or *volume* consumed by the routing network, as well as the number of switches in the network and their capacity.

It is most desirable if the routing algorithm is guaranteed to perform well for all routing problems. Often this is not possible (or, at least, such an algorithm may not be known). In such cases, it is desirable for the algorithm to perform well for a random or average-case routing problem. At the very least, the algorithm should be known to perform well on a special set of commonly-occurring routing problems.

Lastly, we desire routing algorithms that will perform well even if some of the switches in the network are faulty. Fault-tolerant algorithms are not typically used in practice today, although this is in the process of changing, and experimental machines (such as the MIT Transit Machine [15, 23, 24, 68]) are being designed with fault-tolerance as a key performance measure.

3. Basic techniques and approaches

We begin our review of approaches to message routing with a discussion of greedy algorithms. Greedy algorithms are widely used in practice and are known to perform reasonably well on average. Unfortunately, greedy algorithms are also notorious for performing poorly on a variety of worst-case routing problems that often arise in practice.

3.1. Greedy algorithms

Most parallel machines use some form of greedy algorithm when routing messages through a network. Examples of such machines include the BBN Butterfly, IBM RP3, Ametek 2010, Intel Hypercube and Touchstone, and the Thinking Machines CM-1,

CM-2, and CM-5. The algorithms used on these machines are all greedy in the sense that each packet *attempts* to follow a simple shortest path to its destination. In a 2-dimensional array, for example, a packet starting at node (i, j) and destined for node (i', j') would first move through the ith row to node (i, j') and then through the j'th column to its destination. In an N-input butterfly, there is always a unique path of length $\log N$ from every input to every output, and every packet follows such a path in a greedy algorithm on the butterfly. Similar greedy paths can be defined for other hypercubic networks.

In what follows, we will survey what is known about the performance of the greedy algorithm for a variety of networks and routing models. There are many variations of the greedy algorithm depending on what contention-resolution protocols are used (if any) and which greedy paths are used by packets (if there is more than one option), and we will limit our discussion to the general behavior of greedy algorithms. For a more detailed discussion of greedy algorithms on arrays, butterflies, and other hypercubic networks, we refer the reader to [50, Sections 1.7 and 3.4].

3.1.1 Best-case and worst-case performance

Generally speaking, the greedy algorithm performs very well in the best case and very poorly in the worst case, at least for static one-to-one routing problems. For the N-input butterfly and related hypercubic networks, the greedy algorithm can route many N-packet one-to-one routing problems from the inputs to the outputs with edge-disjoint paths in $\log N$ steps, which is optimal. Good examples of such best-case routing problems include spreading and packing problems. A *packing problem* is a routing problem in which all the packets are sent to a contiguous block of outputs so that the relative order of the packets is unchanged. A *spreading problem* is the reverse of a packing problem. An important special case of a packing problem is the *k-cyclic shift permutation* (where the packet at input i is sent to output $(i+k) \bmod N$ for all i). As a consequence, the greedy algorithm can be used to perform a k-cyclic shift of N packets in $\log N$ steps on an N-input butterfly for any k. The greedy algorithm can also be used to route any N-packet monotone routing problem in $2 \log N$ steps by first packing the packets and then spreading them. (A *monotone routing problem* is a one-to-one problem for which the relative order of the packets is unchanged.) Monotone routing problems arise in many applications, including the solution of one-to-many routing problems and the routing of packets that have been presorted according to destination. (For example, see Section 3.5.1.)

Greedy routing also works well on arrays, particularly when the packets have been presorted according to their destination. More generally, if at most one packet in each row is initially destined for each column in a $\sqrt{N} \times \sqrt{N}$ array, then the greedy algorithm routes every packet to its destination using queues of size one in the store-and-forward model in $2\sqrt{N} - 2$ steps, which is optimal for many routing problems on an array (due to diameter constraints).

Unfortunately, the greedy algorithm does not perform very well in the worst case, even if we restrict ourselves to one-to-one routing problems (which we shall do for the

time being). For example, an N-input butterfly takes at least $\Omega(\sqrt{N})$ steps to route an N-packet transpose permutation no matter what routing model is used. (The *transpose permutation* sends the packet at input $i_1 i_2 \cdots i_{((\log N)/2)} j_1 j_2 \cdots j_{((\log N)/2)}$ to output $j_1 j_2 \cdots j_{((\log N)/2)} i_1 i_2 \cdots i_{((\log N)/2)}$.) If we are using the store-and-forward model and are limited to constant-size queues, the routing time can be even worse [67]. Similarly poor performance is exhibited by the greedy algorithm for several other commonly occurring routing problems such as the *bit-reversal permutation* (where the packet at input $i_1 i_2 \cdots i_{\log N}$ goes to output $i_{\log N} i_{\log N - 1} \cdots i_1$). More generally, it is known that any *oblivious algorithm* (i.e., one for which the routing path for each packet is deterministically selected without knowledge of the other packet origins and destinations) will exhibit $\Omega(\sqrt{N}/d)$ time worst-case performance for any N-node degree-d network. (This fact was recently proved by Kaklamanis et al. [37], who improved the well-known $\Omega(\sqrt{N}/d^{3/2})$ bound due to Borodin and Hopcroft [13].)

The greedy algorithm can also be made to perform poorly on an array. In the store-and-forward model, the algorithm can be made to use queues of size $\Omega(\sqrt{N})$ and/or many more than $\Omega(\sqrt{N})$ steps to route the packets to their destinations.[3] In the circuit-switching and cut-through models, the greedy algorithm can be made to use $\Theta(N)$ steps to route N messages, depending on what contention-resolution protocols are used.

3.1.2. Average-case performance

On average, the greedy algorithm performs very well. So well, in fact, that average-case routing problems can be regarded as best-case problems for many networks. Indeed, the probability of encountering a routing problem with performance anything like the worst case is so small that it should never happen in practice; that is, if practical problems were random, which, of course, they are not.

The fact that commonly occurring (in practice) routing problems such as transpose are among the very few routing problems that exhibit worst-case performance can spell trouble for the designer of message routing algorithms. More than once, a greedy routing algorithm has been tested on random routing problems, found to "always" perform well, been implemented, and then found to have some nasty worst-case performance which did not show up the initial testing. Indeed, the message routing domain is one for which a solid understanding of worst-case behavior is crucial since (probabilistically rare) worst-case problems frequently arise in practice.

When analyzing the average-case performance of the greedy algorithm, it is usually assumed that each packet has a random destination (independent of the other packets), which allows for the possibility that several packets might have the same destination. In the static case, it is usually assumed that there are p packets starting at each input (where

[3] For recent work in this area, see "D. China, T. Leighton and M. Tompa, Minimal adaptive routing on the mesh with bounded queue size, in *Proc. 6th ACM Symp. on Parallel Algorithms and Architectures* (1994)".

p is often assumed to be one). In the dynamic setting, it is assumed that packets arrive at each input according to some random process (usually binomial or Poisson).

The following results are representative of what is known about the average-case behavior of the greedy algorithm on an N-input butterfly and an N-node array.

- If there are p packets at each input of a butterfly in the store-and-forward model, then the greedy algorithm (using any nonpredictive contention-resolution protocol such as FIFO) will route all the packets to their destinations within $\log N + o(\log N) + O(p)$ steps using queues of size $o(\log N) + O(p)$ with probability $1 - N^{-\alpha}$ for any constant α. (This result is proved in [50] using techniques from [5, 79, 85, 86, 90].) Whether or not the same result holds if queuesizes are restricted to be constant is not known, although variations of the greedy algorithm that will be discussed later do obtain O(1)-size queues. (See [67] for more information on this problem.)
- If there is one packet at each input of a butterfly in the circuit-switching model, then $\Theta(N/\log N)$ packets are likely to reach their destination without being blocked on the first pass of the greedy algorithm. If q message paths can use each edge of the butterfly, then this bound can be improved to $\Theta(N/\log^{1/q} N)$. (A butterfly where each edge is replaced by a channel of q edges – so that q paths can pass through a channel simultaneously – is called a *q-dilated butterfly*. For more information on these bounds, we refer the reader to [42, 43, 45, 50, 67].)
- In the dynamic store-and-forward model, the expected latency incurred by any packet on the way to its destination in a butterfly is $O(\log N)$, even if the arrival rate of packets is high enough to load the network to within 99% of capacity [83]. Whether or not a similar result can be proved for an algorithm that restricts queues to be of size O(1) is an interesting open question.
- When routing N packets on a $\sqrt{N} \times \sqrt{N}$ array in the store-and-forward model, the probability that a packet is delayed Δ steps is $O(e^{-\Delta/6})$. Moreover, the maximum queuesize will be 4 with high probability [49, 50].
- For dynamic routing problems on a $\sqrt{N} \times \sqrt{N}$ array for which the arrival rate loads the network of 99% of capacity, the probability that a packet is delayed by Δ steps is $e^{-\Theta(\Delta)}$, and the maximum queuesize observed within any window of T steps is at most $O(1 + (\log T/\log N))$ with high probability [49, 50].

3.1.3. Protocols for bounding queuesize and combining

The behavior of the greedy algorithm is the easiest to analyze when queues in the network are allowed to become arbitrarily large, and when combining is not allowed. (All of the results on average-case performance just described fall into this category.) Unfortunately, very little is known about the average-case behavior of the greedy algorithm when the forward progress of a packet is halted by a full queue in the node ahead.

By modifying the greedy algorithm somewhat, however, it is possible to derive an algorithm which guarantees bounded queues, performs combining, and which provably performs well on static random problems. In particular, if each packet is

assigned a random key (which can be assumed to be a function of the destination of the packet), and if the packets passing through each node of the network are restricted so that they pass through the node in sorted order (of the keys) and so that a packet never advances forward into a full queue, then the greedy algorithm can be shown to perform near optimally (even with combining) for many networks, including arrays, hypercubes, and butterflies. This particular variation of the greedy algorithm is known as *Ranade's algorithm* and is analyzed in detail in [50, 56, 57, 78, 79]. Plans for implementation of this algorithm in a parallel machine are currently under way at the University of Paderborn [1]. (Other approaches to bounding queuesizes which are less amenable to combining are described in [67, 73].)

3.2. Converting worst-case routing problems into average-case problems

Because the greedy algorithm is known to perform very well for all but a "few" routing problems, one approach to message routing has been to use the greedy algorithm for most routing problems and to use specially designed routing algorithms for the routing problems where the greedy algorithm fails. Although there has been some success in devising routing algorithms that work well for classes of routing problems that contain the transpose and bit-reversal permutations [69], algorithms that work well whenever the greedy algorithm fails are not known. Methods have been developed using randomness, however, that can be used to convert worst-case routing problems into average-case routing problems with a small amount of overhead. We will describe these methods in what follows. For more information and details, see [50].

3.2.1. Randomizing the memory

Typically, data is stored in memory according to some natural or logical pattern. For example, the i, j entry or block of a matrix might be stored in memory location $\text{bin}(i)|\text{bin}(j)$. Such natural data storage patterns can lead to trouble, however, when we try to access or permute the data using the greedy algorithm on a network such as the butterfly. (For example, we have already seen that the task of transposing a matrix stored in this fashion would result in worst-case performance by the greedy algorithm.)

One method for avoiding such difficulties is to store the data in a random fashion in memory. More precisely, let $h:[1, M] \rightarrow [1, N]$ be a random function where M is the total size of the memory and N is the number of blocks into which the memory is partitioned. Provided that M is much larger than N (which is usually the case), h will evenly spread the M memory locations among the N blocks of memory with high probability. (In particular, $|\{x: h(x)=y\}|$ will be $(M/N)+o(M/N)$ for all y with probability very close to 1.) Hence, we can use h to randomly (and evenly) partition the memory by assigning memory location x to block $h(x)$.

By randomizing (or *hashing*) the memory in the fashion just described, we can convert any one-to-one routing problem into a random routing problem. This is

because each packet of the one-to-one routing problem will now be destined for a random output (where the randomization is provided by the choice of h). (In fact, the conversion works equally well for routing problems that are only locally one-to-one.) Hence, unless we are very unlucky in our choice of h, the transpose and bit-reversal permutations will be converted into routing problems that run in $\log N + o(\log N)$ steps on an N-input butterfly, just like any other average-case routing problem.

Of course, once h is fixed (and the memory is partitioned), there will still be some bad one-to-one routing problems (e.g., routing problems that run in $\Omega(\sqrt{N})$ steps using the greedy algorithm on an N-input butterfly) but we would not be likely to encounter them. This is because the set of bad routing problems is very sparse and (once h is applied) the set of bad routing problems becomes random.

Randomizing the memory also has other benefits. For example, consider a locally one-to-one routing problem where there are a large number of packets (say k) headed for distinct locations within the same block of memory. The particular block of memory then forms a *hot spot* in the network (i.e., a bottle-neck).

Hot spots frequently arise in practice because of the tendency to organize the memory in a highly structured manner (e.g., storing the ith row of a matrix in the ith block of the memory). Hot spots are troublesome for several reasons. For example, it will take at least $k + \log N$ steps to deliver the k packets to the hot spot output. Moreover, the delays and congestion associated with packets headed for a hot spot can seriously delay other packets that are not headed for a hot spot.

Many techniques have been developed for dealing with the hot spots (see [50]) and we will mention some of them in this paper. Perhaps the simplest is to randomize the memory. In particular, as long as each memory location is accessed by at most one packet in a routing problem, then the routing problem is converted into a random routing problem by randomly partitioning the memory. Of course, hashing the memory does not solve the hot spot problem caused by a large number of packets trying to access the same location in memory (such as in the implementation of a concurrent read or concurrent write), but such routing problems can be efficiently handled using combining.

Unfortunately, randomly hashing the memory has some undesirable side effects. For example, desirable aspects of regularity such as *locality* are destroyed when the memory is randomized. In addition, the addressing mechanism for the memory is complicated by hashing since we must be able to locally compute $h(x)$ quickly for any x. This is not an easy task if h is a totally random function. Fortunately, several methods are known for constructing simple functions h which are random enough for routing purposes. (In particular, any $\log N$-wise independent function will usually do. See [39, 50, 79] for more details and pointers.)

Despite the drawbacks, a randomized memory organization is featured in several (still experimental) parallel machines, including the BBN Monarch [81], the Tera Computer [82], and the parallel machine being built at the University of Paderborn [1].

3.2.2. Randomized routing

Because of the negative side effects associated with hashing, hashing is not always appropriate as a tool for converting worst-case routing problems such as transpose into average-case problems. In addition, even if the memory is randomized, there are still worst-case one-to-one routing problems (which can be easily constructed by anyone that knows the hash function h).

For applications where hashing is not appropriate, there is an entirely different approach to routing that does not alter the memory organization at all, and that does not exhibit consistent worst-case behavior for any one-to-one routing problem. The approach is based on the concept of *randomized routing* [90]. In randomized routing, each packet is intially sent to a random destination, and then it is greedily sent to its correct destination. Overall, then, each packet makes two passes through the network.

At first glance, the concept of randomized routing may not seem very appealing. After all, we have converted one routing problem into two (potentially doubling our workload) and we have deliberately sent packets where they do not want to go during the first pass (which seems wasteful). Upon closer examination, however, it becomes clear that randomized routing can be very useful for nasty worst-case problems such as transpose. This is because randomized routing converts any single one-to-one routing problem into a random routing problem followed by the reverse of a random routing problem (which is equivalent to a second random routing problem for the purposes of greedy routing). In other words, randomized routing converts each one-to-one problem into two random problems, which means that any N-packet one-to-one problem can be routed in $2\log N + o(\log N)$ steps on an N-input butterfly with high probability. (Here, the probability of failure depends on the choice of random intermediate destinations and not on the routing problem. If we are unlucky and pick bad intermediate destinations, then we just try again with new random intermediate destinations.)

More generally, we can use randomized routing to solve any routing problem with p packets at each input and p packets destined for each output in $2\log N + O(p) + o(\log N)$ steps on an N-input butterfly with high probability. Randomized routing can also be used to convert worst-case problems on arrays into average-case problems, with analogous results (depending on the diameter of the network).

Unfortunately, randomized routing cannot be used to solve the memory hot spot problem unless we are allowed to combine messages destined for the same output (even if the messages are not destined for the same memory location). Other techniques that involve memory reorganization (such as hashing) are needed to resolve this problem. In addition, randomized routing does not directly exploit locality (if it exists), although variations of technique (such as the routing algorithm implemented in the CM-5 [62]) can be used in a fashion that exploits locality. Lastly, randomized routing uses twice as much time as does the simple greedy algorithm for average-case problems, which is undesirable in practice. Nevertheless, randomized routing is beginning to gain some acceptance in practice. In particular, the new CM-5 is the first parallel machine to utilize the technique in its routing algorithm, and the C104

Transputer routing chip is designed to handle messages with two headers (one being a random intermediate address) [75].

3.3. The universality of hypercubic networks

We have just observed that an N-input butterfly can solve any routing problem with at most p packets starting at each input and at most p packets destined for each output in $O(\log N + p)$ steps with high probability. In fact, this bound is tight, which means that it takes the same amount of time (up to constant factors) to route one permutation as it does $\log N$ permutations on an N-input butterfly. Similar results hold for routing on an N-node hypercube as well as other hypercubic networks. Hence, these networks are most efficient when used to route $\Theta(N \log N)$ packets.

Since the N-input butterfly (or N-node hypercube) can route any $\log N$-relation in $O(\log N)$ steps with high probability, we can use an N-processor hypercube (or N-input butterfly) to simulate an $N \log N$-processor distributed-memory PRAM (or any $N \log N$-node fixed-connection network) with $O(\log N)$ slowdown, which is optimal up to constant factors (since the N-node hypercube has a factor of $\log N$ fewer processors). The simulation is straightforward. The ith processor of the hypercube is responsible for simulating $\log N$ processors (namely, processors $(i-1)\log N + 1, (i-1)\log N + 2, \ldots$, and $i \log N$ of the PRAM) for each i. Each step of the PRAM can be simulated with a $\log N$-to-one routing problem (which takes $O(\log N)$ steps with high probability) and $\log N$ computational steps on the hypercube. Hence, the hypercube and its derivative networks are *universal* in the sense that an N-processor hypercube can do whatever an arbitrary $N \log N$-processor distributed-memory machine can do with only $O(\log N)$ slowdown with high probability. (Technically speaking, the hypercube processors are more powerful than the PRAM processors since the hypercube processors can handle $\log N$ switching decisions per step, but we will ignore this issue for now.)

If hashing and/or combining are used, then the hypercube and its derivative networks can also simulate a CRCW PRAM in a similar fashion. Hence, hypercubic networks are provably close to optimal if the number of processors and wires is the only measure of cost. (Later, we will discuss other cost measures that result in different classes of networks being universal.) This is one of the main reasons that the hypercube and its derivative networks are such popular architectures for parallel machines. (Another reason that they are popular is that they contain many other natural networks, such as arrays and trees, as subgraphs. See [50] for more information and pointers.)

3.4. Randomly wired networks and the importance of expansion

In Section 3.2, we observed that the problems associated with worst-case performance of the greedy algorithm on the butterfly (and related networks) can be largely overcome by randomizing the memory and/or using randomized routing. Recently, an alternative approach has been discovered that is proving to be even more powerful – *randomized wiring*. For example, consider an N-input butterfly network in which the first level of wiring is scrambled so that the ith input is connected to two randomly

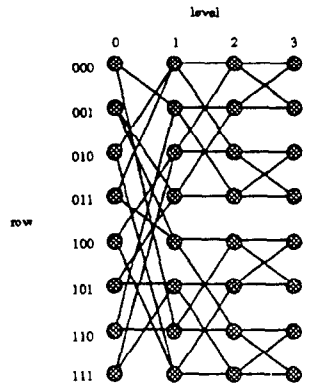

Fig. 2. An 8-input butterfly with a randomly wired first level.

selected nodes on the next level, one in the top $N/2$ nodes and one in the bottom half of the nodes. (In addition, suppose that the connections are constrained so that two inputs are connected to each node at the next level, in order to maintain regularity at the switches.) The remainder of the network is left unchanged (so that the upper half of the remaining network is still a butterfly as is the lower half of the remaining network). For example, see Fig. 2.

It is easy to see that there is a unique path of length $\log N$ from every input to every output in the modified butterfly just described. As in the ordinary butterfly, the path from an input to an output moves upward at a switch on the ith level ($0 \leqslant i < \log N$) if the $(i+1)$st bit of the destination is a 0, and downward otherwise. Hence, the greedy algorithm for routing on the modified butterfly is the same as was used for routing on an ordinary butterfly.

In fact, the only real difference between the modified butterfly and the ordinary butterfly is that the space of worst-case one-to-one routing problems (for the greedy algorithm) is randomized in the modified butterfly. This is because every packet is at a random position (within the top half or bottom half of the butterfly) after the first level of routing (since the connections at the first level are random). Hence, unless we are exceedingly unlucky in our choices for random connections, permutations such as transpose and bit reversal will be routed in $\log N + o(\log N)$ steps by the greedy algorithm on the modified butterfly. Of course, there are still worst-case permutations for the modified butterfly, but we are not likely to encounter them in practice.

It is worth noting that when we randomize wiring at the first level of the butterfly, we do not incur the negative side effects associated with randomized routing (e.g., the factor of two slowdown on average) or randomizing memory (e.g., loss of locality and the need to compute the hash function). In fact, by randomizing the wiring at every level of the butterfly (and adding wires to allow for redundant paths), far stronger properties can be attained. In particular, we can eliminate the existence of the worst-case (i.e., $\Omega(\sqrt{N})$-time) one-to-one routing problems altogether! We will describe how this feat can be accomplished in what follows.

3.4.1. Splitters, expanders, and multibutterflies

The butterfly and modified butterfly just described belong to a larger class of networks that are often referred to as *splitter networks* (or *multistage interconnection networks*). The switches on each level of a splitter network can be partitioned into *blocks*. All of the switches on level 0 belong to the same block. On level 1, there are two blocks, one consisting of the switches that are in the upper $N/2$ rows, and the other consisting of the switches that are in the lower $N/2$ rows. In general, the switches in a block B of size $M = N/2^l$ on level l have neighbors in two blocks B_u and B_l on level $l+1$, where u stands for *upper* and l for *lower*. The upper block B_u contains the switches on level $l+1$ that are in the same rows as the upper $M/2$ switches of B. The lower block B_l consists of the switches that are in the same rows as the lower $M/2$ switches of B. The edges from B to B_u are called the *up* edges, and those from B to B_l are called the *down* edges. The three blocks B, B_u, and B_l and the edges between them are collectively called a *splitter*. The switches in B are called the *splitter inputs*, and those in B_u and B_l are called the *splitter outputs*. In a splitter network with *multiplicity d*, each splitter input is incident to d outgoing up edges and d outgoing down edges, and each splitter output is incident to $2d$ incoming edges. For example, the butterfly and modified butterfly just described are splitter networks with multiplicity 1. In a *d-dilated butterfly*, the d up (and d down) edges incident to each splitter input all lead to the same splitter output, but there are better ways to connect the wires. For example, we have illustrated an 8-input splitter network with multiplicity 2 in Fig. 3. (More generally, we can have splitters with r blocks of outputs where $r \geqslant 2$ is the *radix* of the splitter. For the present discussion, however, we will confine ourselves to splitters with radix 2.)

In a splitter network, each input and output are connected by a single logical (up–down) path through the blocks of the network. For example, Fig. 4 shows the logical path from any input to output 011. In a butterfly (or any other splitter network with multiplicity 1), this logical path specifies a unique path through the network, since only one up and one down edge emanate from each switch. (In fact, a splitter

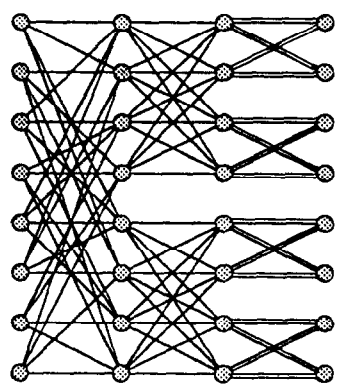

Fig. 3. An 8-input splitter network with multiplicity 2.

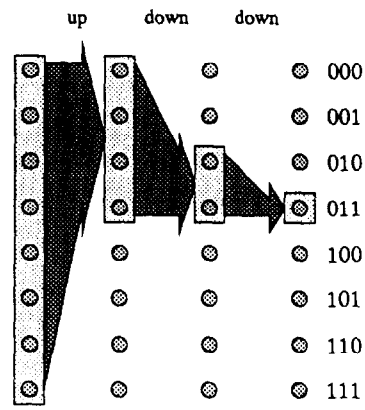

Fig. 4. The logical path from any input to output 011.

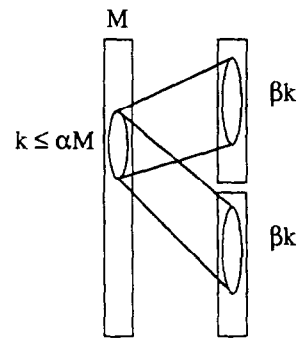

Fig. 5. An M-input splitter with (α, β)-expansion.

network with multiplicity one is very similar to a *delta network* [46].) In a general splitter network with multiplicity d, however, each switch will have d up and d down edges, and each step of the logical path can be taken on any one of d edges. Hence, one logical path can be realized by a myriad of physical paths in a general splitter network.

In what follows, we will be primarily concerned with randomly wired splitter networks. A *randomly wired splitter network* is a splitter network where the up and down edges within each splitter are chosen at random subject to the constraint that each splitter input is incident to d up and d down edges, and each splitter output is incident to $2d$ incoming edges.

The crucial property that randomly wired splitter networks are likely to possess is known as *expansion*. In particular, an M-input splitter is said to have (α, β)-expansion if every set of $k \leqslant \alpha M$ inputs is connected to at least βk up outputs and βk down outputs, where $\alpha > 0$ and $\beta > 1$ are fixed constants. For example, see Fig. 5.

A splitter network is said to have (α, β)-*expansion* if all of its splitters have (α, β)-expansion. More simply, a splitter or a splitter network is said to have *expansion*

if it has (α, β)-expansion for some constants $\alpha > 0$ and $\beta > 1$. A splitter network with expansion is more commonly known as a *multibutterfly* [87], and a *multibutterfly with (α, β)-expansion and multiplicity d* consists of splitters in which each splitter input is incident to d up and d down edges and for which any $k \leqslant \alpha M$ splitter inputs are adjacent to βk splitter outputs (where M is the size of the splitter).

Splitters with expansion are known to exist for any $d \geqslant 3$, and they can be constructed deterministically in polynomial time [36, 64, 87], but randomized wirings typically provide the best possible expansion. In fact, the expansion of a randomly wired splitter will be close to $d-1$ with probability close to 1, provided that α is a sufficiently small constant. (For a discussion of the tradeoffs between α and β in randomly wired splitters, see [51, 87].)

A multibutterfly with (α, β)-expansion is good at routing because one must block βk splitter outputs in order to block k splitter inputs. In classical networks such as the butterfly, the reverse is true: it is possible to block $2k$ inputs by blocking only k outputs. When this effect is compounded over several levels, the effect is dramatic. In a butterfly, a single fault can block 2^l switches l levels back, whereas in a multibutterfly, it takes β^l faults to block a single switch l levels back.

3.4.2. Routing results for multibutterflies and randomly wired splitter networks

Randomly wired splitter networks and multibutterflies have long been known to possess interesting properties, but have only recently been discovered to be an excellent choice as a message routing network. For example, in 1974, Bassalygo and Pinsker [10] used (randomly wired) splitter networks with expansion to construct the first nonblocking network of size $O(N \log N)$ and depth $O(\log N)$. (A *nonblocking network* is a network capable of connecting any unused input to any unused output with a path that does not overlap with existing paths between other inputs and outputs. For example, the telephone system would ideally be a nonblocking network.) Unfortunately, the Bassalygo–Pinsker result did not include a fast on-line algorithm for connecting the input/output pairs and so the result was of limited applicability, and interest in the networks waned. (For a survey of the early work on nonblocking networks and telephone switching networks, see [72].)

Interest in splitter networks with expansion was rekindled in 1989 when Upfal (who named the networks *multibutterflies* – a term attributed to Ron Fagin) showed that any N-input multibutterfly can route any one-to-one problem in the store-and-forward model in $O(\log N)$ steps using a variation of the greedy algorithm [87], and that, by using pipelining, any N-input multibutterfly can route $O(\log N)$ one-to-one problems in $O(\log N)$ steps. Although the proof was complicated and the constants hidden by the Big Oh notation were large, the result was important because the only previously known deterministic on-line linear-hardware $O(\log N)$-step packet routing algorithm [48] requires the use of the AKS sorting circuit [4] (which is even more complicated and has even larger constant factors).

Shortly afterwards, Leighton and Maggs [53, 54] provided a simple analysis of the greedy routing algorithm on multibutterflies, and derived smaller constant factors for

the $O(\log N)$ time bound. More importantly, they also showed that the multibutterfly is highly fault-tolerant. In particular, they proved that no matter how an adversary chooses f switches to fail, there will be at least $N-O(f)$ inputs and $N-O(f)$ outputs between which a simple variant of the greedy algorithm can route any $\log N$ permutations in $O(\log N)$ steps. Note that this is the best that one could hope for in general, since the adversary can always choose to isolate $\Omega(f)$ inputs and $\Omega(f)$ outputs by carefully selecting f faults. In the more commonly studied model of randomly located faults (e.g., see [33]), one can do even better. For example, even if $\Theta(N \log N)$ faults are randomly placed in the multibutterfly, with probability near 1, the network can still deterministically route any permutation on $\Theta(N)$ inputs and outputs. Thus the multibutterfly became the first bounded-degree network known to be able to sustain large numbers of faults with only minimal degradation in performance. (These results were also recently extended to hold for randomly wired splitter networks with multiplicity 2 [55]. The results are of interest since such networks do not have expansion – and, therefore, are not multibutterflies – and they are the networks of choice in practical implementations [15].)

Randomly wired splitter networks have also been found to be very useful for circuit switching. In particular, Arora et al. [8] developed a circuit-switching algorithm for a randomly wired splitter network that guarantees 100% throughput in $O(\log N)$ steps for any one-to-one routing problem. Arora et al. also showed how to on-line route any sequence of paths in a nonblocking fashion using back-to-back randomly wired splitter networks, thereby resolving the problem left open by Bassalygo and Pinsker.[4]

Perhaps the most important feature of the algorithms discovered for message routing on randomly wired splitter networks is that they work as well in practice as they do in theory. In fact, there is a growing body of experimental data that indicates that randomly wired splitter networks outperform more traditional networks with comparable hardware in many important respects [8, 15, 24, 44, 52, 54]. As a consequence, randomly wired networks appear to be an attractive substitute for hypercubic networks in many message routing applications. Construction of a 64-processor switching network based on this approach is currently underway at MIT [15, 16, 24].

3.4.3. Networks with multipath expansion

Chong and Knight [16] recently proposed an alternative to multibutterflies based on the notion of *multipath expansion*. In particular, they suggest construction of a splitter network for which the number of physical paths corresponding to each logical path is maximized. Such networks can be easily constructed and might appear to be superior to networks that are chosen to maximize ordinary expansion for routing purposes. Curiously, networks with multipath expansion need not have

[4] These results have recently been extended in "N. Pippenger, Self-routing superconcentrators, in: *Proc. 25th ACM Symp. on Theory of Computing* (1993) 355–361".

ordinary expansion and vice versa. In fact, networks with maximal multipath expansion can be constructed from ordinary butterflies, and can be shown to be subject to nasty worst-case behavior for large N. The networks have been found to perform well experimentally for $N \leqslant 1024$, however.

3.5. Other approaches to message routing

Thus far, we have focused our attention on some of the simplest and most widely used (or most promising) approaches to message routing. Many other approaches to the problem have been developed, however, and we will briefly survey some of them in what follows.

3.5.1. Routing by sorting

Given a fast algorithm for sorting N items, it is usually easy to derive a fast algorithm for routing N packets. This is because routing N packets to their destinations is often trivial if the packets have been presorted according to their destinations. In particular, for one-to-one routing problems, the greedy algorithm routes presorted packets in optimal time for both arrays and butterflies. This approach is used in the construction of Batcher–Banyon telephone switching networks such as the AT & T Starlite switch [35, 84].

In addition, by presorting packets based on destination, packets with the same destination can be gathered together in a fashion that makes the task of combining greatly simplified. (In fact, the combining can be accomplished using a parallel prefix operation. This approach has been used successfully on the CM-1 and CM-2.) Even if combining is not allowed, the effect of memory hot spots can be localized by presorting packets. For more information on routing by sorting, see [47, 50].

Unfortunately, sorting is a very challenging task in its own right. For a review of sorting algorithms on arrays and hypercubic networks, see [50].

3.5.2. The information dispersal approach to routing

Even though the greedy algorithm performs well for random routing problems, many packets will be delayed by congestion in most every routing problem. Hence, greedy algorithms typically require some form of queueing for most routing problems. By using a technique known as *information dispersal*, we can eliminate the delays associated with congestion for most one-to-one routing problems on butterflies. The idea behind information dispersal is to break up each packet into a collection of subpackets (often $\log N$ subpackets) which are routed in a greedylike fashion to their common destination along different paths. The advantage of the information dispersal approach is that the dispersal of large packets into many small packets tends to result in very balanced communication loads on the edge of the network. As a consequence, the maximum congestion in the network is likely to be so close to the average congestion (which is guaranteed to be low) that there is a good chance that packets will never be delayed at all. In addition, if we encode the contents of a packet into

a collection of subpackets in a redundant fashion, we will be able to make the algorithm highly fault-tolerant since only a fraction of the subpackets will have to reach the destination in order for the original packet to be reconstructed.

However, there are some significant costs associated with information dispersal. For example, by partitioning each packet into many subpackets, we dramatically increase the number of packets overall as well as the number of bits used for addressing and routing information. Hence, the approach is only suitable for applications where typical messages are long and the time to route a packet is closely correlated to the size of the packet. In addition, the method is not particularly useful in applications involving combining.

The information dispersal approach to routing was first proposed and analyzed by Rabin [77]. Rabin's original algorithm was subsequently simplified and strengthened by several researchers [33, 65, 66, 76]. For more information on the approach, see [50].

Information dispersal can also be quite useful as a tool for organizing memory in a way that prevents many of the problems associated with memory contention. As we have already seen, preventing *hot spots* where many packets need to access the same block of memory is one of the most challenging aspects of efficiently simulating a shared-memory machine (such as a PRAM) on a distributed-memory machine (such as a Connection Machine). As was discussed earlier, memory contention problems can be mitigated through combining, presorting packets according to destination, and/or randomizing memory. Information dispersal provides another highly effective tool for dealing with such problems. In particular, by encoding each item of data in the memory into R packets of data (each with $O(1/R)$ of the total size of the original packet) so that only $R/3$ of the packets are needed to reconstruct the original item, we can efficiently spread the storage of each item across R blocks of memory. When we try to write an item, we will be content to successfully write any $2R/3$ of the R pieces. Similarly, when we try to read an item, we will be content to read any $2R/3$ of the pieces (at least $R/3$ of which will be current, which is enough to reconstruct the item). By making R to be large (about $\log N$), contention can be handled by simply dropping packets that are headed for hot spots. (For more information on the information dispersal approach to memory organization, see [9, 50].)

3.5.3. Off-line routing

Thus far in our discussion of message routing, we have focussed our attention on on-line routing algorithms. This is because most parallel machines use on-line algorithms for routing. There are some machines (most notably, the IBM GF11 and the control network of the CM5), however, that make use of off-line routing algorithms.

Off-line routing algorithms differ from on-line algorithms in that the paths to be followed by the packets are computed by a processor with global information about the packet routing problem. Since it usually takes a long time (relatively speaking) to compute routing paths off-line, this approach is only useful if the routing problem is

known beforehand (as is the case with the GF11 – since the programmer must specify all memory access patterns ahead of time), or if the same routing problem will be performed many times, or if very long messages are being sent in a circuit-switching mode.

Finding good routing paths off-line is often very easy for natural networks such as arrays or butterflies. For example, Beneš [11] showed in 1964 that there exist edge-disjoint paths for any N-packet one-to-one routing problem on an N-input *Beneš network* (the network consisting of back-to-back butterflies). (Waksman [91] later showed how to find the paths quickly off-line. See [50] for more information.) Near optimal routing paths can also be found in polynomial time for arbitrary networks, although the methods are more difficult. (See [41, 57, 58] for more information.)

3.5.4. Area-universal networks and fat trees

For most part in this paper, we have measured the size of a network in terms of the number of nodes and wires it contains. While the numbers of processors, switches, and wires in a network are certainly valid measures of network cost, so are the silicon (VLSI) area and wiring volume of the network. The *area* of a network is the total area consumed by the chips and boards that make up the network, and the wiring *volume* is the total amount of three-dimensional space occupied by the network. Of course, the area and volume of a network are highly dependent on the number of nodes and wires contained in the network, but they are also highly dependent on the network itself. For example, an N-input butterfly consumes $\Theta(N^2)$ area and $\Omega(N^{3/2})$ volume (the latter bound is tight if we allow fully three-dimensional wiring), whereas an N-node 2-dimensional array consumes $\Theta(N)$ area and $\Theta(N)$ volume. Hence, an N-input butterfly can be significantly more costly to build than an N-node array for large N.

We have already seen that hypercubic networks are near-optimal (in particular, they are universal) in terms of their performance when compared to other networks with the same number of nodes and wires. Because of their large area and volume requirements, however, hypercubic networks are far from optimal when compared to other networks with the same amount of area or volume. (This is because, for example, we can fit an $N^{3/2}$-processor array into about the same volume as an N-processor hypercube.) Hence, we might ask which networks (if any) have near optimal (or universal) performance when compared to other networks with the same area or volume.

The answer to this question lies with a class of networks known as *fat trees*. A fat tree has a structure resembling that of a complete binary tree, except that the internal nodes and edges of the tree are replaced by clusters of nodes and channels of wires (respectively) whose size increases in the upper levels of the tree. There are also other networks that are area and volume-universal (such as the mesh of trees [50]), but fat trees appear to have the strongest universality properties. In fact, for every A, there is a fat tree with area A that can simulate any other network with area A (or a DRAM with area A) on-line with slowdown $O(\log A)$ with high probability [57].

The routing algorithms used on fat trees are very similar to the algorithms used on back-to-back butterflies, except that locality is exploited wherever possible. In fact, most of the approaches described thus far for butterflies can be directly implemented on fat trees. (For that matter, the butterfly itself can be considered to be a fat tree, albeit a very fat one.) In particular, *randomly wired fat trees* have all the same routing properties as randomly wired splitter networks (provided that the routing problem to be solved does not overload the capacity of internal channels in the fat tree).

As the sizes of parallel machines grow, it is likely that area-universal networks such as fat trees will increase in importance. In fact, fat trees have already been implemented in the new CM-5 in combination with the more traditional butterfly [62].

The fat tree approach to message routing was first proposed by Leiserson [59]. For more information on this subject, see [30, 31, 60]. In particular, [60] provides an excellent survey of area-universal networks.

3.5.5. Cut-through routing on dilated tori

Motivated by the same area and volume constraints that led to the development of fat trees, several architects (led by Dally and Seitz) have opted for a dilated torus in place of a hypercubic network. A *q-dilated torus* is a torus (usually with 2 or 3 dimensions) where every edge is replaced by a channel with q wires. The bisection width of an N-node q-dilated r-dimensional torus is $2qN^{1-1/r}$. Hence, the bisection width of an $N^{1/r}$-dilated r-dimensional torus is the same for all r. In addition, the area of an $N^{1/r}$-dilated r-dimensional torus is about the same for all r (including the case when $r = \log N$, where we have a hypercube). Hence, we would like to choose the value of r for which the routing performance is optimized.

Dilated tori are typically used in conjunction with a cut-through or worm-hole routing algorithm and with long messages. Given a message between two random points u and v in the network, we will need (on average) at least $(rN^{1/r}/4)$ steps to send the message from u to v using the greedy algorithm since the expected distance between two random points in the network is this large. If the message has length L and we use all $N^{1/r}$ wires in each channel along a shortest path from origin to destination to transmit the message in pipelined fashion, the entire message can be sent in about $rN^{1/r}/4 + L/N^{1/r}$ steps on average. Using simple calculus, we find that this value (the average latency of a single message) is minimized by selecting

$$r = \frac{\log N}{\log\left(\frac{2L}{\log N}\right) + \Theta\left(\log\log\left(\frac{2L}{\log N}\right)\right)}.$$

For relatively small values of N and large values of L, r is typically 2 or 3. For example, if $N = 16000$ and $L = 150$ (these are typical numbers for the MIT J-Machine [19]), then the latency is minimized by choosing $r = 3$ (which is why the J-machine is a 3-dimensional torus).

Of course, the preceding analysis neither accounts for delays caused by congestion nor for costs associated with $N^{1/r}$-ary switches. However, similar heuristic calculations have been made [19] that indicate that the 3-dimensional torus is a good routing network when area constraints are taken into account and typical message lengths are long. As a consequence, machines such as the Ametek 2010 and Intel Touchstone are configured as low-dimensional arrays. (Whether or not there is a more rigorous and/or scalable analysis which justifies this choice remains an interesting open question.) For more information on this important approach to message routing, see [19, 20, 21, 22, 70].

Acknowledgment

I would like to thank Alok Aggarwal, Fred Chong, Faith Fich, Charles Leiserson, Danny Sleator and Quentin Stout for helpful comments and pointers. I am particularly grateful to Bruce Maggs for numerous helpful corrections and references.

References

[1] F. Abolhassan, J. Keller and W. Paul, On the cost-effectiveness and realization of the theoretical PRAM model, Tech. Report 09/1991, FB Informatik, Universität des Saarlandes, 1991.

[2] A. Aggarwal, A. Chandra and M. Snir, On communication latency in PRAM computations, in: *Proc. ACM Symp. on Parallel Algorithms and Architectures*; *SIAM J. Comput.*, to appear.

[3] A. Aggarwal, A. Chandra and M. Snir, Communication complexity of PRAMs, *Theoret. Comput. Sci.* **71** (1990) 3–28.

[4] M. Ajtai, J. Komlos and E. Szemeredi, Sorting in $c \log n$ parallel steps, *Combinatorica* **3** (1983) 1–19.

[5] R. Aleliunas, Randomized parallel communication, in: *Proc. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing* (1982) 60–72.

[6] G. Almasi and A. Gottlieb, *Highly Parallel Computing* (Benjamin Cummings, Redwood City, CA, 1989).

[7] B. Alpern, L. Carter, E. Feig and T. Selker, A uniform memory hierarchy model of computation, Tech. Report, IBM Watson, 1990.

[8] S. Arora, T. Leighton and B. Maggs, On-line algorithms for path selection in a nonblocking network, in: *Proc. 22nd Ann. ACM Symp. on Theory of Computing* (1990) 149–158.

[9] Y. Aumann and A. Schuster, Deterministic PRAM simulation with constant memory blow-up and no time-stamps, in: *Proc. 3rd Symp. on the Frontiers of Massively Parallel Computation* (1990) 22–29.

[10] L.A. Bassalygo and M.S. Pinsker, Complexity of an optimum non-blocking switching network without reconnections, *Problems Inform. Transmission* **9** (1974) 64–66.

[11] V. Beneš, Permutation groups, complexes, and rearrangeable multistage connecting networks, *Bell System Tech. J.* **43** (1964) 1619–1640.

[12] G.E. Blelloch, *Vector Models for Data-Parallel Computing* (MIT Press, Cambridge, MA, 1990).

[13] A. Borodin and J. Hopcroft, Routing, merging, and sorting on parallel models of computation, *J. Comput. System Sci.* **30** (1) (1985) 130–145.

[14] A. Chin, Practical issues in parallel complexity, Ph.D. Thesis, Univ. of Oxford, England, 1991.

[15] F. Chong, E. Egozy and A. DeHon, Fault tolerance and performance of multipath multistage interconnection networks, in: T.F. Knight, Jr. and J. Savage, eds., *Advanced Research in VLSI and Parallel Systems 1992* (MIT Press, Cambridge, MA, 1992).

[16] F.T. Chong and T. Knight, Jr., Design and performance of multipath MIN architectures, Tech. Report 64, MIT Artificial Intelligence Laboratory, 1992.

[17] R. Cole and O. Zajicek, The APRAM: incorporating asynchrony into the PRAM model, in: *Proc. 1st ACM Symp. on Parallel Algorithms and Architecture* (1989) 169–178.
[18] R. Cole and O. Zajicek, The expected advantage of asynchrony, in: *Proc. 2nd ACM Symp. on Parallel Algorithms and Architecture* (1990) 85–94.
[19] W. Dally, Network and processor architecture for message-driven computers, in: R. Suaya and G. Birtwhistle, eds., *VLSI and Parallel Computation*, Ch. 3 (Morgan Kaufman, San Mateo, CA, 1990) 140–222.
[20] W. Dally, Express cubes: improving the performance of k-ary n-cube interconnection networks, *IEEE Trans. Comput.*, to appear.
[21] W. Dally, Virtual-channel flow control, *IEEE Trans. Parallel and Distributed Systems*, to appear.
[22] W. Dally and C. Seitz, Deadlock free message routing in multiprocessor interconnection networks, *IEEE Trans. Comput.* **C-36** (1987) 547–553.
[23] A. DeHon, Mbta: modular bootstrapping transit architecture, Tech. Report 17, MIT Artificial Intelligence Laboratory, 1990.
[24] A. DeHon, T. Knight, Jr. and H. Minsky, Fault-tolerant design for multistage routing networks, in: *Proc. Internat. Symp. on Shared Memory Multiprocessing* (Information Processing Society of Japan, 1991).
[25] F. Fich, The complexity of computing on a parallel random access machine, in: J. Reif, ed., *Synthesis of Parallel Algorithms* (Morgan Kaufman, San Mateo, CA, 1993).
[26] F. Fich, R. Impagliazzo, B. Kapron, V. King and M. Kutylowski, Limits on the power of PRAMs with weak forms of write conflict resolution, unpublished manuscript, 1992.
[27] F. Fich, P. Ragde and A. Wigderson, Relations between concurrent write models of parallel computation, *SIAM J. Comput.* **17** (1988) 606–627.
[28] P. Gibbons, The asynchronous PRAM: a semisynchronous model for shared memory MIMD machine, Ph.D. Thesis, Univ. of California, Berkeley, 1989.
[29] A. Gottlieb, An overview of the NYU ultracomputer project, in: J. Dongarra, ed., *Experimental Computing Architecture* (Elsevier, Amsterdam, 1987) 25–95.
[30] R.I. Greenberg, Efficient interconnection schemes for VLSI and parallel computation, Ph.D. Thesis, MIT, 1989.
[31] R.I. Greenberg and C.E. Leiserson, Randomized routing on fat-trees, *Adv. Comput. Res.* **5** (1989) 345–374.
[32] T. Hagerup and T. Radzik, Every robust CRCW PRAM can efficiently simulate a priority PRAM, in: *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures* (1990) 117–124.
[33] J. Håstad, T. Leighton and M. Newman, Fast computation using faulty hypercubes, in: *Proc. 21st Ann. ACM Symp. on Theory of Computing* (1989) 251–263.
[34] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann, San Mateo, CA, 1990).
[35] A. Huang and S. Knauer, Starlite: a wideband digital switch, in: *Proc. GLOBECOM 84* (1984) 121–125.
[36] N. Kahale, Better expansion for Ramanujan graphs, in: *Proc. 32nd Ann. IEEE Symp. on Foundations of Computer Science* (1991) 398–404.
[37] C. Kaklamanis, D. Krizanc and T. Tsantilas, Tight bounds for oblivious routing in the hypercube, in: *Proc. 2nd Ann. ACM Symp. on Parallel Algorithms and Architectures* (1990) 31–36.
[38] A. Karlin, M. Manasse, L. Rudolph and D. Sleator, Competitive snoopy caching, *Algorithmica* **3** (1988) 79–119.
[39] A. Karlin and E. Upfal, Parallel hashing – an efficient implementation of shared memory, in: *Proc. 18th Ann. ACM Symp. on Theory of Computing* (1986) 160–168.
[40] R. Karp, Parallel combinatorial computing, in: J.P. Mesirov, ed., *Very Large Scale Computation in the 21st Century* (SIAM, Philadelphia, PA, 1991) 221–238.
[41] P. Klein, A. Agrawal, R. Ravi and S. Rao, Approximation through multicommodity flow, in: *Proc. 31st FOCS* (1990) 726–737.
[42] R. Koch, Increasing the size of a network by a constant factor can increase performance by more than a constant factor, in: *Proc. 29th Ann. IEEE Symp. on Foundations of Computer Science* (1988) 221–230.
[43] R. Koch, An analysis of the performance of interconnection networks for multiprocessor systems, Ph.D. Thesis, MIT, 1989.

[44] S. Konstantinidou and E. Upfal, Experimental comparison of multistage networks, Tech. Report, IBM Almaden Research Center, 1991.

[45] C. Kruskal and M. Snir, The performance of multistage interconnection networks for multiprocessors, *IEEE Trans. Comput.* **C-32** (1983) 1091–1098.

[46] C. Kruskal and M. Snir, A unified theory of interconnection network structure, *Theoret. Comput. Sci.* **48** (1986) 75–94.

[47] M. Kunde, Routing and sorting on mesh-connected arrays, in: J. Reif, ed., *Proc. 3rd Aegean Workshop on Computing: VLSI Algorithms and Architectures*, Lecture Notes in Computer Science Vol. 319 (Springer, Berlin, 1988) 423–433.

[48] T. Leighton, Tight bounds on the complexity of parallel sorting, *IEEE Trans. Comput.* **C-34** (1985) 344–354.

[49] T. Leighton, Average case analysis of greedy routing algorithms on arrays, in: *Proc. 2nd Ann. ACM Symp. on Parallel Algorithms and Architectures* (1990) 2–10.

[50] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes* (Morgan Kaufman, San Mateo, CA, 1992).

[51] T. Leighton, C.L. Leiserson and M. Klugerman, Theory of parallel and VLSI computation, Research Seminar Series Report MIT/LCS/RSS10, MIT Laboratory for Computer Sciences, 1991.

[52] T. Leighton, D. Lisinski and B. Maggs, Empirical evaluation of randomly wired multistage networks, in: *Proc. IEEE Internat. Conf. on Computer Design* (1990) 380–385.

[53] T. Leighton and M. Maggs, Expanders might be practical: fast algorithms for routing around faults in multibutterflies, in: *Proc. 30th Ann. IEEE Symp. on Foundations of Computer Science* (1989) 384–389.

[54] T. Leighton and B. Maggs, Fast algorithms for routing around faults in multibutterflies and randomly-wired splitter networks *IEEE Trans. Comput.* **41** (5) (1992) 578–587.

[55] T. Leighton and B. Maggs, *Introduction to Parallel Algorithms and Architectures: Expanders, PRAMs and VLSI* (Morgan Kaufman, San Mateo, CA, to appear).

[56] T. Leighton, B. Maggs, A. Ranade and S. Rao, Randomized algorithms for routing and sorting in fixed-connection networks, *J. Algorithms*, to appear.

[57] T. Leighton, B. Maggs and S. Rao, Universal packet routing algorithms, in: *29th Ann. IEEE Symp. on Foundations of Computer Science* (1988) 256–271.

[58] T. Leighton and S. Rao, An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms, in: *29th Ann. IEEE Symp. on Foundations of Computer Science* (1988) 422–431.

[59] C.E. Leiserson, Fat-trees: universal networks for hardware-efficient supercomputing, *IEEE Trans. Comput.* **C-34** (1985) 892–901.

[60] C.E. Leiserson, VLSI theory and parallel supercomputing, in: R.F. Rashid, ed., *Carnegie Mellon University School of Computer Science 25th Anniversary Symp.* (Addison-Wesley, Reading, MA, 1991) 29–44.

[61] C.E. Leiserson and B.M. Maggs, Communication-efficient parallel algorithms for distributed random-access machines, *Algorithmica* **3** (1988) 53–57.

[62] C. Leiserson et al., The network architecture of the connection machine CM-5, *J. Parallel Distributed Computing*, to appear.

[63] H. Li and Q. Stout, Reconfigurable SIMD massively parallel computers, *Proc. IEEE* **79** (1991) 429–443.

[64] A. Lubotzky, R. Phillips and P. Sarnak, Ramanujan graphs, *Combinatorica* **8** (1988) 261–277.

[65] Y.-D. Lyuu, Fast fault-tolerant parallel communication and on-line maintenance using information dispersal, in: *Proc. 2nd Ann. ACM Symp. on Parallel Algorithms and Architectures* (1990) 378–387.

[66] Y.-D. Lyuu, An information dispersal approach to issues in parallel processing, Ph.D. Thesis, Harvard Univ. 1990.

[67] B. Maggs and R. Sitaraman, Simple algorithms for routing on butterfly networks with bounded queues, in: *Proc. 1992 ACM Symp. on Theory of Computing* (1992) 150–161.

[68] H. Minsky, A. DeHon and T.F. Knight Jr., RN1: low-latency, dilated, crossbar router, in: *Hot Chips Symp. III*, 1991.

[69] D. Nassimi and S. Sahni, A self-routing Benes network and parallel permutation algorithms, *IEEE Trans. Comput.* **C-30** (1981) 332–340.

[70] J. Ngai and C. Seitz, A framework for adaptive routing in multicomputer networks, in: *Proc. ACM Symp. on Parallel Algorithms and Architectures* (1989) 1–9.
[71] N. Nishimura, Asynchronous shared memory parallel computation, in: *2nd ACM Symp. on Parallel Algorithms and Architectures* (1990) 76–84.
[72] N. Pippenger, Telephone switching networks, in: *Proc. AMS Symp. in Applied Mathematics*, Vol. 26 (1982) 101–133.
[73] N. Pippenger, Parallel communication with limited buffers, in: *Proc. 25th Ann. IEEE Symp. on Foundations of Computer Science* (1984) 127–136.
[74] N. Pippenger, Communication networks, in: J. Van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A: Algorithms, and Complexity* (MIT Press, Cambridge, 1990) 805–834.
[75] D. Pountain, The transputer strikes back, *Byte Magazine* (1991) 265–275.
[76] F. Preparata, Holographic dispersal and recovery of information, *IEEE Trans. Inform. Theory* **IT-35** (1989) 1123–1124.
[77] M. Rabin, Efficient dispersal of information for security, load balancing, and fault tolerance, *J. ACM* **36** (1989) 335–348.
[78] A. Ranade, How to emulate shared memory, in: *Proc. 28th Ann. IEEE Symp. on Foundations of Computer Science* (1987) 185–194.
[79] A. Ranade, Fluent parallel computation, Ph.D. Thesis, Yale Univ., New Haven, CT, 1988.
[80] A. Ranade, S. Bhatt and L. Johnson, The fluent abstract machine, in: *Advanced Research in VLSI: Proc. 5th MIT Conf.* (MIT Press, Cambridge, MA, 1988) 71–94.
[81] R. Rettberg, W. Crowther, P. Carvey and R. Tomlinson, The monarch parallel processor hardware design, *Computer* **23** (1990) 18–30.
[82] B. Smith, A massively parallel shared memory machine, in: *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, 1991, invited lecture.
[83] G. Stamoulis and J. Tsitsiklis, The efficiency of greedy routing in hypercubes and butterflies, in: *Proc. 3rd Ann. ACM Symp. on Parallel Algorithms and Architectures* (1991) 248–259.
[84] F. Tobagi, Fast packet switch architectures for broadband integrated services digital networks, *Proc. IEEE* **78** (1990) 133–167.
[85] T. Tsantilas, A refined analysis of the Valiant–Brebner algorithm, Tech. Report TR-22-89, Center for Research in Computing Technology, Harvard Univ., 1989.
[86] E. Upfal, Efficient schemes for parallel communication, in: *Proc. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing* (1982) 55–59.
[87] E. Upfal, An $O(\log N)$ deterministic packet routing scheme, in: *Proc. 21st Ann. ACM Symp. on Theory of Computing* (1989) 241–250; *J. ACM*, to appear.
[88] L. Valiant, A bridging model for parallel computation, *Comm. ACM* **33** (8) (1990) 103–111.
[89] L.G. Valiant, General purpose parallel architectures, in: J. Van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A: Algorithms, and Complexity* (MIT Press, Cambridge, 1990) 943–972.
[90] L. Valiant and G. Brebner, Universal schemes for parallel communication, in: *Proc. 13th Annual ACM Symp. on Theory of Computing* (1981) 263–277.
[91] A. Waksman, A permutation network, *J. ACM* **15** (1) (1968) 159–163.