Mapping Vision Algorithms to Parallel Architectures

QUENTIN F. STOUT, MEMBER, IEEE

Invited Paper

Because parallel architectures can vary widely, the problem of mapping a parallel algorithm onto a given parallel machine is generally much harder than the problem of mapping a serial algorithm onto a serial machine. This paper examines some of the problems encountered, emphasizing mappings of vision algorithms onto mesh, hypercube, mesh-of-trees, pyramid, and Parallel Random Access Machines (PRAMs) having many simple processors, each with a small amount of memory. Approaches that have been suggested include simulating one architecture on another, designing ideal algorithms for ideal architectures and simulating the ideal architectures, and using general data movement operations. Each of these is shown to occasionally produce unacceptably inefficient implementations. It appears that as long as PRAMs cannot achieve the desired cost and performance goals, programmers must contend with carefully designing algorithms for specific architectures.

I. Introduction

Programming parallel machines is difficult for a great many reasons. Programmers trained to program sequential computers must unlearn many of their prejudices concerning the best approach to problems. Few languages for parallel programming have been developed, and they are not as mature as serial languages. Debugging parallel algorithms is complicated because it is difficult to determine the state of a parallel machine, and it is difficult to maintain a mental picture of what the correct state should be. Finally, parallel machines vary much more widely than serial ones, making it difficult to transfer programs from one to another.

This paper is concerned with this last difficulty, examining some of the techniques that are currently being used to map algorithms onto specific architectures. The problems and architectures are all concerned with vision and image processing, though many of the observations are of more general applicability. No fully satisfactory solutions are uncovered, and it seems that none will appear soon.

In general, one might describe an algorithm at a great many levels. At very high levels one has paradigms such as divide-and-conquer, dynamic programming, and branch-and-bound, which suggest approaches to classes of problems. As more details are given the paradigm becomes an algorithm for a specific problem, and eventually an implementation for a specific machine. The higher the level of description, the more malleable it is, and hence the more likely that an efficient implementation can be developed for a given target architecture. However, it is also true that the higher the level of description, the more work the programmer must do to translate the description into usable form, which requires greater understanding and decreases programmer productivity.

Starting at a very low level, one may have an algorithm currently running on parallel machine A and want to transform it to run on parallel machine B. The easiest technique is to develop a simulation of A on B, for then all of A's algorithms can be run on B after developing only one simulation. Unfortunately, A may not have been a particularly good architecture for the problem, and the simulation only compounds the inefficiency. At a slightly higher level, one may develop an "ideal" parallel algorithm (along with an "ideal" architecture for the algorithm), and then simulate this ideal algorithm on each target machine. This presumably eliminates starting with an inefficient solution, but requires repeatedly developing simulations of ideal architectures on a given target architecture. For this to be practical, it would be helpful to have programs which can map the ideal architecture onto the target architecture, a problem known as the mapping problem [7], [9], [41]. At a yet higher level one may describe an algorithm in terms of standard operations which manipulate and move the data between processors, and then develop an efficient implementation of each operation for the target machine. This gives more flexibility than in mapping a fixed architecture to the target architecture, but effectively utilizing this flexibility requires more programmer insight.

Each of these approaches will be examined by considering a few examples. Throughout the input will be a black/white image, stored one pixel per processor. This is a *fine-grained* model of parallel computing, as opposed to a coarse-grained model in which there are few processors, each with a large amount of memory. Image-processing

Manuscript received January 30, 1988; revised March 22, 1988. This research was supported by National Science Foundation Grant DCR-85-07851, and an Incentives for Excellence award from Digital Equipment Corporation.

The author is with the Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109-2122, USA.

IEEE Log Number 8822801.

0018-9219/88/0800-0982$01.00 © 1988 IEEE

computers such as the CLIP series, MPP, and Connection Machine are all fine-grained [6], [10], [15], [52], [57], and analyzing fine-grained machines allows us to concentrate on the issues involved in the parallelism, as opposed to the mix of serial and parallel computing inherent in coarse-grained machines. We assume that the machines are SIMD (Single Instruction Multiple Data), in which a central controller issues the same instructions to all processors, since most fine-grained vision-processing machines are of this form. Medium- and coarse-grain machines such as most parallel machines designed for scientific applications, or multiprocessor mainframes, are MIMD (Multiple Instruction Multiple Data) machines in which each processor has its own set of instructions, as well as its own data [17], [35], [43]. Converting between MIMD and SIMD machines is an important mapping problem, but due to space limitations it will not be considered here. We also force to omit any consideration of mapping a larger problem onto a given machine, of dynamically reassigning tasks to processors in response to processor or communication load imbalances, of designing an architecture to be efficient for a collection of algorithms, of simulating multiple copies of an algorithm, or of designing parallel languages to increase portability. Each of these is important and is the subject of current and proposed research [7], [9], [12], [16], [22], [24], [31], [37], [41], [45]–[47], [49], [50], [62].

Analyses will always be of worst-case time. Worst-case time can be unduly pessimistic, but it will be used here because it is easier to analyze. The observations and conclusions about mapping algorithms would hold for expected case time if such a concept could ever be defined for vision problems. Similarly, we concentrate on relatively simple, well-defined problems which are much more analyzable than most real vision algorithms. Our emphasis is not on taking a useful vision algorithm and mapping it to various architectures, but rather analyzing the process of finding such mappings and trying to determine what makes the process complicated for the programmer and the result inefficient for the machine. Vision algorithms suitable for immediate use on real images are even more difficult to transfer and optimize.

In the next section the machine models and notation are introduced. Parallel architectures utilized are the mesh, pyramid, mesh-of-trees, hypercube, and Parallel Random Access Machine (PRAM). Section III considers lower bounds which arise because of the communication capabilities of these architectures, and Section IV considers mappings between the architectures. Section V examines developing ideal parallel algorithms/architectures and mapping them onto target architectures. Section VI considers developing algorithms in terms of general data movement operations, and implementing them on target architectures. Section VII concludes with some observations.

II. Definitions

The input image will be an $n \times n$ black/white image, stored one pixel per processor. To simplify analyses, n will always be an integral power of 2. Analyses will be of worst-case time measured in terms of n, analyzed using O, Ω, and Θ. For positive functions $f(n)$ and $g(n)$, $f = O(g)$ if there are constants $N, D > 0$ such that $f(n) \leq D * g(n)$ for all $n \geq N$, $f = \Omega(g)$ if there are constants $N, C > 0$ such that $C*g(n) \leq f(n)$ for all $n \geq N$, and $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$. The O, Ω, and Θ symbols are sometimes read "order no more than," "order no less than," and "order equal to." We use lg to denote $\log_2$.

The *mesh* computer is a 2-dimensional grid of processors, each connected to its 4 nearest neighbors (except for edge processors which have fewer connections). See Fig. 1(a).

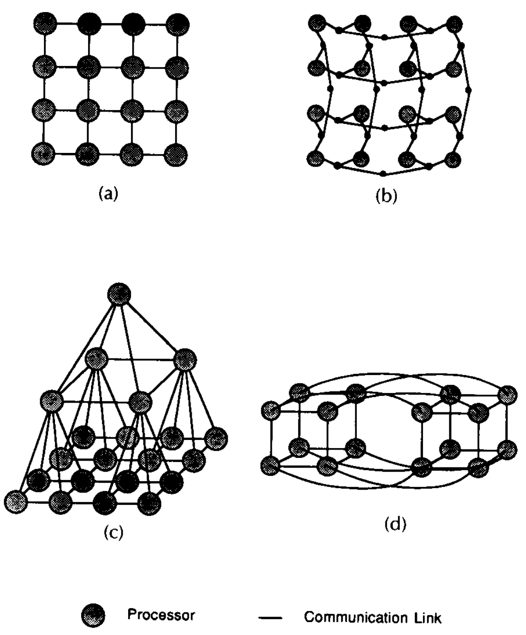

Fig. 1. Some regular parallel computers for 4×4 images. (a) Mesh. (b) Mesh of trees (base mesh connections omitted for clarity). (c) Pyramid. (d) Hypercube.

Throughout it is assumed that the mesh has exactly n^2 processors. There are extensive analyses of meshes as vision architectures, and many have been built for this purpose [6], [8], [15], [26], [39], [33], [57].

The *mesh-of-trees* starts with a base which is a mesh of n^2 processors, and then a tree is added over each row and each column. These trees are disjoint except for their leaves, which are the base processors. See Fig. 1(b). The mesh-of-trees has a total of $3n^2 - 2n$ processors. It was originally proposed as a general-purpose parallel computer ideal for VLSI implementation [56], and is only recently receiving attention as a vision processing architecture [28], [36].

The *pyramid* computer also starts with a base mesh of n^2 processors, denoted level 0, and then adds levels of meshes above. Each level has 1/4 as many processor as the level below, and each processor (except those on the base) is connected to four children on the level below. See Fig. 1(c). The apex of the pyramid is at level $\lg(n)$. The pyramid has a total of $(4/3)n^2 - 1/3$ processors. A few pyramids have been built for vision processing, and there has been a fairly extensive analysis of pyramidal vision algorithms for serial and parallel computers [10], [11], [27], [40], [42], [51], [54], [55].

The *hypercube* computer with n^2 processors has each connected to $\lg(n^2)$ others. Processors are assigned labels which are binary strings of length $\lg(n^2)$, and two processors are connected if and only if their labels differ in exactly one bit. See Fig. 1(d). While most hypercubes are currently being used for scientific applications [43], there is increasing use and analysis of them for vision applications [17], [28], [34], [52].

The mesh, mesh-of-trees, pyramid, and hypercube all have the property that they have a systematic interconnection scheme which can be represented very simply in formal grammers. They have numerous useful properties such as the fact that they can be subdivided into smaller machines of the same type. They (and others like them) are sometimes known as *regular architectures*. Distributed systems such as local area networks are often irregular, and are difficult to coordinate on a single task. Irregular systems will not be considered here.

The *PRAM* (Parallel Random Access Machine) is really a class of architectures. The standard model is as in Fig. 2(a),

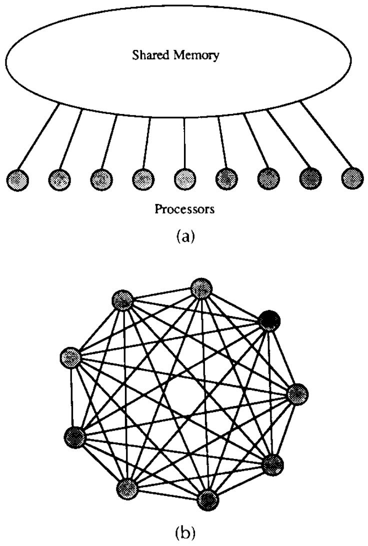

Fig. 2. Parallel random access machines for 3 × 3 images. (a) A shared-memory PRAM. (b) A distributed memory PRAM.

where there is a global shared memory accessible in parallel by all processors. The primary variations in this model are whether or not two or more processors can read the same memory location simultaneously (Concurrent Read, CR, versus Exclusive Read, ER), and whether or not two or more processors can write to the same memory location simultaneously (Concurrent Write, CW, versus Exclusive Write, EW). No processor can read a location while another processor is writing to it. If concurrent writing is allowed, then there are variations as to which value ends up being written into memory. For example, the minimum value may end up there, or one of the values sent may be chosen arbitrarily. For our purposes the latter is sufficient.

Another model of a PRAM is a distributed memory model, as in Fig. 2(b), where all memory is at the processors and processors communicate directly with each other, as opposed to indirectly via shared memory on the standard model. Here processors send messages requesting or transmitting values, instead of reading or writing directly from memory, and again one can consider concurrent versus exclusive options. One can have concurrent sends, in which the same value is sent to all processors on a list (perhaps with restrictions on the form this list can take), versus exclusive sends where only one message at a time may be sent, and concurrent receives, in which multiple simultaneous incoming messages are permtted, versus exclusive receives. In the case of concurrent receives one needs to specify which message reaches the processor. Again, for our purposes it suffices to assume that the message is chosen arbitrarily.

If the memory/processor ratio is large, then the differences between the two types of PRAM are significant when many processors want to read different memory locations. In the former model this is always possible in unit time, but in the latter model they may have to queue up if they are all trying to fetch values stored in the same processor. However, in a fine-grained model where the memory/processor ratio is a constant the differences are quite minor, and for our purposes either model is acceptable. In either model, we assume that there are n^2 processors. The RP3 is a simulation of a shared memory PRAM [35], while the Connection Machine is sometimes described as a simulation of a distributed memory PRAM [52]. Both of these machines are to be used for numerous applications, including vision processing [19, 52].

Finally, the $n \times n$ input image is assumed to be initially present in the mesh, stored one pixel per processor in the standard manner. For the pyramid and mesh-of-trees the image is initially in the base. For the hypercube it is stored using the Gray code discussed in Section IV. For the shared memory PRAM the image starts in the shared memory, and for the distributed memory PRAM the pixels are assigned to processors using row-major ordering.

III. Lower Bounds

One important aspect of designing efficient algorithms and determining constraints on their portability is to understand the various lower bound arguments that can apply. There are primarily three techniques that are used, based on speedup, communication radius, and bandwidth or cutset considerations.

The *speedup bound* means that a machine with P processors should take at least T/P time to solve a problem requiring T time on a single processor. To apply this one must be careful to make sure that the comparisons are fair (e.g., that the single processor have as much memory as all P processors combined), and to understand the anomalies that can occur if the serial algorithm is not the best possible (e.g., some search heuristics may lead the serial processor astray [23]). When applied to sorting, the speedup bound shows that n processors must take $\Omega(\log n)$ time to sort n numbers, no matter how the processors are arranged. For the image processing problems considered here, however, an $n \times n$ image can be processed in $\Theta(n^2)$ time on a serial processor, and hence speedup for n^2 processors gives the trivial lower bound of $\Omega(1)$.

To define communication radius, fix a machine M, let $D(p, q)$ denote the minimum number of communication links on any path from processor p to processor q in M, and let $R(p)$, the radius at p, equal max $\{D(p, q)$: q a processor in $M\}$. $R(p)$ is the largest number of communication links needed to communicate from p to any other processor. The *communication radius* of M is the mimimum value of $R(p)$

over all processors p in M. For example, on the mesh the communication radius is achieved at the four center processors, where each takes n hops to go to the furthest corner. For the pyramid, mesh-of-trees, and hypercube the communication radius is $\Theta(\log n)$. Given any nontrivial problem in which one processor is to end up with a value depending on values initially in all of the processors, such as sorting or determining the average gray level of an image, any algorithm must take time proportional to the communication radius. As will be shown below, each of these architectures can achieve this lower bound on simple problems such as determining average gray level.

Cutset or *bandwidth* lower bounds are established by considering some collection C of processors, counting the number of wires connecting processors in C to the processors not in C, and determining the number of values that must pass over these wires. The time required can be no less than the number of items divided by the number of wires. For example, on a mesh, if each processor starts with an item and these are to be sorted, then considered the region consisting of the left half. There are exactly n wires connecting it to the right half, and in the worst case all $n^2/2$ items on the left must move to the right. Therefore at least $n/2$ time units are needed, so sorting takes $\Omega(n)$ time. Cutting the pyramid or mesh-of-trees in half shows that they too must take $\Omega(n)$ time to sort since they only double the number of wires crossing between the halves. For the hypercube these arguments only show that sorting must take $\Omega(1)$ time, since any collection of $P \leq n^2/2$ nodes has $\Omega(P)$ wires connecting them to the other nodes.

Because the mesh can sort in $\Theta(n)$ time [53], the sorting lower bounds can be attained for the mesh, pyramid, and mesh-of-trees. However, the best known sorting algorithm for the hypercube is bitonic sort, which takes $\Omega(\log^2 n)$ time [5]. This is larger than any of the hypercube lower bounds, and it is an interesting open question whether the hypercube can sort faster.

One slightly different lower bound that will be used later is that a pyramid computer needs $\Omega((n \log n)^{1/2})$ time to move $n \log n$ items from the leftmost column to the rightmost one. This was proven in [27] by an argument that is a blend of the radius and cutset arguments using weighted links.

IV. Stepwise Simulation

The simplest technique for mapping an algorithm designed for architecture A onto a target architecture B is to determine an efficient mapping of A onto B, and then perform a stepwise simulation of A. This has the advantage that algorithms currently in use are presumably already well understood and implemented correctly, and it also has the advantage that once the mapping has been determined, all algorithms for A can be implemented on B. Because of this reusability, it is reasonable for a programmer to spend considerable time optimizing the mapping.

Abstractly measuring the efficiency of a mapping is often done in terms of factors such as the *processor load* (maximum number of processors from A simulated by a single processor of B), *link load* (maximum number of communication links from A mapped onto a single communication link of B), *dilation* (maximum length in B of the image of a communication link in A), and *expansion* (number of processors in B divided by number of processors in A) [7], [9]. However, we only consider mappings between machines having the same number of processors (or the same number of base processors), and because we are analyzing fine-grained simulation each processor can simulate only a constant number of processors. Rather than use indirect measures of mapping efficiency, we will only consider the direct measure, namely the time to simulate a single step. Initialization time can also be significant but will be ignored.

To attain the best possible simulation times, some of the mappings are nontrivial, especially in the subsection on "Regular Architectures Simulating Regular Architectures." Further, due to space limitations many details have been eliminated. Readers who are most interested in an overview of the results may prefer to examine Fig. 5, which summarizes the results in the remainder of this section.

Simulations Involving PRAMs

For each regular architecture, the time needed to simulate a CRCW PRAM is an important upper bound on the time needed to simulate arbitrary architectures. If the PRAM model being simulated is a shared memory model with M memory locations, then the first transformation is to a distributed memory PRAM where each processor simulates M/n^2 shared memory locations. To simulate a distributed memory CRCW PRAM, suppose the concurrent write is such that two or more simultaneous writes to the same location are resolved by taking the smallest value. Each processor trying to write a value v to location i creates a record (i, v). By first sorting the records by their destination, all values bound for the same location are in consecutive locations. If ties are broken by placing the smaller value first, then a record (i, v) contains the value that should be written to i if and only if the record in the previous processor has a different destination. Records not satisfying this condition are eliminated, and the surviving records are routed to their destinations. The concurrent read can similarly be reduced to a fixed sequence of sorting and routing steps, plus a distribution step in which values are distributed to runs of consecutive processors holding records requesting the value from the same location [29].

Using this approach, on each regular architecture considered here a step of a CRCW PRAM can be simulated in the time needed to sort. Therefore the mesh, pyramid, and mesh-of-trees neded $\Theta(n)$ time, and the hypercube can simulate in $\Theta(\log^2 n)$ time. Further, these simulations can be easily modified to finish in the same time using write conflicts such as summing the values written, taking an arbitrary value written, taking the median value written, or taking the mode value written. Since simulating the EREW PRAM is also as time-consuming as sorting, if one is writing PRAM algorithms only as ideal algorithms to be used in simulation, then there is no reason not to use concurrent reads and concurrent writes.

If the CRCW PRAM being simulated has fewer processors than the simulating machine, then it is sometimes possible that simulations can be performed faster than just having a smaller machine simulate the PRAM. For the mesh this is not true, since any collection of P processors in a mesh must have a communication diameter of $\Omega(P^{1/2})$ (and hence a simulation time of $\Omega(P^{1/2})$), which can be attained by using a $P^{1/2} \times P^{1/2}$ mesh. However, a mesh-of-trees with an $n \times n$ base can simulate n PRAM processors in $\Theta(\log n)$ time, versus the $\Theta(n^{1/2})$ time needed by a mesh-of-trees with a base of n processors, by using the diagonal processors to simulate the processors, and the row and column trees to manage communication. If the PRAM processor i is to send to processor j, then the message can travel along the tree in row i to reach column j, where it then travels in the column tree to row j. To finish in logarithmic time it must be that the concurrent write operation can be determined by the column tree in logarithmic time. All concurrent write operations of widespread usage, such as using the minimum or maximum value, using the average value, using the value sent by the processor of smallest or largest index, or using an arbitrary value, satisfy this condition. Similarly, for any concurrent write operation in widespread use, the hypercube can simulate a CRCW PRAM with $n^{2-\epsilon}$ processors, for any fixed $\epsilon > 0$, in $\Theta(\log n)$ time, where the implied multiplicative constant depnds upon ϵ. This is because the hypercube can sort this amount of data in $\Theta(\log n)$ time [34]. Another improvement of some use is the fact that, on the mesh, the basic sorting-based approach to PRAM simulation can be modified so that if it is known that the maximum distance between source and destination processors for any message is d, then the step can be complete in $\Theta(d)$ time.

Reversing the direction of simulation, the simpler EREW PRAM, and hence also the CRCW PRAM, can simulate any fixed interconnection architecture of degree d, with the same number of processors, in $\Theta(d)$ time. This can be done by having d locations for each simulated processor to simulate the input communication links. To simulate one time step the PRAM processor reads and resets these locations, performs the calculations performed by the processor it is simulating, and writes the output values to the appropriate locations. This shows that the EREW PRAM can simulate the mesh, pyramid, and mesh-of-trees in $\Theta(1)$ time. To simulate the hypercube would take $\Theta(\log n)$ time by this procedure, but for all of the algorithms considered here it is known which input link will contain data, and hence only one location need be examined and the simulation takes $\Theta(1)$ time. This is not the only possible model of hypercube algorithms and architectures, and it is easy to conceive of models where an EREW PRAM would take $\Theta(\log n)$ time to simulate one step. For example, Valiant's work on routing assumed that in unit time a hypercube processor could receive an input on each link, decide where to forward it, and send an output on each link [59]. However, for our purposes it is not reasonable to call this a single time step.

Regular Architectures Simulating Regular Architectures

Since the pyramid and mesh-of-trees contain the mesh in their base, they can do a straightforward simulation of it in $\Theta(1)$ time. The hypercube can simulate the mesh by using Gray codes. If G is a Gray code mapping $0..n-1$ to $0..n-1$ (viewing the latter set as bit strings of $\lg(n)$ bits), then the Gray code property is that $G(i)$ and $G((i+1) \bmod n)$ differ by a single bit, for $0 \le i < n$. The mesh is mapped onto the hypercube by mapping processor (i, j) to the hypercube processor with label which is the concatenation of $G(i)$ and $G(j)$. This maps mesh neighbors to neighboring processors in the hypercube, and the simulation takes $\Theta(1)$ time. Notice that any such Gray code map has the property that each row is mapped to a subcube, as is each column. For some of the mappings mentioned later, we need additional properties which depend upon using the reflexive Gray codes, instead of arbitrary Gray codes. If G_b denotes the reflexive Gray code mapping $0..2^b-1$ into binary strings of length b, then they can be recursively defined via $G_1(0) = 0$, $G_1(1) = 1$, and

$$G_b(i) = \begin{cases} 0G_{b-1}(i), & i \le 2^{b-1}-1 \\ 1G_{b-1}(2^b-1-i), & i \ge 2^{b-1} \end{cases} \qquad b > 1.$$

For the mesh to simulate the other architectures, the PRAM simulation result implies that it can simulate each in $O(n)$ time. Further, since each of the other architectures has a communication radius of $\Theta(\log n)$, and the mesh has a communication radius of $\Theta(n)$, it must take $\Omega(n/\log n)$ time to simulate each of the others. For the pyramid there is a natural mapping of the levels above the base down onto its base, where at most one upper processor is mapped onto any base processor (see Fig. 3(a)). Using this mapping, the

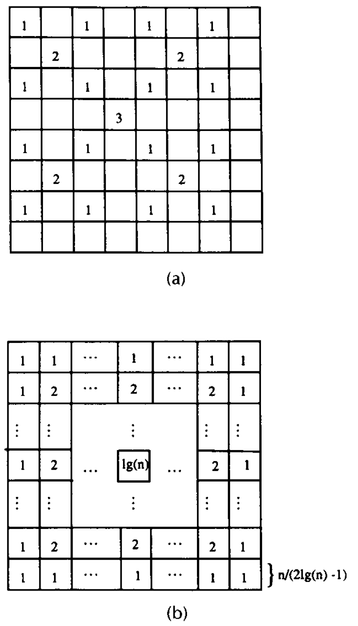

Fig. 3. Mapping the pyramid onto its base. (a) Local mapping of pyramid onto base (numbers indicate level). (b) Squares for global optimal mapping (numbers indicate minimum level).

mesh will take $\Theta(n)$ time since the apex is mapped to a processor at distance $\Theta(n)$ from the processors simulating its children. This mapping can be modified to achieve the lower bound by dividing the base into $(2 \lg n - 1)^2$ squares, as in Fig. 3(b). Now a pyramid node at height h is projected down to the base just as before, and if it is in a square labeled h or greater it is mapped to that base square. Otherwise the nearest square labeled h is located and the pyramid node is mapped to any base processor not already mapped onto. With this mapping adjacent nodes in the pyramid are mapped to the same or adjacent squares, and hence are never more than $\Theta(n/\log n)$ apart. Using the fact mentioned above that the mesh communication can be modified to finish in time proportional to a known upper bound on the distance messages need to travel, pyramid simulation takes $\Theta(n/\log n)$ time per step.

This same approach can be used when simulating the mesh-of-trees on the mesh, finishing in $\Theta(n/\log n)$ time using a similar complicated mapping and in $\Theta(n)$ time using a natural mapping which maps each row and column tree to its base row and column, respectively, using the mapping in Fig. 4(a) When simulating either the pyramid or mesh-of-trees, the simulations finishing in $\Theta(n/\log n)$ time are *globally optimal* in that they minimize the worst possible communication. However, the simpler embeddings have the property that if the algorithm is only using the bottom k levels then each step can be simulated in only $\Theta(2^k)$ time since no two neighboring nodes are more than $\Theta(2^k)$ apart. This situation does not help the globally optimal embedding, which still takes $\Theta(n/\log n)$ time. Since many pyramid and mesh-of-trees algorithms use only a few levels at a time, globally optimal embeddings may not provide the best total simulation times, and the more natural *locally optimal* embeddings may be better. However, this is only true if it is known which levels are being used, and if an intelligent simulation is being employed which utilizes this information. By a *naive simulation* with locally optimal embeddings it is meant that the simulation assumes that communication at all levels is possible, taking $\Theta(n)$ time per step.

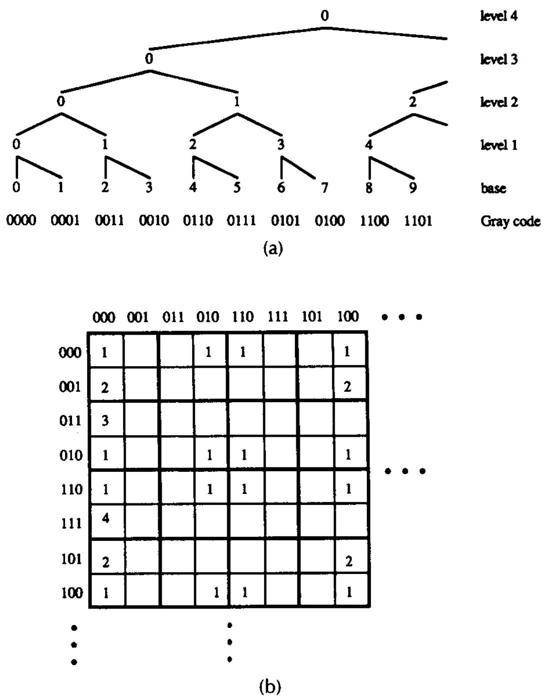

Fig. 4. Mapping into the hypercube via the base. (a) Tree nodes over base nodes they are mapped to (lines indicate parent-child links). (b) Pyramid nodes over base nodes they are mapped to.

For the pyramid to simulate the mesh-of-trees, level i of the pyramid will simulate levels $2i - 1$ and $2i$ of the mesh-of-trees. Partitioning the base into squares of edgelength 2^{2i}, above each base square there are exactly $3*(2^{2i})$ mesh-of-trees processors in row trees at levels $2i - 1$ and $2i$, two per row at level $2i - 1$ and one per row at level $2i$. In the pyramid, above each base square there is a square with exactly $2^i * 2^i$ processors at level i. Each pyramid processor in this square at level i will stimulate three processors from mesh-of-trees row trees at levels $2i - 1$ and $2i$ above the same base square, using any arbitrary assignment. The pyramid processors also each simulate three processors from column trees above the same base square. This mapping is based on techniques from [27], and using data movement operations there it is straightforward to show that any single step of the mesh-of-trees can be simulated in $\Theta(n^{1/2})$ time. Using the $\Omega((n \log n)^{1/2})$ lower bound noted in Section III for the pyramid moving $n \log n$ items from the first column to the last, coupled with the fact that the mesh-of-trees can accomplish this in $\Theta(\log n)$ time, shows that the best possible simulation is $\Theta((n/\log n)^{1/2})$. It seems that the mapping given here is as good as possible, and that a slightly improved use of the techniques in [27] might be able to prove this.

For the hypercube to simulate the pyramid and mesh-of-trees, the basic technique is to map each onto its base and then apply the Gray code mapping to the map the base onto the hypercube. To take advantage of the fact that processors can be adjacent in the hypercube that are images of processors not adjacent in the mesh, for the pyramid the mapping down to the base is done in a slightly altered fashion so that the neighbors will ultimately be mapped to nearby processors in the hypercube. Fig. 4 illustrates the mapping that is used to map each row or column tree down to the base, and the mapping of the pyramid into its base. Using these maps, neighbors in the mesh-of-trees are mapped to processors within 2 of each other in the hypercube, and neighbors in the pyramid are mapped to processors within 3 of each other, and the hypercube can simulate a step of either in $\Theta(1)$ time [28], [48].

For the mesh, pyramid, or mesh-of-trees to simulate the hypercube it seems that no mapping can be better than just arbitrarily mapping the hypercube onto the base and using the PRAM simulation result, taking $\Theta(n)$ time. While it is easy to prove that each must take $\Omega(n/\log n)$ time, the author does not know of a proof that $\Omega(n)$ time is required.

Finally, for the mesh-of-trees to simulate the pyramid it again seems that a simple simulation is as good as possible. For each pyramid processor above the base, determine the base processor it maps to under locally optimal pyramid to mesh mapping, determine the mesh-of-tree processor in a column tree that would map to the same base location under the locally optimal mapping, and map the pyramid processor to this column tree processor. In [31] it is shown that the communication can be arranged so no bottlenecks occur, and the simulation time is $\Theta(\log n)$.

Fig. 5 summarizes the simulation times obtained by the above mappings. In some cases the simulations are quite simple, but in other cases they are less so, and are complicated by the fact that there are globally optimal versus locally optimal solutions for some source/target pairs. The simulation strategy, which is the easiest way to deal with the "dusty deck" problem of converting existing algorithms from one parallel machine to another, is complicated by the fact that naive simulation of all possible communication may be significantly slower than intelligent simulation of only those portions actually used. Achieving the best possible results, even when restricted to the using

Simulator \ Simulated	Mesh	Pyramid	Mesh-of-Trees	Hypercube	CRCW PRAM
Mesh	1	n (local) n/log(n) (global)	n (local) n/log(n) (global)	n	n
Pyramid	1	1	$n^{1/2}$	n	n
Mesh-of-Trees	1	log(n)	1	n	n
Hypercube	1	1	1	1	$\log(n)^2$
CRCW PRAM	1	1	1	1	1

Fig. 5. Simulation times.

stepwise simulation, requires that the programmer understand the algorithm.

V. Ideal Parallel Algorithms

In this section we examine the approach based on developing an ideal parallel algorithm for a given problem, and then mapping this to a target machine. The only restriction that we impose is that the optimal solution must be for a machine with no more than n^2 processors, and that each processor have only a constant amount of memory. While this approach is closely related to the simulations discussed above, it has the possibility of avoiding inefficiencies that arise when the architecture being simulated is not particularly well suited for the problem. However, it has the difficulty that there may not be any such ideal algorithm, or that there will be too many. Further, since it potentially requires developing a new mapping for each problem, to be widely useful the mapping process should be as automated as possible. Some systems for automatically finding maps are described in [7], [41].

Reduction

One of the more fundamental operations used is global *reduction* (or aggregation), in which each processor has some value and these are all combined to form a single result. For example, determining the average gray level of the image can be viewed as a reduction which sums up the gray levels, followed by a division by the number of pixels. Reduction can also be used for problems such as counting the number of pixels with a given property, finding the leftmost, topmost, rightmost, and bottommost black pixels, or determining if all processors have terminated. Reduction is quite simple, and illustrates the principle that generally one wants to minimize communication. It also illustrates that applying this principle is not as straightforward as one would suppose.

Summing the values in the processors will be used as a specific example of the reduction operation. The natural parallel approach to finding the sum is to find the sum of the first half, while simultaneously finding the sum of the second half, and then add the two partial sums. This directly yields a balanced binary tree approach with n^2 leaf nodes, which finishes in exactly 2 lg (n) steps. Values start at the leaf nodes and are passed up to the parents. Each node receiving two values from its children adds them and passes up their sum. After the first step there are only $n^2/2$ numbers remaining to be added, after the second there are only $n^2/4$, etc. This is an "ideal" parallel algorithm/architecture for this problem, assuming that concurrent reads or concurrent writes are forbidden. (See [13] for faster summing algorithms on CRCW PRAMs. If concurrent reads, or concurrent writes are forbidden then it is straightforward to show that any reduction operator must take $\Omega(\log n)$ time.)

If this ideal parallel algorithm is applied to the pyramid, one quickly sees that actually one should divide into 4 pieces, rather than 2, and that these should be the quadrants of the base. Each node will compute the sum of the numbers below it by adding together the four answers passed to it by its children. The algorithm finishes in lg(n) steps, where each step involves 3 additions. Notice that for the common row-major ordering of the base processors, the quadrants do not correspond to contiguous numbers. A naive automatic mapping technique which maps the leaf nodes of the ideal tree directly onto the base nodes of the pyramid with the same number would give a very inefficient algorithm. An efficient solution could be found automatically, but it requires that the mapping program notice that by coalescing pairs of levels in the tree together it creates a tree where each interior node has 4 children. The mapping program must also notice that the pyramid contains such a tree with the apex as its root and the base processors as the leaves, though not with the standard ordering.

Worst problems occur when trying to map onto the mesh-of-trees. Again one tries to reduce the number of partial sums remaining as quickly as possible, but doing so based on halves or qudrants does not seem to work efficiently. To combine 4 numbers stored one per quadrant in the middle of the quadrant takes logarithmic time. This would create an algorithm obeying a recurrence of the form $T(n) = T(n/2) + \Theta(\log n)$, which gives $T(n) = \Theta(\log^2 n)$ time to compute the entire sum. To produce a more efficient algorithm, one notices that in each row the ideal algorithm can be directly applied, finding the row sum and then moving the answer back down the row tree to place it in the leftmost base processor in logarithmic time. Now all partial sums are in the first column, where again the ideal algorithm can be applied using the column tree, completing the problem in logarithmic time. Once again, this solution *might* be discovered by an automatic mapping program if it mapped the top half of the tree onto the first column, and the bottom pieces of the tree onto the rows, and then identified the top of each row tree with that row's leaf in the first column tree, but discovering such mappings seems to be beyond current mapping systems.

For the mesh, the ideal tree approach can be used but will give an algorithm which is slower than optimal by a multiplicative constant. Instead a simple approach of finding the sum in each row (and storing it in the leftmost processor), and then adding these sums along the first column, will give an algorithm requiring only $2(n - 1)$ steps, where each step involves one addition and passing one number. Discovering a reasonable mapping of the binary tree onto the mesh is within the capabilities of automatic mapping systems [7], but the extra overhead of simulating the tree is fairly substantial when compared to the simple approach.

For the hypercube, again the ideal binary tree could be mapped onto the hypercube by coalescing nodes and mapping the resulting quotient graph onto the hypercube, but the results are not as good as can be obtained by an algorithm more natural for the hypercube. This is the "recursive halving" approach of having each node with a high-order bit of 1 send its value to its neighbor with a high-order bit of 0. The nodes receiving a value add it to their value. The number of partial sums remaining is $n^2/2$, and they are all stored in a subcube of 1 smaller dimension. The process is recursively applied, taking $\lg(n^2) = 2\lg(n)$ steps, each taking a constant amount of time.

Notice that the hypercube, or at least the hypercube processors and the communication links used by recursive halving, can also be considered the ideal architecture for reduction, using only n^2 processors instead of the $2n^2 - 1$ used by the binary tree. However, the tree for recursive halving has a processor with degree 2 $\lg(n)$, as opposed to the degree 3 or less for all processors in the binary tree. Had we started with the recursive halving tree as the ideal architecture, the mappings would have been less efficient since the other architectures do not have nodes of logarithmic degree. This would have forced neighbors of the apex to be more than a constant distance away, which would force the simulation to take more than constrant time per step. In the pyramid and mesh-of-trees this would make the algorithm take more than logarithmic time using a naive simulation. Thus the efficiency of simulating an ideal solution depends on the specific "ideal" architecture chosen, and on the target architecture.

Extreme Points

The next example is more sophisticaed than reduction, but still results in significantly reducing the amount of communication used in later stages. We are interested in determining the convex hull of the black pixels, where the *convex hull* of a finite planar set is the smallest convex polygon containing them. It is easy to show that the corners of this polygon are points of the set. See Fig. 6. The corners, called

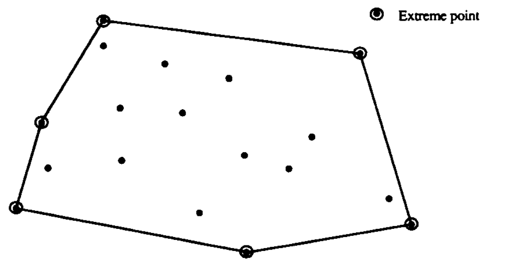

Fig. 6. Extreme points.

the *extreme points*, are a concise representation of the convex hull. Using only them, one can determine properties such as the smallest circle enclosing the figure, a smallest rectangle enclosing the figure, the diameter of the figure, etc. [26].

We will consider only the problem of finding the extreme points of all black pixels, rather than of each individual object. Finding the extreme points of each object typically requires a blend of the approaches used to find the extreme points of all black pixels and to label connected components (discussed below).

One commonly used approach for determining extreme points is to use divide-and-conquer, based on the observation that if the points are partitioned then a point which is extreme for the entire set must be extreme for its partition. This is given in Algorithm 1.

Algorithm 1 *Finding Extreme Points via Divide-and-Conquer*

1) Divide the image in half by a line and find the extreme points of each half (as if each was the entire image). Call these points "candidates."
2) Find the two lines of support between the halves and eliminating all candidates in the center between the lines of support. The remaining candidates are extreme points of the entire image.

Fig. 7 shows the lines of support that need to be determined when merging two halves. They can be determined by a

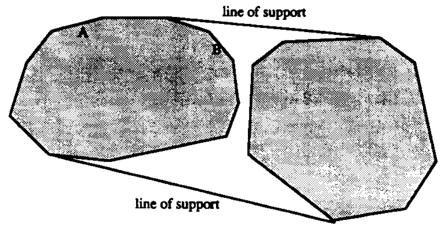

Fig. 7. Lines of support.

logarithmic search on each side, where one step involves sending a description of one edge of the polygon on one side to the other side and determining if all pixels on the other side lie on the same side of the line as the pixels on the sending side. For example, in Fig. 7, if side *A* is the probe, then the reply is yes, while if side *B* is the probe then the reply is no. Based on this, one knows that the support line lies between *A* and *B*.

If step 2) is interpreted as requiring such a binary search, where each step requires sending a value to all processors on the other side and then collecting their replies, then step 2) involves $\Theta(\log n)$ searches, each taking $\Omega(\log n)$ time to transmit the query and collect the replies. Thus the total time for step 2) is $\Omega(\log^2 n)$, and the total for the entire algorithm is $\Omega(\log^3 n)$. This approach can be directly implemented by the pyramid in $\Theta(\log^3 n)$ time, and also by the hypercube and mesh-of-trees in the same time. Further, the tree connections of the pyramid provide a perfect implementation if one alternates between vertical and horizontal separating lines to do the division in step 1).

However, if one does not insist on this reading of the algorithm then faster divide-and-conquer algorithms are possible. First, on the pyramid, hypercube, and mesh-of-trees, when a binary search is being conducted, each time a probe edge is being tested it takes logarithmic time to send the information and then collected the replies. This only uses one level of the machine at a time, so most processors are idle. One can pipeline probes, sending each over one time step after the previous one. By dividing the remaining candidate edges into a logarithmic number of pieces (instead of two), each round still takes logarithmic time, but only $O((\log n)/\log \log n)$ iterations are needed.

Another speedup by a factor of $\log \log n$ can be obtained by combining a logarithmic number of pieces together at one time, rather than two at a time. To obtain both speedups one could combine $\Theta((\log n)^{1/2})$ pieces together simultaneously, using $\Theta((\log n)^{1/2})$ probes from each at each step. In practice, however, the greatly increased difficulty of implementing either of these speedups makes it nearly certain that they would never be used. Further, the extra overhead introduced would require that n be extremely large before the log log n factors would overcome the overhead. Because of this, from now on improvements by factors less than log n will not be discussed.

Significantly better times can be obtained if one uses only vertical separating lines instead of alternating between horizontal and vertical ones. Eventually the recursive application produces regions which are each a single column. In a column, the extreme points are the topmost and bottommost points, which can be determined by simple reduction operations, rather than the slower and more complicated general divide-and-conquer approach. Now, for each candidate in each column, two lines of support are determined. Assuming that a candidate is the topmost point in its column (with the bottommost points having a similar operation), the line of support to all candidate points to the left is determined, as well as the line of support to all candidate points to the right. For each candidate C, its lines of support can be found by determining maximal angles, relative to vertical with C as the origin, from C to any other candidate on the left and to any other candidate on the right. Finally, C is an extreme point only if these lines of support from an angle less than π. This is utilized in Algorithm 2.

Algorithm 2 *Extreme Points via Column Reduction and Complete Interchange*

1) Determine the topmost and bottommost black pixel in each column. Call this "candidates."
2) For each candidate, find the support line to all candidates to the right, and the support line to all candidates to the left. The candidate is extreme if and only if these support lines from an angle of less than π (towards the object).

To implement Algorithm 2 efficiently one needs to be able to perform reductions in each column, transmit the candidates to all other columns, and then again perform column reductions. The column and row trees of the mesh-of-trees provide precisely these connections, and the entire algorithm can be completed in $\Theta(\log n)$ time [28], [36]. Since all the reduction operations are maximums over a predetermined collection of possibilities bounded by a polynominal in n (i.e., either n row coordinates to find the column candidates, or over slopes which can have $O(n^2)$ values), these can be determined in $\Theta(1)$ time on a CRCW PRAM [58]. Therefore the CRCW PRAM can complete this algorithm in $\Theta(1)$ time, though logarithmic time would be needed if the concurrency in either the read or write was prohibited.

The mesh could simulate CRCW PRAM and mesh-of-trees algorithms in $\Theta(n)$ time, if the stepwise simulation of the mesh-of-trees uses the global simulation or intelligently uses the local simulation. Naively using the local simulation would increase the times by a factor of log n. If a stepwise simulation of the pyramid implementation of the original divide-and-conquer aglorithm is performed then the times would increase to $\Theta(n \log^2 n)$ if the global simulation is used, to $\Theta(n \log n)$ if the local simulation is used intelligently, and to $\Theta(n \log^3 n)$ if the local simulation is used naively.

The hypercube can simulate the mesh-of-trees implementation of the column reduction algorithm in $\Theta(\log n)$ time, it can simulate the pyramid implementation of the divide-and-conquer in $\Theta(\log^3 n)$ time, and it can simulate the CRCW PRAM implementation of the column reduction algorithm in $\Theta(\log^2 n)$ time.

For the pyramid one confronts a confusing situation. If one admits concurrent reads and concurrent writes, then the CRCW PRAM provides the "natural" structure for the fastest possible algorithm. If one does not admit concurrent reads or concurrent writes, then the reduction operations are fairly naturally performed by the mesh-of-trees. Whichever of these two possibilities one adopts, applying them to the pyramid creates severe problems. If the pyramid does a stepwise simulation of the mesh-of-trees version it will take $\Theta(n^{1/2})$ time, and stepwise simulation of the CRCW PRAM would encounter the sorting bound and take $\Theta(n)$ time. Thus the pyramid will take significantly longer by executing the fastest parallel algorithm, as opposed to the time required using the simple initial divide-and-conquer. Similarly the mesh-of-trees will perform much slower if it does a stepwise simulation of the faster CRCW PRAM algorithm, for the communication requirements of simulating the PRAM increase the time to $\Theta(n)$.

These observations lead us to conclude that there probably is no ideal parallel algorithm for determining extreme points, at least in the sense that simulating it gives good results on all architectures. Divide-and-conquer provides a useful starting point for developing good extreme point algorithms, but significant variations occur when one considers how to repeatedly divide (e.g., alternate horizontal and vertical, or always use vertical), when to stop using divide-and-conquer and instead apply a different technique on the small pieces (e.g., stop when it is reduced to a single column), and how many pieces can be merged together at one time (e.g., 2, or n). These variations significantly affect the amount of communication required. The faster column reduction algorithm has the property that it uses more communication to combine pieces together, when compared to the original version, and more communication to do the simultaneous column reductions. While reducing communication is an important general goal in designing efficient parallel algorithms, for each architecture there is a point beyond which further reductions do not help. A slow, communication-intensive operation on one architecture, such as determining the topmost and bottommost black pixels in each column on the pyramid, may be fast and exactly matched to the communication capabilities of another architecture, such as the mesh-of-trees.

Labeling

In component labeling, each black pixel is assigned a label, where two black pixels have the same label if and only if they are connected by a path of adjacent black pixels. Two black pixels are considered adjacent if and only if they share an edge ("4-connectivity"), though with only trivial changes the same algorithms work if they are considered adjacent whenever they share a corner ("8-connectivity"). In all of the algorithms, black pixels start with a label that is the concatenation of their row and column coordinates, i.e., it is their number in row-major ordering of the processors. At the end, all pixels in a connected component will have as their label the smallest initial label of any pixel in the component.

On a parallel computer where adjacent pixels are in adjacent processors, an extremely simple algorithm for this problem is to have each black pixel start with its row-major index as its label, and then repeatedly take as its label the minimum of its label and the label of any of its black neighbors. This process continues until no pixels change their label. In each component all pixels adjacent to the pixel with the smallest initial label receive their final label during the first execution of step 2), while those two pixels away receive their label during the second execution, and so on, where an item at internal distance d receives the label at iteration d. (Given a connected object, the *internal distance* between two pixels in the object is 1 less than the minimum number of pixels needed to form a connected path from one pixel to the other.) If the components are small or fairly close to being convex this algorithm can work quite well, but in the worst case it can take $\Theta(n^2)$ time since a figure may have two pixels at internal distance $\Theta(n^2)$ (see Fig. 8).

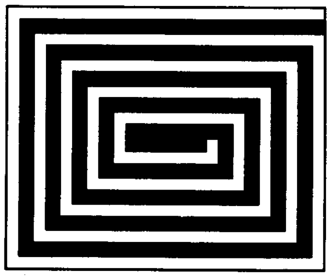

Fig. 8. A bad image for propagation.

A much better approach is to use divide-and-conquer. Apparently the first application of this to images is in Nassimi and Sahni [33], where it is used to provide an optimal mesh algorithm. The algorithm is given in Algorithm 3.

Algorithm 3 *Divide-and-Conquer Component Labeling*

1) Initially each pixel starts with its row-major index as its label.
2) If the image is 1×1 then the algorithm is finished, otherwise divide the picture into quadrants and label the components in each quadrant, ignoring adjacent pixels in other quadrants. (This is a recursive call to the algorithm starting at Step 2.)
3) Combine adjacency information from along the boundaries of the quadrants to decide on the final labels of all components lying in more than one quandrant.
4) Correct the labels of components lying in more than one quadrant.

Fig. 9 shows the labels as they might be assigned by the end of step 2). Notice that the only components which can have more than two labels are those lying in more than one quadrant. This means that step 3) uses only the adjacency information from pixels along the borders of the quadrants, instead of the entire image. This use of a restricted amount of data is crucial in obtaining the desired speed.

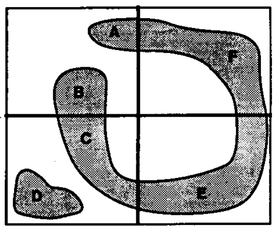

Fig. 9. Labels assigned in quadrants.

In step 3), determining the labels by using the border information is the connected components problem for undirected graphs. This subproblem can be solved in a great many ways. For example, one could construct an adjacency matrix, and use a parallel version of Warshall's algorithm to compute the connected components [61]. This would take $\Omega(n)$ time, and could be accomplished in this time by any of the architectures discussed herein, based on Van Scoy's $\Theta(n)$ time mesh implementation [60].

For most of the architectures, significantly faster solutions can be found by using algorithms based on keeping the graph in the form of a collection of edges, rather than an entire adjacency matrix, since the number of edges is $O(n)$ while the adjacency matrix may have $\Theta(n^2)$ entries. Edge-based algorithms have the disadvantage that their communication follows a very irregular, data-dependent pattern, compared to the elegant communication pattern used in Van Scoy's algorithm, but they compensate by having significantly faster asymptotic times. Most efficient parallel edge-based algorithms are based on variations of Sollin's serial algorithm, which starts with each vertex being a separate "club" and repeatedly merges adjacent clubs together until each component is a single club. This can be implemented in $\Theta(\log n)$ time on a CRCW PRAM [44], and in $\Theta(\log^2 n)$ on a EREW PRAM by simulating the CRCW PRAM algorithm. Mesh implementations of a variant of this finish in $\Theta(n)$ time [38], but a stepwise simulation of the CRCW PRAM algorithm would take an extra factor of $\log n$.

Implementing the divide-and-conquer algorithm on the mesh, using either the adjacency matrix or the efficient edge-based algorithm for the graph subproblem in step 3), yields an algorithm obeying the recurrence $T(n) = T(n/2) + \Theta(n)$, giving $T(n) = \Theta(n)$ [33]. Implementing on the mesh-of-trees, if Van Scoy's algorithm is used then the time is again $\Theta(n)$, but if a stepwise simulation of the CRCW PRAM algorithm is used for step 3) then the time satisfies the recurrence $T(n) = T(n/2) + \Theta(\log^2 n)$, giving $\Theta(\log^3 n)$ time. Here a critical fact is that the CRCW PRAM is solving a problem on only $\Theta(n)$ edges, and can finish in logarithmic time using only $\Theta(n)$ processors. Therefore the mesh-of-trees can simulate each PRAM step in logarithmic time, rather than the $\Theta(n)$ time needed to simulate $\Theta(n^2)$ PRAM processors. Similar comments and timings hold for the hypercube. However, by dividing into $n^{1/2}$ pieces instead of 4, the divide-and-conquer approach can yield a hypercube implementation taking $\Theta(\log^2 n)$ time [14], [29].

For the pyramid the time requirements are more complicated. The best way to simulate n PRAM processors is to simulate them on the middle level of the pyramid, which is a mesh of n processors. There the graph labeling subproblem could be completed in $\Theta(n^{1/2} \log n)$ time, but if instead the optimal mesh algorithm is used it is completed in $\Theta(n^{1/2})$ time. Moving the data to the middle level, and the answers back down again, can also be accomplished in $\Theta(n^{1/2})$ time [27], so the entire algorithm satisfies a recurrence of the form $T(n) = T(n/2) + \Theta(n^{1/2})$, giving $T = \Theta(n^{1/2})$. This algorithm is optimal for the pyramid, as can be seen by considering Fig. 10. In the image X's indicate pixels which

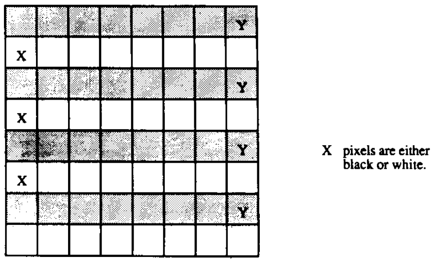

Fig. 10. An image requiring extensive communication.

may or may not be black. Two adjacent Y's will have the same label if and only if the X in the intervening row is black, and hence in constant time after the labeling is finished the Y's can determine if the intervening X is black. Since this data movement requires $\Omega(n^{1/2})$ time (see Section III), component labeling must take $\Omega(n^{1/2})$ time. Notice that the implementation which finishes in this time is not a simulation of any one ideal architecture, but instead moves the data around to achieve the best simulation of a good architecture for a single step (PRAMs along the borders of the quadrants). This is far beyond the capabilities of any automatic mapping system, and would be very difficult for most programmers.

For a CRCW PRAM, the divide-and-conquer algorithm will give a recurrence of the form $T(n) = T(n/2) + \Theta(\log n)$, which is $\Theta(\log^2 n)$. If instead the pixels are just thought of as vertices, with an edge between two of them only if they are adjacent black pixels, then directly applying the graph component labeling algorithm, with no use of divide-and-conquer, will finish in $\Theta(\log n)$ time. Similarly a EREW PRAM would give a time of $\Theta(\log^3 n)$ if the divide and conquer algorithm is used, or $\Theta(\log^2 n)$ if the entire image is viewed as a single graph. A significantly different EREW PRAM algorithm finishes in $\Theta(\log n)$ time, utilizing properties of the planar nature of the image [2].

Once again, we have the situation that if the fastest algorithm, the CRCW PRAM single-graph version, is simulated by some of the architectures then they will end up with far slower implementations than they can obtain from the original divide-and-conquer. On the mesh, pyramid, or mesh-of-trees the time would increase to $\Theta(n \log n)$, and on the hypercube it would increase to $\Theta(\log^3 n)$. Even if one ignores the single-graph CRCW PRAM solution, there remains the problem of identifying the best architecture for the original divide-and-conquer algorithm. Many programmers would just decide that it is too complicated a problem and chose the PRAM of n^2 processors, but then the simulation time jumps up to the time required to sort. A more tractable possibility is to have tree of PRAMs, where the root is a PRAM with $\Theta(n)$ processors identified with borders of the quadrants. Descendants are smaller PRAMs identified with borders of the recursive subdivisions, with different levels in the tree communicating through connections between processors corresponding to the same pixel in different subdivisions. This is a rather formidable ideal architecture which is actually what is utilized in Algorithm 3, but it is unlikely that programmers would feel comfortable using it and it is not at all clear that any automatic mapping system will be able to do a decent mapping of it in the near future.

VI. Data Movement Operations

As the above examples have shown, moving data is a critical part of algorithms. The movement of data to the middle level mentioned in the pyramid implementation of labeling is a significant operation, and in fact similar movement was needed in the other non-PRAM architectures. For example, the mesh-of-trees implementation needs to move the information to the diagonal processors, and then it can efficiently simulate the PRAM graph algorithm for n processors. Such movement is not part of PRAM algorithms, but is critical for efficient implementations on parallel architectures with fixed interconnections. Further, the question of where the data is moved to is highly dependent on the architecture.

Since the data movement is critical in determining the efficiency of implementing algorithms, one approach is to make such movement much more explicit. This enables one to directly confront the problems of optimizing the movement. It also introduces the possibility of reusing previously developed implementations of common movement operations, much as one might reuse a mapping. For example, one might loosely express the column reduction algorithm for convex hulls as in Algorithm 4.

Algorithm 4 *Extreme Points via Column and Crossproduct Reductions*

1) In all columns c simultaneously, $0 \le c \le n-1$, use reduction to determine the location of the topmost black pixel $t(c)$ (the reduction operation is maximum) and bottommost black pixel $b(c)$ (the reduction operation is minimum). In a column with no black pixels, set $t(c) = -\infty$ and $b(c) = +\infty$. Create records $(c, t(c))$ and $(c, b(c))$.
2) Move the n records containing t values, 1 from each column, to locations where crossproduct reduction can be efficiently determined.
3) Using crossproduct reduction, for all c simultaneously determine

   $L(c) = \max\{\text{slope of the line through } (c, t(c)) \text{ and } (d, t(d)): d < c\}$, and

   $R(c) = \min\{\text{slope of the line through } (c, t(c)) \text{ and } (d, t(d)): d > c\}$.

   Then $t(c)$ is an extreme point if and only if $L(c) > R(c)$.
4) Complete steps similar to 2) and 3) for the b values.

The reduction operation used there has been discussed previously. For the crossproduct reduction one has a set S of values, a function f defined on pairs of elements of S, an reduction operation $*$ over the values of f, and wants to compute $F(s) = *\{f(s, t): t \in S\}$ for all s in S. Some uses of the crossproduct operation appear in [27].

Since operations such as reduction and crossproduct reduction are useful in many algorithms, they can be optimized for a given architecture, and then used repeatedly. For example, the pyramid can implement rowwise reduction and a crossproduct reduction over $\Theta(n)$ items in $\Theta(n^{1/2})$ time [27], and therefore can implement this algorithm in $\Theta(n^{1/2})$ time. While this is not as fast as the original divide-and-conquer, it is far better than simulating the CRCW PRAM implementation.

Data movement operations that have been suggested include reduction, broadcasting, multicasting (simultaneous broadcasts with different groups), concurrent read and write with specified tie-breaking procedures in the case of concurrent writes, sorting, crossproduct reduction, matrix multiplication, and prefix (in which there is some semigroup operation *, each processor i starts with some value $v(i)$, and ends up with the value $v(0)*v(1) \cdots *v(i-1)$) [4], [21], [25], [27], [29], [33]. More specialized data movement operations include matrix transposition, sparse matrix read (reduction over all rows simultaneously), list ranking (in which items are on a linked list and the problem is to determine their order position on the list), and path compression (in which items are in upward directed trees and the problem is to have each determine the root of the tree it is in) [13], [20], [29], [33].

Use of data movement operations is fairly natural, in that one develops programs as a sequence of operations, and in fact they may be implemented as a sequence of procedure calls. Data movement operations are used much as data structures are used in serial computers. The data movement approach also decreases the emphasis on identifying the best possible architecture for an algorithm, and hence decreases the inefficiencies that arise due to inappropriate simulations. For example, the mesh-of-trees will directly implement this algorithm in $\Theta(\log n)$ time, and it is irrelevant that a CRCW PRAM can perform the reductions (including the reduction in crossproduct reduction) in constant time.

Use of data movement operations creates some problems, however. Because operations can sometimes be much faster if only a little data is involved, the operations and their implementations tend to proliferate to match different situations. For example, one needs not only reduction, but simultaneous reduction over rows, and perhaps simultaneous reduction over quadrants. One also needs to know the how to simulate n PRAM processors, as well as how to simulate n^2. It seems that each new architecture introduces new special cases which can be exploited. The more algorithms become highly tailored to exploit this, the less general they become and the more they resemble a direct implementation on a given architecture.

VII. Final Remarks

As was noted in the introduction, the mapping of algorithms to architectures can occur at many levels. In general, the higher the level the more likely that an efficient algorithm can be obtained, though at the cost of more programmer effort to develop the implementation for a specific machine. General paradigms are quite useful guides, but provide little specific information concerning a given problem and how to solve it on a given architecture. At the opposite end of the spectrum, code generated for a specific machine may require extensive modifications for another machine of nearly identical architecture. For example, code for an MIMD hypercube may need extensive revision to be usable on an SIMD hypercube. Such retargeting may involve a great amount of detailed work, but should be easier to automate (or semi-automate) than the mappings across architectures considered here.

When transferring algorithms between different architectures, the simplest approach is to develop a simulator of the source architecture on the target architecture, and then use stepwise simulation. If many algorithms had been developed on the source architecture, then by developing a single simulator they all become available on the target architecture. One problem with this approach is that the algorithm may have already lost efficiency in being implemented on the source architecture, and the simulation overhead just compounds this. Another problem is that stepwise simulation may be slower than necessary if the algorithm does not use all the communication links of the source architecture at each step. In this situation an intelligent locally optimal simulation may be superior to a globally optimal one.

One possibility above the level of source-to-target simulations is to map ideal algorithm/architecture solutions onto target architectures. Since each algorithm may have a different ideal architecture, either the programmer must develop a mapping for each algorithm, or else procedures are needed to help automate the mapping. While this has achieved some attention, it does not always produce good results. As the reduction example showed, automatically achieving a good mapping of one regular architecture onto another is a nontrivial problem, though current systems can achieve some success on simple mappings [7]. Further, the notions of "ideal" parallel algorithms or "ideal" architecture for an algorithm do not seem as clear as one would initially think. The reduction example showed that there might be two ideal architectures for a single algorithm, and that the choice made a difference when simulated on other machines. The extreme point and labeling examples showed that it is unlikely that there is any ideal algorithm for them. Ideal in the sense of being fastest is not ideal in the sense of being fastest to simulate.

Another approach is to develop algorithms in terms of data movement operations, and then provide efficient implementations of these on the target architecture. These make the data movement more explicit and easier to optimize, and if sets of generally useful data movement operations can be distilled then they need to be implemented only once on each target machine, and then can be used for multiple problems. Unfortunately, efficient implementations depend on the amount of data involved, as well as the architecture involved, so actually a collection of implementations may be needed for each operation/architecture pair. Selecting a proper implementation requires that extra information be known about the amount of data involved, which complicates their use and requires more of the programmer.

Even the data movement approach does not completely solve the problem of mapping algorithms onto architectures. The data movement capabilities of architectures vary significantly, and it appears to be impossible to write an algorithm in terms of data movement operations and have it yield optimal, or nearly optimal, implementations on all architectures. As the extreme point example showed, the fastest parallel algorithm, even with a data movement approach, will yield a poor solution on the pyramid, and similarly the fastest parallel algorithm for labeling (the CRCW PRAM single-graph algorithm) will yield a poor solution on the pyramid and mesh-of-trees.

One fundamental difficulty with a data movement approach is that operations that take approximately the same time on one machine may be vastly different on another. For example, a EREW PRAM will take $\Theta(\log n)$ time to find the sum of n^2 numbers, to find the sum in each row of an $n \times n$ array of numbers, or to sort n^2 numbers. A hypercube will take $\Theta(\log n)$ time to find the global or row-wise sums, and the best known sorting algorithm takes $\Theta(\log^2 n)$ time. A mesh-of-trees will take $\Theta(\log n)$ time to find the global or row-wise sums, but $\Theta(n)$ time to sort. A pyramid will take $\Theta(\log n)$ time to find the global sum, $\Theta(n^{1/2})$ time to find the row-wise sums, and $\Theta(n)$ time to sort, while the mesh will take $\Theta(n)$ time to perform all three. Suppose a programmer devises two algorithms A and B which are nearly identical but, say, A uses a sort and global sum, while B replaces this with log n iterations of rowwise sums. On the PRAM or mesh A is better, on the mesh-of-trees and pyramid B is preferred, and on the hypercube they have the same times (at least, as measured in general O-notation). It seems impossible to write nontrivial algorithms which take such variations into account and yield an efficient implementation on each architecture. The data movement approach seems to give a decent implementation of an algorithm on a specified architecture, but cannot eliminate the fundamental difficulty that the communication demands of the algorithm may be poorly matched to the communication capabilities of the architecture.

Problems which are as hard as sorting may be somewhat more amenable to general use of PRAM simulation or the data movement approach. For example, given n^2 planar points, determining their extreme points takes $\Omega(n^2 \log n)$ time on a serial computer [63] (and this time can be achieved), and it can be accomplished in $\Theta(\log n)$ time on a EREW PRAM [30]. (CRCW PRAM algorithms finishing in this time appeared in [1], [3].) Stepwise simulation of this will provide a hypercube implementation taking $\Theta(\log^3 n)$ time, and mesh-of-trees, pyramid, and mesh implementations taking $\Theta(n \log n)$ time. While each of these architectures can solve the problem faster by a factor of log n [30], the relative inefficiency is much less than that encountered on the extreme points example for image data. As another example, given two points $(x1, y1)$ and $(x2, y2)$ in the plane, say that the first *dominates* the second if $x1 \geq x2$ and $y1 \geq y2$. Given a set of planar points, the *maximal points* are those that are not dominated by any other. Atallah and Goodrich [4] have given a data movement operation algorithm for determining maximal points which is optimal for all architectures considered here: first sort the points into decreasing order by their x coordinates, breaking ties in decreasing order of their y coordinates. Then use prefix to determine for each point the largest y coordinate of any point in a preceeding processor. A point is maximal if and only if its y coordinate is larger than this value.

However, the use of sorting should have little relevance for vision, particularly for the lower-level operations. While the algorithms in Section V involve extensive data movement, they require significantly less than is involved with sorting all the data. Once the image data is reduced to a representation as a set of points or a graph, the remaining steps may be as communication-intensive as sorting, but the amount of data involved is far less than the number of pixels and the movement can be performed more quickly. Quickly reducing the amount of data, and utilizing locality of information exchange, is critical for efficient vision algorithms. These features lead researchers to propose and utilize image processing architectures such as the pyramid and mesh-of-trees, despite the fact that they are no better than the mesh when they must sort n^2 values. An important exception to the principle of data locality occurs with extensive image rotation or warping, which require significant communication.

Because reducing the amount of data plays such an important role in vision, and because the communication capabilities of architectures vary widely, it is difficult to see how vision algorithms can be easily mapped to varied architectures and provide efficient implementations. If PRAMs with thousands of processors could be built to perform with acceptable speed then this would not be needed, and we would have only the (major) problem of determining how to write efficient vision algorithms for PRAMs. However, for quite some time it will be true that to achieve high performance one must take into account the location of data, even in pseudo-PRAM machines such as the RP3 or Connection Machine [35], [52]. Further, many vision applications, such as real-time navigation, place severe performance demands on vision systems. Use of simulations, or data movement operations, can be helpful in many algorithm mapping cases, but not in all. For the forseeable future programmers solving vision problems with high performance requirements will need to both understand the problem and the architecture in order to develop efficient algorithms.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlang, and C. Yap, "Parallel computational geometry," in *Proc. 25th IEEE Symp. Found. Comp. Sci.*, pp. 468-477, 1985.

[2] A. Agrawal, L. Nekludova, and W. Lim, "A parallel O(log N) algorithm for finding connected components in planar images," in *Proc. 1987 Int'l. Conf. on Parallel Proc.*, pp. 783-786.

[3] M. J. Atallah and M. T. Goodrich, "Efficient parallel solutions to some geometric problems," *J. Parallel and Distrib. Comp.*, vol. 3, pp. 492-507, 1986.

[4] —, "Efficient plane sweeping in parallel," *ACM Symp. Comp. Geometry*, pp. 216-225, 1986.

[5] K. E. Batcher, "Sorting networks and their applications," in *Spring Joint Comp. Conf.*, pp. 307-314, 1968.

[6] —, "Design of a massively parallel processor," *IEEE Trans. Comput.*, vol. C-29, pp. 836-840, 1980.

[7] F. Berman and L. Snyder, "On mapping parallel algorithms into parallel architectures," *J. Parallel and Dist. Computing*, vol. 4, pp. 439-458, 1987.

[8] W. T. Beyer, "Recognition of topological invariants by iterative arrays," Ph.D. Thesis, MIT, 1969.

[9] S. Bokhari, "On the mapping problem," *IEEE Trans. Comput.*, vol. C-30, 1981.

[10] V. Cantoni, M. Ferretti, S. Levialdi, and R. Stefanelli, "PAPIA: Pyramidal architecture for parallel image analysis," in *Proc. 7th Symp. on Comp. Arith.* pp. 237-242, 1985.

[11] V. Cantoni and S. Levialdi, Eds., *Pyramidal Systems for Computer Vision*. New York, NY: Springer-Verlag, 1986.

[12] M. C. Chen, "Very-high-level parallel programming in Crystal," in [Hea], pp. 39-47.

[13] R. Cole and U. Vishkin, "Approximate and exact parallel

scheduling with applications to list, tree and graph problems," in *Proc. 27th IEEE Symp. on Found. Comp. Sci.*, pp. 478-491, 1986.
[14] R. Cypher and J. L. C. Sanz, personal communication.
[15] P. E. Danielsson and S. Levialdi, "Computer architectures for pictorial information systems," *Computer*, vol. 14, pp. 53-67, 1980.
[16] D. Gelernter, "Generative communication in Linda," in *ACM Trans. Prog. Lang. Sys.*, 1985.
[17] M. J. Heath, Ed., *Hypercube Multiprocessors 1987*. SIAM Press, 1987.
[18] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, "Computing connected components on parallel computers," *Comm. ACM*, vol. 22, pp. 461-464, 1979.
[19] R. A. Hummel, "Image processing on the NYU Ultracomputer," in *Proc. Workshop on Algorithm-Guided Parallel Arch. for Auto. Target Recog.*, pp. 111-120, 1984.
[20] S. L. Johnsson, "Communication efficient basic linear algebra computations on hypercube architectures," *J. Parallel and Distrib. Comp.*, vol. 4, pp. 133-172, 1987.
[21] C. Kruskal, L. Rudolph, and M. Snir, "The power of parallel prefix," in *Proc. 1985 Int'l. Conf. Parallel Proc.*, pp. 180-185.
[22] J. T. Kuehn and H. J. Siegel, "Extensions to the C programming language for SIMD/MIMD parallelism," in *Proc. 1985 Int'l. Conf. on Parallel Proc.*, pp. 232-235.
[23] T. H. Lai and S. Sahni, "Anomalies in parallel branch-and-bound algorithms," *Comm. ACM*, vol. 27, pp. 594-602, 1984.
[24] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputers," in *Proc. 1985 Int'l. Conf. on Parallel Proc.*, pp. 393-402.
[25] R. Miller, "Writing SIMD algorithms," in *Proc. 1985 Int'l. Conf. on Computer Design: VLSI in Computers*, pp. 122-125.
[26] R. Miller and Q. F. Stout, "Geometric algorithms for digitized pictures on a mesh-connected computer," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-7, pp. 216-228, 1985.
[27] ——, "Data movement techniques for the pyramid computer," *SIAM J. Comp.*, vol. 16, pp. 38-60, 1987.
[28] ——, "Some graph- and image-processing algorithms for the hypercube," in [Hea], pp. 418-425.
[29] ——, *Parallel Algorithms for Regular Architectures*. Cambridge, MA: MIT Press, 1988.
[30] ——, "Parallel algorithms for convexity," in *Proc. Comp. Vision and Pat. Recogn.*, 1988, to appear.
[31] ——, "Simulating essential pyramids," in *Proc. Comp. Vision and Pat. Recogn.*, 1988, to appear.
[32] T. N. Mudge and T. S. Abdel-Rahman, "Vision algorithms for hypercube machines," *J. Parallel and Distrib. Comp.*, vol. 4, pp. 79-94, 1987.
[33] D. Nassimi and S. Sahni, "Finding connected components and connected ones on a mesh-connected parallel computer," *SIAM J. Comp.*, vol. 9, pp. 744-757, 1980.
[34] ——, "Parallel permutation and sorting algorithms and a new generalized connection network," *J. ACM*, vol. 29, pp. 642-667, 1982.
[35] G. F. Pfister *et al.*, "The IBM research parallel processor prototype (RP3): Introduction and architecture," in *Proc. 1985 Int'l. Conf. on Parallel Proc.*, pp. 764-771.
[36] V. K. Prasanna Kumar and M. Eshaghian, "Parallel geometric algorithms for digitized pictures on mesh of trees," in *Proc. 1986 Int'l. Conf. on Parallel Proc.*, pp. 270-273.
[37] A. P. Reeves, "Parallel Pascal: An extended Pascal for parallel computers," *J. Parallel and Distrib. Computing*, vol. 1, pp. 64-80, 1984.
[38] J. Reif and Q. F. Stout, "Optimal mesh and VLSI algorithms for component labeling and minimal spanning trees," to appear.
[39] A. Rosenfeld, "Parallel image processing using cellular arrays," *Computer*, vol. 16, pp. 14-20, 1983.
[40] ——, *Multiresolution Image Processing and Analysis*. Berlin, West Germany: Springer-Verlag, 1984.
[41] J. H. Saltz and M. C. Chen, "Automated problem mapping: The Crystal runtime system," in [Hea], pp. 130-140.
[42] D. H. Schafer, P. Ho, J. Boyd, and C. Vallejos, "The GAM pyramid," in [Uhr2], pp. 15-42.
[43] C. L. Seitz, "The cosmic cube," *Comm. ACM*, vol. 28, pp. 22-33, 1985.
[44] Y. Shiloach and U. Vishkin, "An O(log n) parallel connectivity algorithm," *J. Algorithms*, vol. 3, pp. 57-67, 1982.
[45] H. J. Siegel *et al.*, "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.*, vol. C-30, pp. 934-947, 1981.
[46] L. Snyder, "Parallel programming and the Poker programming environment," *Computer*, vol. 17, pp. 27-36, 1984.
[47] Q. F. Stout, "Mesh and pyramid computers inspired by geometric algorithms," in *Proc. Workshop on Algorithm-Guided Parallel Arch. for Auto. Target Recog.*, pp. 293-315, 1984.
[48] ——, "Pyramids and hypercubes," in [CaLe], pp. 75-90.
[49] ——, "Supporting divide-and-conquer algorithms for image processing," *J. Parallel and Distrib. Computing*, vol. 4, pp. 95-115, 1987.
[50] "Report of the summer workshop on parallel algorithms and architectures," Supercomputing Research Center Tech. Rep., 1986.
[51] S. L. Tanimoto and A. Klinger, *Structured Computer Vision: Machine Perception through Hierarchical Computation Structures*. New York, NY: Academic Press, 1980.
[52] "The connection machine supercomputer: A natural fit to applications needs," Thinking Machines Corp., 1985.
[53] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," *Comm. ACM*, vol. 20, pp. 263-271, 1977.
[54] L. Uhr, "Layered 'recognition cone' networks that preprocess, classify and describe," *IEEE Trans. Comput.*, vol. C-21, pp. 758-768, 1972.
[55] ——, Ed., *Parallel Computer Vision*. New York, NY: Academic Press, 1987.
[56] J. D. Ullman, *Computational Aspects of VLSI*. Rockville, MD: Computer Science Press, 1984.
[57] S. H. Unger, "A computer oriented towards spatial problems," *Proc. IRE*, vol. 46, pp. 1744-1750, 1958.
[58] L. G. Valiant, "Parallelism in comparison problems," *SIAM J. Computing*, vol. 3, 1975.
[59] ——, "A scheme for fast parallel communication," *SIAM J. Computing*, vol. 11, pp. 350-361, 1982.
[60] F. L. Van Scoy, "The parallel recognition of classes of graphs," *IEEE Trans. Comput.*, vol. C-29, pp. 563-570, 1980.
[61] S. Warshall, "A theorem on boolean matrices," *J. ACM*, vol. 9, pp. 11-12, 1962.
[62] W. I. Williams, "Load balancing and hypercubes: A preliminary look," in [Hea], pp. 108-113.
[63] A. C. Yao, "A lower bound to finding convex hulls," *J. ACM*, vol. 28, pp. 780-787, 1981.

Quentin F. Stout (Member, IEEE) was born on September 23, 1949. He received the B.A. degree from Centre College, Danville, KY, in 1970, and the Ph.D. degree from Indiana University, Bloomington, in 1977.

From 1976 to 1984 he was a member of the Mathematical Sciences Department of the State University of New York at Binghamton. Since 1984 he has been an Associate Professor in the Electrical Engineering and Computer Science Department at The University of Michigan, where he is a founding member of the Advanced Computer Architecture Laboratory. He is currently on the editorial board of the *Journal of Parallel and Distributed Computing*. His research interests include parallel algorithms; parallel architectures; image processing; and problems of describing, implementing, and monitoring parallel algorithms.