JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 4, 95–115 (1987)

Supporting Divide-and-Conquer Algorithms for Image Processing

QUENTIN F. STOUT*

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

Received April 4, 1986

Divide-and-conquer is an important algorithm strategy, but it is not widely used in image processing. For higher-level, symbolic operations it should often be the strategy of choice for parallel computers. It is natural for a machine with a regular interconnection scheme such as a mesh, mesh with broadcasting, tree, pyramid, mesh-of-trees, PRAM, or hypercube, and can be used either on a machine with a pixel per processor or on one with many pixels per processor. However, divide-and-conquer algorithms use parallel computers in a different manner than, say, local edge detection, so machines optimized for local neighborhood algorithms may be poor for divide-and-conquer algorithms. Some characteristics of divide-and-conquer algorithms are examined, along with some of their implications for the design of machines and languages which can support the efficient programming and execution of divide-and-conquer algorithms.

1. INTRODUCTION

Divide-and-conquer is an important algorithm strategy which has been successfully used on a wide range of problems, yet it receives scant attention as a strategy for image processing. This is particularly true for the image processing that is performed on parallel computers. Perhaps this is because divide-and-conquer is not especially useful for low-level local neighborhood operations, the sort of operations on which most parallel image processing computers excel. However, as research advances into using parallel computers for higher-level tasks such as scene recognition or image understanding, it seems that divide-and-conquer will become increasingly useful. To make

* Partially supported by the Naval Research Laboratory under Contract 65-2068-85 and the National Science Foundation under Grant DCR 8507851.

0743-7315/87 $3.00

use of this strategy, though, the hardware and software need to provide support for the requirements of divide-and-conquer algorithms.

This paper examines the divide-and-conquer strategy, giving some examples illustrating why it may be more useful in higher-level tasks, analyzing some of the requirements of such tasks, and deducing some of the implications for machine architectures and software. The viewpoint taken is strongly biased toward aiding algorithm design for high-level algorithms. It is also speculative, since the currently proposed algorithms for higher-level tasks are themselves quite speculative and incomplete. In fact, the uncertainty of the nature of future image understanding algorithms is itself a strong argument for considering the implications of the divide-and-conquer strategy. Divide-and-conquer has proven to be a useful strategy in numerous areas, so it is unwise to omit support for it.

This paper will concentrate on parallel computers which are local memory machines, instead of shared memory machines. That is, each processor will have its own memory, containing, among other things, some of the pixels. If a processor needs to obtain information residing in another processor then the information must be transmitted along communication links. Most current parallel computers are local memory machines, though a few shared memory machines have been built and others are being designed. Most of our observations also hold for shared memory machines.

Throughout, all images will be of size $n \times n$. We use Θ to mean "order exactly," Ω to mean "order at least," O to mean "order no greater than," and o to mean "order strictly less than." For example, $3n^2 + 6 = \Theta(n^2)$, $n*\sin^2(n) + 4 = \Omega(1)$, $n*\sin^2(n) + 4 = O(n)$, and $n*\sin^2(n) + 4 = o(n^2)$. The term *polylog time* will mean time which is $O(\log^k n)$ for some integer k.

2. Divide-and-Conquer

The essence of the divide-and-conquer strategy is quite simple:

> To solve a large instance of a problem, break it into smaller instances of the same problem, and use the solutions of these to solve the original problem.

The fact that the smaller problems are instances of the same problem is what distinguishes divide-and-conquer from the more general top-down strategy.

This strategy is strongly encouraged in texts on data structures and algorithms. For example,

> To summarize, many of the most interesting algorithms that we will encounter are based on the divide-and-conquer technique of combining the solutions of recursively solved smaller subproblems. [Sedgewick [27, p. 52]]

> Perhaps the most important, and most widely applicable, technique for designing efficient algorithms is a strategy called "divide-and-conquer." . . . It is probably the ease

> of discovery of divide-and-conquer algorithms that makes the technique so important, although in many cases the algorithms are also more efficient than more conventional ones. [Aho *et al.* [1, pp. 306–307]]

Efficient divide-and-conquer algorithms have been used in sorting, searching, geometry, graph theory, Fourier transforms, arithmetic, algebra, approximations of NP-hard problems, and many other areas [1, 3, 27]. As Ullman notes, it is also a useful strategy in hardware design [41, Sect. 3.3].

Divide-and-conquer has a large number of variations, depending on the work needed to create the subproblems and the work needed to combine the solutions of the subproblems. For example, most of the work in quicksort occurs while creating the subproblems, but most of the work in mergesort occurs while combining the subproblems. Similarly, the difference among preorder, inorder, and postorder tree traversals is the point at which the work on the original problem ("visiting" a node) is interspersed with the work on the subproblems ("visiting" its children).

Other variations are introduced by differences in the implementation of the strategy. For example, mergesort can be implemented recursively in a top-down manner, or, as is more common, iteratively in a bottom-up manner. In this case, the two approaches give algorithms requiring very similar times. In geometry, sometimes a recursive divide-and-conquer algorithm is converted to an iterative one using a planar sweep [27] to avoid repetitive sorting which can increase the time. Finally, dynamic programming can be considered as a bottom-up implementation of divide-and-conquer which eliminates redundant computations.

When using parallel computers, there are several additional reasons why a divide-and-conquer approach may be particularly useful. First, there may be more data than can be contained in the processors at one time, so the data must be analyzed piecemeal. For example, a 128×128 mesh (see Fig. 1a) of processors able to hold only a single pixel each may need to process an image with 1024×1024 pixels. In such cases it is often useful to analyze subsquares (dividing) and then combine the results (conquering), with particular emphasis on the boundaries of the subsquares.

Second, in some machines the individual processors may be quite large, holding many pixels, in which case often a two-level strategy is needed. We will call such machines *medium-grained* machines, to distinguish them from *fine-grained* machines with only a single (or a few) pixel per processor. (*Coarse-grained* machines have only, say, tens of powerful processors. Such machines will not be discussed herein.) For example, the Massively Parallel Processor (MPP) [2] is a fine-grained machine, while the Cosmic Cube [28] is a medium-grained machine. In medium-grained machines the pixels have already been "divided," typically into subsquares, and often the problem can be then "conquered" by combining serial and parallel algorithms. A serial algorithm is used in each processor (simultaneously) to solve as much of

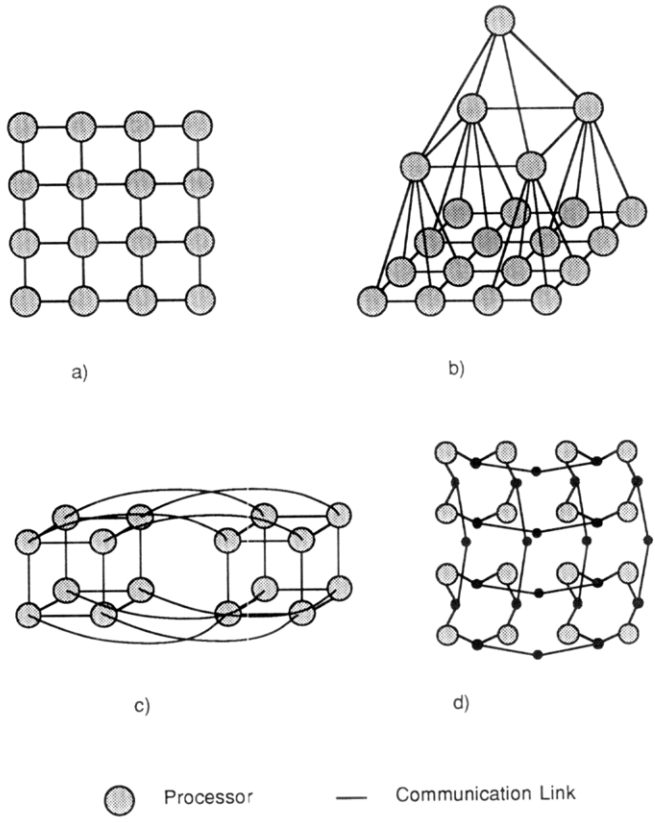

FIG. 1. Some parallel computers. (a) Mesh. (b) Pyramid. (c) Hypercube. (d) Mesh-of-trees (base mesh connections omitted for clarity).

the problem as possible, combined with a parallel algorithm to interchange necessary information. For example, if local median filtering is being performed, then pixel values along the edges of the subsquare in a processor need to be exchanged with the processors containing the neighboring subsquares. The determination of what information needs to be exchanged (a rather trivial consideration in this example) occurs whenever a divide-and-conquer approach is being tried.

Finally, many of the interconnection schemes being used or suggested for massively parallel image processing machines naturally suggest partitioning the machine into similar submachines. Meshes can easily be partitioned into quadrants, each of which is a mesh. Meshes with broadcasting capabilities almost force one to use a divide-and-conquer approach, dividing the mesh into submeshes in which a standard nonbroadcasting mesh algorithm is used, with broadcasting used to combine results of the subproblems [13, 31,

36]. Binary trees can be partitioned into an apex and two subtrees. Pyramids (see Fig. 1b) can be partitioned into the apex and four subpyramids. Binary hypercubes (see Fig. 1c) can be partitioned into halves, each of which is a (binary) hypercube of one smaller dimension. Mesh-of-tree machines (also known as orthogonal trees) can be partitioned into quadrants which are mesh-of-tree machines (see Fig. 1d). Parallel random access machines (PRAMs) can be divided at will into collections of smaller PRAMs, and usually machines which are simulations of PRAMs, such as the Ultracomputer [9] and RP3 [21], can similarly be arbitrarily partitioned, although some partitions have slightly lower performance than others. Further, some machines, such as PASM [29], have been explicitly designed to simplify partitioning.

Meshes and variations of them have been widely discussed and constructed as image processing machines [6, 24], pyramids and their close relatives have been widely discussed, and a few are being built, as image processing machines [5, 18, 26, 38, 40], and hypercubes are now becoming available for arbitrary uses. At least one medium-grained hypercube, at the University of Michigan, is being used for image processing, and some image processing algorithms are being developed for the Connection Machine [43], a fine-grained hypercube. The medium-grained Ultracomputer [9] is also being considered for image processing problems ranging from very low level to very high level, and the fine-grained NON-VON tree machine [10] is being considered for low-level image processing. A few vision-related algorithms for the mesh-of-trees appear in [12, 19], and vision-related algorithms for a mesh with broadcasting appear in [13, 31, 36].

3. Examples

Many divide-and-conquer algorithms have been developed for mesh, pyramid, tree, mesh-of-tree, PRAM, and hypercube computers, but these algorithms have usually originated from algorithm designers solving general problems, rather than from the image processing community which is actually using such machines. As was mentioned earlier, much of this can be attributed to the fact that divide-and-conquer is not needed for simple neighborhood transformations, while the algorithm designers have tended to concentrate on problems requiring more global integration of information.

The simplest divide-and-conquer solutions are tree-like. For example, to find the average gray level in an image, find the average in each quadrant, and average the averages. This approach is suitable for a tree, pyramid, hypercube, mesh-of-trees, or PRAM, taking logarithmic time if there is a pixel per processor. However, it is not a useful example as there are very few problems which can be solved so easily.

3.1. *Black/White Component Labeling*

Probably the most widely discussed nontrivial example is (*connected*) *component labeling* of a black/white image. In this problem, the image is an $n \times n$ array of pixels, each of which is either black or white. Two black pixels are said to be *adjacent* if they share an edge, and they are *connected* if there is a path of adjacent black pixels between them. Connectedness defines equivalence classes of black pixels, where two black pixels are in the same equivalence class if and only if they are connected. These classes are called (*connected*) *components*. The component labeling problem is to assign a label to each black pixel, where two black pixels receive the same label if and only if they are in the same component. Throughout, the pixel at row i, column j will start with a label of $(i-1)n + j$ (its row-major index). The (final) *label of a component* is the smallest label of any of its members.

3.1.1. *Propagation*

On a parallel computer where each processor holds one pixel, and where adjacent pixels are in adjacent processors, a common algorithm for this problem is as follows:

Propagation Component Labeling

1. Each black pixel starts with its row-major label.
2. Each black pixel takes as its new label the minimum of its current label and the current labels of all adjacent black pixels.
3. If no pixels changed their labels in this iteration then the algorithm is done, otherwise go to 2.

In each component, the pixel with the smallest row-major index in the component initially is the only one with its label. In the next cycle all adjacent black pixels acquire its label, in the next cycle all black pixels adjacent to these acquire it, and so on, with the minimum label propagating in a breadth-first manner.

This simple algorithm is quite easy to implement, and as a parallel algorithm has been suggested at least since the 1960s [4]. It has the difficulty that the number of iterations needed can be proportional to n^2, as would occur with a spiral as in Fig. 2. Because of this worst case possibility, either the algorithm must always be run for $\Theta(n^2)$ iterations, each of which takes unit time, or else the algorithm must occasionally intersperse steps in which the processors are checked to see if they have changed their labels. This interspersion can be done on meshes, pyramids, hypercubes, and many other parallel machines with a fixed interconnection network, so that the algorithm finishes in $\Theta(d + c)$ time, where d is the largest internal path distance between any pair of black pixels in the same component and c is the communication diameter of the machine. That is, for all pairs of black pixels p, q, let $d(p, q)$

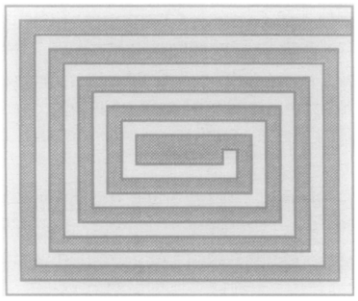

FIG. 2. A difficult image.

be 1 less than the minimum number of black pixels in a path of adjacent black pixels starting at p and ending at q, if p and q are in the same component, and let it be ∞ otherwise. Then

$$d = \max\{d(p, q) : d(p, q) < \infty\}.$$

Similarly, for processors P and Q, let $c(P, Q)$ be the minimum number of communication links which must be traversed in going from P to Q. Then

$$c = \max\{c(P, Q)\}.$$

Spiral-like images should be rare in any reasonable definition of expected case, so $\Theta(n)$ is probably a more realistic expected value of d. If this is so, then for meshes this algorithm will have an expected time of $\Theta(n)$, which is the best possible since $c = \Theta(n)$ and all algorithms potentially involving combining information from throughout the image must take $\Omega(c)$ time. However, for trees, pyramids, mesh-of-trees, PRAMs, and hypercubes, $c = o(n)$, and hence a propagation algorithm taking $\Theta(n)$ time is not necessarily optimal in the expected case. In the worst case the propagation algorithm is not even optimal for the mesh.

3.1.2. *Divide-and-Conquer*

For machines with a sublinear communication diameter, or to guarantee better worst case times on a mesh, there is a well-known divide-and-conquer solution which is much better than the propagation algorithm. It is based on Sollin's component labeling algorithm for graphs, a parallel version of which was given by Hirschberg [8]. Apparently the first application of this to images is that of Nassimi and Sahni [20], who used it to provide an optimal mesh algorithm. The algorithm is as follows:

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 quadrant.
4. Correct the labels of components lying in more than one quadrant.

Figure 3 shows the labels as they might be assigned by the end of Step 2. Note 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 data reduction is crucial in obtaining the desired speed.

This algorithm has been used to derive optimal algorithms for meshes, taking $\Theta(n)$ time on an $n \times n$ mesh [20], and pyramids, taking $\Theta(n^{1/2})$ time on a pyramid with an $n \times n$ base [18]. For a tree with n^2 leaves it gives an optimal algorithm taking $\Theta(n)$ time, and for mesh-of-trees, hypercubes, and PRAMs it gives algorithms taking poly-log time [12, 19].

A similar algorithm can be used if the pixels have some gray level. In this case one could choose some threshold value and set pixels black if their intensity is below the threshold, and white otherwise. The *gray-level connectedness problem* is to decide if the blacks always form a single connected component, no matter what threshold is used. This problem is a variant of the fuzzy connectedness problem [23]. One could try all thresholds, but if each pixel has a different gray level then this would take $\Omega(n^2)$ time. Instead, by slightly modifying the above algorithm, for all the machines mentioned above gray-level connectedness can be decided as quickly as black/white component labeling [35].

The above algorithm was described as if the machine had one pixel per processor, but it can easily be implemented on a medium-grained machine

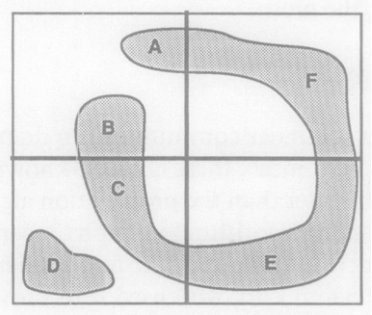

FIG. 3. Labels assigned in quadrants.

where each processor stores a small subsquare of the image. However, using a purely recursive divide-and-conquer approach may be somewhat slower than a blended approach which uses divide-and-conquer to reduce the size of the image until each piece is in a single processor, and then uses a fast serial algorithm within each processor to label its piece. (Even on a single processor a properly implemented pure divide-and-conquer approach gives an optimal algorithm, as measured by O-notation, but the constants involved may be significantly higher than other serial algorithms.) If each processor stores a $k \times k$ subimage, then Step 2 should be changed to read:

2. If the image is $k \times k$ or smaller then use a serial algorithm within each processor. 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.)

Note that it is very easy to incorporate a serial algorithm for small subproblems into the parallel divide-and-conquer algorithm.

3.2. *Convexity*

Given a black/white image with its components labeled, the *all-components extreme point problem* is to find the extreme points of each black component, that is, to find the black pixels in each component which are the corners of the smallest convex polygon containing the component. This problem is useful because the extreme points form a compact representation of a convex set, permitting faster algorithms for determining additional properties of the set such as its elongatedness or diameter. We first analyze the simpler problem of finding the extreme points of all of the black pixels. Parallel algorithms for these problems have appeared in [9, 16, 17, 19, 31] and in numerous other papers.

The algorithm is based on the simple observation that if one has a set of points and partitions it into subsets, then each extreme point of the original set is an extreme point of its subset. Using this, the structure of the algorithm is quite straightforward:

Extreme Points via Divide-and-Conquer

1. If the image is 2×2 then each black pixel is an extreme point and the algorithm is finished. Otherwise, divide the image into quadrants and find the extreme points of each component restricted to each quadrant. (A pixel's label is kept with the representation of the pixel throughout.)
2. Collect the extreme points from the quadrants for those components which lie in more than one quadrant, and eliminate those which are not extreme points of their component.

As for component labeling, this divide-and-conquer approach yields good algorithms for meshes [16], trees, pyramids [18], meshes with broadcasting [13, 31, 36], mesh-of-trees, hypercube [12, 19], and PRAMs [9]. Further, as

before, for medium-grained machines a serial algorithm can easily be utilized when the recursion has subdivided the image into pieces which fit into single processors.

3.3. *Multiresolution Algorithms*

One increasingly popular technique for image processing is the use of multiresolution images and algorithms [25]. Given an $n \times n$ image, an $(n/2) \times (n/2)$ image is formed by partitioning the original image into blocks of 4 pixels and averaging their values to get one new pixel value. This process is recursively repeated to get a sequence of images, the smallest of which is 1×1. (We have used averaging to get new pixel values, but other operations such as taking a median or lowest value can also be used to obtain the smaller images.) This sequence of images can then be used for a variety of operations, with much less computational effort than is required if all operations occur on the original image.

For example, suppose one has an image and is trying to determine where the sky, horizon, and road are. A multiresolution approach is to use one of the smaller, lower-resolution images to get initial estimates of the boundaries of the different objects, and then to follow these initial positions down through higher resolutions to find more exact boundaries. Quadrants of the larger images at higher resolutions can be processed in parallel, where there is communication between quadrants if the target objects cross their borders. The initial estimate is found on a smaller image so it requires less computation than an estimate based on the original image, and from then on only portions of the higher-resolution images are being examined.

Multiresolution image processing can be viewed as a variation of divide-and-conquer where the emphasis has been placed on the role of a specific data structure (the multiple images at different resolutions) which supports a divide-and-conquer approach. Perhaps not all authors would agree that multiresolution image processing is a variation of divide-and-conquer, but its similarity to divide-and-conquer is readily apparent. Currently multiresolution image processing is often thought of as being intimately tied to the use of pyramid computers, which naturally store the different images at different levels. However, multiresolution image processing can be fruitfully utilized on a wide variety of parallel machines, particularly if the proper software support is provided.

3.4. *Further Examples*

The approach of dividing an image into quadrants, solving subproblems in the quadrants, and combining the subproblem solutions, has been applied to yield good parallel algorithms for several other problems involving images. The approach can be used to find the nearest neighboring black component

for each black component, to find an encasing rectangle of minimal area for each black component, to find an encasing circle of minimal area for each black component, to decide if the black pixels form a straight line, to find a minimal path in a maze, and to find the diameter of each black component [16–18]. A more complex problem also solved via several uses of divide-and-conquer is to determine if two images are topologically isomorphic [32], i.e., to see if one can be deformed into the other.

An important example not directly related to image processing is sorting. Sorting forms the basis of several data movement operations, including random access read, random access write, and path compression for directed trees [20]. While these operations do not seem to be related to image processing, they actually play a crucial role in the optimal mesh and pyramid component labeling algorithms and the algorithms mentioned in the preceding paragraph. This point is expanded upon in Section 4.3.

Other examples include various graph algorithms such as deciding tree isomorphism, computing certain functions on trees, and component labeling of arbitrary graphs [32, 34]. Again, while these may not initially seem to be related to images, they are important ingredients of the optimal mesh algorithm for deciding if a pair of images are topologically isomorphic [32]. Further, since many artificial intelligence and higher-level image understanding algorithms make extensive use of graph algorithms for matching and searching problems, it is reasonable to expect that parallel graph algorithms will become increasingly important as more high-level image understanding tasks are attempted on parallel machines.

4. Requirements

If one examines the divide-and-conquer algorithms given or referenced above, it becomes clear that they are not performing quite the same operations as, say, local median filtering. This section examines some of these differences and makes suggestions concerning the hardware and software support needed to program and execute such algorithms efficiently.

4.1. *Spatial Subdivision*

The second step of the algorithm in Section 3.1.2 and the first step of the algorithm in Section 3.2 are typical steps in image processing divide-and-conquer algorithms, in that they partition the image into quadrants and then the operations proceed in each quadrant as if the quadrant is the entire image. To incorporate such steps, one needs software which allows such spatial subdivision to be easily described, and hardware which efficiently supports variations among the quadrants. First the hardware problem will be discussed, and then the software.

4.1.1. *Conditional Neighbors*

After the image has been subdivided into quadrants, the pixels along the middle lines must temporarily behave as if their neighbor across the middle line were not present. This is easily accomplished in an MIMD machine, but it can cause difficulty in some SIMD machines. For an SIMD machine to have some of its processors temporarily ignore the fact that they actually have a neighbor in a given direction, it is necessary that the processors have a masking ability to ignore instructions which would have them access neighbors which are not part of the same partition, or else they need the ability to set pointers or flags of some sort which indicate which neighbors to use on certain operations. Note that this ability is far less useful in an algorithm in which all processors access their neighbors in the same manner, as occurs with many local neighborhood operations.

A mask could either be broadcast to processors based on their coordinates, or computed from local data. The ability to compute the mask from local data would be useful in other steps, such as when a pixel needs to decide with which of its neighbors it should exchange labels. It often happens that a specific neighbor is picked based on local data, and then that neighbor is repeatedly used for certain communications. For example, in component labeling, once a black pixel has determined that a neighbor is white, it never sends any labels to that neighbor.

The ability to set a pointer or flag to indicate which neighbor to use can sometimes be more efficient than a masking ability. If the algorithm is at a point where each processor is sending data to at most one neighbor, then some implementations of masking would require the controller on a mesh computer to issue four instructions roughly saying "If you are communicating with your North neighbor then send it your data," "If you are communicating with your West neighbor then send it your data," etc. With the ability to set a flag or pointer indicating which neighbor gets the communication, only one instruction, of the form "Send your data to your flagged neighbor," need be sent. This is a relatively minor point, but because some algorithms rely so heavily on near-neighbor communication, appropriate support of conditional neighbors may significantly decrease the time. (However, since it can decrease time only by a multiplicative factor, it will not be apparent in an O-notational analysis.)

4.1.2. *Software*

While the pseudo-code algorithm outlines used in Section 3 are fairly natural descriptions, this would not be the case if they had to be translated into the languages currently in use on parallel machines. Most medium-grained machines are currently programmed in some serial language such as FORTRAN, augmented with a few communication primitives. In such a lan-

guage a global overview of the algorithm, as given in Section 3, must be replaced with the specific instructions to each processor on how to accomplish its task. While this must ultimately occur, it does not seem that the programmer should be required to do the translation.

Some fine-grained SIMD image processing machines can be programmed in languages such as Parallel Pascal [22], which provides some global overview of computations by specifying actions on global arrays with one entry per processor. However, Parallel Pascal does not directly support writing algorithms which subdivide global arrays into arrays global to quadrants, so again the programmer is forced to translate natural divide-and-conquer algorithms into a prespecified format not designed to support the writing of such algorithms.

One language which is more adapted to divide-and-conquer is Occam [44], in that Occam makes it easier to describe mapping algorithms and data onto processors, and can do so in a recursive manner. However, even Occam does not directly allow one to describe, say, an image as an array spread across the processors, so some of the global overview capabilities of Parallel Pascal are missing in Occam.

Besides providing a capability for describing data structures such as image arrays spread across processors, and providing a capability for describing recursive algorithms which can operate on such data structures both globally and in a subdivided manner, languages and/or programming environments also need to explicitly deal with the actual sizes of image subarrays on individual processors and with images too large to fit on the parallel machine at one time. To be portable, algorithms need to be written where the size of an image subarray that fits on a single processor is a constant which is initialized at execution time. It may be that the compiler or loader should determine this size by examining the memory requirements of the program and other data structures, combined with a knowledge of the memory size of the processors.

Explicitly dealing with images too large to fit on the machine at one time is currently an unsolved research problem, in that it is very difficult to determine how to do this efficiently. This problem involves an understanding of the algorithm involved, along with the characteristics of the processors and external memory devices. While it certainly will not be solved immediately, there is a need to increase the support a language (or programming environment) can provide to simplify this task. For example, the MPP implementation of Parallel Pascal explicitly avoids such support.

4.2. *Bandwidth*

An extremely important feature of a divide-and-conquer algorithm on a parallel computer is that it focuses attention on the information which one

region needs from another. This factor is often used to establish lower bounds on the performance of algorithms for parallel computers and VLSI. Conversely, given algorithms and the amount of information they transfer, and given a desired performance goal, the minimum acceptable bandwidth between regions can be determined, and from this machines can be designed. For example, this approach is used in Leiserson's Fat-Trees [14], which are a class of tree machines which efficiently simulate other parallel machines by providing the same bandwidth between regions. A related approach, where one uses bandwidth considerations to choose a best design within a class of parallel machines, appears in Stout [36].

Miller and Stout use bandwidth considerations to establish the somewhat nonobvious optimality of the divide-and-conquer pyramid component labeling algorithm in [18]. Bandwidth considerations also show that a pyramid can sort or rotate an image no faster than a mesh (to within a small multiplicative constant). To see this, one merely notes that a pyramid sliced through its center has at most $2n$ wires crossing the cut, while sorting or rotating an image will require moving $\Theta(n^2)$ items across the cut, thus requiring $\Omega(n)$ time. The same bandwidth considerations show that simply adding a global bus, or even a bus per row and column, cannot significantly improve the sorting or image rotation time of a mesh. They also show that a simple tree is a poor architecture for image processing since cutting one wire leading to the root disconnects the tree into halves, but as Leiserson noted it can be turned into a potentially suitable one by increasing the bandwidth toward the root [14].

Note that all of these architectures have a communication diameter which is $O(\log n)$, but that this small diameter is not sufficient to guarantee logarithmic, or even polylogarithmic, algorithms for image rotation. The diameter, which usually is an easily determined function of the size, is often used as simplistic evidence that an architecture is "good." While a small diameter usually guarantees that simple problems such as determining the average gray level or counting the number of black pixels can be done quickly, it is no guarantee of good performance on nontrivial problems.

Many divide-and-conquer algorithms for images show that the number of data which must be transferred between quadrants is $O(n)$, instead of $\Omega(n^2)$. This was true for the component labeling problem, the gray-level connectedness problem, the single-component convexity problem, and for all of the image related problems mentioned in Section 3.4. For example, for component labeling, the algorithm in Section 3.1.2 shows that only information about labels on the boundary of quadrants need be exchanged with other quadrants. This reduces the data between quadrants to $\Theta(n)$, and the example in Fig. 4 shows that this much information must be exchanged in some bad situations. For deciding convexity of a single component, it is easy to see that each quadrant has at most one extreme point in each row and column, so the amount of information that must be exchanged is $O(n)$. Using a bit of

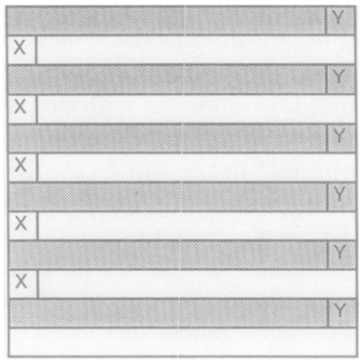

FIG. 4. Proper Y labels require knowledge of whether X's are black or white.

number theory, one can show that actually each quadrant can have at most $\Theta(n^{2/3})$ extreme points, and that this bound is the best possible [42].

Based on these observations, one can design pyramid-like computers with sufficient bandwidth between quadrants that all of these problems can be solved in poly-log time, even in the worst case, while still trying to retain some of the wiring cost advantages of a pyramid as compared to, say, a hypercube [18]. Some designs achieving this performance appeared in [32], and many others are possible. Mesh-of-trees machines [12, 17] also provide exactly enough bandwidth between quadrants to solve these problems in poly-log worst case time, even though they cannot sort in poly-log time. However, when using bandwidth considerations to help design a machine, one must be careful to include the proper problems. For example, while the mesh-of-trees can be used to determine the extreme points of a single component in poly-log time, it cannot always determine the extreme points of all components in poly-log time. The reason for this is that an image as in Fig. 5 has

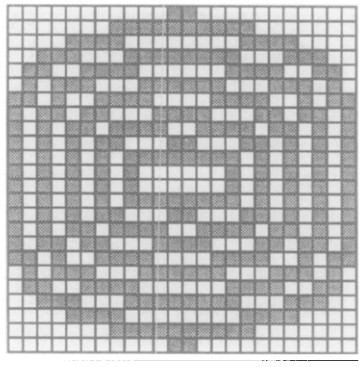

FIG. 5. An image with many components and many extreme points.

$\Theta(n)$ components crossing the middle, each of which might have $\Theta(n^{2/3})$ extreme points. Since the mesh-of-trees has only $\Theta(n)$ wires crossing the middle, it will take $\Omega(n^{2/3})$ time.

One interesting question in connection with this is the expected amount of information which must be passed between regions. If this is significantly lower than the amount needed in the worst case for nonadjacent regions, then it may be possible that a standard pyramid can have poly-log expected time on many of these problems, even though its worst case time is much slower. It is an interesting and useful research problem to determine a good model of expected image complexity for several of the image problems mentioned herein, since from such a model one might be able to suggest new architectures which obtain the desired performance with an inexpensive design.

Bandwidth considerations are one reason to consider hypercube machines for image processing problems, particularly in a research environment where numerous algorithms are being tried. When a hypercube with p processors is partitioned into two subcubes, there are $p/2$ wires between the halves. Thus the hypercube can solve some problems, such as sorting and image rotation, in poly-log time even though potentially all the data must be exchanged between the subproblems. This makes the hypercube much more powerful than the mesh or pyramid, or than any of the proposed broadcasting based modifications of these architectures. Since a hypercube can efficiently simulate each of the other architectures [37], it provides a useful general-purpose image processing machine, though its numerous long wires make it more expensive than the other designs.

4.3. *Data Movement Operations*

The actual implementations of the divide-and-conquer algorithms introduce important aspects which may be overlooked initially. For example, an interesting aspect of Step 3 of the divide-and-conquer component labeling algorithm is the fact that the relabeling decisions inside the subsquare are very symbolic, and in fact use a general graph labeling algorithm. These data no longer resemble pixels, and almost all of the initial geometric properties of the data have been replaced with graph properties. Processors in the subsquare are not performing many arithmetic operations, but are instead using comparisons and pointer manipulation. Therefore instruction sets need to provide a sufficiently rich set of operations for such symbol manipulation.

Another feature of the divide-and-conquer component labeling algorithm is that an optimal mesh algorithm must complete Step 3 in $O(n)$ time. Nassimi and Sahni [20] accomplished this by moving the $O(n)$ words of information to a subsquare. Moving to and from the subsquare takes $O(n)$ time. The subsquare has a communication diameter of only $\Theta(n^{1/2})$, which enabled

the relabeling decisions within the subsquare to be performed in $o(n)$ time. Therefore the total time for the mesh implementation of Step 3 was $O(n)$. Similar movement was needed in the optimal pyramid implementation [18], but to a subsquare above the base.

The movement of the data to and from the subsquare may be slightly unexpected, but it is needed so that the resulting graph problem can be solved in a region with as small a communication diameter as possible. The movement is simple in that it is possible to predetermine the subsquare to which items are moved, and to give easy, efficient routing algorithms. The operations used in the graph algorithm are not like this, in that they are concerned with finding the information associated with a given key, without knowing in advance the index of the processor holding the record with the given key. In [20], operations to accomplish this are called random access read, random access write, and path compression, and are based on using efficient sorting algorithms [37]. These operations have been used and extended in many other mesh algorithms [16], and several related pyramid, hypercube, and mesh-of-tree operations have been developed [18, 19]. Other common data movement operations are broadcast and report functions, which are usually implemented in a tree-like manner. In broadcasting there is some information which must be sent to all processors, while in reporting there are values at all processors which must be collected together, perhaps combining them together with some semigroup operation such as addition or minimization.

The role of these data movement operations is crucial in the divide-and-conquer algorithms for all of the image related problems mentioned in Section 3. To achieve desired levels of speed on these operations, it may be that special hardware support is necessary. For SIMD machines this may involve additional microcode for the controller, while for MIMD machines it may encourage the use of separate communications processors at each node, where the communication processor has the microcode for the operations.

Whether or not the operations are directly supported by hardware, the software needs to supply such operations. The operations are too complicated for most users to develop on their own, particularly if they are primarily interested in quickly trying new algorithms and approaches instead of discovering efficient ways to move data. Work is needed to develop a fairly complete collection of data movement operations, in that a wide variety of algorithms need to be examined to see what operations are needed. The operations might be directly supplied in some language especially designed for expressing parallelism, or might be offered via preprocessors for some serial language like FORTRAN. Currently neither approach is being used, except in locations where a few standard subprograms have been written for operations such as simple broadcasting or reporting.

The data movement operations are an important tie between global and

local views of the computation. Algorithms can be described as local computations interspersed with global data movement, where the programmer should not have to worry about the instructions an individual processor must execute in order to perform the global data movement. This style of programming makes code more transportable across architectures since, while the specific instructions performed by the processors will change, and the time of the operation might also change, the logical effect of data movement operations such as broadcasting a word of data to all processors is independent of the architecture. If the individual data movement operations are efficiently coded, then the resulting algorithms can be efficient on many architectures. Since there will be only a fairly small set of data movement operations, they will form a small core of procedures which must be efficiently implemented on each new machine, and then divide-and-conquer algorithms for general machines can be ported to the specific one. This point of view is further expanded in [15], where it is shown how to code the component labeling algorithm explicitly in terms of such data movement operations. The (O-notational) speed of the resulting algorithm is also given for a variety of architectures.

5. Conclusion

We have shown that divide-and-conquer has been used for a variety of algorithms on meshes, trees, pyramids, mesh-of-trees, PRAMs, and hypercubes, and that one may expect that they will be used for higher-level image processing in the future. We have shown that divide-and-conquer algorithms use neighbor processors in a more conditional way than they are used by local neighborhood algorithms, that a programmer must be able to specify in a natural manner the spatial subdivision of an image and its mapping onto a machine, that divide-and-conquer algorithms can point to the bandwidth needed between regions, and that sophisticated data movement operations are important when actually implementing divide-and-conquer algorithms.

Divide-and-conquer provides an interesting method for interrelating the global and local actions of computations. If proper software support is provided, then programmers of local memory machines will have the ability to describe global data structures which can be acted on either uniformly or differentially depending on their position, and they will have a collection of data movement operations which manipulate data throughout the entire machine, as well as retaining the current ability to describe the local computations. Even for problems which are not solved via divide-and-conquer, these seem to be desirable features for any language which is intended to support rapid prototyping of new advanced algorithms for image processing and image understanding. The development of languages allowing one to

express and exploit parallelism is stated as a critical goal of parallel computing in [45].

Some hardware systems already support some of the desirable hardware features mentioned, but these factors are certainly not universally supported. This is particularly true among SIMD image processing machines where the emphasis has been on low-level image processing. Further, language and/or programming environment support for the software issues raised here is quite meager. Languages and programming environments such as those described in [7, 11, 22, 30, 44] support a few of the desiderata noted here, but none of them provides the full support desired, and none are in widespread use. If programmers are going to be able to easily develop high-level algorithms using standard algorithm techniques such as divide-and-conquer, then the hardware and software must take many more of these factors into account.

REFERENCES

1. Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, 1983.
2. Batcher, K. E. Design of a massively parallel processor. *IEEE Trans. Comput.* **C-29** (1980), 836–840.
3. Bentley, J. L. Multidimensional divide-and-conquer. *Comm. ACM* **23** (1980), 214–229.
4. Beyer, W. T. Recognition of topological invariants by iterative arrays. Ph.D. thesis, Massachusetts Institute of Technology, 1969.
5. Cantoni, V., Ferretti, M., Levialdi, S., and Stefanelli, R. PAPIA: Pyramidal architecture for parallel image analysis. *Proc. 7th Symp. on Comp. Arith.*, 1985, pp. 237–242.
6. Danielsson, P. E., and Levialdi, S. Computer architectures for pictorial information systems. *IEEE Comput.* **14** (1981), 53–67.
7. Gelernter, D. Generative communication in Linda. *ACM Trans. Progr. Lang. Systems* (1985).
8. Hirschberg, D. S., Chandra, A. K., and Sarwate, D. V. Computing connected components on parallel computers. *Comm. ACM* **22** (1979), 461–464.
9. Hummel, R. A. Image processing on the NYU Ultracomputer. *Proc. Workshop on Algorithm-Guided Parallel Arch. for Auto. Target Recog.*, 1984, pp. 111–120.
10. Ibrahim, H. A. H. Some image understanding algorithms on fine-grained tree-structured SIMD machines: Extended abstract. *Proc. Workshop on Algorithm-Guided Parallel Arch. for Auto. Target Recog.*, 1984, pp. 121–142.
11. Kuehn, J. T., and Siegel, H. J. Extensions to the C programming Language for SIMD/MIMD parallelism. *Proc. 1985 Int'l. Conf. on Parallel Proc.*, pp. 232–235.
12. Prasanna Kumar, V. K., and Eshaghian, M. Parallel geometric algorithms for digitized pictures on mesh of trees. *Proc. 1986 Int'l. Conf. on Parallel Proc.*, pp. 270–273.
13. Prasanna Kumar, V. K., and Raghavendra, C. S. Image processing on enhanced mesh connected computers. *Proc. Comp. Arch. for Pattern Anal. and Image Database Man,* 1985, pp. 243–247.

14. Leiserson, C. E. Fat-trees: Universal networks for hardware-efficient supercomputers. *Proc. 1985 Int'l. Conf. on Parallel Proc.*, pp. 393–402.
15. Miller, R. Writing SIMD algorithms. *Proc. 1985 Int'l. Conf. on Computer Design: VLSI in Computers*, pp. 122–125.
16. Miller, R., and Stout, Q. F. Geometric algorithms for digitized pictures on a mesh-connected computer. *IEEE Trans. PAMI* **PAMI-7** (1985), 216–228.
17. Miller, R., and Stout, Q. F. Pyramid computer algorithms for determining geometric properties of figures. *Proc. Symp. on Computational Geometry*, 1985, pp. 263–271.
18. Miller, R., and Stout, Q. F. Data movement techniques for the pyramid computer. *SIAM J. Comput.* **16** (1987), in press.
19. Miller, R., and Stout, Q. F. Data movement operations for the mesh-of-trees and hypercube networks. Submitted for publication.
20. Nassimi, D., and Sahni, S. Finding connected components and connected ones on a mesh-connected parallel computer. *SIAM J. Comput.* **9** (1980), 744–757.
21. Pfister, G. F., Brantley, W. C., George, D. A., Harvey, S. L., Kleingelder, W. J., McAuliffe, K. P., Melton, E. A., Norton, V. A., and Weiss, J. The IBM Research Parallel Processor Prototype (RP3): Introduction and architecture. *Proc. 1985 Int'l. Conf. on Parallel Proc.*, pp. 764–771.
22. Reeves, A. P. Parallel Pascal: An extended Pascal for parallel computers. *J. Parallel Distrib. Comput.* **1** (1984), 64–80.
23. Rosenfeld, A. Fuzzy digital topology. *Inform. and Control* **40** (1979), 76–87.
24. Rosenfeld, A. Parallel image processing using cellular arrays. *IEEE Comput.* **16** (1983), 14–20.
25. Rosenfeld, A. *Multiresolution Image Processing and Analysis.* Springer-Verlag, Berlin, 1984.
26. Schaefer, D. H., Gun, Z. M., Harris, V. J., II, and Wilcox, G. C. The PMMP—A pyramid of MPP processing elements. Tech. rep., George Mason University, 1985.
27. Sedgewick, R. *Algorithms.* Addison-Wesley, Reading, MA, 1983.
28. Seitz, C. L. The Cosmic Cube. *Comm. ACM* **28** (1985), 22–33.
29. Siegel, H. J., Siegel, L. J., Kemmerer, F. C., Mueller, P. T., Jr., Smalley, H. E., Jr., and Smith, S. D. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Trans. Comput.* **C-30** (1981), 934–947.
30. Snyder, L. Parallel programming and the Poker programming environment. *Computer* **17** (1984), 27–36.
31. Stout, Q. F. Broadcasting in mesh-connected computers. *Proc. 1982 Conf. on Inform. Sci. and Systems*, pp. 85–90.
32. Stout, Q. F. Topological matching. *Proc. 15th ACM Symp. on Theory of Computing*, 1983, pp. 24–31.
33. Stout, Q. F. Mesh and pyramid computers inspired by geometric algorithms. *Proc. Workshop on Algorithm-Guided Parallel Arch. for Auto. Target Recog.* (1984), pp. 293–315.
34. Stout, Q. F. Tree-based graph algorithms for some parallel computers. *Proc. 1985 Int'l. Conf. on Parallel Proc.*, pp. 727–730.
35. Stout, Q. F. Properties of divide-and-conquer algorithms for image processing. *Proc. Computer Arch. for Pattern Anal. and Image Database Man.*, 1985, pp. 203–210.
36. Stout, Q. F. Meshes with multiple buses, *Proc. 27th Found. of Comp. Sci.*, 1986, pp. 264–273.
37. Stout, Q. F. Pyramids and hypercubes. In Cantoni, V., and Levialdi, S. (Eds.). *Pyramidal Systems for Computer Vision.* Springer-Verlag, New York, 1986, pp. 75–89.

38. Tanimoto, S. L., and Klinger, A. *Structured Computer Vision: Machine Perception through Hierarchical Computation Structures.* Academic Press, New York, 1980.

39. Thompson, C. D., and Kung, H. T. Sorting on a mesh-connected parallel computer. *Comm. ACM* **20** (1977), 263–271.

40. Uhr, L. Layered 'recognition cone' networks that preprocess, classify and describe. *IEEE Trans. Comput.* **C-21** (1972), 758–768.

41. Ullman, J. D. *Computational Aspects of VLSI.* Comput. Sci. Press, Rockville, MD, 1984.

42. Voss, K., and Klette, R. On the maximum number of edges of convex digital polygons included into a square. Friedrich-Schiller-Universitat Jena, Nr. N/82/6, 1982.

43. The Connection Machine Supercomputer: A natural fit to applications needs. Thinking Machines Corp., 1985.

44. Occam II, Inmos Corp., 1986.

45. Report of the summer workshop on parallel algorithms and architectures. Supercomputing Research Center Tech. Rep., 1986.