JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **35,** 173–190 (1996)
ARTICLE NO. 0079

Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study

DAVID A. BADER[1] AND JOSEPH JÁJÁ[2]

Institute for Advanced Computer Studies and Department of Electrical Engineering, University of Maryland, College Park, Maryland 20742

This paper presents efficient and portable implementations of two useful primitives in image processing algorithms, histogramming and connected components. Our general framework is a single-address space, distributed memory programming model. We use efficient techniques for distributing and coalescing data as well as efficient combinations of task and data parallelism. Our connected components algorithm uses a novel approach for parallel merging which performs drastically limited updating during iterative steps, and concludes with a total consistency update at the final step. The algorithms have been coded in SPLIT-C and run on a variety of platforms. Our experimental results are consistent with the theoretical analysis and provide the best known execution times for these two primitives, even when compared with machine-specific implementations.

1. PROBLEM OVERVIEW

Given an $n \times n$ image with k gray levels on a p-processor machine ($p \leq n^2$), we wish to develop efficient and portable parallel algorithms to perform various useful primitive image processing computations. Efficiency is a performance measure used to evaluate parallel algorithms. This measure provides an indication of the effective utilization of the p processors relative to the given parallel algorithm. For example, an algorithm with an efficiency near one runs approximately p times faster on p processors than the same algorithm on a single processor. Portability refers to code that is written independent of low-level primitives reflecting machine architecture or size. Our goal is to develop portable algorithms that are scalable in terms of both image size and number of processors, and that provide high performance even when run on a sequential processor.

Our first algorithm computes the histogramming of an image; i.e., the output consists of an array $H[0, ..., k-1]$ held in a single processor such that $H[i]$ is equal to the number of pixels in the image with gray level i. Without loss of generality, k is assumed to be a power of 2. The second algorithm performs the connected components of images [2, 9, 11, 12, 15, 17, 18, 20, 33]. The task of connected component labeling is cited as an important object recognition problem in the DARPA Image Understanding benchmarks [40, 32], and also can be applied to several computational physics problems such as percolation [35, 8] and various cluster Monte Carlo algorithms for computing the spin models of magnets, such as the two-dimensional Ising spin model [3, 7, 34]. All pixels with gray level (or "color") 0 are assumed to be background, while pixels with color > 0 are foreground objects. A connected component in the image is a maximal collection of like-colored pixels such that a path exists between any pair of pixels in the component. Note that we are using the notion of 8-connectivity, meaning that two pixels are adjacent if and only if one pixel lies in any of the eight positions surrounding the other pixel, or 4-connectivity, in which only the north, east, south, and west pixels are adjacent. Each colored pixel in the image will receive a positive integer label; pixels will have the same label if and only if they belong to the same connected component. Also, all 0-pixels will receive a label of 0.

The majority of previous parallel histogramming algorithms are architecture- or machine-specific and do not port well to other platforms. Table I shows some previous running times for histogramming algorithms on parallel machines. Note that several of these machines are special purpose image processing machines. The last column corresponds to a normalized measure of the amount of work per pixel, where the total work is defined to be the product of the execution time and the number of processors. In order to normalize the results between fine- and coarse-grained machines, we divide the number of processors in the fine-grained machines by 32 to compute the work per pixel site.

As with the histogramming algorithms, most of the previous connected components parallel algorithms as well are architecture- or machine-specific, and do not port easily to other platforms. Table II shows some previous running times for implementations of connected components for images using parallel machines. Note that the image used in each of these benchmarks is the DARPA II Image Understanding Benchmark Image shown in Fig. 1. Again, several of these machines are special purpose image processing machines. The second last column corresponds to

[1] The support by NASA Graduate Student Researcher Fellowship NGT-50951 is gratefully acknowledged.

[2] Supported in part by NSF Grant CCR-9103135 and NSF HPCC/GCAG Grant BIR-9318183.

0743-7315/96 $18.00

TABLE I
Implementation Results of Parallel Histogramming Algorithms

Year	Researcher(s)	Machine	Processors	Image size	Time	Work per pixel
1980	Marks [28]	AMT DAP	1024	32×32	17.25 ms	539 μs
1983	Potter [31]	Goodyear MPP	16384	128×128	16.4 ms	513 μs
1984	Grinberg *et al.* [16]	3-D machine	16384	256×256	1.7 ms	13.3 μs
1987	Ibrahim *et al.* [19]	NON-VON 3	16384	128×128	2.16 ms	67.5 μs
1990	Nudd *et al.* [30]	Warwick Pyramid	16K base	256×256	237 μs	2.47 μs
1991	Jesshope [23]	AMT DAP 510	1024	512×512	86 ms	10.5 μs
1994	Bader and JáJá [4, 5]	TMC CM-5	16	512×512	12.0 ms	732 ns
		IBM SP-2	16	512×512	9.82 ms	599 ns
		Intel Paragon	8	512×512	20.8 ms	635 ns
		Meiko CS-2	4	512×512	15.2 ms	231 ns

a normalized measure of the amount of work per pixel, where the total work is defined to be the product of the execution time and the number of processors.

Section 2 describes the algorithmic model used to analyze the algorithms whereas Section 3 describes the input images used, as well as the data layout on the parallel platforms. The histogramming algorithm is presented in Section 4, and the binary connected components algorithm is described in Section 5. Section 6 generalizes the connected components algorithm to multi-gray-level images.

The experimental data obtained reflect the execution times from implementations on the Cray T3D, Thinking Machines CM-5, IBM SP-1 and SP-2, Meiko CS-2, and the Intel Paragon, with the number of parallel processing nodes ranging from 16 to 128 for each machine when possible. The algorithms are implemented in Split-C [13], a

TABLE II
Implementation Results of Parallel Connected Components of DARPA II Image (512×512)

Year	Researcher(s)	Machine	PEs	Time	Work/pix	Notes
1989	Kanade and Webb [24]	Warp	10	4.34 s	166 μs	Shrink/expand
1989	Weems *et al.* [39]	Alliant FX-80	8	7.225 s	220 μs	
		Sequent Symmetry 81	8	15.12 s	461 μs	
		Warp	10	3.98 s	152 μs	
		TMC CM-2	32768	140 ms	547 μs	
1992	Choudhary and Thakur [10]	Intel iPSC/2	32	1.914 s	234 μs	Multidim. divide & conquer (partitioned input)
				1.649 s	201 μs	Multidim. divide & conquer (complete im./PE)
				2.290 s	280 μs	Multidim. divide & conquer (cmplt. + collect. comm.)
		Intel iPSC/860	32	1.351 s	165 μs	Multidim. divide & conquer (partitioned input)
				1.031 s	126 μs	Multidim. divide & conquer (complete im./PE)
				947 ms	116 μs	Multidim. divide & conquer (cmplt. + collect. comm.)
		Encore Multimax	16	521 ms	31.8 μs	Multidim. divide & conquer (partitioned input)
1994	Choudhary and Thakur [11]	TMC CM-5	32	456 ms	55.7 μs	Multidim. divide & conquer (partitioned input)
				398 ms	48.6 μs	Multidim. divide & conquer (complete im./PE)
				452 ms	55.2 μs	Multidim. divide & conquer (cmplt. + collect. comm.)
1994	Bader and JáJá [4–6]	TMC CM-5	16	474 ms	28.9 μs	
			32	368 ms	44.9 μs	
		IBM SP-1	4	370 ms	5.65 μs	
		IBM SP-2-TH	4	260 ms	3.97 μs	
		IBM SP-2-WD	4	243 ms	3.71 μs	
		Meiko CS-2	4	627 ms	9.57 μs	
		Cray T3D	4	470 ms	7.17 μs	

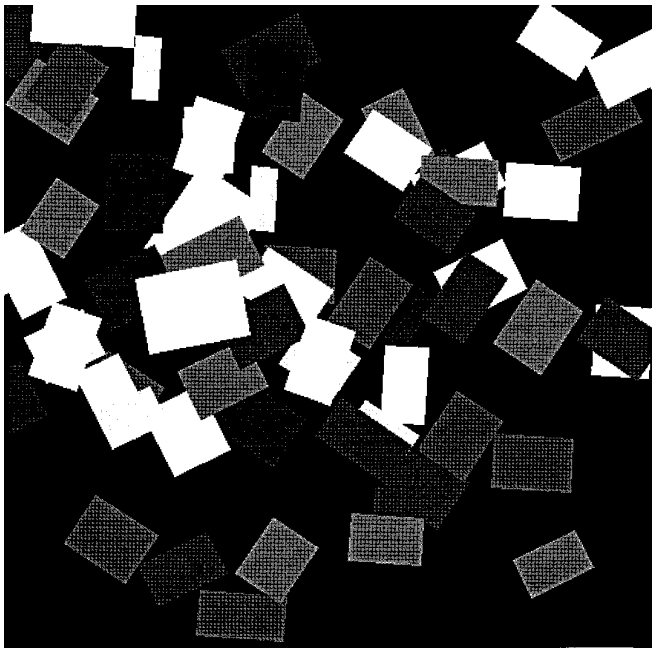


FIG. 1. DARPA II Image understanding benchmark test image.

parallel extension of the C programming language which follows the SPMD (single program multiple data) model on these parallel machines.

2. THE PARALLEL COMPUTATION MODEL

We use a simple model [21, 22] to analyze the performance of our parallel algorithms. Each of our hardware platforms can be viewed as a collection of powerful processors connected by a communication network that can be modeled as a complete graph on which communication is subject to the restrictions imposed by the latency and the bandwidth properties of the network. We view a parallel algorithm as a sequence of local computations interleaved with communication steps, and we allow computation and communication to overlap. We account for communication costs as follows.

Assuming no congestion, the transfer of a block consisting of m contiguous words between two processors takes $\tau + \sigma m$ time, where τ is a bound on the latency of the network and σ is the time per word for a processor to inject or receive data from the network.

Using this cost model, we can evaluate the communication time $T_{\text{comm}}(n, p)$ of an algorithm as a function of the input size n, the number of processors p, and the parameters τ and σ. The coefficient of τ gives the total number of times collective communication is used, and the coefficient of σ gives the maximum total amount of data exchanged between a processor and the remaining processors. This communication model is close to a number of similar models (e.g., [14, 37, 1]) that have recently appeared in the literature and seems to be well suited for designing parallel algorithms on current high performance platforms.

We define the computation time $T_{\text{comp}}(n, p)$ as the maximum time it takes a processor to perform all the local computation steps. In general, the overall performance $T_{\text{comp}}(n, p) + T_{\text{comm}}(n, p)$ involves a tradeoff between $T_{\text{comp}}(n, p)$ and $T_{\text{comm}}(n, p)$. Our aim is to develop parallel algorithms that achieve $T_{\text{comp}}(n, p) = O(T_{\text{seq}}/p)$ such that $T_{\text{comm}}(n, p)$ is minimum, where T_{seq} is the complexity of the best sequential algorithm. The important point to notice is that, in addition to scalability, our optimization criterion requires that the parallel algorithm be an efficient sequential algorithm (i.e., the total number of operations of the parallel algorithm is of the same order as T_{seq}).

Two useful data movement patterns, **transpose** and **broadcast,** are discussed next, and their analyses will be included as communication primitives in the algorithms that follow.

Given a $q \times p$ array on a p-processor machine, where p divides q, the **transpose** (also, all-to-all personalized communication) consists of rearranging the data such that the first q/p rows of elements are destined to the first processor, the second q/p rows to the second processor, and so on, with the last q/p rows of the matrix destined to the last processor. An efficient matrix transposition algorithm consists of p iterations such that, during iteration i, each processor P_t prefetches the appropriate block of q/p elements from processor $P_{(t+i) \bmod p}$.

Next, an efficient parallel algorithm (**bcast**) is given which takes q elements on a single processor and broadcasts them to the other $p - 1$ processors using just two **transpose** primitives.

Performance analysis given will reflect the execution times from implementations on the T3D, CM-5, SP-2, and CS-2, each with $p = 32$ parallel processing nodes. The algorithms are written in Split-C, a parallel extension of the C programming language, primarily intended for distributed memory multiprocessors. Split-C can express the capabilities of our model and provides constructs to express data layout and **split-phase** assignments. The **split-phase** assignment operator, :=, prefetches data from the specified remote location into local memory. Computation can be overlapped with the remote request, and the **sync**() function allows each processor to stall until all data prefetching is complete. The Split-C language also supplies a **barrier**() function for the global synchronization of the processors.

2.1. *Analysis for the* **transpose** *Primitive*

The analysis for the **transpose** primitive is similar to that of the LogP model analysis [14]. The algorithm to perform a $q \times p$ matrix **transpose** on a p-processor machine operates as follows. The data layout of matrix A is straightforward; each column i of q elements is stored on processor i, for $i \in \{0, ..., p - 1\}$. Note that the first index of A contains the processor number, while the second index provides the element offset in that processor.

Processor i runs the following program:

ALGORITHM 1. **transpose** *Communication Primitive*
Input:
$\{A\}$ is the $q \times p$ input array.
begin
1. **For** loop = 0 **to** $p - 1$ **do**:
   1.1 **Set** $r = (i + \text{loop})$ **mod** p;
   1.2 **Prefetch** $A^{\mathrm{T}}[i][(r * q/p) \cdots ((r + 1) * q/p) - 1] := A[r][(i * q/p) \cdots ((i + 1) * q/p) - 1]$;
2. **Sync**()
end

Each prefetch in Step 1.2 requests a block of q/p elements. Since each processor prefetches $p - 1$ blocks of q/p each, this **transpose** algorithm will take $\tau + \sigma(p - 1)q/p$ communication complexity, or

$$\begin{cases} \mathrm{T}_{\mathrm{comm}}(n, p) = \tau + \left(q - \dfrac{q}{p}\right)\sigma; \\ T_{\mathrm{comp}}(n, p) = O(q). \end{cases} \tag{1}$$

2.2. Experimental Results for the **transpose** Primitive

Performance graphs for the **transpose** primitive execution times using SPLIT-C on a 32-processor CM-5, SP-2, and CS-2 and 8 processor Paragon are given in Fig. 7. These results also show the attained data bandwidth[3] per processor for the **transpose** primitive. For large enough data sets on the CM-5, we achieve an average bandwidth of 7.62 MB/s per processor, which is more than three-fourths of the maximum user-payload bandwidth per processor of 12 MB/s per processor [26]. This is consistent with the results achieved by other research teams that have achieved 6.4 MB/s per processor (Culler, [14]), and 7.72 MB/s per processor (Ranka, [38]) for similar data movements on the CM-5. Note that some of these cited results are for low-level implementations using message passing algorithms. For large enough data sets, the SP-2 achieves greater than 24.8 MB/s per processor for the **transpose** primitive, using a high performance switch hardware rated by the vendor as having a peak node to node bandwidth of 40 MB/s [25]. The Meiko CS-2 achieves greater than 10.7 MB/s per processor. Note that the CS-2 result is much less than the maximum attainable bandwidth of 50 MB/s per processor [29] because our SPLIT-C version has not been fully optimized to make use of the architecture's communications coprocessor. The 8 processor Paragon achieves greater than 88.6 MB/s per processor, with the maximum hardware bandwidth given by Intel as 175 MB/s per processor and application peak bandwidth as 135 MB/s per processor [27].

2.3. Analysis for the Broadcasting Primitive (**bcast**)

An efficient algorithm to broadcast q elements from a single processor to p processors (the **bcast** primitive) is based on the **transpose** communication primitive, where q is assumed to be larger than p. Processor 0 holds the q elements to be broadcast in the first column of matrix A. We compute the **transpose** of A, thus giving every processor q/p elements. Each processor then locally rearranges the data so that an additional **transpose** will result in each processor holding a copy of all the q elements in its column of A [21].

The following algorithm runs on processor i:

ALGORITHM 2. **bcast** *Communication Primitive*
Input:
$\{A\}$ is the $q \times p$ input array, with only the 0th column as valid data.
begin
1. **For** loop = 0 **to** $p - 1$ **do**:
   1.1 **Set** $r = (i + \text{loop})$ **mod** p;
   1.2 **Prefetch** $A^{\mathrm{T}}[i][(r * q/p) \cdots ((r + 1) * q/p) - 1] := A[r][(i * q/p) \cdots ((i + 1) * q/p) - 1]$;
2. **Sync**()
3. **For** loop = 0 **to** $p - 1$ **do**:
   3.1. **Set** $r = (i + \text{loop})$ **mod** p;
   3.2. **Prefetch** $A[i][(r * q/p) \cdots ((r + 1) * q/p) - 1] := A^{\mathrm{T}}[r][0 \cdots q/p - 1]$;
4. **Sync**()
end

Notice that at the end of Step 2, only the first q/p elements in each column are valid. Because of this, we specialize the **transpose** in Step 3 to prefetch only this first block from every other processor.

The analysis of the **bcast** primitive is simple. Since this algorithm just performs two transpositions, the complexities of the broadcasting algorithm are

$$\begin{cases} \mathrm{T}_{\mathrm{comm}}(n, p) = 2\left(\tau + \left(q - \dfrac{q}{p}\right)\sigma\right); \\ T_{\mathrm{comp}}(n, p) = O(q). \end{cases} \tag{2}$$

2.4. Experimental Results for the **bcast** Primitive

The performance graphs for broadcasting using the **transpose** primitive on a 32-processor CM-5, SP-2, and CS-2 and an 8-processor Paragon are given in Fig. 7. As expected, these graphs show that the **bcast** primitive takes roughly twice the time of the **transpose** communication primitive. In addition, these figures show the attained data bandwidth per processor for this broadcasting algorithm. As expected, we achieve approximately the same results as those of the **transpose** algorithm on both machines.

3. IMAGE (DATA) LAYOUT AND TEST IMAGES

A straightforward data layout is used in these algorithms for all platforms. The input image is an $n \times n$ matrix of integers. We assign tiles of the image as equally as possible among the processors. If p is an even power of 2, i.e.,

[3] Note that throughout this paper, the rate of "MB/s" will always represent 10^6 bytes per second.

$p = 2^d$, for even d, the processors will be arranged in a $\sqrt{p} \times \sqrt{p}$ logical grid. For future reference, we will denote the number of rows in this logical grid as v and the number of columns as w. For odd d, we assign the number of rows of the logical processor grid to be $v = 2^{\lfloor d/2 \rfloor}$, and the number of columns to be $w = 2^{\lceil d/2 \rceil}$. Each processor initially owns a tile of size $(n/v) \times (n/w)$. For future reference, we assign $q = n/v$ and $r = n/w$. We assume that the p processors are labeled consecutively from 0 to $p - 1$ and are assigned in row-major order to the logical processor grid just described.

Several test images have been used to test the correctness and the performance of the algorithms presented here. These test images, shown in Fig. 2, are generated at runtime, with images 1–4, 7, and 9 augmented to the needed image size, while images 5, 6, and 8 are scaled appropriately. Figure 1 is a 512×512 image with 256 gray-levels from the Second DARPA Image Understanding Benchmark [40]. The histogramming algorithm is assumed to be correct because $\sum_{i=0}^{k-1} H[i] = n^2$, and for regular patterns, it is easy to verify that each $H[i]/n^2$ equals the percentage of area that gray level i covers in the image. Verifying the connected components algorithm is more difficult. In addition to the DARPA Benchmark Image, we include the most widely used test patterns for binary images. A catalog of nine automatically generated scalable images is used, as shown in Fig. 2, and include horizontal, vertical, and forward- and back-slanting diagonal bars, a cross, a filled disc, concentric circles with thickness, four squares inset from the four corners, and a dual-spiral pattern, a "difficult" image [36].

4. HISTOGRAMMING

Histogramming is a useful image processing primitive. One application is histogram normalization (or equalization), a technique that flattens the histogram and, thus, improves the contrast of an image by "spreading out" colors which might be too clumped together for human visual distinction. There are also several image segmentation techniques based upon detection of peaks and valleys in the histogram.

Let k be the number of gray levels in the $n \times n$ input image X; without loss of generality, k is assumed to be a power of 2. Note that this implies that for $k \geq p$, the value of k/p is an integer ≥ 1. Our histogramming algorithm is quite simple. The first step consists of creating an array $H_i[0, ..., k - 1]$ on every processor i, such that each proces-

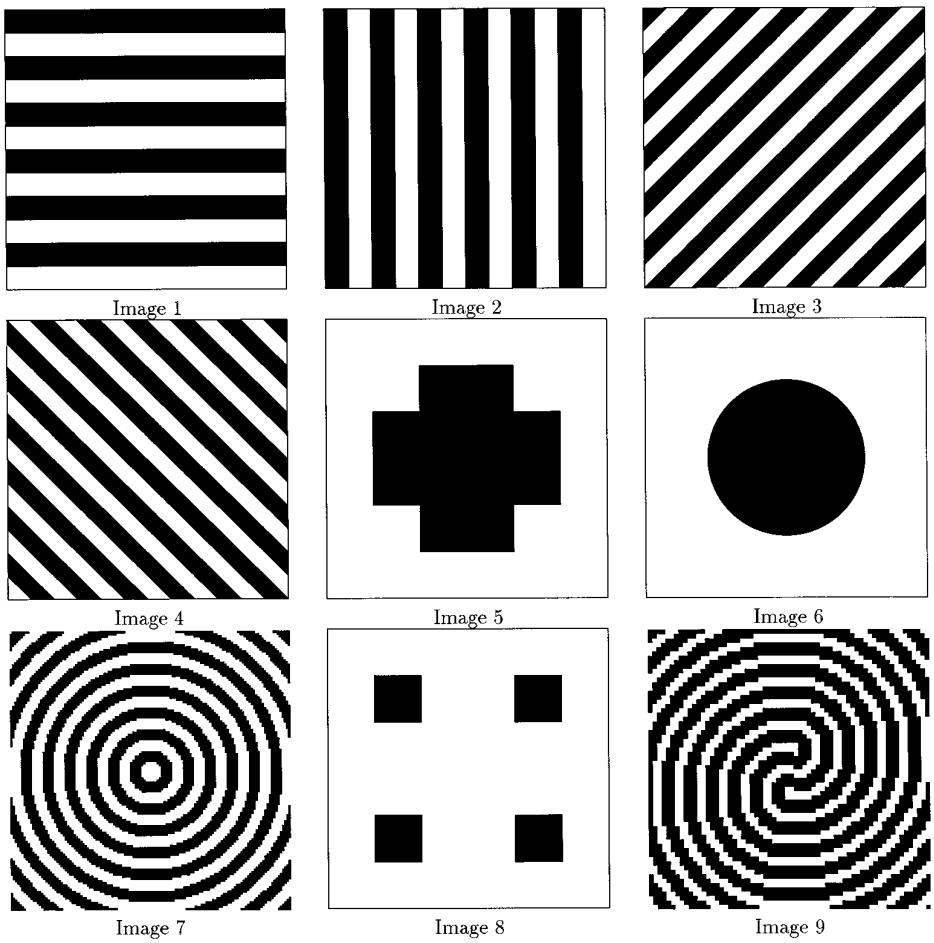


FIG. 2. Binary test images.

sor tallies the number of gray levels in its own $(n/v) \times (n/w)$ subimage into its array $H_i[\star]$. The purpose of the next step is to rearrange the data so that the tallies of each gray level reside on the same processor. If $k < p$ we use a truncated **transpose** primitive to put each row into a processor, P_i, $0 \le i \le k - 1$. If $k \ge p$ we **transpose** k/p rows of the local histograms into each processor, such that each processor, P_i, has all the intermediate sums needed to compute $H[i\,k/p]$ through $H[(i + 1)k/p - 1]$. The routing step is followed by local computations of the histogram, which can be done in $O(k)$ operations. Next, one processor, P_0, prefetches the results by doing a circular data movement, as described in Section 2, and outputs the k-bar histogram of the image.

The communication complexity can be estimated as follows. Two main communication steps are used in our algorithm. The first is a **transpose** primitive of the $k \times p$ histogram array and takes $T_{\text{comm}}(n, p) = \tau + (k - \max(k/p, 1))\sigma$. The second communication collects the histogram bars on a single processor and takes $T_{\text{comm}}(n, p) \le \tau + (k - \max(k/p, 1))\sigma$. Thus, the histogramming algorithm has the following complexities:

$$\begin{cases} T_{\text{comm}}(n, p) \le 2(\tau + k\sigma); \\ T_{\text{comp}}(n, p) = O\left(\dfrac{n^2}{p} + k\right). \end{cases} \tag{3}$$

4.1. Experimental Results for Histogramming

The above analysis indicates that, for fixed p and k, the communication complexity is independent of the problem size. Hence, as n increases, we expect the local computation to dominate.

The histogramming algorithm has been implemented on a CM-5 with $p = 16$, 32, 64, and 128 processors, and the algorithm's performance is plotted in Fig. 3 for 256 gray levels for images ranging from 32×32 to 4096×4096 pixels in size, and expanded in Fig. 9 for 128×128 to 1024×1024 images on 32 and 64 processors. Corresponding performance graphs are given for the IBM SP-2 in Fig. 13. Plots indicate quadratic performance as a function of n for fixed p, and scalability in terms of p. Hence, our theoretical analysis is supported.

Please refer to the plot in Fig. 3 for an illustration of the scalability of the histogramming algorithm's performance. Since computation dominates for large n, the algorithm runs as $O(n^2/p)$. We have plotted n^2 vs time for four configurations of the CM-5. The resulting plot shows the linear relationship between time and image size for each fixed p. Also, when the number of processors doubles, the running time approximately halves.

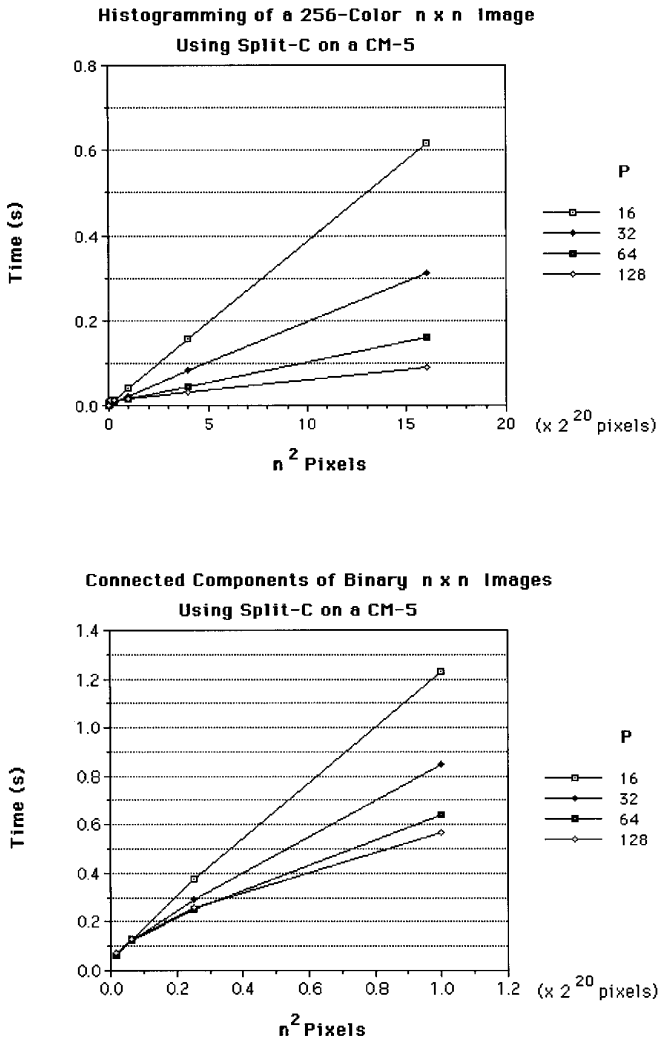


FIG. 3. Histogramming and connected components scalability on the CM-5.

5. CONNECTED COMPONENTS OF BINARY IMAGES

The high-level strategy of our connected components algorithm uses the well-known divide and conquer technique. Divide and conquer algorithms typically use a recursive strategy to split problems into smaller subproblems, and, given the solutions to these subproblems, merge the results into the final solution. It is common to have either an easy splitting algorithm and a more complicated merging, or vice versa, a hard splitting followed by an easy merging. In our parallel connected components algorithm, the splitting phase is trivial and implicit, while the merging process requires more work.

Each processor holds a unique tile of the image, and hence can find the initial connected components of its tile by using a standard sequential algorithm based upon breadth-first search. Next, the algorithm iterates $\log p$ times,[4] alternating between combining the tiles in **horizontal merges** of vertical borders and **vertical merges** of horizontal borders, with the number of horizontal merges equal to $\log w$ and the number of vertical merges equal to $\log v$, since $\log p = \log(v * w) = \log v + \log w$. Our

[4] Note that throughout this paper "$\log x$" will always be the logarithm of x to the base $b = 2$, i.e., $\log_2 x$.

algorithm uses novel techniques to perform the merges and to update the labels. We start by describing the initial sequential connected components algorithm.

5.1. Initialization and Sequential Connected Components

The initialization consists of entirely local operations on each processor. Pixels on each tile are examined in row-major order fashion. If a pixel is an unmarked, colored pixel, a breadth-first search procedure starting at that pixel labels all connected like-colored pixels within that tile with a globally unique label. When a pixel is visited in the labeling procedure, it becomes marked. During the initial row-major order search, for 8-connectivity, only the four pixels to the right, below-left, below, and below-right need to be examined for connectivity. For 4-connectivity, only the pixels to the right and below need to be examined. This sequential connected components algorithm runs in $O(|V| + |E|)$ where $|V|$ is the number of vertices, and $|E|$ is the number of edges searched. Since $|E| \leq 8\ |V|$, this algorithm runs in $O(|V|) = O(n^2/p)$ time. The result is an array of nonnegative integers corresponding to the unique label values of the connected components in the subimage.

The initial labeling of each pixel with local offset (i, j) in the processor with logical grid position (I, J) will be $(Iq + i)n + (Jr + j) + 1$. This labeling ensures that each processor will obtain unique labelings across the subimages after running the sequential connected components step, without having to do any communication among the processors. Thus, the initialization step runs in

$$T_{\text{comp}}(n, p) = O\left(\frac{n^2}{p}\right). \tag{4}$$

5.2. Merging Algorithm—Overview

Now we are ready to begin the merging phase. As mentioned above, we merge the p subimages into larger and larger image sections with consistent labelings. There will be $\log p$ iterations since we cut the number of uncombined subimages in half during each iteration. Unlike previous connected components algorithms, we use a technique which identifies processors as either **group managers** or **clients** during each phase. The group managers have the task of organizing the retrieval of boundary data, performing the merge, and creating the list of label changes. Once the group managers broadcast these changes to their respective clients, all processors must use the information to update their **tile hooks**, data structures which point to connected components on local tile borders. See Figure 5 for an illustration of the **tile hook** data structure in which three tile hooks contain the information needed to update the border pixels. The clients assist the group managers by participating in the coalescing of data during each merge phase. Finally, the complete relabeling is performed at the very end using information from the tile hooks.

Without loss of generality, we first perform a horizontal merge along every other vertical border, then a vertical merge along every other horizontal border, alternating orientation until we have merged all the tiles into one consistent labeling. We merge vertical borders exactly $\log w$ times, where w is the number of columns in the logical processor grid. Similarly, we merge horizontal borders exactly $\log v$ times, where v is the number of rows in the logical processor grid.

The merging algorithm for a horizontal merge is similar to that of a vertical merge. Most of the code is identical, except for substituting ''up'' for ''left'' and ''down'' for ''right.'' However, one nontrivial change relates to identifying during each iteration which processors will be **group managers** and which will be **clients**, concepts defined precisely in the following section.

5.3. Merging Algorithm—Group Managers' Task

We perform $\log p$ merge iterations, alternating between horizontal and vertical merge phases. Let t represent the current merge phase iteration, with $1 \leq t \leq \log p$. For each odd merge iteration t, $1 \leq t \leq \log p$, we will perform the $((t + 1)/2)$th horizontal merge phase, and similarly, for each even merge iteration t, $1 < t \leq \log p$, we will perform the $(t/2)$th vertical merge phase.

During each merge, a subset of the processors will act as **group managers**. These designated processors will prefetch the necessary border information along the column (or row) that each is located upon in the logical processor grid, set up an equivalent graph problem, solve the sequential connected components graph problem, note any changes in the labels, and store these changes $((\alpha_i, \beta_i)$ pairs) in a shared structure. Each **client** decides which processor is its current group manager and waits until the list of label changes is ready. Each retrieves the list, and finally, all processors make the necessary updates to a proper subset of their labels.

During odd merge iterations t, the horizontal merge phases, a processor is a **group manager** if it resides in the logical grid with both

- a row index whose binary representation ends with a 0 followed by $((t + 1)/2 - 1)$ 1's (or just ending in a 0 when $t = 1$), and
- a column index whose binary representation ends in $(t + 1)/2$ 0's.

Similarly, during the even merge iterations t, the vertical merge phases, a processor is a **group manager** if it resides in the logical grid with both

- a row index whose binary representation ends in $t/2$ 0's, and
- a column index whose binary representation ends with a 0 followed by $(t/2 - 1)$ 1's (or just ending in a 0 when $t = 2$).

An example data layout and merge is given in Fig. 4. This image of size 512×512 is distributed onto a 4×8 logical processor grid, with each tile being 128×64 pixels in

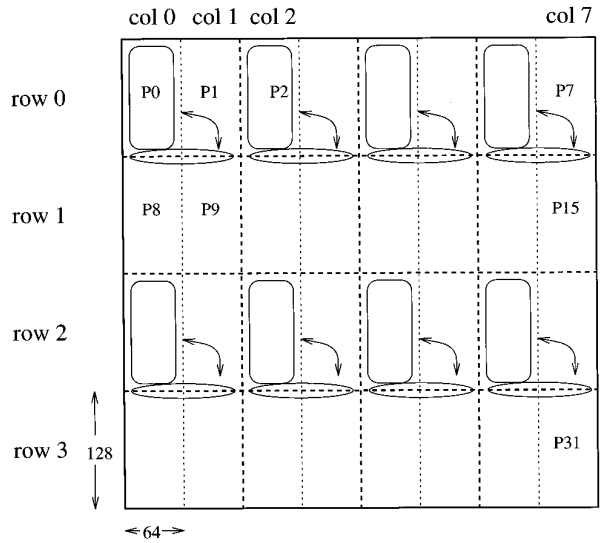


FIG. 4. Data layout of a 512×512 image on 32 processors—vertical merge ($t = 2$). Merge Phase 2: Circled processors are group managers. Dotted borders were merged in Phase 1. Circled borders will be merged in Phase 2.

size. This example shows the second merge step, a vertical merge, for $t = 2$. Group managers are, thus, any processor sitting in the logical processor grid with both last bits of the row and column indices' binary representation equal to "0." These group managers, along with their respective borders to be merged, are circled in this figure. Suppose now that $p \geq 128$, and we are at the $t = 7$th merge phase, which will be a horizontal merge. A processor in this case is a group manager if it is in a logical grid position whose row index's binary representation ends with 0111, and whose column index's binary representation ends with 0000.

For a horizontal merge, the group manager will prefetch the pixel colors and labels from the vertical borders to be merged, which spans across $2^{(t+1)/2}$ rows of processors. There are $2q$ $(= 2n/\sqrt{p})$ pixels per processor row in the border to be merged, meaning that $2q2^{(t+1)/2} - q$ pixels and an equal number of labels need to be prefetched from the clients, while q pixels and q labels are locally available. Thus, each prefetch in the horizontal merge can be done in $T_{\text{comm}}(n, p) \leq \tau + 4q2^{(t+1)/2}\sigma$ and $T_{\text{comp}}(n, p) = O((n/\sqrt{p})2^{(t+1)/2})$.

Similarly for a vertical merge, the group manager will prefetch the pixel colors and labels from the horizontal borders to be merged, which spans across $2^{t/2}$ columns of processors. There are $2r$ pixels per processor column in the border to be merged, meaning that $2r2^{t/2} - r$ pixels and an equal number of labels need to be prefetched from the clients for each iteration, while r pixels and r labels are locally available. Thus, each prefetch in the vertical merge can be done in $T_{\text{comm}}(n, p) \leq \tau + 4r2^{t/2}\sigma$ and $T_{\text{comp}}(n, p) = O((n/\sqrt{p})2^{t/2})$.

Note that the running time of this prefetching is improved by using a second processor, called a **shadow** manager, which is designated as the processor adjacent to the group manager, directly across the border being merged. Using this implementation, both the group and shadow manager prefetch only their side of the border, respectively, and sort each border by label. The reasons for this sorting will be described below. The group manager then prefetches the sorted results from the shadow manager and continues on with the algorithm. From this point on, the shadow manager reverts back to being a client of this group manager.

The total complexities for prefetching summed up over the $\log w$ horizontal merges and the $\log v$ vertical merges are

$$\begin{cases} T_{\text{comm}}(n,p) \leq \displaystyle\sum_{\rho=1}^{\log v} (\tau + 4q2^{\rho}\sigma) + \sum_{\kappa=1}^{\log w} (\tau + 4r2^{\kappa}\sigma) \\ \qquad\qquad \leq \tau \log p + 8n\sigma; \\ T_{\text{comp}}(n,p) = O\left(\displaystyle\sum_{\rho=1}^{\log v} (4q2^{\rho}) + \sum_{\kappa=1}^{\log w} (4r2^{\kappa})\right) \\ \qquad\qquad = O(n). \end{cases} \tag{5}$$

The merging problem is converted into finding the connected components of a graph represented by the border pixels. We use an adjacency list representation for the graph, and add vertices to the graph representing colored pixels. Two types of edges are added to the graph. First, pixels are scanned down the left (or upper) border, and edges are strung linearly down the list between pixels containing the same connected component label. The same is done for pixels on the right (or lower) border. The second step adds edges between pixels of the left (upper) and right (lower) border which are both like-colored pixels and adjacent to each other. We scan down the left column (upper row) elements, and if we are at a colored pixel, we check the pixels in the right column (lower row) adjacent to it. In order to add the first type of edges, the pixels are sorted according to their label for both the left (upper) and right (lower) border by using radix sort.[5] Note the discussion above regarding the use of a shadow manager. A secondary processor is used to prefetch and sort the border elements on the opposite side of the border from the group manager, and the results are then sent to the group manager. This sort takes $T_{\text{comp}}(n, p) = O(|V|)$ steps for a border of $|V|$ nodes.[6] The maximum number of edges attached to each vertex in this graph is at most five; two edges in its own column to pixels above and below of the same label plus the three adjacent pixels in the right col-

[5] Note that whenever radix sort is mentioned in this paper, the actual coding uses the standard UNIX quicker-sort function for smaller sorts, and radix sort for larger sorts, using whichever sorting method is fastest for the given input size.

[6] Our radix sort uses four passes; each pass will sort on one byte of the 32-bit key by using 256 buckets.

umn. Thus, inserting an edge into the adjacency list takes at most five steps, and we add at most $5|V|$ edges. For each horizontal merge step, the number of vertices $|V| \le 2q2^{(t+1)/2}$, and for each vertical merge step, $|V| \le 2r2^{t/2}$. Thus, the construction of this graph summed over all the iterations of the connected components algorithm takes $T_{\text{comp}}(n, p) = O\left(\sum_{\rho=1}^{\log v} (2q2^{\rho}) + \sum_{\kappa=1}^{\log w} (2r2^{\kappa})\right) = O(n)$.

A sequential breadth-first search based connected components algorithm finds the connected components of this graph. It runs in $O(|V| + |E|)$ steps, with $|V|, |E| = O(q2^{(t+1)/2})$ for horizontal merges and $O(r2^{t/2})$ for vertical merges. The pixels in this graph are then scanned again, and any changes in the labeling (α changing to label β) are eventually stored in a sorted array of all unique changes (α_i, β_i). The following algorithm describes the procedure for creating the sorted array of label changes from the original arrays.

- *Step* (1). Copy all label pairs, (α, β), where label α has changed to label β, into a contiguous array.
- *Step* (2). Radix sort this array, using α as the sorting index.
- *Step* (3). Scan down the sorted array, copying all unique (α, β) pairs into a new array.

There are at most $2|V|$ changes, so Steps (1), (2), and (3) take $O(|V|)$ time. Thus, the creation of the sorted array of label changes takes $O(|V|)$ time. Summing over the $\log p$ steps, this is equivalent to $T_{\text{comp}}(n, p) = O\left(\sum_{\rho=1}^{\log v} (2q2^{\rho}) + \sum_{\kappa=1}^{\log w} (2r2^{\kappa})\right) = O(n)$.

The array structure is actually two contiguous arrays, one holding the obsolete labels (α's) and the other holding the corresponding new labels (β's). The size of these arrays of α's and β's is also placed into a shared memory location.

Now each processor hits a barrier and waits until all processors have completed their tasks. After the barrier, a group manager will update its pixels' labels in $O(n^2/p)$ by the following procedure.

After the initial tile labelings, but before the merging iterations, each processor creates a sorted array of **hooks** to each local component containing a border pixel of the tile. There will be exactly one hook for each of these components, including the initial label of that component and the offset address in the tile of any pixel in that component. This is done as follows:

- *Step* (1). For each colored pixel on the tile border with offset position (i, j)

  (1.1). Place $(label[(i, j)], (i, j))$ at the next position of an array.
- *Step* (2). Radix sort this array, using *label* as the sorting index.
- *Step* (3). Scan down the sorted array, copying all unique $(label[(i, j)], (i, j))$ pairs into a new array.

This initialization takes computational complexity of $O(n/\sqrt{p})$ for each of Steps (1), (2), and (3), yielding a total of $T_{\text{comp}}(n, p) = O(n/\sqrt{p})$.

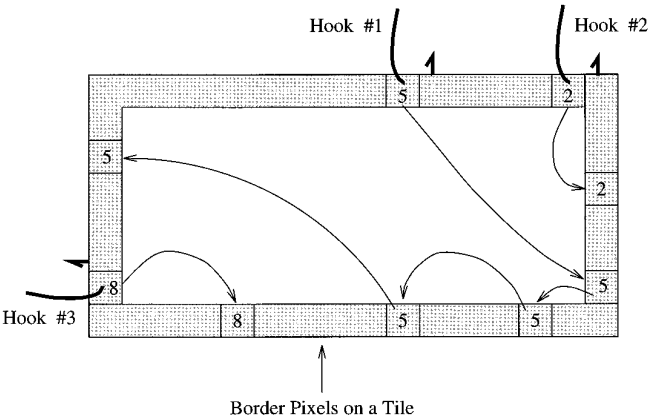


FIG. 5. An example of tile hooks.

At the conclusion of each of the $\log p$ merging steps, only the labels of pixels on the border of each tile are updated. There is no need to relabel interior pixels since they are not used in the merging stage. Only boundary pixels need their labels updated. The procedure is simple; for each colored pixel on the boundary, we perform a binary search of the list of label changes in $T_{\text{comp}}(n, p) = O((n/\sqrt{p}) \log |V|)$ per step. The total computational complexity over the $\log p$ merging iterations is then $O\left(\sum_{\rho=1}^{\log v} [(n/\sqrt{p}) \log(2q2^{\rho})] + \sum_{\kappa=1}^{\log w} [(n/\sqrt{p}) \log(2r2^{\kappa})]\right) = O((n/\sqrt{p}) \log n \log p)$.

At the end of the last merging step, each processor must update its interior pixel labels. Each hook described above is compared with the current label at the hook's offset position index. If the hook's label $label[i]$ is different from the current label at position i, the processor runs a breadth-first search relabeling technique beginning at pixel i, relabeling all the connected pixels' labels to the new label. Since there is only one hook per tile component on the border, the breadth-first search relabeling procedure takes $O(n^2/p)$ time.

The total complexity associated with updating the labels

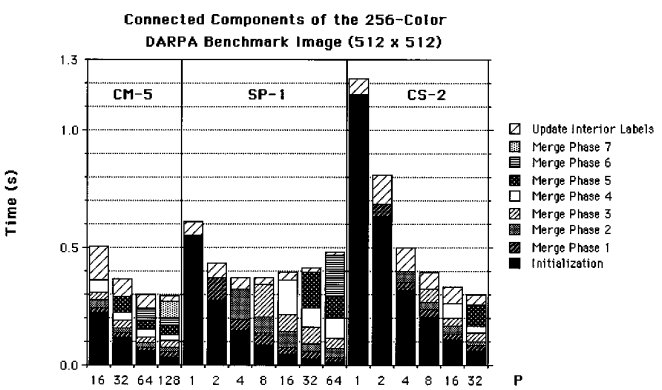


FIG. 6. Connected components of the 512×512 DARPA Benchmark image on various machines.

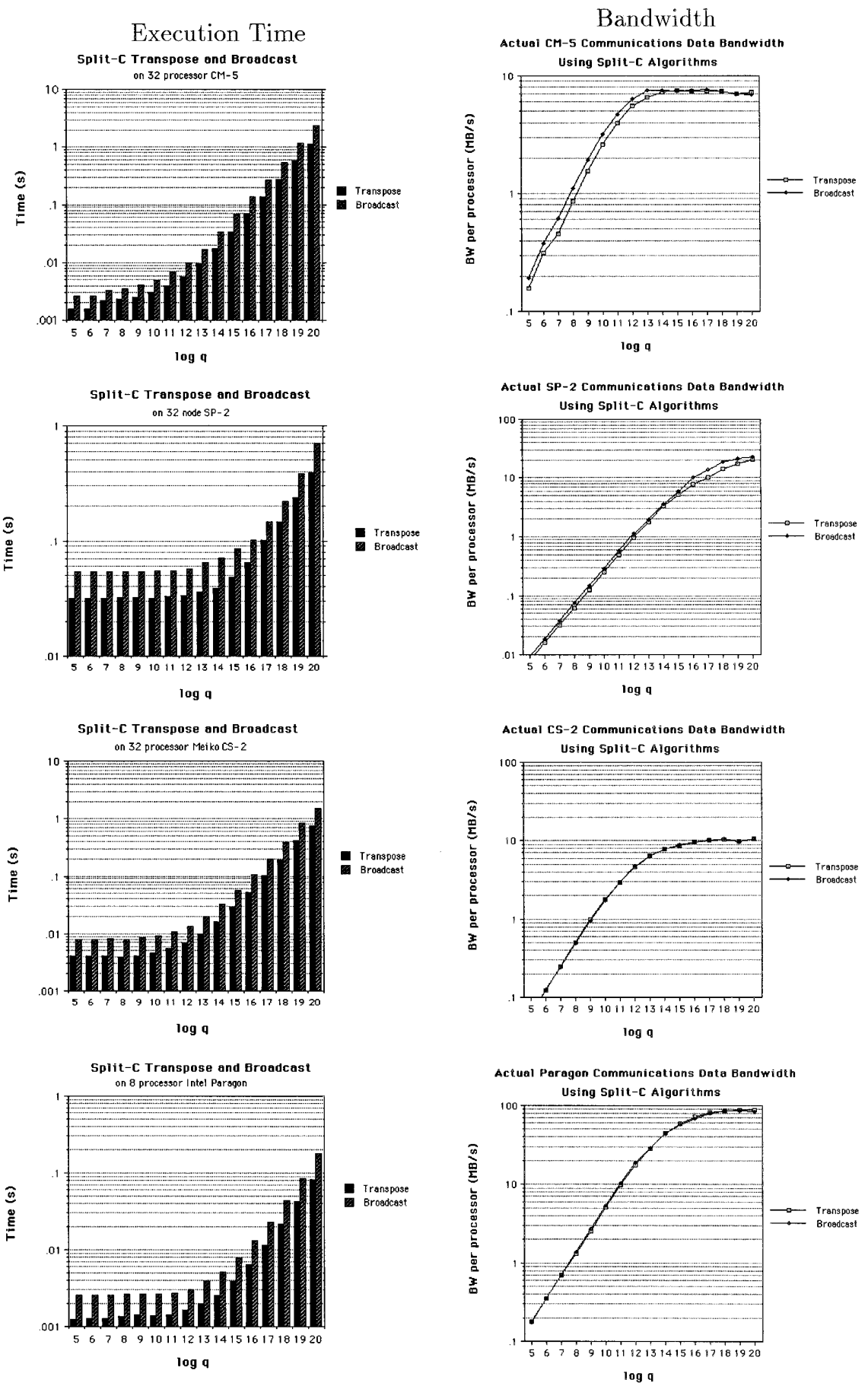


FIG. 7. Transpose and broadcasting performance graphs.

of each tile is $T_{\text{comp}}(n, p) = O((n/\sqrt{p})\log(n/\sqrt{p}) + (n/\sqrt{p})\log n \log p + n^2/p) = O(n^2/p)$, assuming $p \le n$ for large enough n. For $n \ge 128$, $p \le n/8$ is sufficient.

After each merging step label update, a manager hits another barrier, waiting for the end of this iteration. In summary, the group managers' routine has the following complexities:

$$\begin{cases} T_{\text{comm}}(n,p) \le \tau \log p + 8n\sigma; \\ T_{\text{comp}}(n,p) = O\left(\dfrac{n^2}{p} + n\right). \end{cases} \tag{6}$$

5.4. Merging Algorithm—Clients' Task

The client processors are any processors not selected to run the group manager tasks during the current iteration. Each client calculates the logical processor grid address of the manager in charge of its border to be merged and waits for the first barrier. After this barrier, each client prefetches the size (*chSize*) of the list of change pairs from its manager in $T_{\text{comm}}(n, p) \le \tau + 2^t\sigma$, where $(t + 1)/2$ and $t/2$ are the numbers of vertical and horizontal merges, respectively, performed inclusively during the tth merge phase.

Next, each client prefetches a block of *chSize* (α, β) change pairs from its manager. This is done in $T_{\text{comm}}(n, p) \le \tau + 2(2^t)2q2^{(t+1)/2}\sigma$ for horizontal merges, and $T_{\text{comm}}(n, p) \le \tau + 2(2^t)2r2^{t/2}\sigma$ for vertical merges, since there are at most $2q2^{(t+1)/2}$ (or $2r2^{t/2}$) changes, and exactly $(2^t - 1)$ processors requesting these change pairs from each group manager. The client processors use the same procedure described in the previous section for relabeling their border pixels at the end of each merge iteration, and the interior pixels after the final merge. After each pixel label update, each client hits another barrier and waits for the end of this iteration. Over the log p

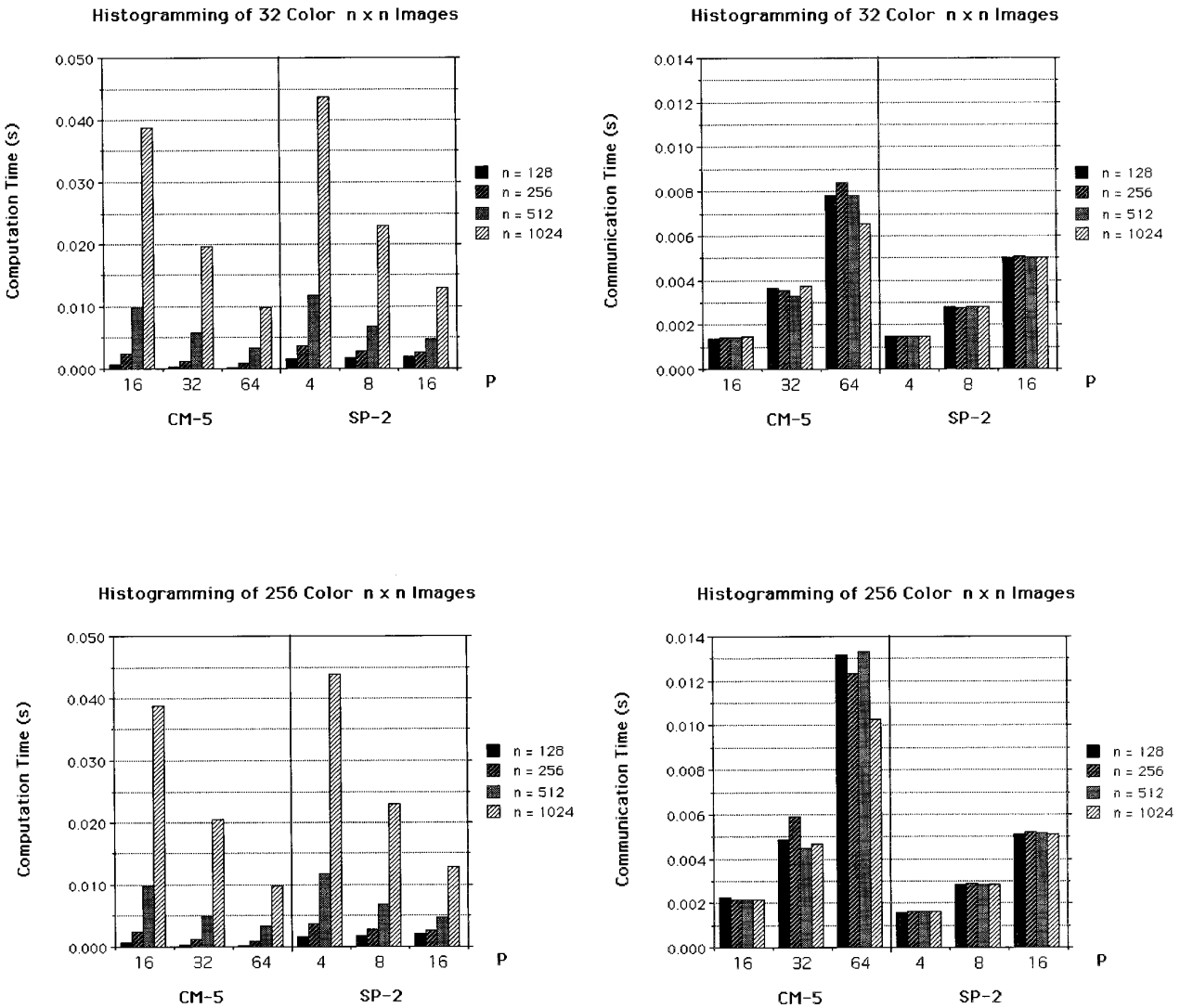


FIG. 8. Histogramming performance.

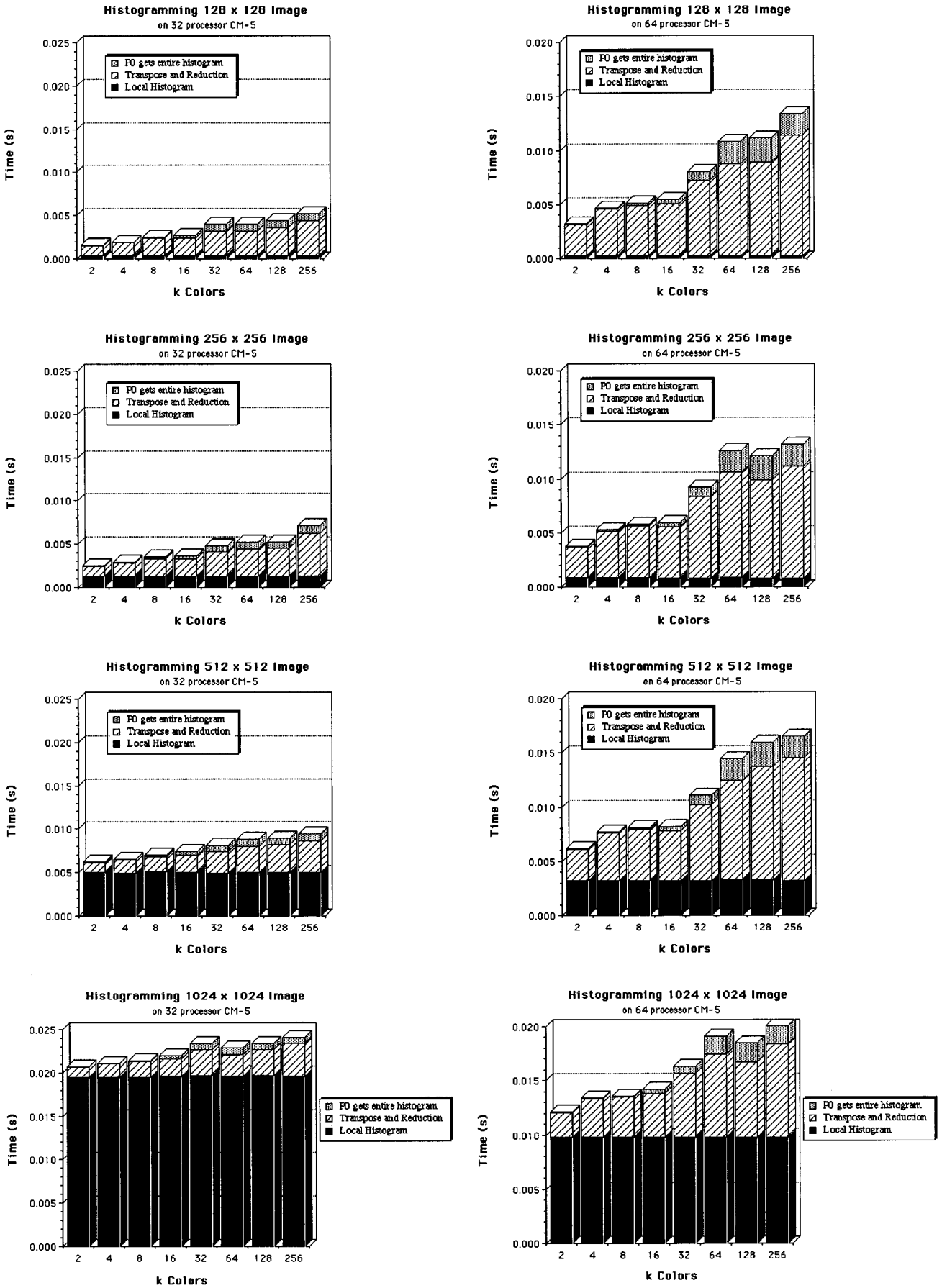


FIG. 9. Histogramming algorithm performance graph on the CM-5, with $p = 32$ and $p = 64$ in the left and right columns, respectively. Rows 1, 2, 3, and 4 correspond to images of size 128×128, 256×256, 512×512, and 1024×1024, respectively.

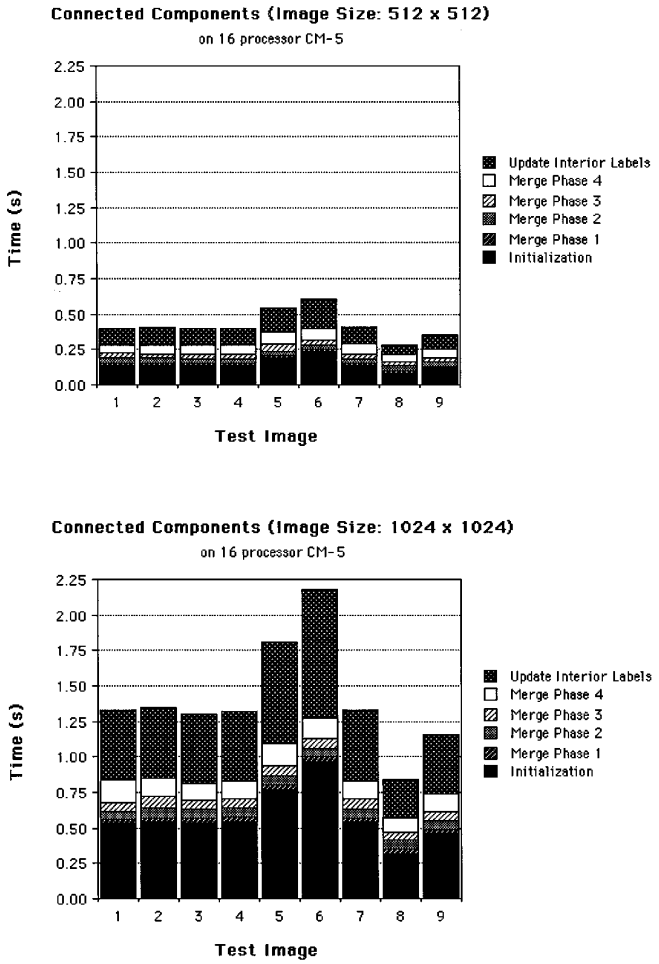


FIG. 10. Connected components algorithm performance graph on the CM-5 ($p = 16$).

iterations, the clients' routine has the following complexities:

$$\begin{cases} T_{\text{comm}}(n,p) \le \displaystyle\sum_{l=1}^{\log p} [\tau + 2^l\sigma] + \sum_{\rho=1}^{\log v} [\tau + 2(2^{2\rho})2q2^{\rho}\sigma] \\ \qquad\qquad + \displaystyle\sum_{\kappa=1}^{\log w} [\tau + 2(2^{2\kappa-1})2r2^{\kappa}\sigma] \\ \qquad\qquad \le (2\log p)\tau + (14np + 2p)\sigma; \\ T_{\text{comp}}(n,p) = O\left(\displaystyle\sum_{l=1}^{\log p} 2^l + \sum_{\rho=1}^{\log v} [2(2^{2\rho})2q2^{\rho}] \right. \\ \qquad\qquad \left. + \displaystyle\sum_{\kappa=1}^{\log w} [2(2^{2\kappa-1})2r2^{\kappa}]\right) + O\left(\frac{n^2}{p}\right) \\ \qquad\qquad = O\left(\frac{n^2}{p} + np\right). \end{cases} \tag{7}$$

Clearly, for large p, this is not an optimal procedure for distributing the list of change pairs from a group manager to the respective clients. If a manager has $f(i) - 1$ clients at the end of iteration i, $0 \le i < \log p$, instead of sending the entire list of $c(i)$ change pairs to $f(i) - 1$ processors, a distribution algorithm based on the **transpose** communication primitive can be used. Using this algorithm, a manager will send blocks of size $c(i)/f(i)$ to each of $f(i)$ processors during the first phase. Each of the $f(i)$ processors repeat this operation by concurrently sending its block to the other processors, in a circular fashion. The complexities for this are

$$\begin{cases} T_{\text{comm}}(n,p) \le 2\left(\tau + \left(c(i) - \dfrac{c(i)}{f(i)}\right)\sigma\right) \\ T_{\text{comp}}(n,p) = O\left(\dfrac{c(i)}{f(i)}\right). \end{cases} \tag{8}$$

The clients' complexities are thus improved to

$$\begin{cases} T_{\text{comm}}(n,p) \le \displaystyle\sum_{l=1}^{\log p} [\tau + 2^l\sigma] + \sum_{\rho=1}^{\log v} [2(\tau + 2q2^{\rho}\sigma)] \\ \qquad\qquad + \displaystyle\sum_{\kappa=1}^{\log w} [2(\tau + 2r2^{\kappa}\sigma)] \\ \qquad\qquad \le (3\log p)\tau + (16n + 2p)\sigma; \\ T_{\text{comp}}(n,p) = O\left(\displaystyle\sum_{l=1}^{\log p} 2^l + \sum_{\rho=1}^{\log v} \left[\frac{2q2^{\rho}}{2^{2\rho}}\right] \right. \\ \qquad\qquad \left. + \displaystyle\sum_{\kappa=1}^{\log w} \left[\frac{2r2^{\kappa}}{2^{2\kappa-1}}\right]\right) + O\left(\frac{n^2}{p}\right) \\ \qquad\qquad = O\left(\frac{n^2}{p}\right). \end{cases} \tag{9}$$

5.5. Parallel Complexity for Connected Components

Thus, for $p \le n$, the total complexities for the parallel connected components algorithm are

$$\begin{cases} T_{\text{comm}}(n,p) \le (4\log p)\tau + (24n + 2p)\sigma \\ \qquad\qquad = (4\log p)\tau + O\left(\dfrac{n^2}{p}\right)\sigma; \\ T_{\text{comp}}(n,p) = O\left(\dfrac{n^2}{p}\right). \end{cases} \tag{10}$$

Clearly, the computational complexity is the best possible asymptotically. As for the communication complexity, it seems that intuitively a latency factor τ has to be incurred during each merge operation, and hence the factor $(\log p)\tau$.

5.6. Experimental Results for Connected Components

Our theoretical analysis indicates that our connected components algorithm is scalable whenever $p \le n/c$, where

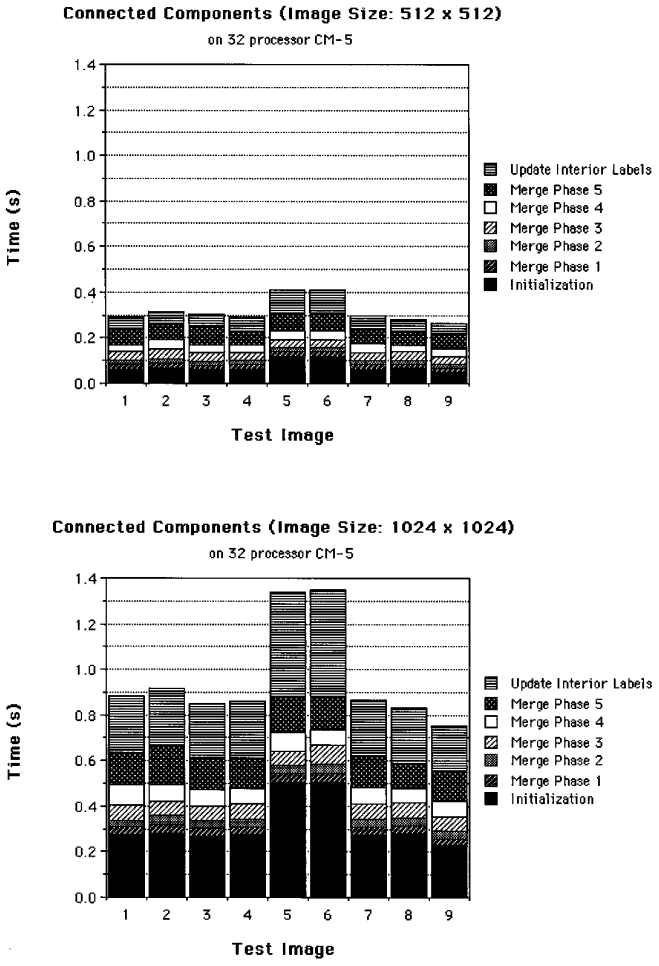


FIG. 11. Connected components algorithm performance graph on the CM-5 ($p = 32$).

c is approximately 26 from the first expression in (10). We have implemented our algorithm in SPLIT-C; the resulting performance on the CM-5 is plotted for images ranging from 128×128 to 1024×1024 pixels in size in Figs. 10–12 for $p = 16$, 32, and 64 processors. Figure 3 presents the summary on the performance of our connected components algorithm on the CM-5 and clearly shows the scalability of our algorithm. Comparable results for execution on the IBM SP-2 are given in Fig. 14. See [4, 5] for additional performance results.

6. CONNECTED COMPONENTS OF GRAY SCALE IMAGES

An $n \times n$ image with k gray levels, $(0, ..., k-1)$, similarly can have its connected components labeled. A 0-pixel is assumed to be background, while each component is the set of like-colored connected pixels. Our algorithm for gray scale connected components of images is based upon the binary image algorithm in the previous section. Again, there will be three phases, an initial labeling, a merge of subimages, and a final updating of interior labels. The details are very similar to those of the binary case and can be found in [4].

Results for the 256-gray level DARPA Image Understanding Benchmark image of size 512×512 pixels, shown in Fig. 1, are given in Fig. 6 for $p = 16$ to 128 processors on the CM-5, and for a wide range of configurations on the SP-1 and Meiko CS-2 parallel machines.

7. IMPLEMENTATION NOTES

Note that the performance graphs for the CM-5, Figs. 3, 6, and 7–12, are for SPLIT-C (version 1.2) programs linked with the CM-5 CMMD Message Passing Libraries (version 3.2). Figure 6 uses the message passing library MPL on the IBM SP-1, and Figs. 7, 13, and 14 are for the IBM SP-2 with thin nodes and also MPL. Figures 6 and 7 are run on a Meiko CS-2 with SPLIT-C linked with the Elan Widgets message passing library. Note that our port of SPLIT-C to the CS-2 results in less than optimal performance because this SPLIT-C installation has not been fully optimized to make use of Elan, the low level communica-

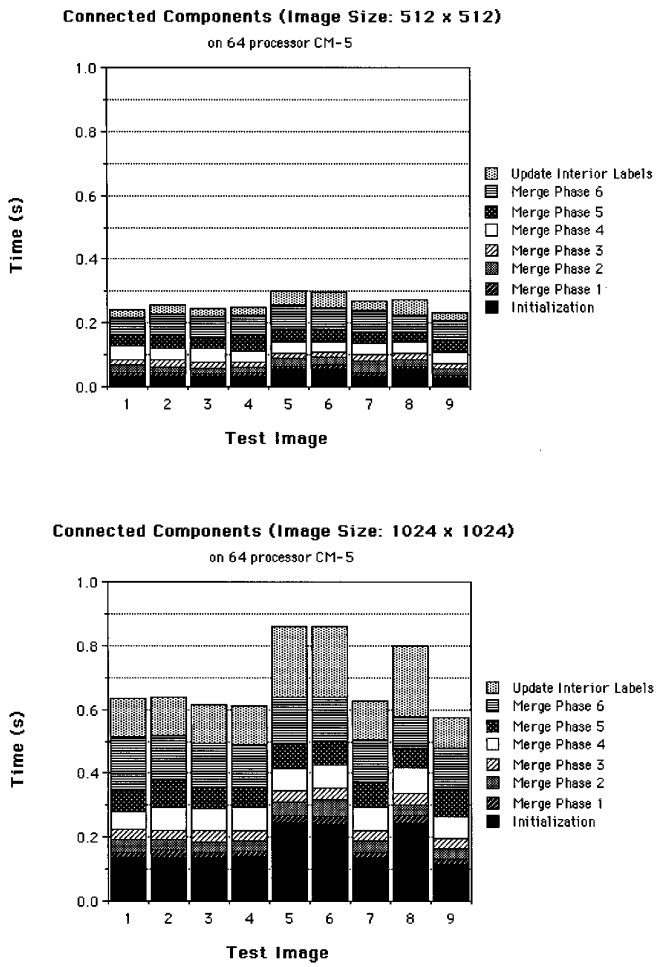
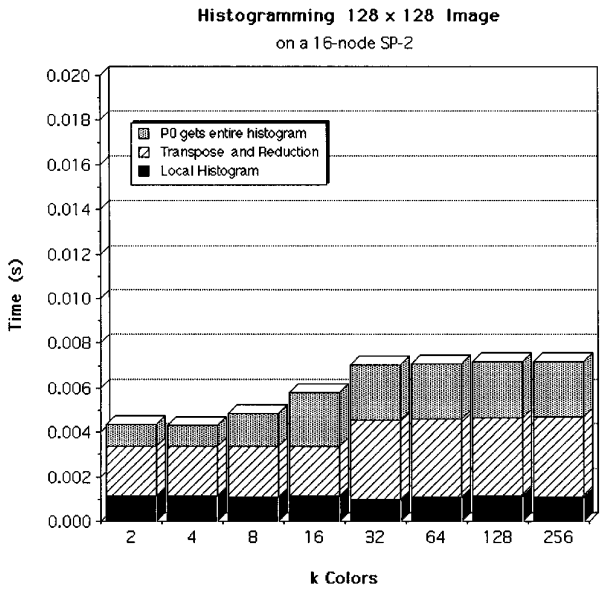
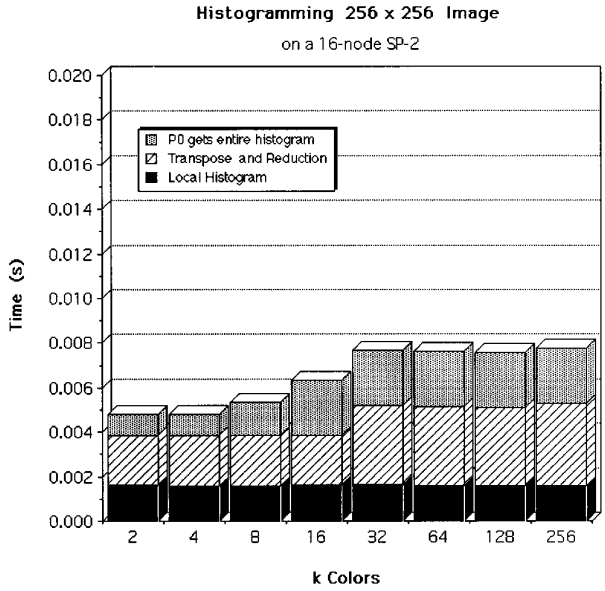


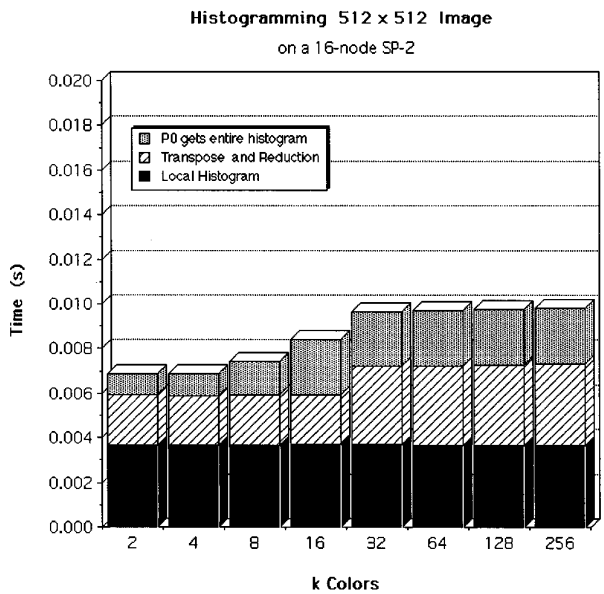
FIG. 12. Connected components algorithm performance graph on the CM-5 ($p = 64$).



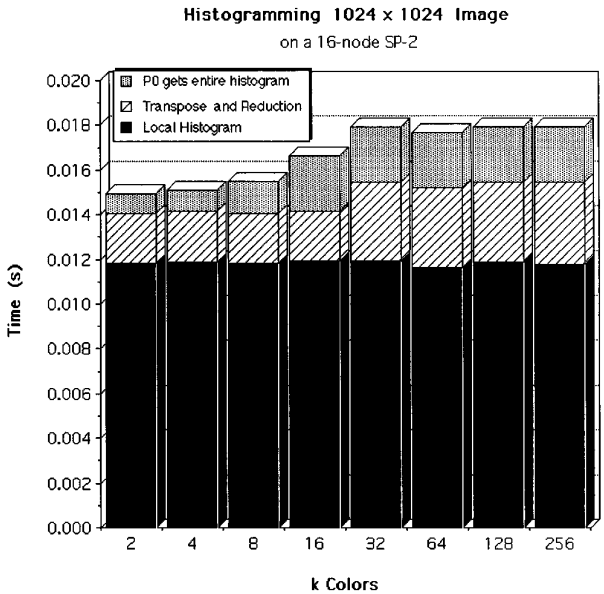
128×128 Images



256×256 Images



512×512 Images



1024×1024 Images

FIG. 13. Histogramming algorithm performance graph on the SP-2 ($p = 16$).

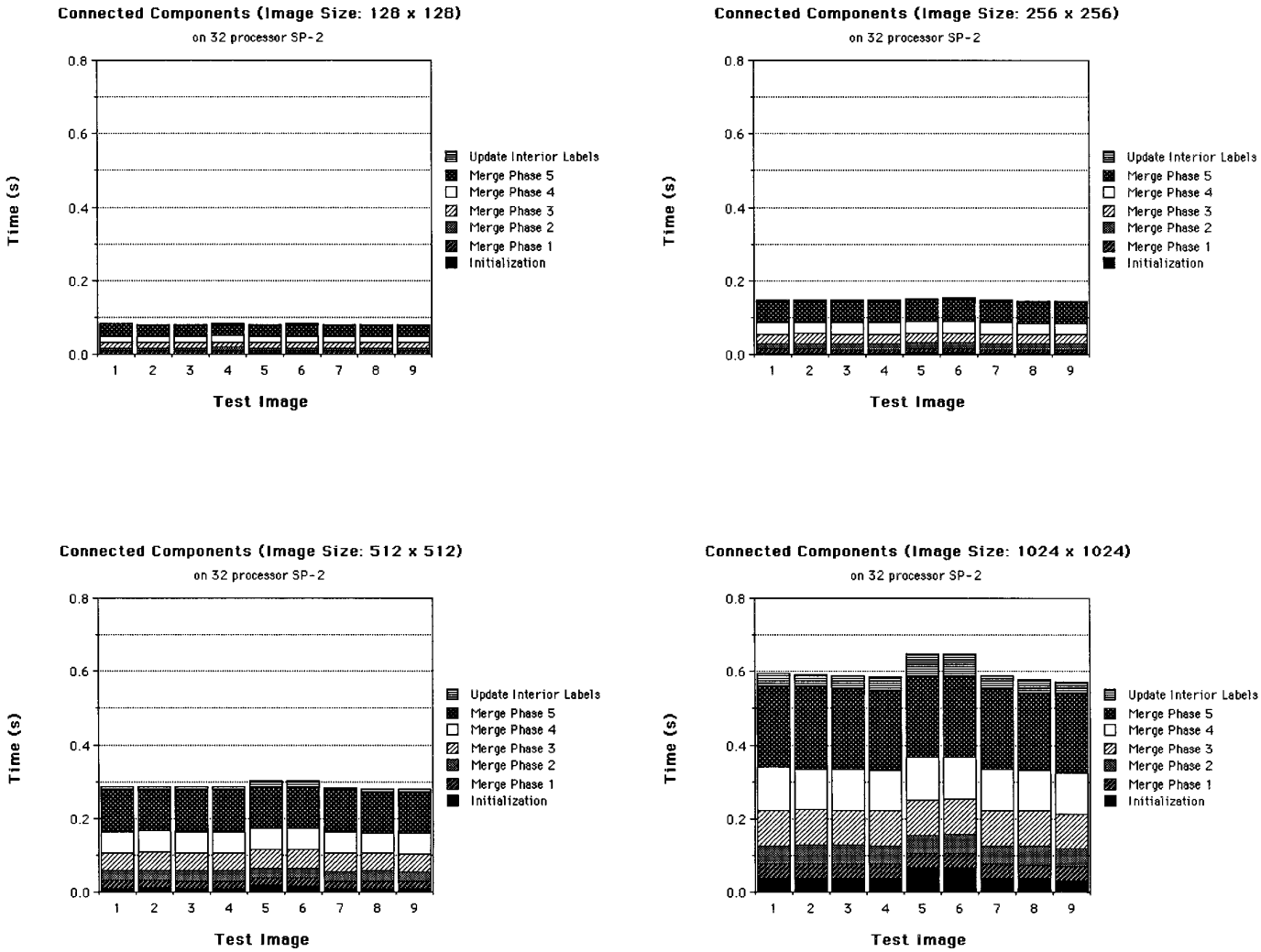


FIG. 14. Connected components algorithm performance graph on the SP-2 ($p = 32$).

tions library. We expect results using an optimized platform shortly. Results for the 8-processor Intel Paragon in Fig. 7 use the PAM message passing libraries, the Paragon Active Messages platform from UC Berkeley.

ACKNOWLEDGMENTS

We recognize Charles Weems at the University of Massachusetts for providing the DARPA test image suite. We thank the CASTLE/SPLIT-C group at the University of California, Berkeley, especially for the help and encouragement from David Culler, Arvind Krishnamurthy, and Lok Tin Liu, and the use of UC Berkeley's 64-processor CM-5 and 8-processor Paragon for testing purposes. Computational support on Berkeley's TMC CM-5 and Intel Paragon was provided by NSF Infrastructure Grant CDA-8722788. We also thank the Numerical Aerodynamic Simulation Systems Division of NASA's Ames Research Center for use of their 160-node IBM SP-2-WN. We also acknowledge Argonne National Labs for use of their 128-node IBM SP-1. Also, Klaus Schauser, Oscar Ibarra, Chris Scheiman, and David Probert of UC Santa Barbara provided help and access to the UCSB 64-node Meiko CS-2. The Meiko CS-2 Computing Facility was acquired through NSF CISE Infrastructure Grant CDA-9218202, with support from the College of Engineering and the UCSB Office of Research, for research in parallel computing. Arvind Krishnamurthy provided additional help with his port of SPLIT-C to the Cray Research T3D. The Jet Propulsion Lab/Caltech 256-node Cray T3D Supercomputer used in this investigation was provided by funding from the NASA Offices of Mission to Planet Earth, Aeronautics, and Space Science. We also acknowledge William Carlson and Jesse Draper from the Center for Computing Science for writing the parallel compiler AC (version 2.6) on which the T3D compiler port has been based. We acknowledge the use of the UMIACS 16-node IBM SP-2-TN2, which was provided by an IBM Shared University Research award and by NSF Academic Research Infrastructure Grant CDA9401151.

Please see `http://www.umiacs.umd.edu/research/EXPAR` for additional performance information. In addition, all the code used in this paper is freely available for interested parties from our anonymous ftp site, `ftp://ftp.umiacs.umd.edu/pub/dbader`.

REFERENCES

1. Alexandrov, A., Ionescu, M., Schauser, K., and Scheiman, C. LogGP: Incorporating long messages into the LogP Model—One step closer towards a realistic model for parallel computation. *7th Annual ACM Symposium on Parallel Algorithms and Architectures,* Santa Barbara, CA, 1995, pp. 95–105.
2. Alnuweiri, H., and Prasanna, V. Parallel architectures and algorithms for image component labeling. *IEEE Trans. Pattern Anal. Mach. Intell.* **14** (1992), 1014–1034.

3. Apostolakis, J., Coddington, P., and Marinari, E. New SIMD algorithms for cluster labeling on parallel computers, *Int. J. Mod. Phys. C* **4** (1993), 749.
4. Bader, D. A., and JáJá, J. Parallel algorithms for image histogramming and connected components with an experimental study. Technical Report CS-TR-3384 and UMIACS-TR-94-133, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, December 1994. *J. Parallel Distrib. Comput.,* to appear.
5. Bader, D. A., and JáJá, J. Parallel algorithms for image histogramming and connected components with an experimental study. *Fifth ACM SIGPLAN Symposium of Principles and Practice of Parallel Programming,* Santa Barbara, CA, 1995, pp. 123–133.
6. Bader, D. A., JáJá, J., Harwood, D., and Davis, L. S. Parallel algorithms for image enhancement and segmentation by region growing with an experimental study. Technical Report CS-TR-3449 and UMIACS-TR-95-44, Institute for Advanced Computer Studies (UMIACS), University of Maryland, College Park, MD, 1995. Presented at the 10th International Parallel Processing Symposium, Honolulu, 1996, pp. 414–423.
7. Baillie, C. F., and Coddington, P. D. Cluster identification algorithms for spin models—Sequential and parallel. *Concurrency: Practice Experience* **3,** 2 (1991), 129–144.
8. Brower, R. C., Tamayo, P., and York, B. A parallel multigrid algorithm for percolation clusters. *J. Statist. Phys.* **63** (1991), 73.
9. Chin, F. Y., Lam, J., and Chen, I-N. Efficient parallel algorithms for some graph problems. *Comm. ACM,* **25,** 9 (1982), 659–665.
10. Choudhary, A., and Thakur, R. Evaluation of connected component labeling algorithms on shared and distributed memory multiprocessors. *Proceedings of the 6th International Parallel Processing Symposium,* 1992, pp. 362–365.
11. Choudhary, A., and Thakur, R. Connected component labeling on coarse grain parallel computers: An experimental study. *J. Parallel Distrib. Comput.* **20,** 1 (Jan. 1994), 78–83.
12. Copty, N., Ranka, S., Fox, G., and Shankar, R. V. A data parallel algorithm for solving the region growing problem on the Connection Machine. *J. Parallel Distrib. Comput.* **21,** 1 (Apr. 1994), 160–168.
13. Culler, D. E., Dusseau, A., Goldstein, S. C., Krishnamurthy, A., Lumetta, S., Luna, S., von Eicken, T., and Yelick, K. *Introduction to Split-C,* version 1.0 edition. Computer Science Division—EECS, University of California, Berkeley, CA, 1994.
14. Culler, D. E., Karp, R. M., Patterson, D. A., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. LogP: Towards a realistic model of parallel computation. *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* 1993.
15. Dillencourt, M. B., Samet, H., and Tamminen, M. Connected component labeling of binary images. Technical Report CS-TR-2303, Computer Science Department, University of Maryland, 1989.
16. Grinberg, J., Nudd, G. R., and Etchells, R. D. A cellular VLSI architecture. *IEEE Comput.* **17,** 1 (1984), 69–81.
17. Han, Y., and Wagner, R. A. An efficient and fast parallel-connected component algorithm. *J. Assoc. Comput. Mach.* **37,** 3 (1990), 626–642.
18. Hirschberg, D. S., Chandra, A. K., and Sarwate, D. V. Computing connected components on parallel computers. *Comm. ACM,* **22,** 8 (1979), 461–464.
19. Ibrahim, H. A., Kender, J. R., and Shaw, D. E. Low-level image analysis tasks on fine-grained tree-structured SIMD machines. *J. Parallel Distrib. Comput.* **4** (1987), 546–574.
20. JáJá, J. *An Introduction to Parallel Algorithms.* Addison–Wesley, New York, 1992.
21. JáJá, J., and Ryu, K. W. The block distributed memory model. Technical Report CS-TR-3207, Computer Science Department, University of Maryland, College Park, January 1994. *IEEE Trans. Parallel Distrib. Systems,* to appear.
22. JáJá, J. F., and Ryu, K. W. The block distributed memory model for shared memory multiprocessors. *Proceedings of the 8th International Parallel Processing Symposium,* Cancún, Mexico, 1994, pp. 752–756. [Extended abstract]
23. Jesshope, C. R. Parallel computers—Architectures and programming. In Vaughan, R. A. (Ed.), *Pattern Recognition and Image Processing in Physics.* Scottish Universities Summer School in Physics, New York, 1990, pp. 205–234.
24. Kanade, T., and Webb, J. A. Parallel vision algorithm design and implementation 1988 end of year report. Technical Report CMU-RI-TR-89-23, The Robotics Institute, Carnegie Mellon University, 1989.
25. Kuzela, J. M. *IBM POWERparallel System—SP2 Performance Measurements.* Power Parallel Systems, IBM, 1994.
26. Leiserson, C. E., Abuhamdeh, Z. S., Douglas, D. C., Feynman, C. R., Ganmukhi, M. N., Hill, J. V., Hillis, W. D., Kuszmaul, B. C., St. Pierre, M. A., Wells, D. S., Wong, M. C., Yang, S.-W., and Zak, R. The network architecture of the Connection Machine CM-5. 1992. [Extended abstract]
27. Liu, L. T. Personal communication. 1994.
28. Marks, P. Low-level vision using an array processor. *Comput. Graphics Image Process.* **14** (1980), 281–292.
29. Meiko. *Computing Surface—Communications Network Overview,* Manual 84-cb041 edition. Meiko World Inc., Concord, MA, 1993.
30. Nudd, G. R., Atherton, T. J., Francis, N. D., Howarth, R. M., Kerbyson, D. J., Packwood, R. A., and Vaudin, G. J. A hierarchical multiple-SIMD architecture for image analysis. *Proceedings of the 10th International Conference on Pattern Recognition.* Atlantic City, NJ, 1990, pp. 642–647.
31. Potter, J. L. Image processing on the Massively Parallel Processor. *IEEE Comput.* **16,** 1 (1983), 62–67.
32. Rosenfeld, A. A report on the DARPA image understanding architectures workshop. *Proceedings of the 1987 Image Understanding Workshop.* 1987, pp. 298–302.
33. Samet, H., and Tamminen, M. A general approach to connected component labeling of images. Technical Report CS-TR-1649, Computer Science Department, University of Maryland, 1986.
34. Sokal, A. D. New numerical algorithms for critical phenomena (multigrid methods and all that). In Landau, D. P., Mon, K. K., and Schuttler, H.-B. (Eds.), *Computer Simulation Studies in Condensed Matter Physics: Recent Developments.* Springer-Verlag, Berlin, 1988.
35. Stauffer, D. *Introduction to Percolation Theory.* Taylor & Francis, Philadelphia, 1985.
36. Stout, Q. F. Supporting divide-and-conquer algorithms for image processing. *J. Parallel Distrib. Comput.* **4** (1987), 95–115.
37. Valiant, L. G., A bridging model for parallel computation. *Comm. ACM* **33,** 8 (1990), 103–111.
38. Wang, J.-C., Lin, T.-H., and Ranka, S. Distributed scheduling of unstructured collective communication on the CM-5. Personal communication, 1994.
39. Weems, C., Riseman, E., Hanson, A., and Rosenfeld, A. A report on the results of the DARPA integrated image understanding benchmark exercise. *Image Understanding Workshop.* 1989, pp. 165–192.
40. Weems, C., Riseman, E., Hanson, A., and Rosenfeld, A. The DARPA image understanding benchmark for parallel computers. *J. Parallel Distrib. Comput.* **11** (1991), 1–24.

DAVID A. BADER holds a joint NSF/NRC research associateship position at the Institute for Advanced Computer Studies at the University of Maryland (UMIACS) and NASA Goddard Space Flight Center. He completed his B.S. (with high honors) in computer engineering and his M.S.E.E. at Lehigh University, Bethlehem, PA, in 1990 and 1991, respectively, and obtained his doctorate in electrical engineering in 1996 from the University of Maryland at College Park. Currently, his research interests

include experimental parallel algorithms for data communication, combinatorial and image processing problems, and high performance computing with hierarchical networks. He is a member of Eta Kappa Nu, Tau Beta Pi, Omicron Delta Kappa, IEEE, and ACM.

JOSEPH JÁJÁ received the Ph.D. in applied mathematics from Harvard University in 1977. He currently holds the positions of Director of Institute for Advanced Computer Studies and Professor of Electrical Engineering at the University of Maryland, College Park. He has published numerous articles on parallel algorithms, image processing, combinatorial and algebraic complexity, and VLSI signal processing. His current research interests are in the general area of high performance computing with a particular emphasis on distributed systems interconnected by high-speed networks. He is the author of the book *An Introduction to Parallel Algorithms* published by Addison–Wesley, 1992. He is currently a subject area editor on parallel algorithms for the *Journal of Parallel and Distributed Computing*, and an associate editor for *IEEE Transactions on Parallel and Distributed Systems.*

Received March 6, 1995; revised February 6, 1996; accepted February 12, 1996