Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study (Extended Abstract)

David A. Bader[*]
dbader@eng.umd.edu

Joseph JáJá[†]
joseph@umiacs.umd.edu

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

Abstract

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. More efficient implementations of SPLIT-C will likely result in even faster execution times.

Keywords: Histogramming, Connected Components, Image Processing, Image Understanding, Scalable Parallel Processing, Parallel Algorithms.

1 Problem Overview

Given an $n \times n$ image with k grey levels on a p processor machine ($p \leq n^2$), we wish to develop efficient and portable parallel algorithms to perform various 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 resources 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 independently 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, when run on distributed memory multiprocessors.

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 grey level i. The second algorithm performs the connected components of images ([1], [6], [7], [12], [14]). The task of connected component labeling is cited as important in object recognition in the DARPA Image Understanding benchmarks ([28]), and also can be applied to several computational physics problems such as percolation ([5], [25]) and various cluster Monte Carlo algorithms for computing the spin models of magnets such as the two-dimensional Ising spin model ([2], [4]). All pixels with grey 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 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 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 support by NASA Graduate Student Researcher Fellowship No. NGT-50951 is gratefully acknowledged.

[†]Supported in part by NSF grant No. CCR-9103135 and NSF HPCC/GCAG grant No. BIR-9318183.

PPOPP '95 Santa Clara, CA USA
© 1995 ACM 0-89791-701-6/95/0007...$3.50

The majority of previous parallel histogramming algorithms are architecture- or machine-specific and do not port well to other platforms (e.g. [17], [23], [13]). Using parallel machines such as the Thinking Machines CM-5, IBM SP-1 and SP-2, Meiko CS-2, and Intel Paragon, our algorithm takes between 230 and 730 nanoseconds of work per pixel, while the best previous implementations, all on special purpose image processing or SIMD machines, have taken between 2.5 to 540 microseconds of work per pixel, where the total work is defined to be the product of the execution time and the number of processors.

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 (e.g. [10], [18], [24], [11], [29], [22], [13]). On parallel machines such as the CM-5, SP-1 and SP-2, CS-2, and Paragon, our connected components algorithm uses between 3.5 and 50 microseconds of work per pixel, while recent implementations on identical architectures have needed 50 to 300 microseconds of work per pixel, where the total work is defined to be the product of the execution time and the number of processors. See [3] for an explicit table of both the histogramming and connected component implementation results.

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 Thinking Machines CM-5, IBM SP-1 and SP-2, Meiko CS-2, and Intel Paragon. The histogramming algorithm is presented in Section 4, and the connected components algorithm is described in Section 5.

The experimental data obtained reflect the execution times from implementations on the TMC 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 shared memory model algorithms are written in SPLIT-C [8], a shared memory programming model language which follows the SPMD (single program multiple data) model on these parallel machines, and the source code is available for distribution to interested parties.

2 Block Distributed Memory Model

We use the Block Distributed Memory (BDM) Model ([15], [16]) as a computation model for developing and analyzing our parallel algorithms on distributed memory machines. This model allows the design of algorithms using a single address space and does not assume any particular interconnection topology. The model captures performance by incorporating a cost measure for interprocessor communication induced by remote memory accesses. The cost measure includes parameters reflecting memory latency, communication bandwidth, and spatial locality. This model allows the initial placement of data and prefetching.

The complexity of parallel algorithms is evaluated in terms of two measures: the computation time $T_{comp}(n,p)$, and the communication time $T_{comm}(n,p)$. The measure $T_{comp}(n,p)$ refers to the maximum of the local computations performed on any processor as measured on the standard sequential model. The communication time $T_{comm}(n,p)$ refers to the total amount of communications time spent by the algorithm in accessing remote data. Using the BDM model, an access operation to a remote location takes $\tau + 1$ time, and l prefetch read operations can be executed in $\tau + l$ time, where τ is the normalized maximum latency of any message sent in the communications network. No processor can send or receive more than one word at a time.

Two useful data movement patterns, matrix transposition and broadcasting, are discussed next, and their analyses will be included as primitives in the algorithms that follow.

Given a $q \times p$ matrix on a p processor machine, where p divides q, the matrix transposition consists of rearranging the data such that the first $\frac{q}{p}$ rows of elements are moved to the first processor in row-major order form, the second $\frac{q}{p}$ rows to the second processor, and so on, with the last $\frac{q}{p}$ rows of the matrix moved 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 $\frac{q}{p}$ elements from processor $P_{(t+i)\bmod p}$. The complexities for matrix transposition are

$$\begin{cases} T_{comm}(n,p) & = \tau + \left(q - \frac{q}{p}\right); \\ T_{comp}(n,p) & = O(q). \end{cases} \tag{1}$$

An efficient BDM algorithm which takes q elements on a single processor and broadcasts them to the other $p-1$ processors uses just two matrix transpositions.

Performance analysis reflects the execution times from implementations on the 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 the BMD model and provides a shared global address space, 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.

Performance graphs for matrix transposition and broadcasting execution times using SPLIT-C on a 32 processor CM-5, SP-2, and 8 processor Paragon are given in Figure 7. These figures also show the attained data bandwidth[1] per processor for the transpose algorithm. 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 [20]. This is comparable to the results achieved by other research teams that have achieved 6.4 MB/s per processor (Culler et.al., [9]), and 7.72 MB/s per processor (Wang et.al., [27]) 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 matrix transposition algorithm, using a high performance switch hardware rated by the vendor as having a peak node to node bandwidth of 40 MB/s [19]. 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 [21].

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 two, i.e. $p = 2^d$, 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 $v = 2^{\lfloor \frac{d}{2} \rfloor}$, and the number of columns to be $w = 2^{\lceil \frac{d}{2} \rceil}$. Each processor initially owns a tile of size $\frac{n}{v} \times \frac{n}{w}$. For future reference, we assign $q = \frac{n}{v}$ and $r = \frac{n}{w}$. 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. Figure 3 is a 512×512, 256 grey-level, image from the Second DARPA Image Understanding Benchmark [28]. 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 grey 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 patterns for binary images. A catalog of nine automatically generated scalable images is used [3], 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 [26].

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

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.

Let k be the number of grey levels in the $n \times n$ input image X, and without loss of generality, k is assumed to be a power of two. Note that this implies that for $k \geq p$, the value of $\frac{k}{p}$ 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 processor tallies the number of grey levels in its own $\frac{n}{v} \times \frac{n}{w} = \frac{n^2}{p}$ subimage into its array $H_i[\star]$. The purpose of the next step is to rearrange the data so that the tallies of each grey level reside on the same processor. If $k < p$ we use a truncated transpose to put each row into a processor, P_i, $0 \leq i \leq k-1$. If $k \geq p$ we transpose $\frac{k}{p}$ rows of the local histograms into each processor, such that each processor, P_i, has all the intermediate sums needed to compute $H\left[i\frac{k}{p}\right]$ through $H\left[(i+1)\frac{k}{p} - 1\right]$. 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 matrix transposition of the $k \times p$ histogram array and takes $T_{comm}(n,p) = \tau + \left(k - \max\left(\frac{k}{p}, 1\right)\right)$. The second communication collects the histogram bars on a single processor and takes $T_{comm}(n,p) \leq \tau + \left(k - \max\left(\frac{k}{p}, 1\right)\right)$. Thus, the histogramming algorithm has the following complexities:

$$\left\{ \begin{array}{lcl} T_{comm}(n,p) & \leq & 2(\tau + k); \\ T_{comp}(n,p) & = & O\left(\frac{n^2}{p} + k\right). \end{array} \right. \quad (2)$$

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 Figure 5 for 256 grey levels for images ranging from 32×32 to 4096×4096 pixels in size. Corresponding performance graphs are given for the IBM SP-1 and SP-2 in [3]. 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 Figure 5 for an illustration of the scalability of the histogramming algorithm's performance. Since computation dominates for large n, the algorithm runs as $O\left(\frac{n^2}{p}\right)$. 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 double, the running time approximately halves.

5 Connected Components

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, following by 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[2], 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 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 $O(|V|) = O\left(\frac{n^2}{p}\right)$ time. The result is an array of positive 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 $T_{comp}(n, p) = O\left(\frac{n^2}{p}\right)$.

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

5.2 Merging Algorithm - Overview

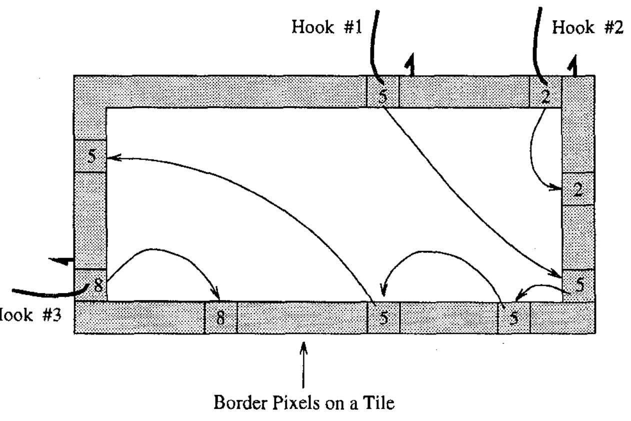


Figure 1: An example of Tile Hooks

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 **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 1 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. For each odd merge iteration t, $1 \le t \le \log p$, we will perform the $(\frac{t+1}{2})^{\text{th}}$ horizontal merge phase, and similarly, for each even merge iteration t, $1 < t \le \log p$, we will perform the $(\frac{t}{2})^{\text{th}}$ vertical merge phase.

Let t represent the current merge phase iteration, with $1 \le t \le \log p$. Thus, there will be $\log w$ and $\log v$ horizontal and vertical merge phases, respectively.

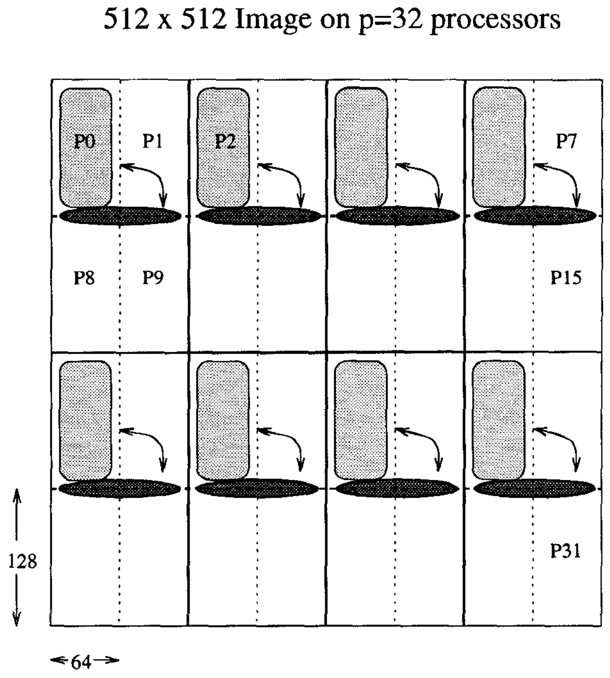
During each merge, a subset of the processors will act as **group managers**. These designated processors will prefetch the necessary border information along the column (or row) that they are located upon in the logical processor grid, setting up an equivalent graph problem, running a sequential connected components algorithm on the graph, noting any changes in the labels, and storing these changes ((α_i, β_i) pairs) in a shared structure. The **clients** decide who their current group manager is and wait until the list of label changes is ready. They retrieve the list, and all processors make the necessary updates to a proper subset of their labels.

During odd merge iterations t, the horizontal merge phases, processors are **group managers** if they reside in the logical grid with both

- row numbers whose binary representation end with a 0 followed by $(\frac{t+1}{2} - 1)$ 1's (or just ending in a 0 when $t = 1$), and
- column numbers whose binary representation end in $\frac{t+1}{2}$ 0's.

Similarly, during the even merge iterations t, the vertical merge phases, processors are **group managers** if they reside in the logical grid with both

- row numbers whose binary representation end in $\frac{t}{2}$ 0's, and
- column numbers whose binary representation end with a 0 followed by $(\frac{t}{2} - 1)$ 1's (or just ending in a 0 when $t = 2$).



Merge Phase 2: shaded processors are group managers.

Dotted borders were merged during Phase 1.

Circled borders will be merged in Phase 2.

Figure 2: Data Layout of a 512 × 512 Image on 32 Processors - Vertical Merge ($t = 2$)

An example data layout and merge is given in Figure 2. This image is 512 × 512, distributed onto a 4 × 8 logical processor grid, with each tile being 128 × 64 pixels in 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 numbers' binary representation equal to '0'. These group managers, along with their respective borders to be merged, are circled in this figure. Suppose now that $p \ge 128$, and we are at the $t = 7^{\text{th}}$ 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 number's binary representation ends with 0111, and whose column number'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 span across $2^{\frac{t+1}{2}}$ rows of processors. There are $2q$ ($= \frac{2n}{\sqrt{p}}$) pixels per processor row in the border to be merged, meaning that $2q2^{\frac{t+1}{2}} - q$ 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 $T_{comm}(n,p) \le \tau + 4q2^{\frac{t+1}{2}}$ and $T_{comp}(n,p) = O\left(\frac{n}{\sqrt{p}}2^{\frac{t+1}{2}}\right)$.

Similarly for a vertical merge, the group manager will prefetch the pixel colors and labels from the horizontal borders to be merged, which span across $2^{\frac{t}{2}}$ columns of processors. There are $2r$ pixels per processor column in the border to be merged, meaning that $2r2^{\frac{t}{2}} - r$ 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 $T_{comm}(n,p) \le \tau + 4r2^{\frac{t}{2}}$ and $T_{comp}(n,p) = O\left(\frac{n}{\sqrt{p}}2^{\frac{t}{2}}\right)$.

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 $T_{comm}(n,p) \le \tau \log p + 8n$ and $T_{comp}(n,p) = O(n)$.

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[3]. 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 $T_{comp}(n,p) = O(|V|)$ steps for a border of $|V|$ nodes[4]. 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 column. 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 $|V| \le 2q2^{\frac{t+1}{2}}$, and for each vertical merge step, $|V| \le 2r2^{\frac{t}{2}}$. Thus, the construction of this graph summed over all the iterations of the connected components algorithm takes $T_{comp}(n,p) = O(n)$.

A sequential breadth-first search based connected components algorithm computes the connected components of this graph. It runs in $O(|V| + |E|)$ steps, with $|V|, |E| = O\left(q2^{\frac{t+1}{2}}\right)$ for horizontal merges and $O\left(r2^{\frac{t}{2}}\right)$ 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). Now all the processors hit a barrier and wait until everyone has completed their tasks. After the barrier, the group manager will update its pixels' labels in $O\left(\frac{n^2}{p}\right)$.

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 will binary search the list of label changes in $T_{comp}(n,p) = O\left(\frac{n}{\sqrt{p}}\log|V|\right)$ per step. The total computational complexity over the $\log p$ merging iterations is then $O\left(\frac{n}{\sqrt{p}}\log n \log p\right)$.

At the end of the last merging step, each processor must update its interior pixel labels. Each hook described above is compared to 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 will run 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 $O\left(\frac{n^2}{p}\right)$ time.

The total complexity associated with updating the la-

[3]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.

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

bels of each tile is $T_{comp}(n,p) = O\left(\frac{n^2}{p}\right)$ assuming $p \leq n$ for large enough n. For $n \geq 128$, $p \leq \frac{n}{8}$ is sufficient.

After each merging step label update, the manager hits another barrier, waiting for the end of this iteration.

In summary, the group managers' routine has a communications complexity $T_{comm}(n,p) \leq \tau \log p + 8n$ and a computation complexity $T_{comp}(n,p) = O\left(\frac{n^2}{p} + n\right)$.

5.4 Merging algorithm - clients' task

The client processors are any processor not selected in the current iteration to run the group manager tasks. These processors calculate the logical processor grid address of the manager in charge of their border to be merged and wait for the first barrier. After this barrier, the clients prefetch the size (*chSize*) of the list of change pairs from the manager in $T_{comm}(n,p) \leq \tau + 2^t$, where $\frac{t+1}{2}$ and $\frac{t}{2}$ are the number of vertical and horizontal merges, respectively, performed inclusively during the t^{th} merge phase.

Next, the clients prefetch a block of *chSize* (α, β) change pairs from the manager. This is done in $T_{comm}(n,p) \leq \tau + 2\,(2^t)\,2q2^{\frac{t+1}{2}}$ for horizontal merges, and $T_{comm}(n,p) \leq \tau + 2\,(2^t)\,2r2^{\frac{t}{2}}$ for vertical merges, since there are at most $2q2^{\frac{t+1}{2}}$ (or $2r2^{\frac{t}{2}}$) 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, the clients hit another barrier, and wait for the end of this iteration.

Over the $\log p$ iterations, the clients' routine a communications complexity $T_{comm}(n,p) \leq (2\log p)\tau + 14np + 2p$ and a computation complexity $T_{comp}(n,p) = O\left(\frac{n^2}{p} + np\right)$.

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, $0 \leq i < \log p$, instead of sending the entire list of $c(i)$ change pairs to $f(i) - 1$ processors, a distribution algorithm based on the matrix transposition can be used. Using this algorithm, the manager will send blocks of size $\frac{c(i)}{f(i)}$ to each of $f(i)$ processors during the first phase. All of the $f(i)$ processors repeat this operation by concurrently sending their block to the other processors, in a circular fashion. The complexities for this are $T_{comm}(n,p) \leq 2\left(\tau + c(i) - \frac{c(i)}{f(i)}\right)$ and $T_{comp}(n,p) = O\left(\frac{c(i)}{f(i)}\right)$.

The clients' complexities are thus improved to $T_{comm}(n,p) \leq (3\log p)\tau + 16n + 2p$ and $T_{comp}(n,p) = O\left(\frac{n^2}{p}\right)$.

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

$$\begin{cases} T_{comm}(n,p) & \leq (4\log p)\tau + (24n + 2p) \\ & \leq (4\log p)\tau + O\left(\frac{n^2}{p}\right); \\ T_{comp}(n,p) & = O\left(\frac{n^2}{p}\right). \end{cases} \quad (3)$$

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 $(\log p)\tau$.

5.5 Experimental Results for Connected Components

Our theoretical analysis indicates that our connected components algorithm is scalable whenever $p \leq \frac{n}{c}$, where c is approximately 26 from the first expression in (3). We have implemented our algorithm in SPLIT-C on the CM-5; detailed performance plots for images ranging from 128×128 to 1024×1024 pixels in size are given in [3]. Figure 5 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-1 and SP-2 are given in [3].

This connected components algorithm easily extends to grey level images. Again, a 0-pixel is assumed to be background, while each component is the set of like-colored connected pixels. The complexity for this algorithm remains the same as for binary images (3) and is both efficient and optimal. Results for the 256-grey level DARPA Image Understanding Benchmark image of size 512×512 pixels, shown in Figure 3, are given in Figure 4 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.

6 Implementation Notes

Note that the performance graphs for the CM-5, Figures 5, 7, and 4, are for SPLIT-C (version 1.2) programs linked with the CM-5 CMMD Message Passing Libraries (version 3.2), Figure 4 uses the IBM SP-1 with the message passing libraries MPL, and Figure 7 uses the IBM SP-2 with wide nodes and also MPL. Figures 7 and 4 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 communications library. We expect results using an optimized platform shortly. Figure 7 is implemented on an 8-processor Intel Paragon, using the PAM message passing libraries, the Paragon Active Messages platform from

UC Berkeley. SPLIT-C has also been ported to the Cray T3D, and performance results are expected shortly. Note that **absolutely no** coding modifications were made to the application code used on the various platforms.

7 Acknowledgements

We would like to thank the UMIACS parallel systems staff for their help and machine maintenance while developing this research on the 32-processor UMIACS CM-5, the CASTLE group at Berkeley, especially the help and encouragement from Arvind Krishnamurthy, Lok Tin Liu, David Culler, Steve Luna, and Rich Martin, and the use of UC Berkeley's 64-processor CM-5 and 8-processor Paragon for testing purposes, and Charles Weems at the University of Massachusetts for providing the DARPA test image suite. Computational support on Berkeley's TMC CM-5 and Intel Paragon was provided by NSF Infrastructure Grant number CDA-8722788. We also thank Toby Harness and the Numerical Aerodynamic Simulation Systems Division of NASA's Ames Research Center for use of their 128-processor CM-5 and 128-node (all wide) IBM SP-2.

We thank Argonne National Labs for use of their 128-node IBM SP-1, and the Maui High Performance Computing Center for use of their 400-node IBM SP-2 machine. William Gropp, from the Mathematics and Computer Science Division of Argonne National Labs, provided significant help with the IBM SP-1 message passing interface. Also, Klaus Schauser and David Probert of University of California, Santa Barbara, provided access to the 64-node UCSB Meiko CS-2. The Meiko CS-2 Computing Facility was acquired through NSF CISE Infrastructure Grant number CDA-9218202, with support from the College of Engineering and the UCSB Office of Research, for research in parallel computing.

Please see `http://www.umiacs.umd.edu/~dbader` for additional performance information.

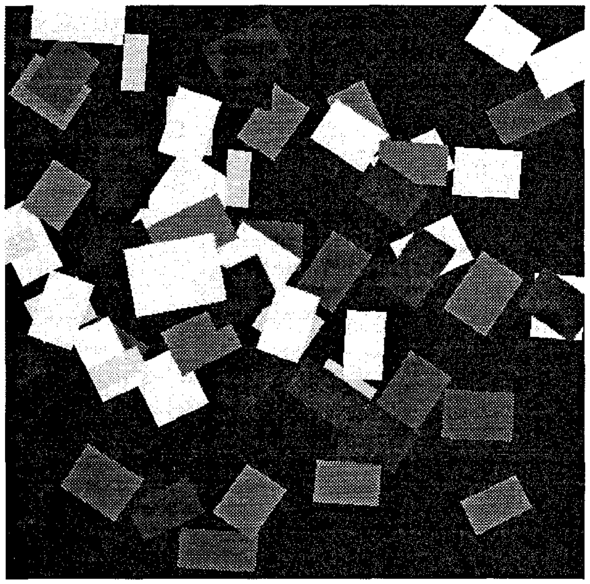


Figure 3: 512 × 512 DARPA Benchmark Test Image

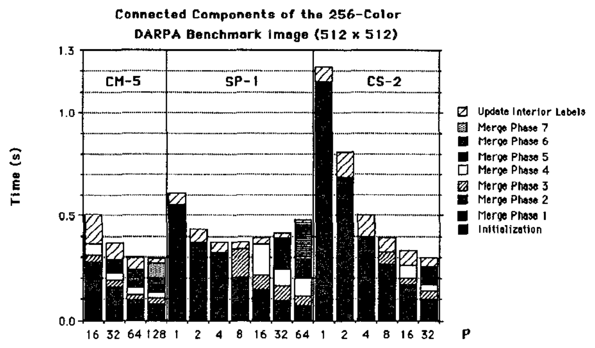


Figure 4: Connected Components Performance of the DARPA Benchmark Image on Various Machines

References

[1] H. Alnuweiri and V. Prasanna. Parallel Architectures and Algorithms for Image Component Labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14:1014–1034, 1992.

[2] J. Apostolakis, P. Coddington, and E. Marinari. New SIMD Algorithms for Cluster Labeling on Parallel Computers. *Int. J. Mod. Phys. C*, 4:749, 1993.

[3] D. A. Bader and J. JáJá. 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.

[4] C.F. Baillie and P.D. Coddington. Cluster Identification Algorithms for Spin Models - Sequential and Parallel. *Concurrency: Practice and Experience*, 3(2):129–144, 1991.

[5] R.C. Brower, P. Tamayo, and B. York. A Parallel Multigrid Algorithm for Percolation Clusters. *Journal of Statistical Physics*, 63:73, 1991.

[6] A. Choudhary and R. Thakur. Connected Component Labeling on Coarse Grain Parallel Computers: An Experimental Study. *Journal of Parallel and Distributed Computing*, 20(1):78–83, January 1994.

[7] N. Copty, S. Ranka, G. Fox, and R.V. Shankar. A Data Parallel Algorithm for Solving the Region Growing Problem on the Connection Machine. *Journal of Parallel and Distributed Computing*, 21(1):160–168, April 1994.

[8] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, S. Luna, T. von Eicken, and K. Yelick. *Introduction to Split-C.* Computer Science Division - EECS, University of California, Berkeley, version 1.0 edition, March 6, 1994.

[9] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[10] H. Embrechts, D. Roose, and P. Wambacq. Component Labelling on a MIMD Multiprocessor. *CVGIP: Image Understanding*, 57(2):155–165, March 1993.

[11] B. Falsafi and R. Miller. Component Labeling Algorithms on an Intel iPSC/2 Hypercube. In *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 159–164, Charleston, SC, April 1990.

[12] Y. Han and R.A. Wagner. An Efficient and Fast Parallel-Connected Component Algorithm. *JACM*, 37(3):626–642, 1990.

[13] H.A. Ibrahim, J.R. Kender, and D.E. Shaw. Low-Level Image Analysis Tasks on Fine-Grained Tree-Structured SIMD Machines. *Journal of Parallel and Distributed Computing*, 4:546–574, 1987.

[14] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, New York, 1992.

[15] J. JáJá and K.W. Ryu. The Block Distributed Memory Model. Technical Report CS-TR-3207, Computer Science Department, University of Maryland, College Park, January 1994.

[16] J.F. JáJá and K.W. Ryu. The Block Distributed Memory Model for Shared Memory Multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 752–756, Cancún, Mexico, April 1994. (Extended Abstract).

[17] C.R. Jesshope. Parallel Computers - Architecutures and Programming. In R.A. Vaughan, editor, *Pattern Recognition and Image Processing in Physics*, pages 205–234. Scottish Universities Summer School in Physics, New York, 1990.

[18] J.J. Kistler and J.A. Webb. Connected Components With Split and Merge. In *Proceedings of the 5th International Parallel Processing Symposium*, pages 194–201, Anaheim, CA, April 1991.

[19] J.M. Kuzela. *IBM POWERparallel System - SP2 Performance Measurements.* Power Parallel Systems, IBM, October 1994.

[20] C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, W.D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S. Wells, M.C. Wong, S.-W. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. (Extended Abstract), July 28, 1992.

[21] L.T. Liu. Personal communication. November 1994.

[22] M. Manohar and H.K. Ramapriyan. Connected Component Labeling of Binary Images on a Mesh Connected Massively Parallel Processor. *Computer Vision, Graphics, and Image Processing*, 45(2):133–149, 1989.

[23] G.R. Nudd, T.J. Atherton, N.D. Francis, R.M. Howarth, D.J. Kerbyson, R.A. Packwood, and G.J. Vaudin. A Hierarchical Multiple-SIMD Architecture for Image Analysis. In *Proceedings of the 10th International Conference on Pattern Recognition*, pages 642–647, Atlantic City, NJ, June 1990.

[24] D. Parkinson. Experiments in Component Labeling in a Parallel Computer. In V.K. Prasanna Kumar, editor, *Parallel Architectures and Algorithms for Image Understanding*, pages 209–225. Academic Press, Inc., New York, 1991.

[25] D. Stauffer. *Introduction to Percolation Theory.* Taylor and Francis, Philadelphia, PA, 1985.

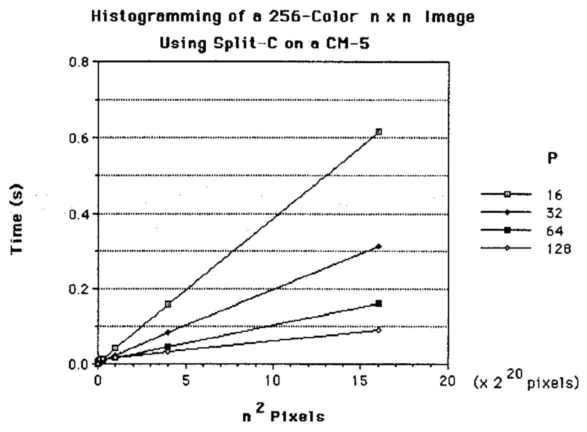
[26] Q.F. Stout. Supporting Divide-and-Conquer Algorithms for Image Processing. *Journal of Parallel and Distributed Computing*, 4:95–115, 1987.

[27] J.-C. Wang, T.-H. Lin, and S. Ranka. Distributed Scheduling of Unstructured Collective Communication on the CM-5. Technical Report CRPC-TR94502, Syracuse University, Syracuse, NY, 1994.

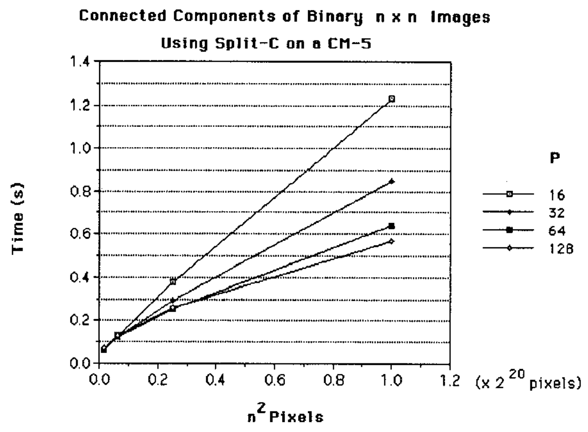
[28] C. Weems, E. Riseman, A. Hanson, and A. Rosenfeld. An Integrated Image Understanding Benchmark: Recognition of a $2\frac{1}{2}$ D "Mobile". In *Image Understanding Workshop*, pages 111–126, Cambridge, MA, April 1988.

[29] C. Weems, E. Riseman, A. Hanson, and A. Rosenfeld. A Report on the Results of the DARPA Integrated Image Understanding Benchmark Exercise. In *Image Understanding Workshop*, pages 165–192, May 1989.

A Performance Results

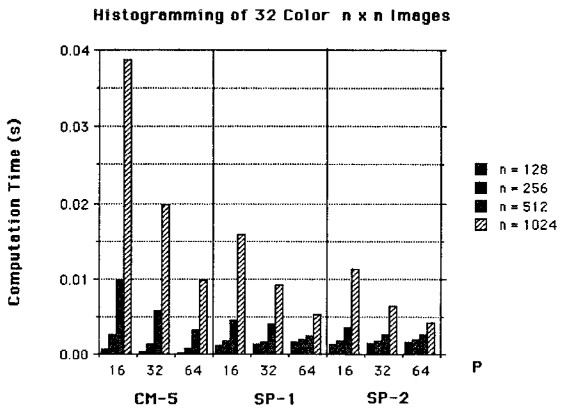


Histogramming Algorithm Performance Graph for $k <= 256$

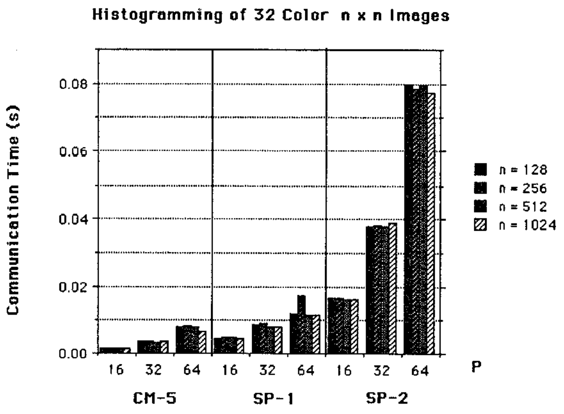


Connected Components for Binary $n \times n$ Images

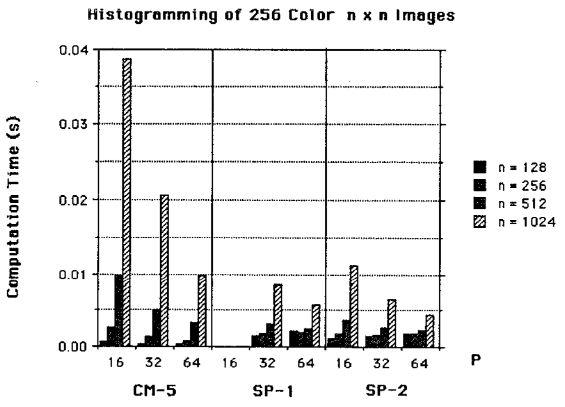
Figure 5: Scalability of Histogramming and Connected Components Scalability on the CM-5



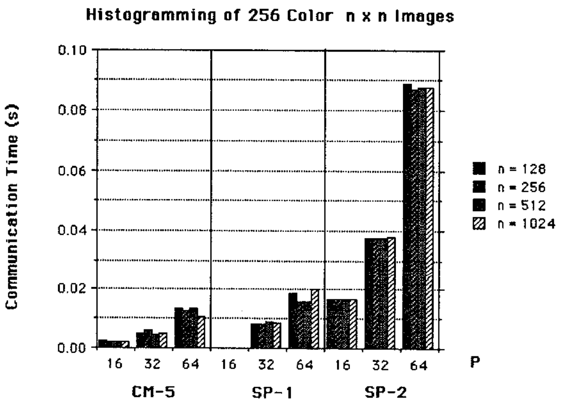
Computation Time



Communication Time



Computation Time



Communication Time

Figure 6: Scalability of Histogramming Performance on 32 and 256 Color Images

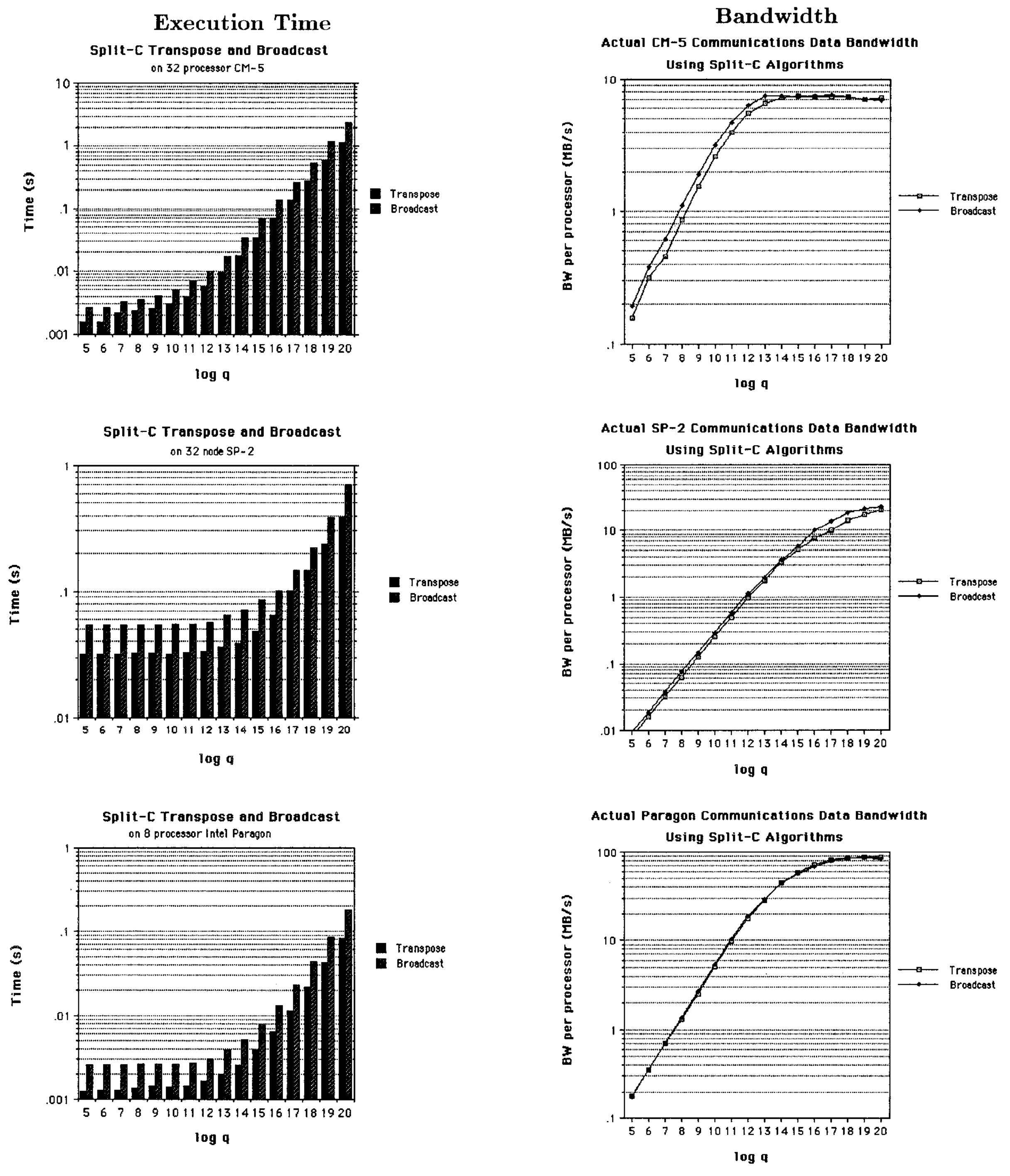


Figure 7: Transposition and Broadcasting Performance Graphs on Various Machines