The Journal of Supercomputing, 10, 141–168 (1996)

Parallel Algorithms for Image Enhancement and Segmentation by Region Growing, with an Experimental Study

DAVID A. BADER,* JOSEPH JÁJÁ,* DAVID HARWOOD, AND LARRY S. DAVIS**

{dbader, joseph, harwood, lsd}@umiacs.umd.edu

Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742

(Received May 1995; final version accepted July 1996.)

Abstract. This paper presents efficient and portable implementations of a powerful image enhancement process, the Symmetric Neighborhood Filter (SNF), and an image segmentation technique that makes use of the SNF and a variant of the conventional connected components algorithm which we call *δ-Connected Components*. We use efficient techniques for distributing and coalescing data as well as efficient combinations of task and data parallelism. The image segmentation algorithm makes use of an efficient connected components algorithm based on a novel approach for parallel merging. The algorithms have been coded in SPLIT-C and run on a variety of platforms, including the Thinking Machines CM-5, IBM SP-1 and SP-2, Cray Research T3D, Meiko Scientific CS-2, Intel Paragon, and workstation clusters. Our experimental results are consistent with the theoretical analysis (and provide the best known execution times for segmentation, even when compared with machine-specific implementations). Our test data include difficult images from the Landsat Thematic Mapper (TM) satellite data.

Keywords: Parallel algorithms, image processing, region growing, image enhancement, image segmentation, Symmetric Neighborhood Filter, connected components, parallel performance.

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 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 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.

Image-processing applications are well-suited to high-performance computing techniques because of their regular input organization (typically multidimensional arrays of discrete values) and spatial locality properties; for example, pixels near each other tend to be of similar color. Images used for analysis are produced from a variety of applications, for

* Also affiliated with the Department of Electrical Engineering.

** Also affiliated with the Department of Computer Science and the Center for Automation Research.

example, remote sensing of the Earth, detection of surface defects in industrial manufacturing, and military target recognition. Some of the processing is low-level, such as image calibration or enhancement, while other analyses are intermediate- or high-level, such as segmenting an image into objects or regions and classifying each. We now introduce our work on image enhancement and segmentation.

Image segmentation algorithms cluster pixels into homogeneous regions, which, for example, can be classified into categories with higher accuracy than could be obtained by classifying the individual pixels. Region growing is a class of techniques used in image segmentation algorithms in which, typically, regions are constructed by an agglomeration process that adds (merges) pixels to regions when those pixels are both adjacent to the regions and similar in property (most simply intensity) (e.g., [18, 22, 35, 61, 66]). Each pixel in the image receives a label from the region-growing process; pixels will have the same label if and only if they belong to the same region. Our segmentation algorithm makes use of an efficient and fast parallel connected components algorithm based on a novel approach for merging (a detailed theoretical and experimental analysis of this algorithm can be found in previous work [8]). Typically in region-growing algorithms a region's border is susceptible to erroneous merging at its weakest point, which can be aggravated by several factors, including noise, blur, and lighting. Thus it becomes extremely important to enhance an image before the region-growing process. We next describe a new image enhancement filter that preserves edges as well as smoothes the interior of regions.

In real images, natural regions have significant variability in gray level. Noise, introduced from the scanning of the real scene into the digital domain, will cause single-pixel outliers. Also, lighting changes can cause a gradient of gray levels in pixels across the same region. Because of these and other similar effects, we preprocess the image with a stable filter, the Symmetric Neighborhood Filter (SNF) [36], that smooths out the interior pixels of a region to a near-homogeneous level. Also, due to the relative motion of the camera and the scene, as well as aperture effects, edges of regions are usually blurred so that the transition in gray levels between regions is not a perfect step over a single pixel, but ramps from one region to the other over several pixels. Our filter is, additionally, an edge-preserving filter that detects blurred transitions such as these and sharpens them while preserving the true border location as best as possible. Most preprocessing filters will smooth the interior of regions at the cost of degrading the edges or, conversely, detect edges while introducing intrinsic error on previously homogeneous regions. However, the SNF is an edge-preserving smoothing filter that performs well for simultaneously sharpening edges and smoothing regions. In addition, it is an iterative filter that also can be tuned to retain thin-image structures corresponding, for example, to rivers and roads. A variety of SNF operators have been studied, and we chose a single-parameter version that has been shown to perform well on remote sensing applications.

The majority of previous parallel implementations of the SNF filter are architecture- or machine-specific and do not port well to other platforms (e.g., [31, 46, 47, 48, 56]). For example, Webb [57] gives an implementation of a 15×15 SNF filter on the CMU Warp, a 10-processor linear systolic array, which takes 4.76 seconds on a 512×512 image. We present our SNF filter execution timings in Section 5. In comparison, on a 32-processor

TMC CM-5, we take less than 165 milliseconds per iteration operating on an image of equivalent size.

After the image is enhanced by the SNF, we use a variant of the connected components algorithm for gray-level images, called *δ-Connected Components*, to combine similar pixels into homogeneously labeled regions producing the final image segmentation. As with the SNF implementations, most previous parallel algorithms for segmentation do not port well to other platforms (e.g., [27, 42, 43, 54, 61, 62, 63]).

There is a vast literature on the implementations of parallel segmentation algorithms based upon connected components. Some of these algorithms only operate on binary images [33, 65, 50, 29, 2, 15, 42, 19, 30, 45, 44, 3, 53] and thus are not useful for a large class of image analysis problems from multigray-level images. Most of the previous connected components algorithms for gray-level images [32, 22, 21, 13, 20, 60, 41] are machine-dependent and thus are not efficient on current parallel platforms. In addition, both binary and gray-level connected components algorithms strictly label regions based upon pixel color and locality. Thus interior variation in regions as well as noise and blur cause the standard algorithms to fail when useful segmentation of real imagery is required. Several parallel algorithms have been developed that attempt to overcome these difficulties by adding additional conditions. For example, Dehne and Hambrusch [26] present mesh and hypercube algorithms for binary images with the notion of k-width connectivity, such that two 1-pixels (a and b) belong to the same k-width component if and only if there exists a path of width k such that pixel a is one of the k start pixels and b is one of the k end pixels of this path. Also, Hambrusch et al. [32] describe a mesh algorithm for gray-level images that accepts two input parameters for range (ϵ) and adjacency (δ) such that in any labeled component, the maximum difference between any two pixels is ϵ and the maximum difference between any two adjacent pixels is δ.

Our approach is the first high-level parallel algorithm and implementation for gray-level images that is both (1) simple, using a single relaxation parameter, δ, such that in any labeled components the maximum difference between any two adjacent pixels is no greater than δ and (2) efficient, using drastically limited merging steps in which image tile boundaries are first labeled consistently, and then a final label update is propagated inward. In addition, our algorithm does not belong to any of the four parallelized versions of standard sequential connected component methods [42], including the Nearest-Neighbor Propagation, Shrink-Expand, Boundary Following, and Union-Find approaches.

The experimental data obtained reflect the execution times from implementations on the TMC CM-5, IBM SP-1 and SP-2, Meiko CS-2, Cray Research T3D, and the Intel Paragon, with the number of parallel processing nodes ranging from 16 to 128 for each machine when possible. The parallel algorithms are written in SPLIT-C [23], a parallel extension of the C programming language that follows the SPMD model on these parallel machines, and the source code is available for distribution to interested parties.

The organization of this paper is as follows. In Section 2 we address the algorithmic model and various primitive operations that are used to analyze the algorithms. Section 3 is a discussion of the test images and data layout on the parallel machines. Our segmentation process overview, which includes a discussion of SNF and 1-Nearest Neighbor filters and the δ-Connected Components algorithm, is given in Section 4. Finally, in Sections 5 and 6

we describe the parallel implementations of the Symmetric Neighborhood Filter algorithm and δ-Connected Components, respectively, and present algorithmic analyses and empirical results.

2. The Model for Parallel Computation

In this section we describe the simple model that we use for analyzing the performance of parallel algorithms. Our model is based on the fact that current 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. A parallel algorithm consists of a sequence of local computations interleaved with communication steps, where we allow computation and communication to overlap. We account for communication costs as follows.

The transfer of a block consisting of m contiguous words between two processors, assuming no congestion, takes $\tau + \sigma m$ time, where τ is a bound on the latency of the network and σ is the time per word at which a processor can inject or receive data from the network. Note that the bandwidth per processor is inversely proportional to σ. We assume that the bisection bandwidth is sufficiently high to support block permutation routing among the p processors at the rate of $\frac{1}{\sigma}$ per processor. In particular, for any subset of q processors, a block permutation among the q processors takes $\tau + \sigma m$, where m is the size of the largest block. Similar to MPI and other message-passing standards, we assume that communication and computation can be overlapped. This cost model can be justified by our earlier work [39, 40, 9, 10, 11, 7].

Using this cost model, we can evaluate the communication time $T_{comm}(n,p)$ 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., the BSP [55], LogP [25], and LogGP [1] models) that have recently appeared in the literature, but significant differences exist. Our model is extended to include a collection of communication primitives that makes our model considerably easier to use than the BSP or the LogP models [7].

We define the computation time $T_{comp}(n,p)$ as the maximum time any processor takes to perform all the local computation steps. In general, the overall performance $T_{comp}(n,p) + T_{comm}(n,p)$ involves a tradeoff between $T_{comm}(n,p)$ and $T_{comp}(n,p)$. Our aim is to develop parallel algorithms that achieve

$$T_{comp}(n,p) = \mathrm{O}\left(\frac{T_{seq}}{p}\right)$$

such that $T_{comm}(n,p)$ is minimum, where T_{seq} is the complexity of the best sequential algorithm. Such optimization has worked very well for the problems we have looked at, but

other optimization criteria are possible. 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}).

The complete set of collective communication primitives is discussed fully in earlier work [7]. Next, we present three of these useful primitives, **transpose**, **broadcast**, and **reduce**, which are needed by the algorithms presented in this paper.

2.1. *Transpose Communication Primitive*

Given a $q \times p$ array on a p processor machine, where p divides q, the **transpose** consists of rearranging the data such that the first $\frac{q}{p}$ rows of elements are moved to the first processor, the second $\frac{q}{p}$ rows to the second processor, and so on, with the last $\frac{q}{p}$ rows of the array moved to the last processor. An efficient **transpose** algorithm consists of p iterations such that, during iteration i, $(1 \leq i \leq p-1)$, each processor P_t gets the appropriate block of $\frac{q}{p}$ elements from processor $P_{(t+i) \bmod p}$. The parallel algorithm and analysis for the **transpose** data movement are given in earlier work [8, 9, 7] and are similar to that of the LogP model [25]. The **transpose** primitive has the following complexity:

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

2.2. *Broadcasting Communication Primitive*

Another useful data movement primitive is to broadcast a block of data from a single processor to the remaining processors. An efficient algorithm [8, 9, 7, 39] takes q elements on a single processor and broadcasts them to the other $(p-1)$ processors using just two **transpose** primitives. We start by discussing the case when there are more than p elements to be broadcast.

An efficient algorithm to **broadcast** the q elements is based on the **transpose** primitive, where q is assumed to be larger than p. Processor r holds the q elements to be broadcast in the first column of array A. We compute the **transpose**(A) primitive, thus giving every processor $\frac{q}{p}$ 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 [39].

The analysis of this **broadcast** primitive is simple. Since this algorithm just performs two **transposes**, the complexity of the broadcasting algorithm is

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

Performance analysis given in earlier work [8, 7] reflects the execution times from implementations on the CM-5, SP-2, and CS-2, each with $p = 32$ parallel processing nodes.

SPLIT-C can express the capabilities of the parallel model and provides constructs to express data layout and **split-phase** assignments. The **split-phase** assignment operator, :=, prefetches data from the specified remote address into a 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.

2.3. Reduce Communication Primitive

The **reduce** communication primitive takes a parallel input array A and an associative operator, $\oplus$, and returns the value of $\sum_{i=0}^{p-1} A[i]$, where $\sum$ uses the associative operation $\oplus$. Parallel computers can handle this efficiently [14], and SPLIT-C implements this as a primitive library function. A simple algorithm consists of $p-1$ rounds that can be pipelined [39]. Each processor P_i initializes a local sum to $A[i]$. During round r, each processor then reads $A[(i+r) \bmod p]$, for $1 \leq r \leq p-1$, and adds this value to the local sum. Since these rounds can be realized with $p-1$ nonblocking read operations, the resulting complexity is

$$\begin{cases} T_{comm}(n,p) & \leq \tau + (p-1)\sigma; \\ T_{comp}(n,p) & = O(p). \end{cases} \tag{3}$$

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.

Our test images shown in Appendixes A and B are divided into two categories, artificial and real, respectively. The artificial images given in Figures A.1 and A.2 range in size from 128×128 to 512×512 pixels. We use Landsat satellite data to represent real images; Figure B.1 is from band 5 of a South American scene, and Figure B.2 is band 4 taken from a view of New Orleans. Both of these images are 256 gray-level, 512×512 pixel arrays from single bands of the Landsat Thematic Mapper (TM) satellite data.

4. Image Segmentation—Overview

Images are segmented by running several phases of the SNF enhancement algorithm, followed by several iterations of the 1-Nearest Neighbor (1-NN) filter and finally the δ-Connected Components. See Figure 1 for a dataflow diagram of the complete segmentation

process. The following subsections describe these image-processing algorithms, beginning with the SNF algorithm.

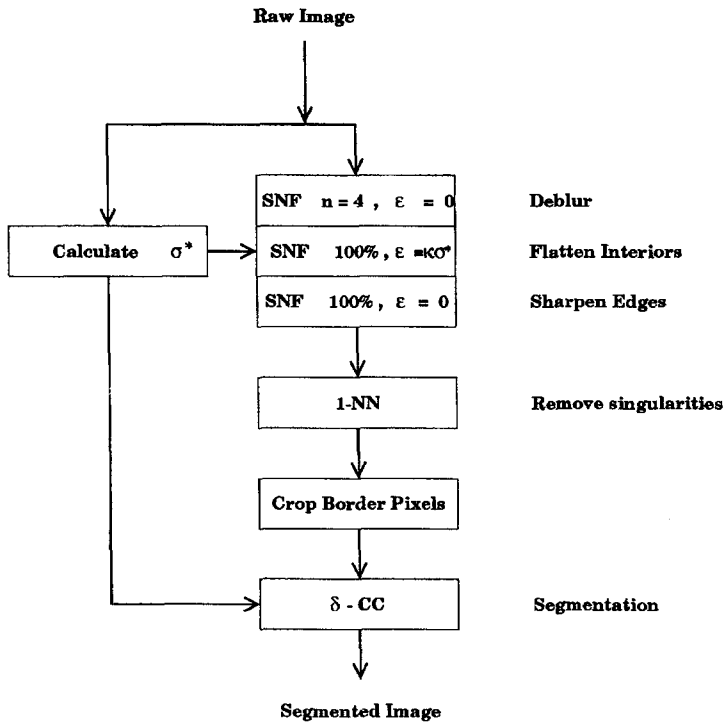


Figure 1. Segmentation process.

4.1. Symmetric Neighborhood Filter

Due to interior variation as well as noise and blur, regions in real images are seldom homogeneous in gray level and sharp along their borders. Preprocessing the image with an enhancement filter that reduces these effects will yield better segmentation results.

The SNF enhancement is a stable filter that is applied either for a fixed number of iterations or until stopping criteria (defined below) are reached, and takes the single parameter ϵ, as follows. The SNF filter compares each pixel to its 8-connected neighbors. (Note that the 1-pixel image boundary is ignored in our implementation.) The neighbors are inspected in symmetric pairs around the center, that is, N $\sim$ S, W $\sim$ E, NW $\sim$ SE, and NE $\sim$ SW; see Figure 2 for a diagram of a 3×3 neighborhood centered around a pixel, with the symmetric pairs colored the same. Using each pair and the center pixel, one of the three in each of the

four comparisons is selected using the following criteria. Assume without loss of generality that the pair of pixels are colored A and B and that $A > B$ (see Figure 3). If the center pixel (with value x) falls within region R_A, that is, $\frac{A+B}{2} < x \leq A + \epsilon$, then we select A. Likewise, if the center pixel falls within region R_B, that is, $B - \epsilon \leq x < \frac{A+B}{2}$, then we select B. And if x is midway between A and B, we simply select x, which is the average. Finally, if x is an outlier with respect to A and B so that $x > A + \epsilon$ or $x < B - \epsilon$, we leave x fixed. The four selected pixels are then averaged together, and finally the center pixel is replaced by the mean of this average and the center pixel's current gray-level value. This latter average is similar to that of a damped gradient descent, which yields a faster convergence.

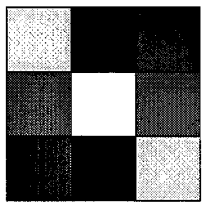


Figure 2. Symmetric pairs of pixels.

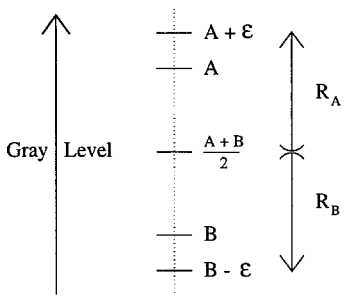


Figure 3. Selection of SNF pixels.

The first phase of segmentation is a combination of three iterative SNF filters. The first step runs for a small number of iterations (e.g., four) with $\epsilon = 0$ and is used to preserve edges. We define σ^* to be the median of the standard deviations of all 3×3 neighborhoods centered around each nonborder pixel in the image. See our previous work [10, 11] for a parallel median algorithm. To flatten the interior of regions, SNF iterates with $\epsilon = \kappa\sigma^*$, where κ is typically set to 2.0 for this application. The stopping criteria for this iterative filter occurs when the percentage of "fixed" pixels reaches 100.0 %, this percentage has not changed for three iterations, or when we reach 200 iterations, whichever comes first. Finally, we sharpen the borders of regions with SNF using $\epsilon = 0$, again stopping the iterative process when the pixels have fixed, as defined above. The resulting image has near-homogeneous regions with sharp transitions between bordering regions. While the SNF enhancement filter performs

qualitatively well for flattening region interiors and sharpening their borders, single-pixel outliers (perhaps due to noise) are not touched. Next, we describe our method for handling such pixels.

4.2. 1-Nearest Neighbor Filter

Single-pixel regions rarely can be classified, even under the best circumstances. Therefore, we prefer to filter these out as our last enhancement stage. A typical 1-Nearest Neighbor filter removes single-pixel outliers by replacing each pixel in the image with the value of one of its adjacent pixels that is closest to its own gray level. Note that one application of the 1-Nearest Neighbor filter may cause small neighborhoods of pixels to oscillate. For example, two adjacent pixels with values A and $A + \Delta$ surrounded by a region of more than Δ levels above or below would never stabilize. Therefore, we apply the 1-Nearest Neighbor as an iterative filter, stopping when the input and output images are identical. For faster convergence we use a damped approach (similar to the SNF) that assigns an output pixel to the mean of its original and nearest-neighbor values. Typically, we converge in roughly six to eight iterations.

Since no image enhancement occurs along the pixels of image borders, we crop the border so that additional segmentation techniques will not use this raw data to merge dissimilar regions via paths through the noisy, uncorrected pixels. For this application we crop the border by a width of three pixels. With the enhanced image we are now ready to present a new segmentation algorithm that combines the pixels into regions.

4.3. δ-Connected Components

The image-processing problem of determining the connected components of images is a fundamental task of imaging systems (e.g., [2, 21, 22, 28, 34, 37, 38]). The task of connected component labeling is cited as a fundamental computer vision problem in the DARPA Image Understanding benchmarks [49, 58, 60], and also can be applied to several computational physics problems such as percolation [16, 52] and various cluster Monte Carlo algorithms for computing the spin models of magnets such as the two-dimensional Ising spin model [5, 12, 51]. 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 uniformly 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. Each pixel in the image will receive a label; pixels will have the same label if and only if they belong to the same connected component. Also, all background pixels will receive a label of 0.

It is interesting to note that, in the previous paragraph, we defined connected components as a maximal collection of uniform color pixels such that a *path* existed between any pair of pixels. The conventional algorithm assumes that there is a connection between two adjacent pixels if and only if their gray-level values are identical. We now relax this connectivity

rule and present it as a more general algorithm called *δ-Connected Components*. In this approach we assume that two adjacent pixels with values x and y are connected if their absolute difference $|x - y|$ is no greater than the threshold δ. Note that setting the parameter δ to 0 reduces the algorithm to the classic connected components approach. This algorithm is identical in analysis and complexity to the conventional connected components algorithm, as we are merely changing the criterion for checking the equivalence of two pixels.

For the final phase in the segmentation process, δ-Connected Components is applied to the enhanced image, using $\delta = \kappa\sigma^*$, where the values of κ and σ^* are the same as those input to the enhancement filters. The analysis for the δ-Connected Components algorithm is given in Section 6, equation (7). Thus we have an efficient algorithm for image segmentation on parallel computers. The results of the segmentation process on our test images are described next.

4.4. Test Images

We use the Landsat Thematic Mapper (TM) raw satellite data for our test images. Each test image is a 512 × 512 pixel subimage from a single TM band. Figure B.1 shows a subimage from band 5, an image from South America, and Figure B.2 is taken from band 4 of New Orleans data. These images have 256 gray levels and also have postprocessing enhancement of the brightness for visualization purposes in this paper. We have applied SNF enhancement to these images, and the results appear below the original images. For the band 5 data, Figure B.1 shows the results of the enhancement, with both the full image, and an enlargement of a structure in the river of this image. A further segmentation with $\delta = \kappa\sigma^*$ using the δ-Connected Components algorithm is given at the bottom of Figures B.1 and B.2.

5. Symmetric Neighborhood Filter—Parallel Implementation

Most common enhancement filters will smooth the interior of regions at the cost of degrading the edges or find edges while introducing intrinsic error on previously homogeneous regions. However, the Symmetric Neighborhood Filter (SNF) is an edge-preserving smoothing filter, meaning that it performs well for both sharpening edges and flattening regions. The SNF is a convergent filter that can be run for a predetermined number of iterations or until a percentage of the image pixels are fixed in gray level. A variety of SNF operators have been studied, and we chose a single-parameter version that has been shown to perform well. Previous parallel implementations of the SNF have been based on special-purpose image-processing platforms, including data-parallel SIMD machines such as the TMC CM-2 and the MasPar MP-1 [46, 47], video-rate VLSI implementations [48], pipelined computers [31], and systolic linear arrays such as the Warp [4, 56, 57].

A useful data movement needed for this 3 × 3 local SNF filter is the fetching of tile-based *ghost cells* [24, 64], which contain shadow copies of border pixels from adjacent tiles. These ghost cells are used in the selection process when recalculating each tile's border. Suppose each processor is allocated a $q \times r$ pixel tile from the image. In total there will be

eight ghost cell arrays as follows. The first two ghost cell arrays, **ghostN** and **ghostS**, each hold r pixels, and the second two, **ghostW** and **ghostE**, hold q pixels each. In addition, four single pixel ghost cells for diagonal neighboring pixels are **ghostNW**, **ghostNE**, **ghostSE**, and **ghostSW**. An example of these ghost cells is pictured in Figure 4.

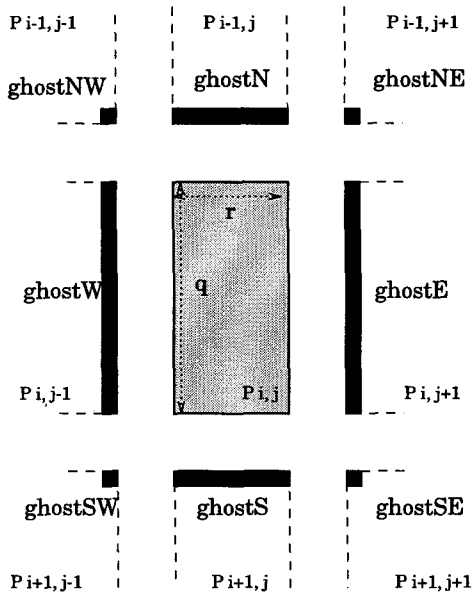


Figure 4. An example of ghost cells.

The analysis for filling the ghost cells is simple. We can divide the operation into eight separate data movements, one for each direction. Since each movement is a block permutation, it can be routed efficiently using the parallel model in $\leq \tau + m\sigma$ communication cost, where m is the block size. Thus the filling of the north and south ghost cell arrays each take $T_{comm}(n,p) \leq \tau + r\sigma$, the east and west ghost cell arrays each take $T_{comm}(n,p) \leq \tau + q\sigma$, and the diagonal four ghost cells each take $T_{comm}(n,p) \leq \tau + \sigma$. Therefore, the entire ghost cell fetching operation takes

$$\begin{cases} T_{comm}(n,p) & \leq & 8\tau + \left(4\frac{n}{\sqrt{p}} + 4\right)\sigma; \\ T_{comp}(n,p) & = & \mathrm{O}\left(\frac{n}{\sqrt{p}}\right). \end{cases} \tag{4}$$

We are now ready to present the parallel algorithm for the Symmetric Neighborhood Filter.

The following is an SPMD algorithm for an iteration of SNF on processor i:

Algorithm: Symmetric Neighborhood Filter
{ i } is my processor number;
{ p } is the total number of processors, labeled from 0 to $p-1$;

$\{ A \}$ is the $n \times n$ input image.
$\{ \epsilon \}$ is input parameter.
begin
0. Processor i gets an $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ tile of image A, denoted A_i.
1. **Prefetch** ghost cells.
2. **For each** local pixel $A_{i,<x,y>}$ that has not fixed yet, using ϵ, compute $B_{i,<x,y>}$, the updated pixel value. **Decide** if local pixel position $< x, y >$ is now fixed.
3. **Set** f_i equal to the number of local pixels that have remained fixed.
4. $f = \mathbf{reduce}(f_i, +)$; that is, $f = \sum_{i=0}^{p-1} f_i$.
5. **Output** $\frac{f}{n^2} \times 100\%$.
end

For each iteration of the SNF operator on a p-processor machine, the theoretical analysis is as follows. The complexities for Step 1 and Step 4 are shown in (4) and (3), respectively. Steps 2 and 3 are completely local and take $O\left(\frac{n^2}{p}\right)$. Thus for $p \leq n$, the SNF complexities are

$$\begin{cases} T_{comm}(n,p) & \leq & 9\tau + \left(4\frac{n}{\sqrt{p}} + 3 + p\right)\sigma; \\ T_{comp}(n,p) & = & O\left(\frac{n}{\sqrt{p}} + p\right) + O\left(\frac{n^2}{p}\right) \\ & = & O\left(\frac{n^2}{p}\right). \end{cases} \tag{5}$$

Figure 5 shows the convergence of the SNF enhancement during the second phase of the smoothing filter. As can be seen, there is a fast convergence of the pixels asymptotically close to 100% fixed. Because fixed pixels are not recalculated, the time per iteration quickly ramps down from approximately 165 ms/iteration to 26 ms/iteration on a 512×512 TM image.

The complexity of an iteration of the 1-Nearest Neighbor filter is simple, namely, a fetch of ghost cells and one pass through the image tile on each processor. The ghost cell analysis is given in (4), and the update of pixels takes $O\left(\frac{n^2}{p}\right)$. Therefore, the 1-Nearest Neighbor algorithm has complexities

$$\begin{cases} T_{comm}(n,p) & \leq & 8\tau + \left(4\frac{n}{\sqrt{p}} + 4\right)\sigma; \\ T_{comp}(n,p) & = & O\left(\frac{n^2}{p}\right). \end{cases} \tag{6}$$

6. δ-Connected Components of Gray-Scale 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 easy

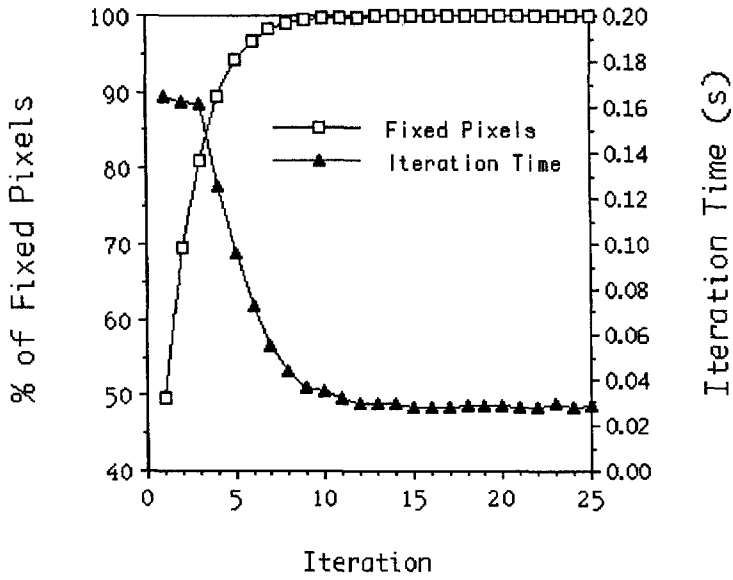
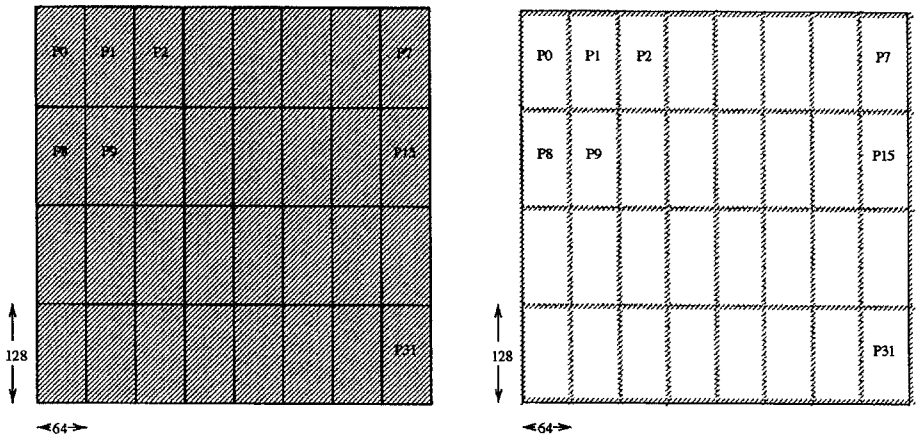


Figure 5. Convergence and timings of SNF for a 512 × 512 image on a 32-processor CM-5.

merging. In our parallel labeling algorithm the splitting phase is trivial and implicit, while the merging process requires more work. The algorithm contains three major phases. In the first phase labelings of each tile are performed concurrently. During the second phase processors perform a drastically limited merging operation such that at the conclusion, a mesh of tile borders is labeled consistently (see Figure 6). Finally and concurrently, each processor recolors its interior pixels using the consistent tile border labelings.

Each processor initially holds a unique tile of the image and hence can label the connected components of its tile by using a standard sequential algorithm based upon a breadth-first search. Next, the algorithm iterates $\log p$ times,[1] alternating between combining the tiles in *horizontal merges* of vertical borders and *vertical merges* of horizontal borders. Our algorithm uses novel techniques to perform the merges and to update the labels. We will attempt to give an overview of this algorithm (a complete description can be found in previous work [8, 9, 7]).

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 that 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



a) Every pixel is updated. b) Only the border pixels are updated.

Figure 6. Image is 512×512 on $p = 32$ processors. After each merging step, most previous labeling algorithms update every pixel (a). Our algorithm performs drastically limited merging steps such that only a mesh of tile borders is relabeled (b).

changes to their respective clients, all processors must use the information to update their *tile hooks*, data structures that point to connected components on local tile borders. See Figure 7 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.

An example data layout and merge is given in Figure 8. This image of size 512×512 is 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 $t = 2$.

During each merge a subset of the processors will act as group managers. (The group managers, along with their respective borders to be merged, are circled in Figure 8.) 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. Each client decides which processor is its current group manager and waits until the list of label

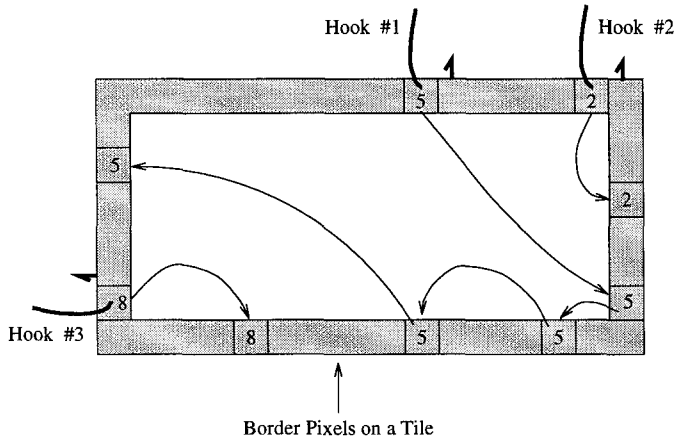


Figure 7. An example of tile hooks.

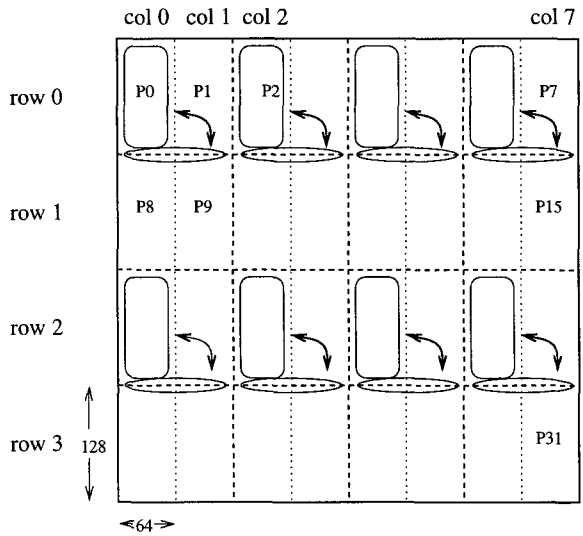


Figure 8. Data layout of a 512×512 image on $p = 32$ processors — vertical merge ($t = 2$). Circled processors are group managers. Dotted borders were merged in Phase 1, and circled borders will be merged in Phase 2.

changes is ready. Each retrieves the list, and finally all processors make the necessary updates to a proper subset of their labels.

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 that are adjacent to each other and differ by no greater than δ in gray level. 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.[2] 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.

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. Taking advantage of this, we do not need to propagate boundary labels inward to recolor a tile's interior pixels after each iteration of the merge. This is one of the attractive highlights of our newly proposed algorithm, namely, the drastically limited updates needed during the merging phase.

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.

6.1. *Parallel Complexity for δ-Connected Components*

Thus for $p \leq n$ the total complexities for the parallel δ-Connected Components algorithm are [8]

$$\begin{cases} T_{comm}(n,p) & \leq (4\log p)\tau + (24n+2p)\,\sigma = (4\log p)\tau + \mathrm{O}\left(\frac{n^2}{p}\right)\sigma; \\ T_{comp}(n,p) & = \mathrm{O}\left(\frac{n^2}{p}\right). \end{cases} \tag{7}$$

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

The majority of previous connected components parallel algorithms are architecture- or machine-specific, and do not port easily to other platforms. Table 1 shows some previous running times for parallel implementations of connected components on the DARPA II image given in Figure A.1. The second to 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.

Our implementation also performs better compared with other recent parallel region-growing codes [22]. Note that this implementation uses data-parallel Fortran on the TMC

Year	Researcher(s)	Machine	PEs	Time (s)	Work/Pix	Notes
1989	Kanade and Webb [41]	Warp	10	4.34	166 μs	Shrink/expand
1989	Weems et al. [59]	Alliant FX-80	8	7.225	220 μs	
		Sequent Symmetry 81	8	15.12	461 μs	
		Warp	10	3.98	152 μs	
		TMC CM-2	32768	0.140	547 μs	
1992	Choudhary and Thakur [20]	Intel iPSC/2	32	1.914	234 μs	Multidim. divide & conquer (partitioned input)
				1.649	201 μs	Multidim. divide & conquer (complete image/PE)
				2.290	280 μs	Multidim. divide & conquer (cmplt. + collect. commun.)
		Intel iPSC/860	32	1.351	165 μs	Multidim. divide & conquer (partitioned input)
				1.031	126 μs	Multidim. divide & conquer (complete image/PE)
				0.947	116 μs	Multidim. divide & conquer (cmplt. + collect. commun.)
		Encore Multimax	16	0.521	31.8 μs	Multidim. divide & conquer (partitioned input)
1994	Choudhary and Thakur [21]	TMC CM-5	32	0.456	55.7 μs	Multidim. divide & conquer (partitioned input)
				0.398	48.6 μs	Multidim. divide & conquer (complete image/PE)
				0.452	55.2 μs	Multidim. divide & conquer (cmplt. + collect. commun.)
1994	Bader and JáJá [8]	TMC CM-5	32	0.368	44.9 μs	
		IBM SP-1	4	0.370	5.65 μs	
		IBM SP-2-WD	4	0.243	3.71 μs	
		Meiko CS-2	2	0.809	6.17 μs	
			32	0.301	36.7 μs	
1995	Bader et al. (this paper)	IBM SP-2-TH	4	0.260	3.97 μs	
			8	0.257	7.84 μs	
			16	0.285	17.4 μs	
		IBM SP-2-WD	4	0.245	3.74 μs	
			8	0.238	7.26 μs	
			16	0.262	16.0 μs	
		TMC CM-5	16	0.474	28.9 μs	
		Meiko CS-2	4	0.627	9.57 μs	
			8	0.393	12.0 μs	
			16	0.351	21.4 μs	
			32	0.317	38.7 μs	
		CRAY T3D	2	0.472	3.60 μs	
			4	0.470	7.17 μs	
			8	0.479	14.6 μs	

Table 1. Implementation results of parallel connected components of the DARPA II image (512×512).

CM-2 and CM-5 machines, and lower-level implementations on the CM-5 using Fortran with several message-passing schemes. For example, Figure A.2 shows two of the more difficult images from the study by Copty et al. [22] that are segmented by region growing. Image 3 is a 256-gray level 128 × 128 image, containing six homogeneous circles. Image 6 is a binary 256 × 256 image of a tool. Tables 2 and 3 show the comparison of execution times for Images 3 and 6, respectively. Because these images are noise-free, our algorithm skips the image enhancement task. Notice that our algorithms are faster by several orders of magnitude than those of Copty et al. [22] on the CM-5 with 32 processors.

Table 2. Implementation results of segmentation algorithm on image 3 from Copty et al. [22], six gray circles (128 × 128 pixels).

Year	Researcher(s)	Machine	PEs	Time (s)	Work/Pix	Notes
1994	Copty et al. [22]	TMC CM-2	8192	13.911	217 ms	Data parallel
			16384	9.650	302 ms	Data parallel
		TMC CM-5	32	42.931	83.9 ms	Data parallel
				9.567	18.5 ms	Message passing, comm1
				5.537	10.8 ms	Message passing, comm2
1995	Bader et al. (this paper)	TMC CM-5	16	0.0816	79.7 μs	
			32	0.0720	141 μs	
		IBM SP-2-WD	4	0.0629	15.4 μs	
		Meiko CS-2	4	0.0996	24.3 μs	
			8	0.0909	44.4 μs	

Table 3. Implementation results of segmentation algorithm on image 6 from Copty et al. [22], a binary tool (256 × 256 pixels).

Year	Researcher(s)	Machine	PEs	Time (s)	Work/Pix	Notes
1994	Copty et al. [22]	TMC CM-2	8192	20.538	80.2 ms	Data parallel
			16384	13.955	109 ms	Data parallel
		TMC CM-5	32	77.648	37.9 ms	Data parallel
				12.290	6.00 ms	Message passing, comm1
				7.334	3.58 ms	Message passing, comm2
1995	Bader et al. (this paper)	TMC CM-5	16	0.223	54.4 μs	
			32	0.175	85.5 μs	
		IBM SP-2-TH	4	0.202	12.3 μs	
			8	0.187	22.8 μs	
			16	0.177	43.2 μs	
		IBM SP-2-WD	4	0.194	11.8 μs	
			8	0.176	21.5 μs	
		Meiko CS-2	4	0.414	25.3 μs	
			8	0.274	33.5 μs	
			16	0.204	49.8 μs	
		CRAY T3D	4	0.396	24.2 μs	

The scalability of the segmentation algorithm running on the 512 × 512 Landsat TM band 5 subimage, shown in Figure B.1, is given in Figure 9 for various machine configurations

of the CM-5, CS-2, SP-2, and T3D. For this image the first, second, and third phases of SNF iterate 4, 26, and 34 times, respectively. Also, the 1-Nearest Neighbor task contains 8 iterations.

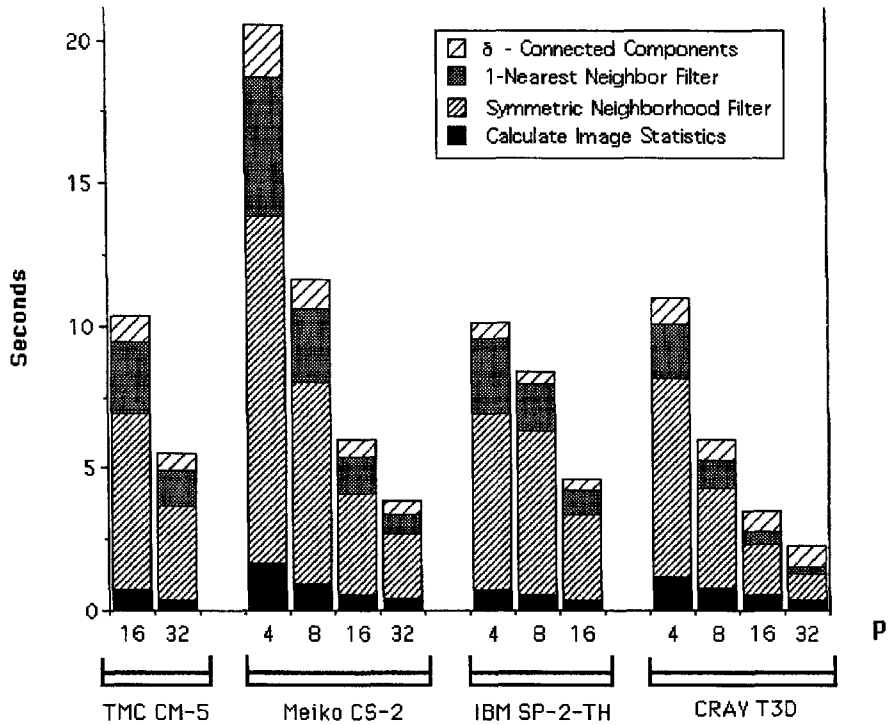


Figure 9. Scalability of the segmentation algorithm for the 512 × 512 Landsat TM band 5 image.

Results are given in Table 4 for a larger 1024 × 1024 subimage of the same view. Note that the SNF and 1-Nearest Neighbor filters are iterative and data-dependent, with timings that ramp down after the initial iteration; thus only the slowest timing for a single iteration is reported. Figure 10 shows the scalability of the segmentation algorithm running on the 1024 × 1024 subimage, with various machine configurations of the CM-5, SP-2, and T3D. For this image the first, second, and third phases of SNF iterate 4, 56, and 47 times, respectively. Also, the 1-Nearest Neighbor task contains 11 iterations. For both the 512 × 512 and 1024 × 1024 images and on each machine, the total execution time for the segmentation process scales with respect to the number of processors.

Table 5 compares the best-known sequential code for SNF to that of the parallel implementation. Again, this test uses the 1024 × 1024 image and performs a total of $4+56+47 = 107$ iterations. The sequential tests are performed on fast workstations dedicated to a single user and reflect only the time spent doing the filter calculations. These empirical results show

Table 4. Segmentation execution time (in ms) for the 1024 × 1024 Landsat TM band 5 image.

Machine	PEs	Decide Noisy	Calc. σ^*	Max. SNF Iter.	Max. 1-NN Iter.	Crop	δ-CC
TMC CM-5	16	576	2347	1300	1275	124	3218
	32	292	1232	657	641	62.4	1963
IBM SP-2-TH	4	963	1875	1185	1288	66.8	1828
	8	563	1142	679	72.9	34.0	1320
	16	287	675	349	377	17.7	885
IBM SP-2-WD	4	958	1795	1130	1272	64.3	1734
	8	554	1063	645	713	32.8	1113
	16	283	628	339	365	17.0	839
	32	169	596	202	212	9.71	721
CRAY T3D	4	918	3880	1460	1209	48.9	3083
	8	324	1021	586	472	24.5	1788
	16	162	1075	298	236	12.4	1287
	32	81.3	712	147	118	6.19	1083
	64	40.6	506	73.7	59.4	3.10	1019

our segmentation algorithm scaling with machine and problem size and exhibiting superior performance on several parallel machines when compared with state-of-the-art sequential platforms.

Table 5. Total SNF execution time (in seconds) for the 1024 × 1024 Landsat TM band 5 image.

Machine	PEs	Time (s) for 107 Iter.
Sun Sparc 10 - Model 40	1	104
Sun Sparc 20 - Model 50	1	83.6
IBM SP-2-TH	1	78.2
DEC AlphaServer 2100 4/275	1	48.1
TMC CM-5	16	35.2
	32	18.5
IBM SP-2-TH	4	30.9
	8	24.4
	16	12.5
CRAY T3D	4	45.3
	8	20.9
	16	10.6
	32	5.35

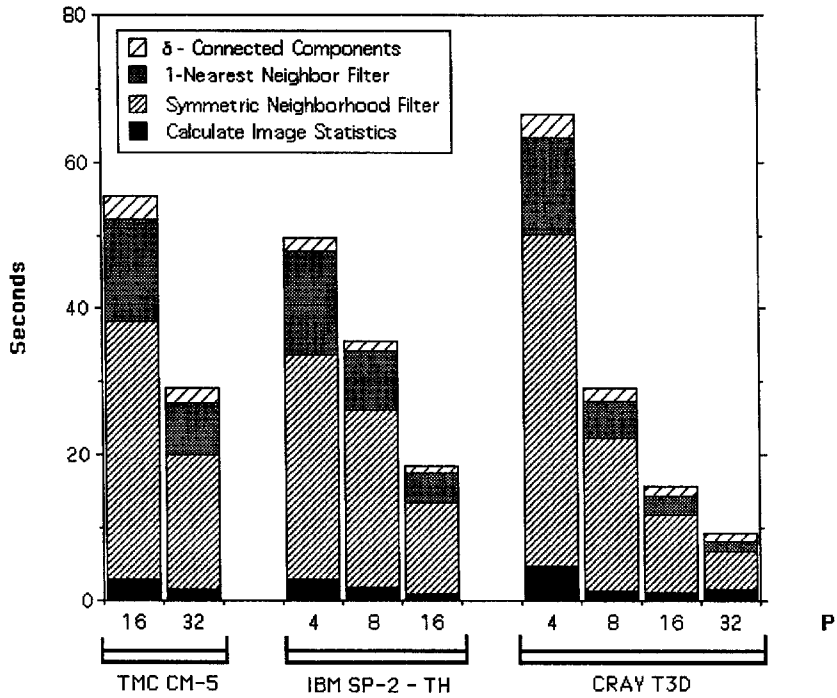


Figure 10. Scalability of the segmentation algorithm for the 1024 × 1024 Landsat TM band 5 image.

7. Implementation Notes

Note that the performance results for the CM-5 are for SPLIT-C (version 1.2) programs linked with the CM-5 CMMD Message Passing Libraries (version 3.2), and IBM SP-2 results use MPL for message passing. The Meiko CS-2 port of SPLIT-C uses the Elan communications libraries. For the CRAY T3D, SPLIT-C is built on top of AC (version 2.6) [17] and SHMEM from Cray Research.

Acknowledgments

D.A. Bader gratefully acknowledges the support by NASA Graduate Student Researcher Fellowship No. NGT-50951. We acknowledge support of J. JáJá in part by NSF grant No. CCR-9103135, and of JáJá, D. Harwood, and L.S. Davis by NSF HPCC/GCAG grant No. BIR-9318183.

We would like to thank the CASTLE/SPLIT-C group at UC Berkeley, especially for the help and encouragement from David Culler, Arvind Krishnamurthy, Lok Tin Liu, Steve Luna, and Rich Martin. Computational support on UC Berkeley's 64-processor TMC CM-5 and 8-processor Intel Paragon was provided by NSF Infrastructure Grant number CDA-8722788. We also thank the Numerical Aerodynamic Simulation Systems Division of NASA's Ames Research Center for use of their 128-processor CM-5 and 160-node IBM SP-2-WD.

We recognize Charles Weems at the University of Massachusetts for providing the DARPA test image suite, and Nawal Copty at Syracuse University for providing additional test images.

Also, Klaus Schauser, Oscar Ibarra, and David Probert of the 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.

Additional thanks goes to Argonne National Labs for allowing the use of their 128-node IBM SP-1 and to the Maui High Performance Computing Center for the 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.

Arvind Krishnamurthy provided additional help with his port of SPLIT-C to the Cray Research T3D [6]. 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 (formerly the Supercomputing Research Center) for writing the parallel compiler AC (version 2.6) [17] on which the T3D port of SPLIT-C has been based.

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/EXPAR`. We encourage other researchers to compare their findings with our results for similar inputs.

Appendix A

Test Images of Artificial Scenes

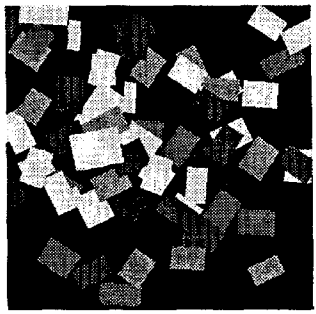


Figure A.1. DARPA II Image Understanding Benchmark test image (512 × 512).

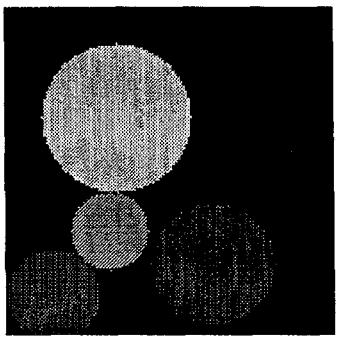


Image 3 (128 × 128)

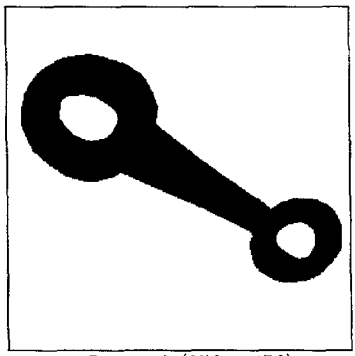


Image 6 (256 × 256)

Figure A.2. Test images from Copty et al. [22].

Appendix B

Test Images of Real Scenes

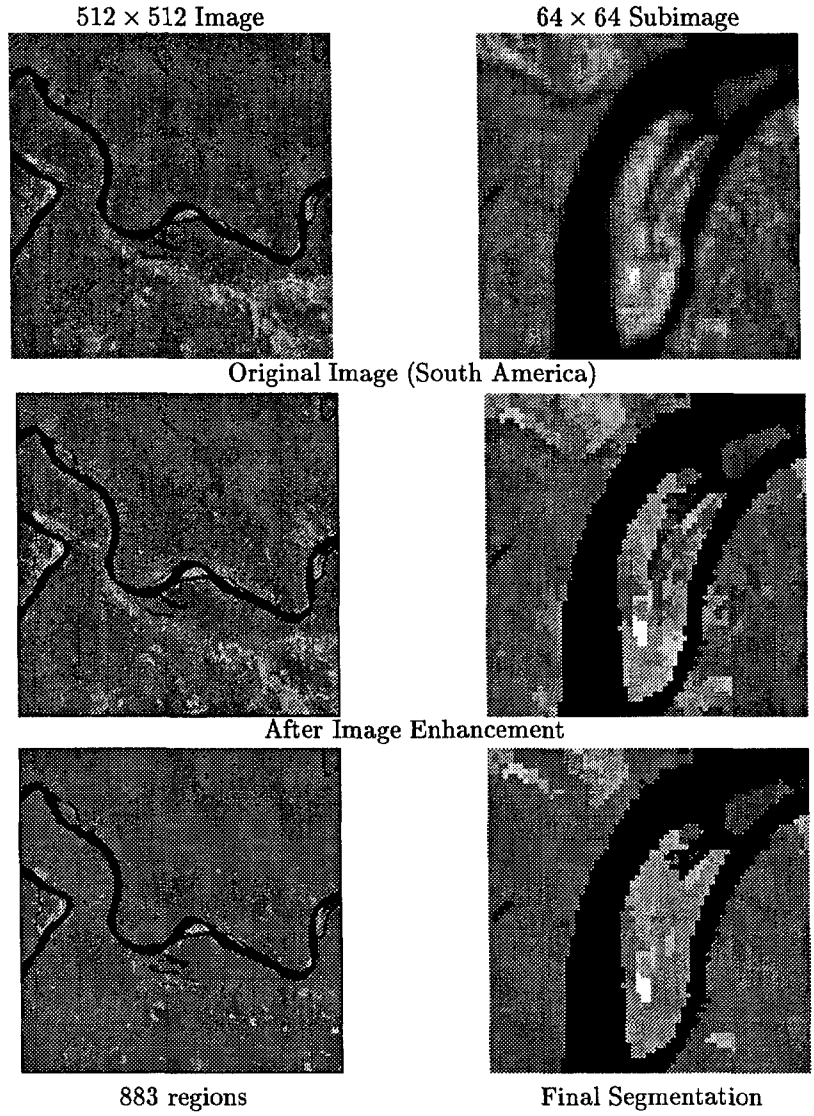


Figure B.1. Landsat TM band 5 images.

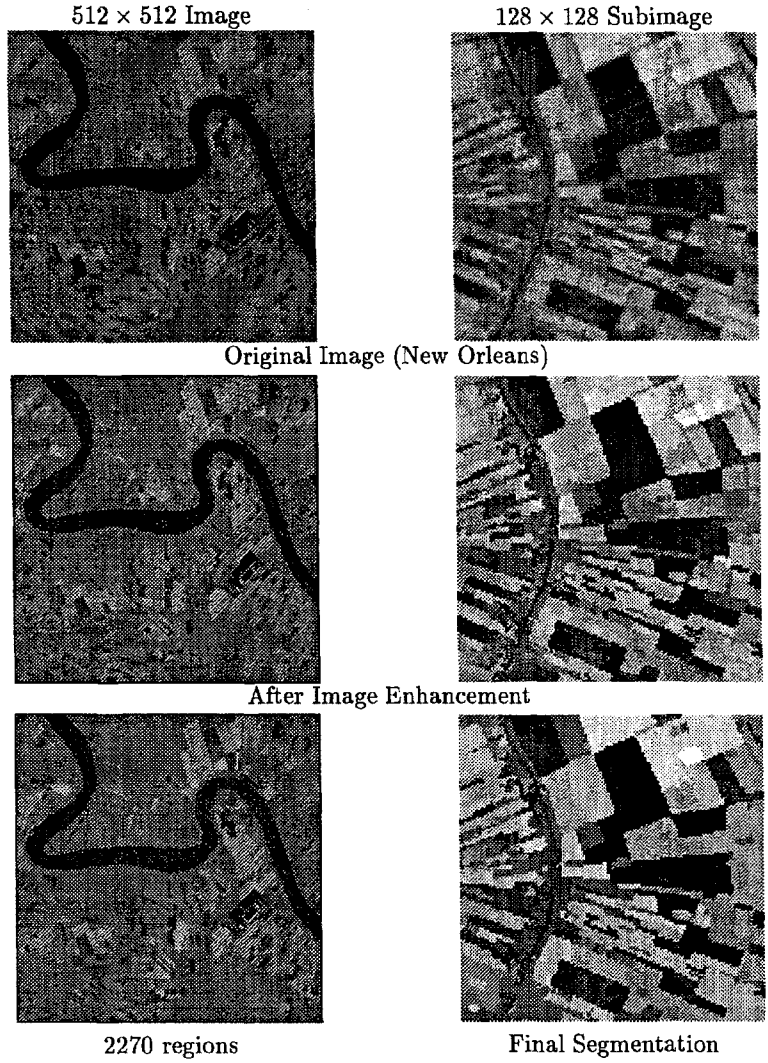


Figure B.2. Landsat TM band 4 images.

Notes

1. Note that throughout this paper $\log x$ will always be the logarithm of x to the base $b = 2$, namely, $\log_2 x$.
2. 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, depending on which sorting method is fastest for the given input size.

References

1. A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model — One step closer towards a realistic model for parallel computation. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, Santa Barbara, Calif., July 1995.
2. 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.
3. H.M. Alnuweiri and V.K. Prasanna Kumar. Efficient image computations on VLSI architectures with reduced hardware. In *Proceedings of the 1987 Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence*, pages 192–199, Seattle, October 1987.
4. M. Annaratone, E. Arnould, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, and J.A. Webb. The Warp computer: Architecture, implementation, and performance. *IEEE Transactions on Computers*, C-36:1523–1538, 1987.
5. J. Apostolakis, P. Coddington, and E. Marinari. New SIMD algorithms for cluster labeling on parallel computers. *International Journal of Modern Physics C*, 4:749, 1993.
6. R.H. Arpaci, D.E. Culler, A. Krishnamurthy, S.G. Steinberg, and K. Yelick. Empirical evaluation of the CRAY-T3D: A compiler perspective. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 320–331, Santa Margherita Ligure, Italy, June 1995.
7. D.A. Bader. On the design and analysis of practical parallel algorithms for combinatorial problems with applications to image processing. Ph.D. thesis, Department of Electrical Engineering, University of Maryland, College Park, April 1996.
8. 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. *Journal of Parallel and Distributed Computing*, 35(2): 173-190, 1996.
9. D.A. Bader and J. JáJá. Parallel algorithms for image histogramming and connected components with an experimental study. In *Fifth ACM SIGPLAN Symposium of Principles and Practice of Parallel Programming*, pages 123–133, Santa Barbara, Calif., July 1995.
10. D.A. Bader and J. JáJá. Practical parallel algorithms for dynamic data redistribution, median finding, and selection. Technical Report CS-TR-3494 and UMIACS-TR-95-74, UMIACS and Electrical Engineering, University of Maryland, College Park, Md., July 1995.
11. D.A. Bader and J. JáJá. Practical parallel algorithms for dynamic data redistribution, median finding, and selection. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 292–301, Honolulu, April 1996.
12. 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.
13. P.K. Biswas, J. Mukherjee, and B.N. Chatterji. Component labeling in pyramid architecture. *Pattern Recognition*, 26(7):1099–1115, 1993.
14. G.E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.
15. S.H. Bokhari and H. Berryman. Complete exchange on a circuit switched mesh. In *Proceedings of Scalable High Performance Computing Conference*, pages 300–306, Williamsburg, Va., April 1992.
16. R.C. Brower, P. Tamayo, and B. York. A parallel multigrid algorithm for percolation clusters. *Journal of Statistical Physics*, 63:73, 1991.
17. W.W. Carlson and J.M. Draper. AC for the T3D. Technical Report SRC-TR-95-141, Center for Computing Sciences, Bowie, Md., February 1995.
18. Y.-L. Chang and X. Li. Adaptive image region-growing. *IEEE Transactions on Image Processing*, 3(6):868–872, 1994.
19. V. Chaudhary and J.K. Aggarwal. On the complexity of parallel image component labeling. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 183–187, August 1991.
20. A. Choudhary and R. Thakur. Evaluation of connected component labeling algorithms on shared and distributed memory multiprocessors. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 362–365, March 1992.
21. 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.

22. 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.
23. 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.
24. D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Portland, Ore., November 1993.
25. 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.
26. F. Dehne and S.E. Hambrusch. Parallel algorithms for determining k-width connectivity in binary images. *Journal of Parallel and Distributed Computing*, 12:12–23, 1991.
27. H. Derin and C.-S. Won. A parallel image segmentation algorithm using relaxation with varying neighborhoods and its mapping to array processors. *Computer Vision, Graphics, and Image Processing*, 40:54–78, 1987.
28. M.B. Dillencourt, H. Samet, and M. Tamminen. Connected component labeling of binary images. Technical Report CS-TR-2303, Computer Science Department, University of Maryland, College Park, Md., August 1989.
29. H. Embrechts, D. Roose, and P. Wambacq. Component labelling on a MIMD multiprocessor. *CVGIP: Image Understanding*, 57(2):155–165, March 1993.
30. 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, S.C., April 1990.
31. R. Goldenberg, W.C. Lau, A. She, and A.M. Waxman. Progress on the prototype PIPE. In *Proceedings of the 1987 Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence*, pages 67–74, Seattle, October 1987.
32. S. Hambrusch, X. He, and R. Miller. Parallel algorithms for gray-scale digitized picture component labeling on mesh-connected computer. *Journal of Parallel and Distributed Computing*, 20:56–68, 1994.
33. F. Hameed, S.E. Hambrusch, A.A. Khokhar, and J.N. Patel. Contour ranking on coarse grained machines: A case study for low-level vision computations. Technical report, Purdue University, West Lafayette, Ind., November 1994.
34. Y. Han and R.A. Wagner. An efficient and fast parallel-connected component algorithm. *Journal of the ACM*, 37(3):626–642, 1990.
35. R.M. Haralick and L.G. Shapiro. Image segmentation techniques. *Computer Vision, Graphics, and Image Processing*, 29:100–132, 1985.
36. D. Harwood, M. Subbarao, H. Hakalahti, and L.S. Davis. A new class of edge-preserving smoothing filters. *Pattern Recognition Letters*, 6:155–162, 1987.
37. D.S. Hirschberg, A.K. Chandra, and D.V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.
38. J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, New York, 1992.
39. 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. To appear in *IEEE Transactions on Parallel and Distributed Systems*.
40. 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)
41. T. Kanade and J.A. Webb. Parallel vision algorithm design and implementation 1988 end of year report. Technical Report CMU-RI-TR-89-23, The Robotics Institute, Carnegie Mellon University, August 1989.
42. 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, Calif., April 1991.
43. H.T. Kung and J.A. Webb. Mapping image processing operations onto a linear systolic machine. *Distributed Computing*, 1:246–257, 1986.
44. J.J. Little, G. Blelloch, and T. Cass. Parallel algorithms for computer vision on the Connection Machine. In *Image Understanding Workshop*, pages 628–638, Los Angeles, February 1987.

45. 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.
46. P.J. Narayanan. Effective use of SIMD machines for image analysis. Ph.D. thesis, Department of Computer Science, University of Maryland, College Park, Md., 1992.
47. P.J. Narayanan and L.S. Davis. Replicated data algorithms in image processing. Technical Report CAR-TR-536/CS-TR-2614, Center for Automation Research, University of Maryland, College Park, Md., February 1991.
48. M. Pietikäinen, T. Seppänen, and P. Alapuranen. A hybrid computer architecture for machine vision. In *Proceedings of the 10th International Conference on Pattern Recognition, Volume 2*, pages 426–431, Atlantic City, N.J., June 1990.
49. A. Rosenfeld. A report on the DARPA Image Understanding Architectures Workshop. In *Proceedings of the 1987 Image Understanding Workshop*, pages 298–302, 1987.
50. H. Shi and G.X. Ritter. A fast algorithm for image component labeling with local operators on mesh connected computers. *Journal of Parallel and Distributed Computing*, 23:455–461, 1994.
51. A.D. Sokal. New numerical algorithms for critical phenomena (Multi-grid methods and all that). In *Proceedings of the International Conference on Lattice Field Theory*, Tallahassee, Fla., October 1990. (*Nucl. Phys. B (Proc. Suppl.)*, 20:55, 1991.).
52. D. Stauffer. *Introduction to Percolation Theory*. Taylor and Francis, Philadelphia, 1985.
53. M.H. Sunwoo, B.S. Baroody, and J.K. Aggarwal. A parallel algorithm for region labeling. In *Proceedings of the 1987 Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence*, pages 27–34, Seattle, October 1987.
54. J.C. Tilton and S.C. Cox. Segmentation of remotely sensed data using parallel region growing. In *Ninth International Symposium on Machine Processing of Remotely Sensed Data*, pages 130–137, West Lafayette, Ind., June 1983.
55. L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
56. R.S. Wallace, J.A. Webb, and I.-C. Wu. Machine-independent image processing: Performance of Apply on diverse architectures. *Computer Vision, Graphics, and Image Processing*, 48:265–276, 1989.
57. J.A. Webb. Architecture-independent global image processing. In *Proceedings of the 10th International Conference on Pattern Recognition, Volume 2*, pages 623–628, Atlantic City, N.J., June 1990.
58. 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, Mass., April 1988.
59. 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.
60. C. Weems, E. Riseman, A. Hanson, and A. Rosenfeld. The DARPA Image Understanding Benchmark for parallel computers. *Journal of Parallel and Distributed Computing*, 11:1–24, 1991.
61. T. Westman, D. Harwood, T. Laitinen, and M. Pietikäinen. Color segmentation by hierarchical connected components analysis with image enhancement by Symmetric Neighborhood Filters. In *Proceedings of the 10th International Conference on Pattern Recognition*, pages 796–802, Atlantic City, N.J., June 1990.
62. M. Willebeek-LeMair and A.P. Reeves. Region growing on a highly parallel mesh-connected SIMD computer. In *The 2nd Symposium on the Frontiers of Massively Parallel Computations*, pages 93–100, Fairfax, Va., October 1988.
63. M. Willebeek-LeMair and A.P. Reeves. Solving nonuniform problems on SIMD computers: Case study on region growing. *Journal of Parallel and Distributed Computing*, 8:135–149, 1990.
64. R. Williams. Parallel load balancing for parallel applications. Technical Report CCSF-50, Concurrent Supercomputing Facilities, California Institute of Technology, November 1994.
65. W. Yong and M.L. Brady. Efficient component labeling on SIMD mesh processors. In *Proceedings of the International Conference on Parallel Processing*, pages III–31 — III–34, August 1994.
66. S.W. Zucker. Region growing: Childhood and adolescence. *Computer Graphics and Image Processing*, 5:382–399, 1976.