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

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

Joseph JáJá‡
joseph@umiacs.umd.edu

David Harwood†
harwood@umiacs.umd.edu

Larry S. Davis§
lsd@umiacs.umd.edu

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

May 12, 1995

Abstract

This paper presents efficient and portable implementations of a useful image enhancement process, the Symmetric Neighborhood Filter (SNF), and an image segmentation technique which makes use of the SNF and a variant of the conventional connected components algorithm which we call δ-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. The image segmentation algorithm makes use of an efficient connected components algorithm which uses 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. More efficient implementations of Split-C will likely result in even faster execution times.

Keywords: Parallel Algorithms, Image Processing, Region Growing, Image Enhancement, Image Segmentation, Symmetric Neighborhood Filter, Connected Components, Parallel Performance.

*Also affiliated with Department of Electrical Engineering. The support by NASA Graduate Student Researcher Fellowship No. NGT-50951 is gratefully acknowledged.

†Supported by NSF HPCC/GCAG grant No. BIR-9318183.

‡Also affiliated with Department of Electrical Engineering. Supported in part by NSF grant No. CCR-9103135 and NSF HPCC/GCAG grant No. BIR-9318183.

§Also affiliated with the Department of Computer Science and the Center for Automation Research; supported by NSF HPCC/GCAG grant No. BIR-9318183.

1 Problem Overview

Given an $n \times n$ image with k grey levels on a p processor machine ($p \le n^2$), 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 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. [10], [13], [21], [41], [44]). 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 algorithm makes use of an efficient and fast parallel connected components algorithm which uses a novel approach for merging. For a detailed theoretical and experimental analysis of this algorithm, please refer to [4].

In real images, natural regions have significant variability in grey level. Noise, introduced from the scanning of the real scene into the digital domain, will cause single pixels outliers. Also, lighting changes can cause a gradient of grey 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) [22], that smooths out the interior pixels of a region to a near-homogeneous level. Also, due to relative motion of the camera and the scene, as well as aperture effects, edges or regions are usually blurred so that the transition in grey 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 which can detect 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 which performs well for both edge-sharpening and region smoothing. It is an iterative filter which also can be tuned to retain thin image structures corresponding, e.g., to rivers, roads, etc. A variety of SNF operators have been studied, and we chose a single parameter version which 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. [19], [30], [31], [32], [37]). For example, [38] 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 Figure 10. 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 grey 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. [17], [28], [29], [36], [41], [42]).

Section 2 addresses the algorithmic model and various primitive operations we use to analyze the algorithms. Section 3 discusses the test images, as well as the data layout on the parallel machines. Our segmentation process overview which includes discussion of SNF and 1-Nearest Neighbor filters, and the δ-Connected Components algorithm, is given in Section 4. Finally, Sections 5 and 6 describe the parallel implementations of the Symmetric Neighborhood Filter algorithm and δ-Connected Components, respectively, and present algorithmic analyses and empirical results.

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 shared memory model algorithms are written in Split-C [14], 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 ([25], [26]) 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 will be 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 overall 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.

2.1 Matrix Transposition

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, 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, $(1 \leq i \leq p-1)$, each processor P_t prefetches the appropriate block of $\frac{q}{p}$ elements from processor $P_{(t+i) \bmod p}$. The BDM algorithm and analysis for the matrix transpose data movement is given in [4] and is similar to that of the LogP model [16]. This matrix transpose algorithm has the following complexity:

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

2.2 Broadcasting

Another useful data movement primitive is broadcasting. An efficient BDM algorithm is given [4], [25] which takes q elements on a single processor and broadcasts them to the other $p-1$ processors using just two matrix transpositions.

An efficient algorithm to broadcast q elements from a single processor to p processors is based on matrix transposition, where q is assumed to be larger than p. Processor 0 holds the q elements to be broadcast in the first column of matrix A. We compute the matrix transpose of A, thus, giving every processor $\frac{q}{p}$ elements. Each processor then locally rearranges the data so that an additional matrix transpose will result in each processor holding a copy of all the q elements in its column of A [25].

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

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

Performance analysis given in [4] 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.

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 Appendix A are divided into two categories, artificial and real. The artificial images, given in Figures 6 and 7, range in size from 128×128 to 512×512 pixels. We use Landsat satellite data to represent real images; Figure 8 is from band 5 of a South American scene, and Figure 9 is band 4 taken from a view of New Orleans. Both of these images are 256 grey 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 filter, and finally, δ-Connected Components. See Figure 1 for a dataflow diagram of the complete segmentation process.

4.1 Symmetric Neighborhood Filter

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

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, i.e. N $\sim$ S, W $\sim$ E, NW $\sim$ SE, and NE $\sim$ SW; see Figure 2 for diagram of a 3×3

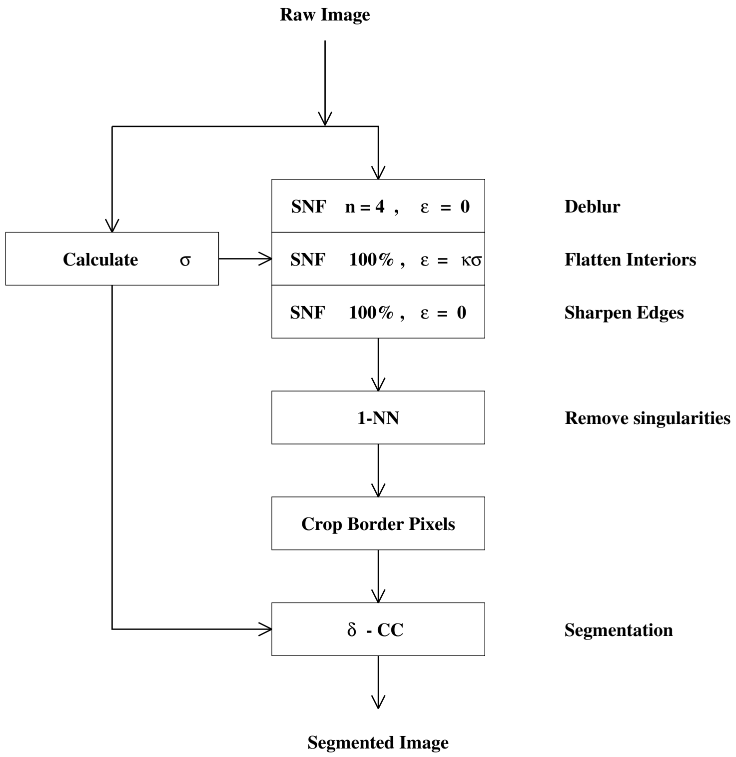

Figure 1: Segmentation Process

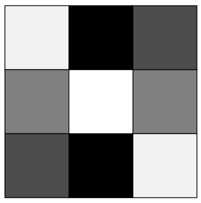

Figure 2: Symmetric Pairs of Pixels

neighborhood centered around a pixel, with the symmetric pairs colored the same. Essentially, the one pixel in each pair closest to the center in grey level is selected, but only if its intensity is within ϵ of the center pixel, otherwise, the center pixel's value is used. If the center pixel is equidistant from the pair, or is a local minima or maxima, its value is selected instead. The collection of four selected pixels are averaged together, and finally, the center pixel is replaced by the mean of this average and the center pixel's current grey level. This latter average is similar to that of a damped gradient descent which yields a faster convergence.

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 non-border pixel in the image. See [5] for a parallel median algorithm. To flatten the interior of regions, SNF is iterated 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.

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. The 1-Nearest Neighbor filter removes single pixel outliers by replacing each pixel in the image with the mean of its value and the grey level value of an adjacent pixel which is closest to its current value. Note that one application of the 1-Nearest Neighbor filter may cause small neighborhoods of pixels to oscillate. 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 which 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.

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. [1], [12], [13], [18], [20], [23], [24]). The task of connected component labeling is cited as a fundamental computer vision problem in the DARPA Image Understanding benchmarks ([33], [39]), and also can be applied to several computational physics problems such as percolation ([8], [35]) and various cluster Monte Carlo algorithms for computing the spin models of magnets such as the two-dimensional Ising spin model ([3], [6], [34]). 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 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 grey 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.

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 8 shows a subimage from band 5, an image from South America, and Figure 9 is taken from band 4 of New Orleans data. These images have 256 grey levels and also have post-processing 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 8 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 8 and 9.

5 Symmetric Neighborhood Filter - Parallel Implementation

Most common enhancement filters will smooth the interior of regions at the cost of 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 edge sharpening and region flattening. The SNF is a convergent filter which can be run for a predetermined number of iterations, or until a percentage of the image pixels are fixed in grey level. A variety of SNF operators have been studied, and we chose a single parameter version which has been shown to perform well. Previous parallel implementations of the SNF have been based around special purpose image processing platforms, including data parallel SIMD machines such as the TMC CM-2 and the MasPar MP-1 ([30] and [31]), video-rate VLSI implementations ([32]), pipelined computers

([19]), and systolic linear arrays such as the Warp ([2], [37], and [38]).

A useful data movement needed for this 3×3 local SNF filter is the fetching of tile-based **ghost cells** ([15], [43]) which contain shadow copies of neighboring tiles' pixel borders. These ghost cells are used in the selection process when recalculating our tile's border. Suppose each tile of the image allocated to a processor is $q \times r$ pixels. We have four ghost cell arrays, **ghostN** and **ghostS** which hold r pixels each, and **ghostW** and **ghostE** which 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 3.

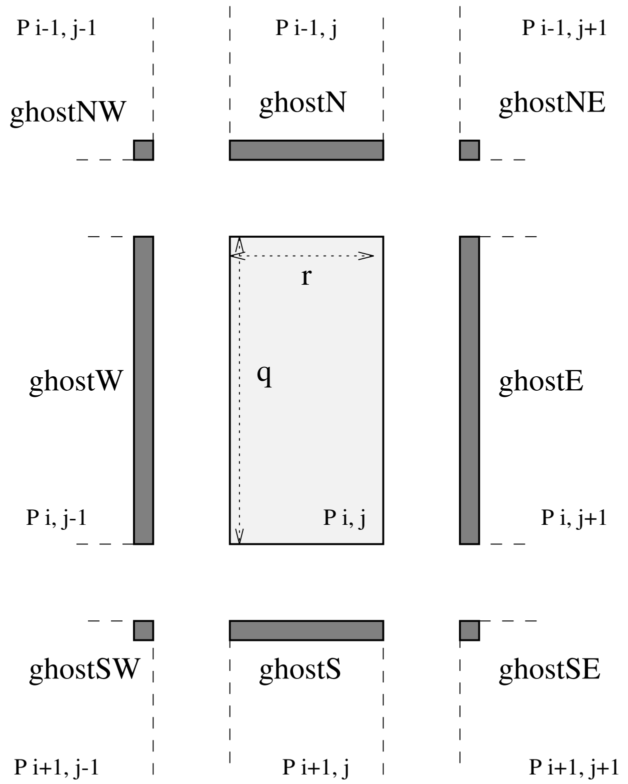

Figure 3: An example of Ghost Cells

The analysis for the prefetching of ghost cells is simple. We can divide the prefetching into eight separate data movements, one for each direction. Since each movement is a permutation, i.e. it has a unique source and destination, it can be routed with little or no contention. The prefetching of the north and south ghost cell arrays each take $T_{comm}(n,p) \le \tau + r$, the east and west ghost cell arrays each take $T_{comm}(n,p) \le \tau + q$, and the diagonal four ghost cells each take $T_{comm}(n,p) \le \tau + 1$. Thus, the entire ghost cell prefetching takes

$$\begin{cases} T_{comm}(n,p) & \le & 8\tau + 4\frac{n}{\sqrt{p}} + 4; \\ T_{comp}(n,p) & = & O\left(\frac{n}{\sqrt{p}}\right). \end{cases} \tag{3}$$

A second data movement needed for SNF is the **reduction** operation. Each processor i has a data value, Z_i, and we need the value of $Z_0 \oplus Z_1 \oplus \ldots \oplus Z_{p-1}$, where $\oplus$ is any associative operator. Parallel computers can handle this efficiently [7], and SPLIT-C implements this as a primitive library function. A simple algorithm consists of $p-1$ rounds that can be pipelined [25]. Each processor P_i initializes a local sum to Z_i. During round r, each processors then reads $Z_{(i+r) \bmod p}$, for $1 \leq r \leq p-1$, and adds this value to the local sum. Since these rounds can be realized with $p-1$ pipelined prefetch read operations, the resulting complexity is

$$\begin{cases} T_{comm}(n,p) & \leq \quad \tau + p - 1; \\ T_{comp}(n,p) & = \quad O(p). \end{cases} \tag{4}$$

An SPMD algorithm for an iteration of SNF on Processor i:

Algorithm 1 *Symmetric Neighborhood Filter*
Block Distributed Memory Model Algorithm.

Input:
{ 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.
{ ϵ } 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. **Reduce** $f = \sum_{i=0}^{p-1} f_i$.

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 (3) and (4), 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

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

Figure 10 in Appendix B 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 (3), and the update of pixels takes $O\left(\frac{n^2}{p}\right)$. Therefore, the 1-Nearest Neighbor algorithm has complexities

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

6 δ-Connected Components of Greyscale 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, 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[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; for a complete description, see [4].

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 4 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

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

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.

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.

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 adjacent to each other and differ by no greater than δ in grey 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.

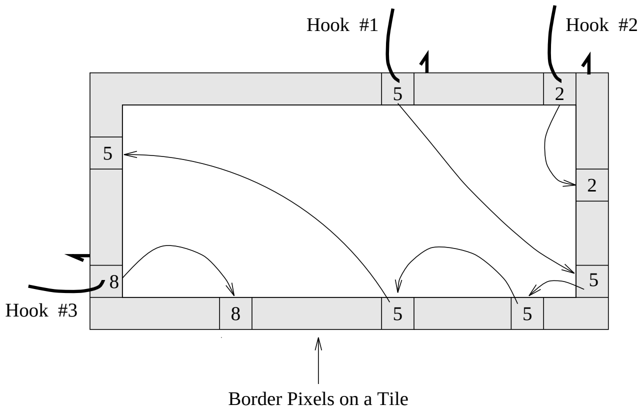

Figure 4: An example of Tile Hooks

At the conclusion of each of the $\log p$ merging steps, only the labels of pixels on the border of each tile are updated. There is no need to relabel interior pixels since they are not used in the merging

[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, using whichever sorting method is fastest for the given input size.

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 [4]

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

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 I shows some previous running times for parallel implementations of connected components on the "DARPA II Image" given in Figure 6. 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 ([13]). Note that this implementation uses data parallel Fortran on the TMC CM-2 and CM-5 machines, and lower-level implementations on the CM-5 using Fortran with several message passing schemes. For example, Figure 7 shows two of the more difficult images from [13] which are segmented by region growing. Image 3 is a 256-grey level 128×128 image, containing seven homogeneous circles. Image 6 is a binary 256×256 image of a tool. Tables II and III 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.

Execution timings for segmentation of the 512×512 Landsat TM band 5 subimage, shown in Figure 8, are given in Table IV. Corresponding results are given in Table V 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

Year	Researcher(s)	Machine	PE's	Time	work/pix	Notes
1989	Kanade and Webb [27]	Warp	10	4.34 s	166 μs	shrink/expand
1989	Weems, Riseman, Hanson, and Rosenfeld [40]	Alliant FX-80	8	7.225 s	220 μs	
		Sequent Symmetry 81	8	15.12 s	461 μs	
		Warp	10	3.98 s	152 μs	
		TMC CM-2	32768	140 ms	547 μs	
1992	Choudhary and Thakur [11]	Intel iPSC/2	32	1.914 s	234 μs	multi-dim. divide & conquer (partitioned input)
				1.649 s	201 μs	multi-dim. divide & conquer (complete im./PE)
				2.290 s	280 μs	multi-dim. divide & conquer (cmplt. + collect. comm.)
		Intel iPSC/860	32	1.351 s	165 μs	multi-dim. divide & conquer (partitioned input)
				1.031 s	126 μs	multi-dim. divide & conquer (complete im./PE)
				947 ms	116 μs	multi-dim. divide & conquer (cmplt. + collect. comm.)
		Encore Multimax	16	521 ms	31.8 μs	multi-dim. divide & conquer (partitioned input)
1994	Choudhary and Thakur [12]	TMC CM-5	32	456 ms	55.7 μs	multi-dim. divide & conquer (partitioned input)
				398 ms	48.6 μs	multi-dim. divide & conquer (complete im./PE)
				452 ms	55.2 μs	multi-dim. divide & conquer (cmplt. + collect. comm.)
1994	Bader and JáJá [4]	TMC CM-5	32	368 ms	44.9 μs	
		IBM SP-1	4	370 ms	5.65 μs	
		IBM SP-2-WD	4	243 ms	3.71 μs	
		Meiko CS-2	2	809 ms	6.17 μs	
			32	301 ms	36.7 μs	
1995	Bader et al. (this paper)	IBM SP-2-TH	4	260 ms	3.97 μs	
			8	257 ms	7.84 μs	
			16	285 ms	17.4 μs	
		IBM SP-2-WD	4	245 ms	3.74 μs	
			8	238 ms	7.26 μs	
			16	262 ms	16.0 μs	
		TMC CM-5	16	474 ms	28.9 μs	
		Meiko CS-2	4	627 ms	9.57 μs	
			8	393 ms	12.0 μs	
			16	351 ms	21.4 μs	
			32	317 ms	38.7 μs	
		Cray T3D	2	472 ms	3.60 μs	
			4	470 ms	7.17 μs	
			8	479 ms	14.6 μs	

Table I: Implementation Results of Parallel Connected Components of DARPA II Image (512×512)

a single iteration is reported. Figure 5 shows 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. Table VI compares the best-known sequential code for SNF to that of the parallel implementation. Again, this test uses the 1024×1024 image, and iterates with the counts specified above. 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 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.

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) [9] and SHMEM from Cray Research.

Year	Researcher(s)	Machine	PE's	Time	work/pix	Notes
1994	Copty et al. [13]	TMC CM-2	8192	13.911 s	217 ms	data parallel
			16384	9.650 s	302 ms	data parallel
		TMC CM-5	32	42.931 s	83.9 ms	data parallel
				9.567 s	18.5 ms	message passing, comm1
				5.537 s	10.8 ms	message passing, comm2
1995	Bader et al. (this paper)	TMC CM-5	16	81.6 ms	79.7 μs	
			32	72.0 ms	141 μs	
		IBM SP-2-WD	4	62.9 ms	15.4 μs	
			8	76.0 ms	37.1 μs	
		Meiko CS-2	4	99.6 ms	24.3 μs	
			8	90.9 ms	44.4 μs	
			16	88.8 ms	86.7 μs	

Table II: Implementation Results of Segmentation Algorithm on Image 3 from [13], seven grey circles (128×128)

The source code for the parallel algorithms presented in this paper is available for distribution to interested parties.

8 Acknowledgements

We would like to thank the UMIACS parallel systems staff for their help and machine maintenance while developing this research on the 16-node IBM SP-2 and 32-processor UMIACS CM-5, and the CASTLE group at UC Berkeley, especially the help and encouragement from Arvind Krishnamurthy, Lok Tin Liu, David Culler, 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 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 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.

Additional thanks goes to Argonne National Labs for allowing use of their 128-node IBM SP-1, and to 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, Oscar Ibarra, 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.

Year	Researcher(s)	Machine	PE's	Time	work/pix	Notes
1994	Copty et al. [13]	TMC CM-2	8192	20.538 s	80.2 ms	data parallel
			16384	13.955 s	109 ms	data parallel
		TMC CM-5	32	77.648 s	37.9 ms	data parallel
				12.290 s	6.00 ms	message passing, comm1
				7.334 s	3.58 ms	message passing, comm2
1995	Bader et al. (this paper)	TMC CM-5	16	223 ms	54.4 μs	
			32	175 ms	85.5 μs	
		IBM SP-2-TH	4	202 ms	12.3 μs	
			8	187 ms	22.8 μs	
			16	177 ms	43.2 μs	
		IBM SP-2-WD	4	194 ms	11.8 μs	
			8	176 ms	21.5 μs	
			16	164 ms	40.0 μs	
		Meiko CS-2	4	414 ms	25.3 μs	
			8	274 ms	33.5 μs	
			16	204 ms	49.8 μs	
			32	193 ms	94.2 μs	
		Cray T3D	4	396 ms	24.2 μs	
			8	443 ms	54.1 μs	

Table III: Implementation Results of Segmentation Algorithm on Image 6 from [13], a binary tool (256×256)

Arvind Krishnamurthy provided additional help with his port of Split-C to the Cray Research T3D. The Jet Propulsion Lab/Caltech 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 Supercomputing Research Center for writing the parallel compiler AC on which the T3D port of Split-C has been based.

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

Machine	PE's	Decide Noisy	Calc. σ	Max. SNF iter.	Max. 1-NN iter.	Crop	δ-CC
TMC CM-5	16	145	605	318	317	30.3	911
	32	74.0	307	162	161	15.3	575
IBM SP-2-TH	4	253	496	303	339	17.0	537
	8	170	347	194	215	9.13	401
	16	88.9	266	101	118	5.24	351
IBM SP-2-WD	4	251	466	301	339	16.5	515
	8	165	322	190	205	8.66	382
	16	87.8	255	96.7	108	4.88	333
	32	59.7	348	66.5	72.2	3.16	322
Meiko CS-2	4	294	1335	681	615	71.7	1800
	8	169	772	361	323	36.2	1057
	16	86.1	488	186	167	17.8	640
	32	52.7	376	177	111	9.05	488
Cray T3D	4	161	1012	296	235	12.3	932
	8	81.1	697	147	118	6.16	757
	16	40.6	496	73.6	59.4	3.10	706
	32	20.6	353	36.9	29.5	1.59	734

Table IV: Segmentation Execution Time (in ms) for 512×512 band 5 image

Machine	PE's	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

Table V: Segmentation Execution Time (in ms) for 1024×1024 band 5 image

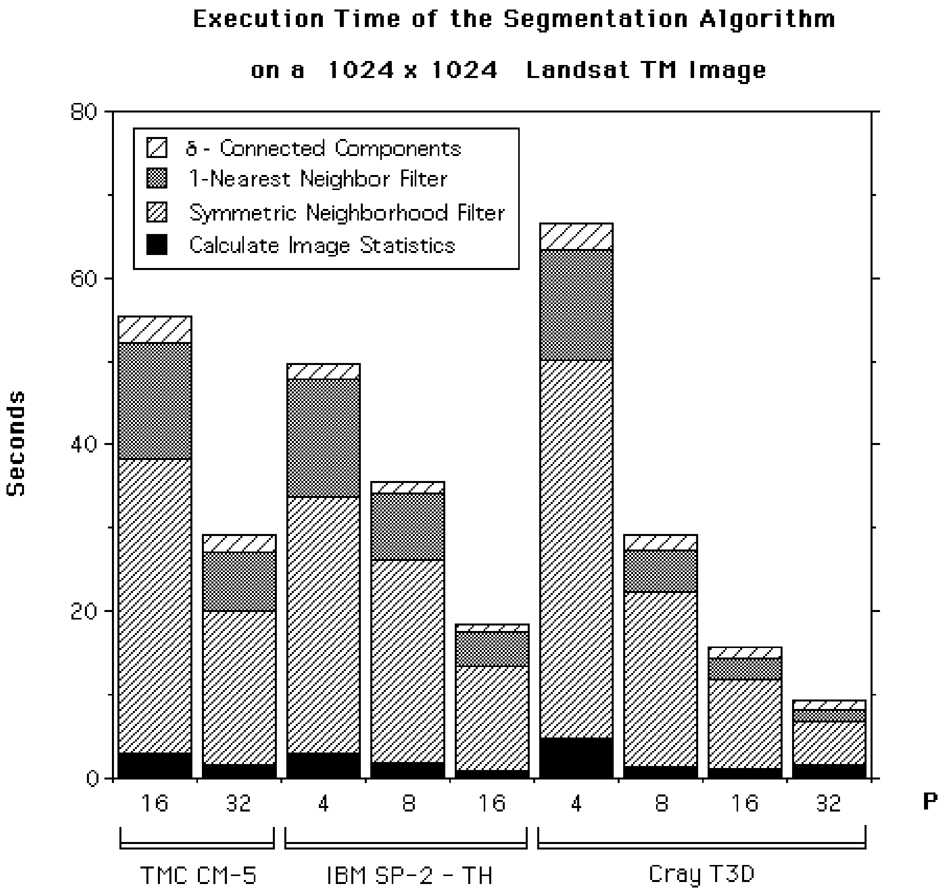

Figure 5: Scalability of the Segmentation Algorithm

Machine	PE's	Time (sec.) 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

Table VI: Total SNF Execution Time (in seconds) for 1024 $\times$ 1024 band 5 image

A Test Images

A.1 Artificial Scenes

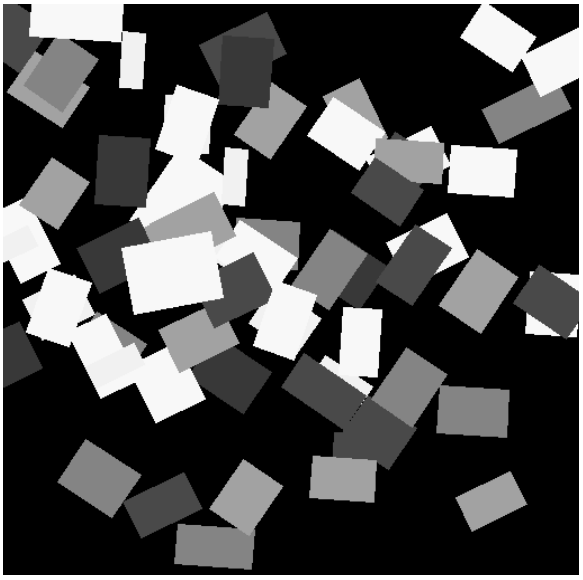

Figure 6: DARPA II Image Understanding Benchmark Test Image (512×512)

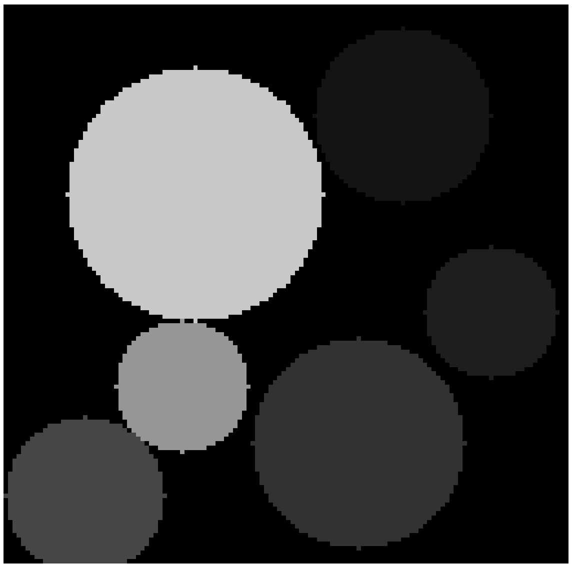

Image 3 (128×128)

Image 6 (256×256)

Figure 7: Test Images from [13]

A.2 Real Scenes

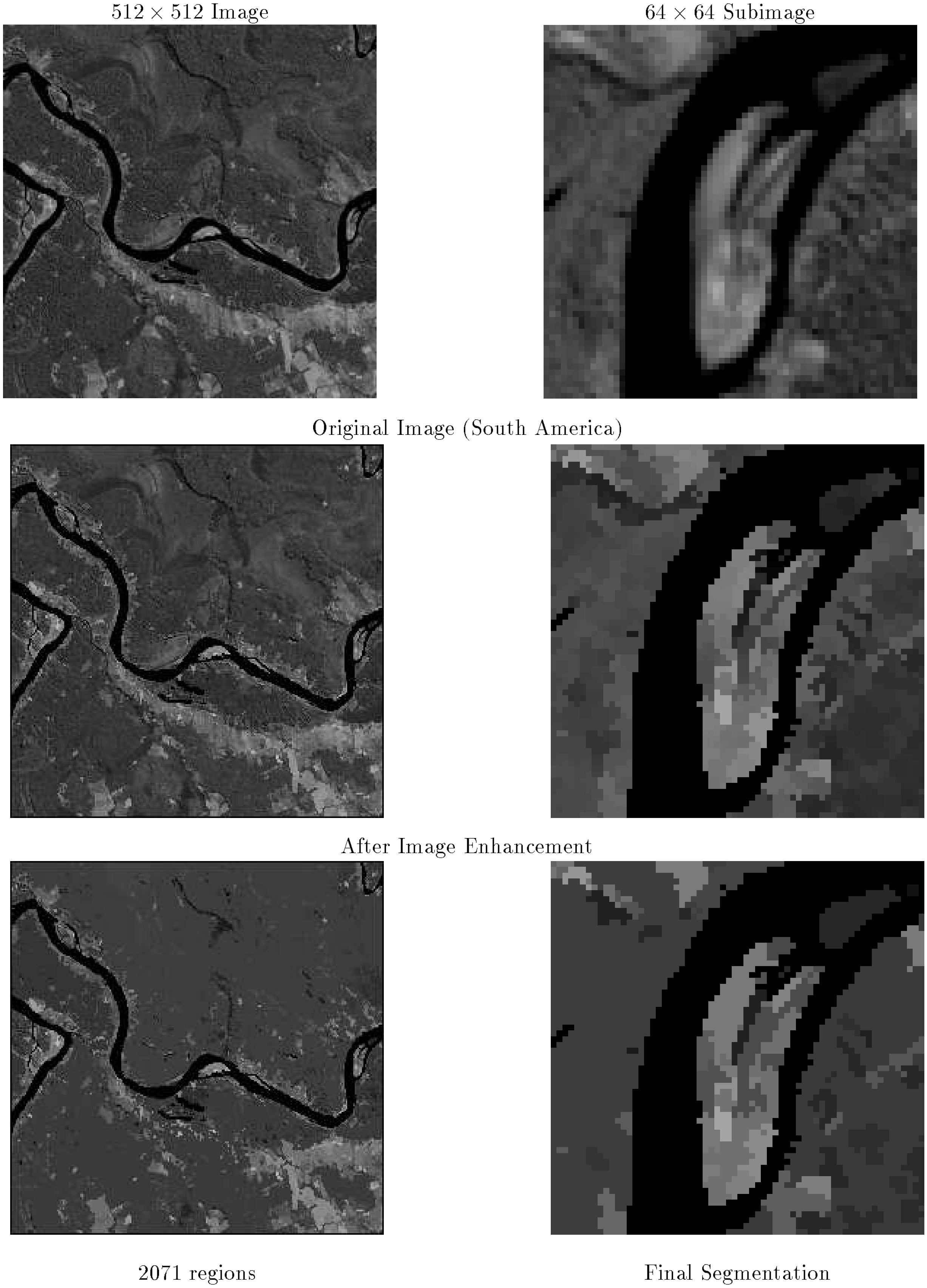

Figure 8: Landsat TM Band 5 Images

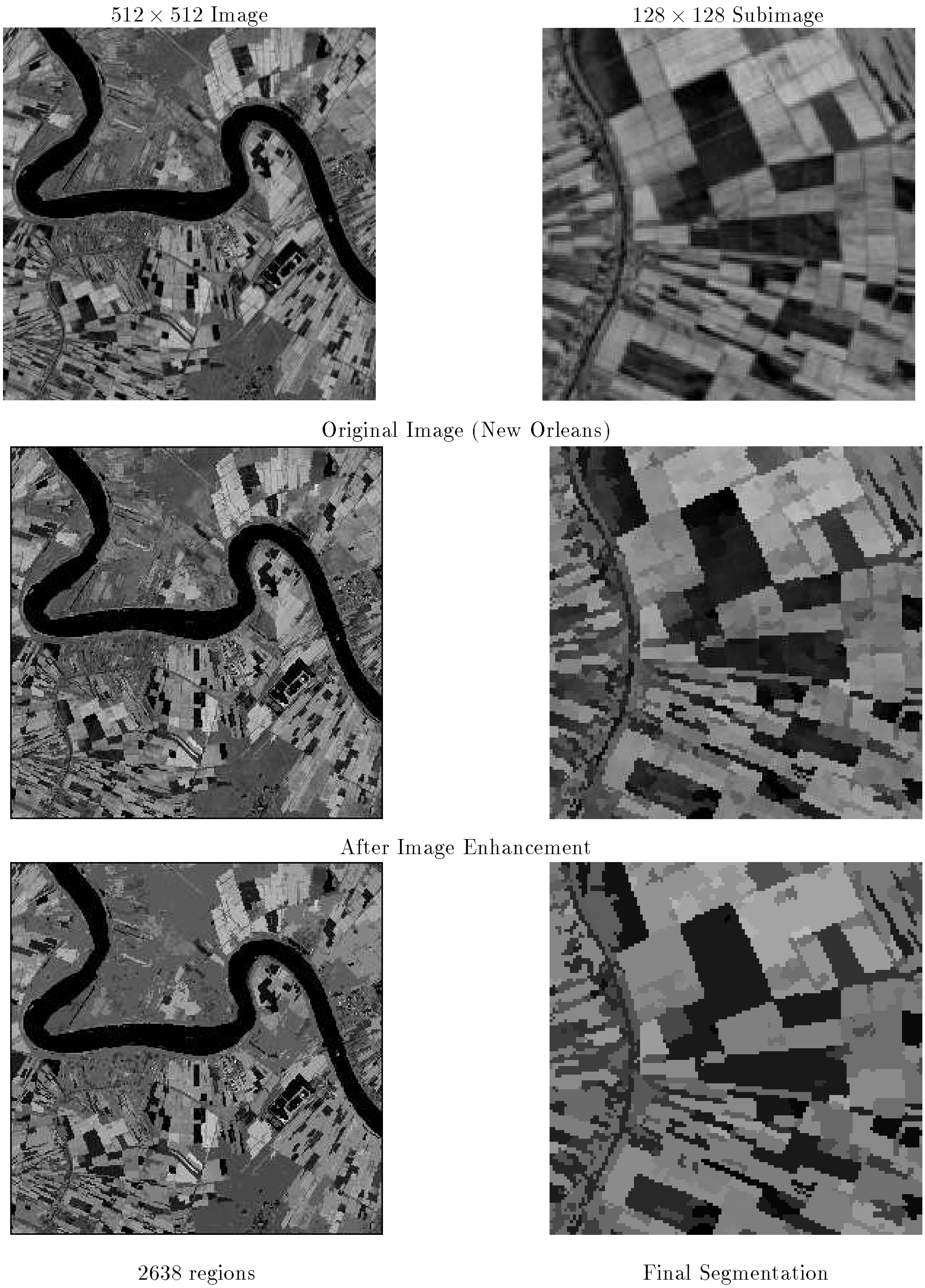

Figure 9: Landsat TM Band 4 Images

B Convergence and Execution Time for Band 5 Image

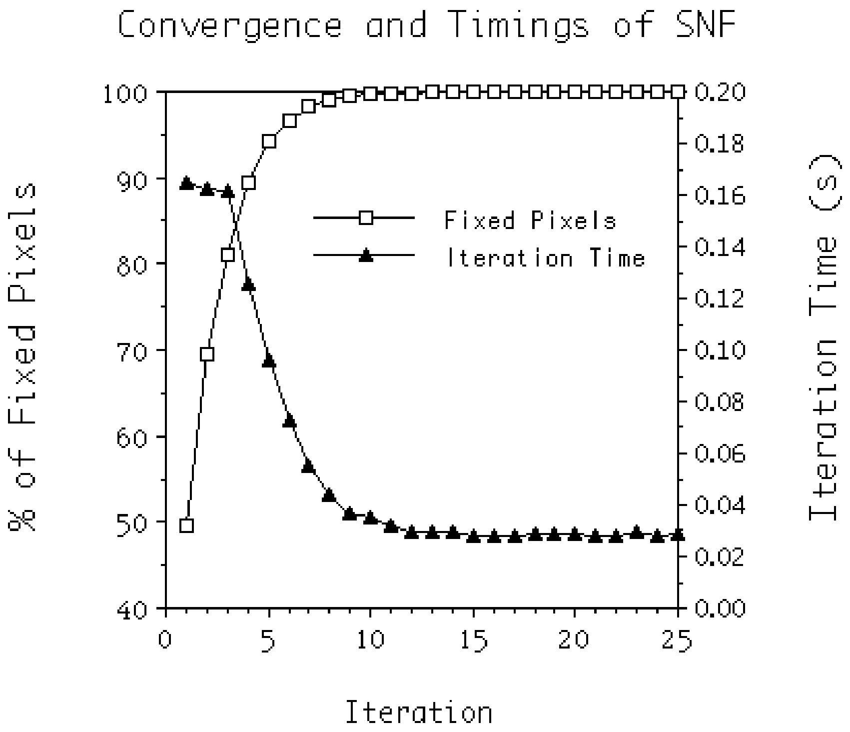

Figure 10: SNF Statistics for a 512 $\times$ 512 Image on a 32-processor CM-5

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] 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.

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

[4] 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. To be presented at the Fifth ACM SIGPLAN Symposium of Principles and Practice of Parallel Programming, Santa Barbara, CA, July 1995.

[5] D. A. Bader and J. JáJá. A Practical Parallel Algorithm for Median Finding, Selection, and Load Balancing. In preparation., 1995.

[6] 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.

[7] G.E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

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

[9] W.W. Carlson and J.M. Draper. AC for the T3D. Technical Report SRC-TR-95-141, Supercomputing Research Center, Bowie, MD, February 1995.

[10] Y.-L. Chang and X. Li. Adaptive Image Region-Growing. *IEEE Transactions on Image Processing*, 3(6):868–872, 1994.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, S. Luna, T. von Eicken, and K. Yelick. Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Portland, OR, November 1993.

[16] 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.

[17] 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.

[18] 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, August 1989.

[19] 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, WA, October 1987.

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

[21] R.M. Haralick and L.G. Shapiro. Image Processing Techniques. *Computer Vision, Graphics, and Image Processing*, 29:100–132, 1985.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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).

[27] 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.

[28] 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.

[29] H.T. Kung and J.A. Webb. Mapping Image Processing Operations Onto a Linear Systolic Machine. *Distributed Computing*, 1:246–257, 1986.

[30] P.J. Narayanan. *Effective Use of SIMD Machines for Image Analysis.* PhD thesis, Department of Computer Science, University of Maryland, College Park, MD, 1992.

[31] 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.

[32] 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, NJ, June 1990.

[33] A. Rosenfeld. A Report on the DARPA Image Understanding Architectures Workshop. In *Proceedings of the 1987 Image Understanding Workshop*, pages 298–302, 1987.

[34] 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, Fl, October 1990. (*Nucl. Phys. B (Proc. Suppl.)* 20:55, 1991.).

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

[36] 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, IN, June 1983.

[37] 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.

[38] 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, NJ, June 1990.

[39] 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.

[40] 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.

[41] 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, NJ, June 1990.

[42] 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.

[43] R. Williams. Parallel Load Balancing for Parallel Applications. Technical Report CCSF-50, Concurrent Supercomputing Facilities, California Institute of Technology, November 1994.

[44] S.W. Zucker. Region Growing: Childhood and Adolescence. *Computer Graphics and Image Processing*, 5:382–399, 1976.