NPAC Technical Report SCCS-397

Solving the Region Growing Problem on the Connection Machine

Nawal Copty Sanjay Ranka Geoffrey Fox
Ravi Shankar
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244

Abstract

Region growing is a general technique for image segmentation, where image characteristics are used to group adjacent pixels together to form regions. The region growing problem is a representative of one type of loosely synchronous problems, known as adaptive irregular problems, whose data objects evolve during the computation in a time synchronized manner.

This paper presents a parallel algorithm for solving the region growing problem based on the split and merge approach. The algorithm was implemented on the Connection Machine, models CM-2 and CM-5, in the data parallel and message passing programming models. The performance of these implementations is examined and compared.

Although the region growing problem belongs to a class of irregular problems, no new, sophisticated data structures were required to solve the problem. Only one and two-dimensional arrays of data were used. The algorithm is therefore highly portable on a wide variety of SIMD and MIMD architectures.

Keywords: Region growing, Split and merge, Parallel processing, Data parallel, Message passing, and Connection machine.

1 The Region Growing Problem

Region growing is a general technique for image segmentation. Image characteristics are used to group adjacent pixels together to form regions. Regions are then merged with other regions to *grow* larger regions. A region might correspond to a world object or a meaningful part of one [2].

The merging of pixels or regions to form larger regions is usually governed by a **homogeneity criterion** that must be satisfied. A variety of homogeneity criteria have been

investigated for region growing. If $f(x, y)$ is the image intensity (or gray level) at the pixel with coordinate (x, y), then the **pixel range** homogeneity criterion $H(R)$, for a region R, is defined as follows:

$$H(R) = \begin{cases} \textit{true}, \text{ if for all point pairs } (x_1, y_1) \text{ and } (x_2, y_2) \text{ in } R, \\ \|f(x_1, y_1) - f(x_2, y_2)\| < T. \\ \\ \textit{false}, \text{ otherwise.} \end{cases}$$

This particular homogeneity criterion requires that the range between the minimum and maximum intensities within a region R, not exceed a threshold value T. T could be a constant for the entire image, or could have different values for different parts of the image.

Unlike the binary image component labeling problem which has a unique solution that is unaffected by the order of computation, the solution to the region growing problem depends on the order of merges that take place and requires the evaluation of all possible merges at each merge step.

There are many approaches for solving the region growing problem [1, 2, 6, 9, 14]. Ballard [2] classifies the various techniques into three classes:

1. *Local techniques.* Pixels are placed in a region based on their properties or the properties of their local neighbors.

2. *Global techniques.* Pixels are placed in a region based on properties of large numbers of pixels distributed throughout the image.

3. *Splitting and merging techniques.* Graph structures are used to represent the regions and boundaries, and both local and global merging and splitting criteria are used.

The effectiveness of a particular region growing algorithm depends on the application area and the input image. If the image is sufficiently simple, local techniques can be effective, while on very difficult images even the most sophisticated techniques may not produce a satisfactory segmentation.

This paper presents a parallel algorithm for solving the region growing problem based on the split and merge approach proposed by Horowitz and Pavlidis [7]. The algorithm aims to reduce the number of merge steps required to identify the regions in the image by using a preprocessing split stage.

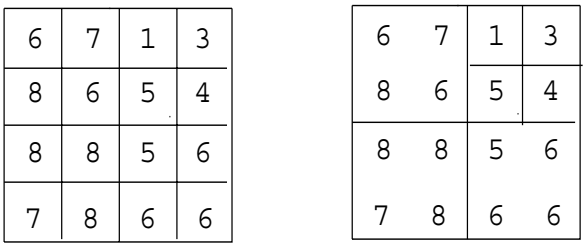
While previous parallel implementations of the split and merge approach have used dynamic or tree structures to represent the regions in the image [12, 13], our implementations use only one and two-dimensional arrays. We also introduce an element of randomness to the merging of regions when selecting a partner in case of a tie; this significantly reduced the execution time of our algorithm. A component labeling algorithm described by Hambrusch et al. [5] alternately uses extra selection rules that reduce the possibility of a tie during merging. These extra selection rules increase the execution time, but could produce bettwe solutions for certain input images.

2 The Split and Merge Approach

The split and merge approach solves the region growing problem in two stages: the split stage and the merge stage. The split stage is a preprocessing stage that aims to reduce the number of merge steps required to solve the problem.

2.1 The Split Stage

In the **split stage**, an $N \times N$ image is partitioned into square regions which conform to the homogeneity criterion. At first, each pixel is considered a homogeneous square region of size 1×1. Then every group of four adjacent pixels are tested for homogeneity. If the homogeneity criterion is satisfied, the pixels are combined into one larger square region of size 2×2. Next, every group of four adjacent square regions of size 2×2 are tested for homogeneity. If the homogeneity criterion is satisfied, the four square regions are combined into one larger square region of size 4×4, and so on... The split stage terminates when the whole image is one square region of size $N \times N$, or when no more square regions can be merged. While splitting, the top left-hand corner pixel of each square region is designated as the region representative and is assigned a unique ID. Figure 1 shows the square regions produced by the split stage for a 4×4 image, where the threshold value $T = 3$. The numbers in the image represent pixel intensities.



Square regions: (a) at start of the split stage; (b) after first and final split iteration

Figure 1:
The Split Stage

2.2 The Merge Stage

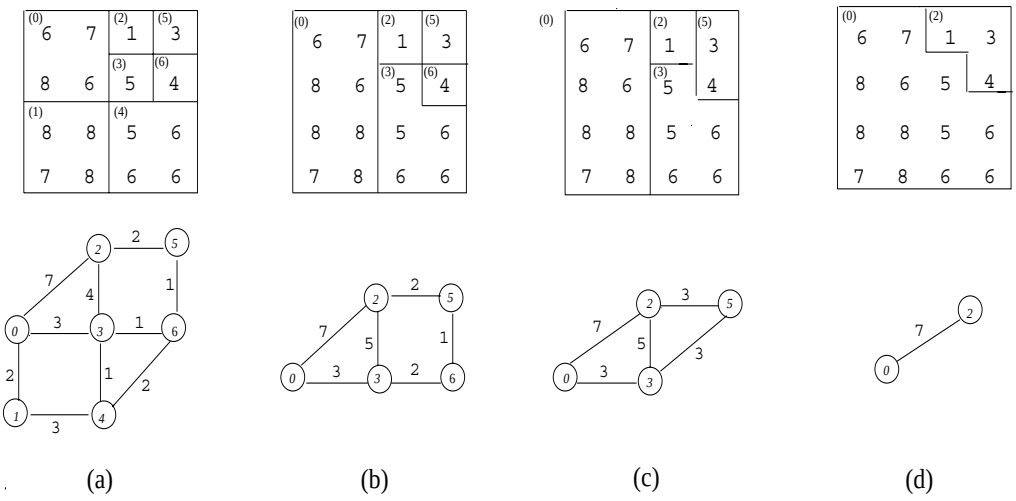
In the **merge stage** of the split and merge approach, the square regions determined by the split stage are iteratively merged into larger and larger regions which conform to the homogeneity criterion. The merge continues until no more merges are possible.

The merge is achieved by reformulating the region growing problem as a weighted, un-directed graph problem, where the vertices of the graph represent the regions in the image, and the edges represent the neighboring relationships between these regions. That is, an edge e exists between two vertices v and w of the graph, if and only if the regions represented by v and w share a common boundary. The weight of the edge e is the

difference between the maximum and minimum pixel values in the union of the two regions represented by v and w.

Obviously, only vertices connected by edges satisfying the homogeneity criterion can be merged. In one merge iteration, each region selects for merging its neighbor that *best satisfies* the homogeneity criterion. This "best merge" approach yields better results by minimizing the increase in range with each merge [13]. A tie may be broken by selecting the neighbor with the smallest (largest) ID, or by selecting a neighbor at random, as will be discussed in the next section. Two regions actually merge *if their merge choices are mutual*. That is, two regions must select each other in order for them to merge. Once two regions merge, the region with the smaller ID becomes the representative of the two, and the vertices and edges of the graph are updated. The merge stage terminates when no more edges satisfying the homogeneity criterion exist in the graph.

Figure 2 shows the different regions obtained and their corresponding graphs in each iteration of the merge stage, for the 4×4 image of Figure 1. Ties are broken by selecting the neighbor with the smallest ID. The small numbers in parenthesis in the corners of the regions denote the region IDs.



Regions: (a) at start of the merge stage; (b) after first merge iteration; (c) after second merge iteration; (d) after third and final merge iteration

Figure 2:
Merges That Take Place When Ties are Broken by Choosing the Neighbor With the Smallest ID

2.3 Resolving Ties at Random

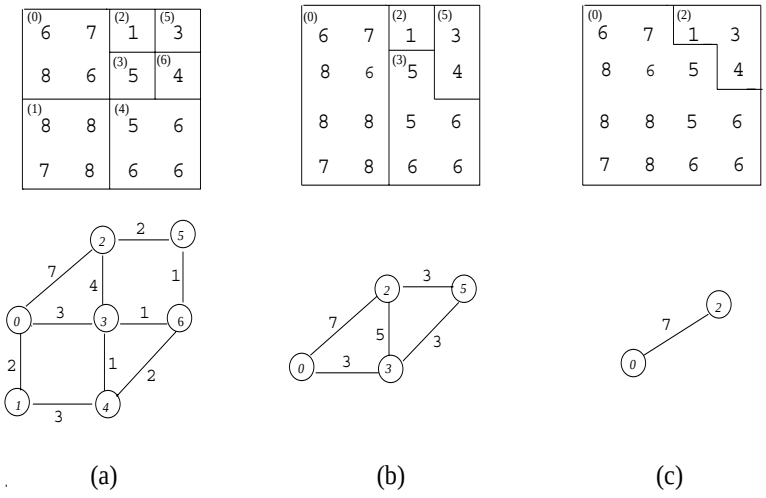
The region growing problem is a representative of a type of loosely synchronous problems, known as adaptive irregular problems, whose data objects evolve during the computation in a time synchronized manner [4]. The problem exhibits a dynamic behavior that starts with a high degree of parallelism that very rapidly diminishes to a much lower degree of

parallelism.

In order to increase the degree of parallelism in the algorithm, we introduced an element of randomness to our parallel implementations. In case of a tie during the merge stage, the tie is broken by selecting a neighbor *at random* instead of selecting the neighbor with the smallest (largest) ID.

Figures 2 and 3 illustrate the difference between the two approaches. In Figure 2(a), both regions 3 and 5 tie for merging with region 6, since merging with either of the two regions best satisfies the homogeneity criterion for region 6 (i.e. produces the least increase in pixel range for region 6, as indicated by the weights of the edges). In Figure 2, region 6 chooses to merge with region 3, since ties are broken by choosing the neighbor with the smallest ID. But no merge actually takes place, since region 3 chooses to merge with region 4.

If, instead, ties were broken at random, then in the first merge iteration regions 5 and 6 could merge, at the same time as regions 3 and 4; and the merge stage could take 2 iterations instead of 3, as illustrated by Figure 3.



Regions: (a) at start of the merge stage; (b) after first merge iteration ; (c) after second and final merge iteration

Figure 3:
Merges That *Could* Take Place When Ties are Broken at Random

Experimentally, the random approach in breaking ties proved to be significantly faster than the approach of selecting the neighbor with the smallest (largest) ID, as shown in Table I in Section 8.1. This is due to the fact that the random approach generally results in a larger number of merges per merge iteration, while the approach of selecting the neighbor with the smallest (largest) ID imposes a serialization on the order of merges.

3 The Connection Machine

The region growing problem was implemented on two distinct models of the Connection Machine: the CM-2 and CM-5. In what follows, the architecture of each of these models will be briefly described.

3.1 Model CM-2

The Connection Machine, model CM-2, is a massively parallel computer that belongs to the range of SIMD (Single Instruction Multiple Data) machines. A 32K CM-2 consists of 2048 chips, each containing 16 bit-serial processors plus associated memory, for a total of 32,768 (2^{15}) processors. The chips form an 11-dimensional hypercube, each chip having 11 wires connecting it to other chips. Within each chip, the processors are connected in a 4×4 grid.

The CM-2 operates under the programmed control of a front end computer that provides the program development and execution environment. All CM-2 programs execute on the front end; during the course of the execution, the front end issues instructions to the CM-2 processors. The CM-2 processors work in lock step. The algorithm designer need not worry about synchronizing the processors or balancing the work load.

The CM-2 can be programmed using the CM Fortran language, which is essentially standard Fortran 77 supplemented with the array processing extensions of Fortran 90. An array that is used only as a set of scalars is stored and processed in the front end in the normal serial manner, while an array that is referenced as an object is stored in the CM-2 processors and processed in parallel.

3.2 Model CM-5

The Connection Machine model CM-5 is an MIMD machine composed of a control processor (also known as partition manager), and tens or hundreds of processing nodes connected together in the form of a *fat tree* [8]. Every processing node is a general-purpose computer that can fetch and interpret its own instruction stream, execute arithmetic and logical instructions, calculate memory addresses, and perform interprocessor communication.

The control processor has the same general capability as a processing node, but is specialized for performing administrative functions rather than computational ones. It manages a partition composed of a number of processing nodes and is responsible for scheduling user tasks, allocating resources, and servicing I/O requests for that partition.

The CM-5 supports both the data parallel and message passing models of programming. For the data parallel model, the CM-5 provides the CM Fortran language. The partition manager executes all of the CM Fortran that is Fortran 77, while the nodes execute all the array extensions drawn from Fortran 90.

For the message passing model, the CM-5 provides the CMMD library, which is a collection of routines that permit cooperative message passing among the processing nodes. CMMD supports a version of message passing known as *host/node* programming, where a *host* program runs on the partition manager, and independent copies of a *node* program run on each of the processing nodes.

4 The Data Parallel Implementation

In the data parallel model of execution, the same CM Fortran program can be executed on both the CM-2 and the CM-5 without modification. In the CM-2 case, the SIMD hardware directly supports the data parallel model, since the synchronization of the processors is built into the architecture, while in the CM-5 case, compilers, assemblers, and other system software have to deal with the many "housekeeping" details such as load balance and synchronization of operations.

The data parallel implementation of the split and merge region growing algorithm consists of the following steps:

1. The two-dimensional pixel image is repeatedly split into homogeneous square regions. The split stage stops when the whole image is one homogeneous square region, or when no more merges are possible.

2. For each square region in the pixel image, a corresponding graph vertex is created, and for each pair of neighboring square regions, an edge is created. Edges that do not satisfy the homogeneity criterion are de-activated.

3. A region determines its neighboring region that best satisfies the homogeneity criterion. In the case of a tie, one of the neighboring regions is chosen *at random*. Two regions merge if their merge choices are mutual. In one merge iteration, several region pairs can merge at the same time without conflicting with each other.

4. The vertices and edges of the graph are updated to reflect the new regions in the image. Edges that do not satisfy the homogeneity criterion are de-activated.

5. If there still exist any active edges, then steps 3 and 4 are repeated. Otherwise, the program terminates.

5 The Message Passing Implementation

In contrast to the data parallel model of execution, the message passing model requires the programmer to explicitly specify the detailed behavior of individual processors operating asynchronously. Many of the facilities provided by the system software in the data parallel model are exchanged for the ability to program each node individually and to make explicit decisions on data layout, synchronization, and load balancing.

The message passing implementation of the split and merge region growing algorithm is a hand-coded translation of the data parallel one. The message passing implementation consists of the following steps:

0. The image is mapped to the node processor grid such that each processor receives an $\frac{N}{P1} \times \frac{N}{P2}$ sub-image of the original image. This partitioning maintains adjacency between neighboring blocks of the image.

1. Each node processor independently splits its $\frac{N}{P1} \times \frac{N}{P2}$ sub-image and determines the homogeneous square regions within it.

2. Each node processor sets up the vertices and edges of the graph associated with its sub-image. Boundary information is exchanged so that edges connected to vertices in other processors are created.

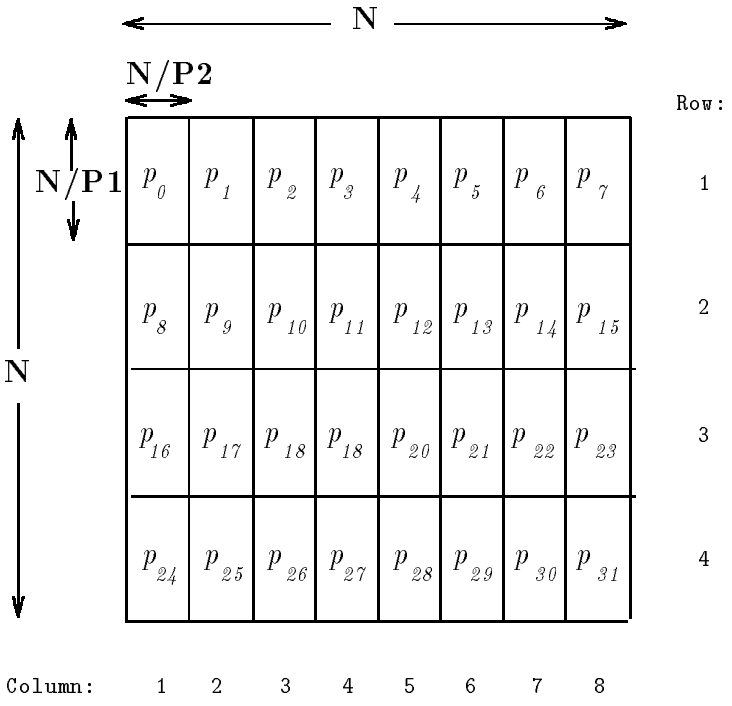
3. The node processors cooperate to merge the regions determined so far in the image.

4. The node processors cooperate to update the vertices and edges of their graphs.

5. If there still exist any active edges in any of the node processors, then steps 3 and 4 are repeated. Otherwise, the host and node programs terminate.

5.1 Partitioning of the Pixel Image

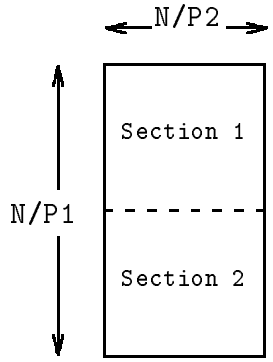
In the message passing implementation, the pixel image is partitioned equally among the node processors. Given a pixel image of size $N \times N$, and $P1 \times P2$ node processors (where each of $P1$ and $P2$ divides N), the pixel image is mapped to the processor grid such that each processor receives an $\frac{N}{P1} \times \frac{N}{P2}$ sub-image of the original image. This partitioning maintains adjacency between neighboring blocks of the image. The row and column numbers of a node processor in the grid are given by (Self address DIV $P2$) + 1 and (Self address MOD $P2$) + 1 respectively, where DIV is the integer division operation, and MOD is the modulo operation. The following diagram shows an $N \times N$ image partitioned among a 4×8 processor grid ($P1 = 4$, $P2 = 8$). The processors are numbered p_0, p_1, ..., p_{31}.



In the split stage, each node processor works independently on its $\frac{N}{P1} \times \frac{N}{P2}$ sub-image, and determines the homogeneous square regions within it. If the sub-image within the

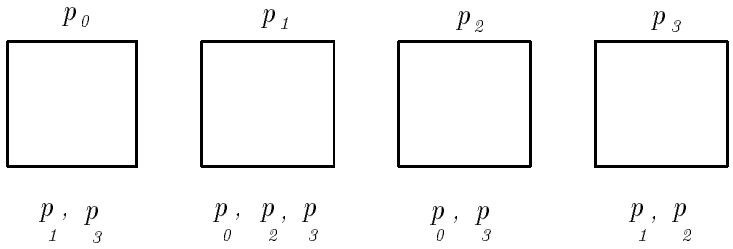
processor is rectangular in shape, then it is divided into square sections, and the split stage is applied independently to each of these sections in turn.

In the following diagram, $P2$ is assumed to be twice $P1$. The $\frac{N}{P1} \times \frac{N}{P2}$ sub-image in one processor is divided into two square sections.



5.2 Irregular Communication

At several points in the message passing implementation, irregular communication is required, whereby each of the node processors sends zero or more messages to other processors, in an irregular fashion. For example, we may have the following situation with 4 processors each having a list of destinations to send to:



In this case, an efficient communication scheme is needed whereby messages are sent and received without causing deadlock.

Two different communication schemes were investigated. The first, called **Linear Permutation (LP)** [11], uses synchronous (blocked) message passing. In this scheme, each node obtains a copy of the communication matrix, using a global concatenation operation. Then, in step i, $0 < i < Q$, processor p_k sends a message to processor $p_{(k+i) \ MOD \ Q}$, and receives a message from processor $p_{(k-i) \ MOD \ Q}$, where Q is the total number of node processors. The sender and receiver nodes are blocked until the message is transmitted. The steps of the Linear Permutation algorithm are as follows:

For all processors p_k, $0 \leq k \leq Q-1$, *in parallel do*
for $i = 1$ *to* $Q-1$ *do*
  Processor p_k sends a message to processor $p_{(k+i) \ MOD \ Q}$
  Processor p_k receives a message from processor $p_{(k-i) \ MOD \ Q}$

endfor

If processor p_k does not have any message to send to processor $p_{(k+i) \ MOD \ Q}$ then it does not participate in the send operation to that processor. Similarly, if processor p_k does not have any message to receive from processor $p_{(k-i) \ MOD \ Q}$ then it does not participate in the receive operation from that processor.

The second communication scheme uses **asynchronous** message passing. In this scheme, a node that wishes to send or receive a message does not block while waiting for its partner node. A node announces its intention to send or receive a message, and then pursues other computation until the message is ready to be sent and received. When both sender and receiver nodes are ready, the system interrupts whatever else is happening on the nodes and the message is transmitted. The steps of the asynchronous communication algorithm are as follows:

- Using a global reduction operation, each node determines the number of messages it must receive from the other nodes.
- Every node sends all the messages it wishes to send to other nodes, asynchronously.
- Every node receives the required number of messages.

In order to reduce the communication overhead in both schemes, whenever a processor needs to send more than one message to the same destination, all the messages are concatenated together and sent as one large message.

6 Data Structures

In implementing the split and merge algorithm for solving the region growing problem, no sophisticated data structures were needed to solve the problem. Only one and two-dimensional arrays were used to represent the various data items required. Two-dimensional arrays were used to store the gray levels as well as other information pertaining to the pixels, such as the pixel column and row numbers and whether a pixel is a region representative or not. One-dimensional arrays were used to store information about the vertices and edges of the graph modeling the problem.

To illustrate the way in which data is stored in the various arrays, consider the following square regions obtained at the end of the split stage, where the threshold value $T = 3$.

(0) 6	7	(2) 1	(5) 3
8	6	(3) 5	(6) 4
(1) 8	8	(4) 5	6
7	8	6	6

Information on vertices is stored in one-dimensional arrays, as follows:

Region ID:	0	1	2	3	4	5	6
Min. pixel value:	6	7	1	5	5	3	4
Max. pixel value:	8	8	1	5	6	3	4

Information on edges is stored in one-dimensional arrays, as follows:

ID of first region of Edge:	0	0	0	1	2	2	3	3	4	5
ID of second region of Edge:	1	2	3	4	3	5	4	6	6	6
Min. pixel value in union of 2 regions:	6	1	5	5	1	1	5	4	4	3
Max. pixel value in union of 2 regions:	8	8	8	8	5	3	6	5	6	4
Edge active?	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes

7 Complexity

Given an $N \times N$ pixel image, the complexity of the parallel split and merge algorithm depends on the number of processors used and the number of iterations required to find the regions in the image. This number in turn depends on the shape and size of those regions.

7.1 The Split Stage

In the best case, when every pixel is a region by itself, only one split iteration is required. In the worst case, when the whole image is one homogeneous square region, $log(N)$ split iterations are required.

CM-2 Implementation:

Suppose that P processors are used by the data parallel implementation on the CM-2, and P is smaller than N^2. At the beginning of the split stage, each pixel is considered a square region, and the first split iteration can be done in $\frac{N^2}{P}$ steps. In the second split iteration, there are $O(\frac{N^2}{4})$ square regions, and this iteration can be done in $O(\frac{N^2}{4 \times P})$ steps, and so on, until the number of square regions becomes $\leq P$. When this occurs, each iteration can be done in one step, and there will be at most $log(P)$ of these iterations. Thus the complexity of the split stage in the data parallel implementation on the CM-2 is given by $O(\frac{N^2}{P} + logP)$.

CM-5 Implementations:

In the data parallel and message passing implementations on the CM-5, the first $log\frac{N^2}{P}$ split iterations are done locally, while the last $logP$ iterations require communication. Assuming that communication in each of the last $logP$ split iterations requires $O(\tau)$ time units, where τ is the setup time, then the total time for the split stage is $O(\frac{N^2}{P}+(\tau \times logP))$. If the split stage is stopped after $log\frac{N^2}{P}$ iterations, then the time is $O(\frac{N^2}{P})$.

7.2 The Merge Stage

The number of iterations needed to complete the merge stage of the algorithm is upper bounded by the maximum number of sub-regions that must be merged to connect any single region in the image. If a region consists of r sub-regions, then it will require at least $log(r)$ merge iterations. In the worst case, when only one pair of regions is merged in each iteration, it will require $r-1$ merge iterations.

The total time for the merge stage depends on the number of square regions in the image at the beginning of the merge stage, and the number of regions in the image at the end of the merge stage. Let R_i and R_f denote these two numbers, respectively. Suppose that the number of regions is reduced by a factor of k at every step in the merge stage ($1 \leq k \leq 2$). Then the number of iterations required is $log_k\frac{R_i}{R_f}$.

The number of edges, E, and the number of regions, R_i, at the beginning of the merge stage can be derived by Euler's formula [3]: $V+R_i-E=2$, where V is the total number of corners of the square regions. Since $E=V+R_i-2$ and $V \leq 4 \times R_i$, then $R_i \leq E \leq 5 \times R_i$. Thus, the number of edges is linearly proportional to the number of regions.

CM-2 Implementation:

Suppose that P processors are used by the data parallel implementation on the CM-2. Then the total time required for any step of the merge stage in which E edges are active is $\frac{E}{P}\times$ (Cost of a Random Access Write + Cost of a Random Access Read).

The time taken by a Random Access Read and a Random Access Write of B data elements on a P-processor hypercube is:

$$= \begin{cases} O(logP), & \text{if } B \leq P \\ O(\frac{B \times logB}{P}), & \text{if } B \geq P^{1+\epsilon}, \epsilon > 0 \end{cases}$$

If we assume that the number of active edges decreases by a factor of k in each iteration of the merge stage (same as for number of regions), then the total time required for the merge stage, assuming $B \leq P$ in every iteration, is:

$$O(logP \times log_k \frac{R_i}{R_f})$$

The total time required in the general case is:

$$O((\frac{R_i \times logR_i}{P} + \frac{\frac{R_i}{2} \times log\frac{R_i}{2}}{P} + ...) + logP \times log_k \frac{R_i}{R_f})$$

$$= O(\frac{R_i \times logR_i}{P} + logP \times log_k \frac{R_i}{R_f})$$

Note that this is a very loose complexity analysis ($\frac{B \times logB}{P}$ is only valid for $B \geq P^{1+\epsilon}, \epsilon > 0$).

CM-5 Implementations:

In the data parallel and message passing implementations on the CM-5, each merge step of the algorithm requires a many-to-many communication. The complexity of the many-to-many communication is difficult to analyze, since it depends on the number of the messages sent by every processor, which in turn depends on the image.

8 Performance

The data parallel implementation (CM Fortran) of the split and merge algorithm was executed on both a 16K CM-2 and a 32-node CM-5, while the message passing implementation (F77 + CMMD) was executed on a 32-node CM-5.

A variety of images were used to test the various implementations. Pictures of the images (Image1 - Image 6) are shown at the end of this paper.

8.1 Comparison of the Smallest-ID and Random Approaches in Resolving Ties

Table I compares the smallest-ID and random approaches in resolving ties during the merge stage. The table presents the execution time and the number of iterations required by the merge stage of the data parallel implementation (CM Fortran) on the CM-5, using each of the two approaches. Invariably, in all of the images, the random approach in resolving ties proved to be faster than the approach of selecting the region with the smallest ID. Similar results were obtained for the message passing implementation on the CM-5, as well as the data parallel implementation on the CM-2.

Data Parallel Implementation on the CM-5 (32 nodes):

	Merge Stage (Smallest-ID Approach)		Merge Stage (Random Approach)	
	Time (sec)	Iterations	Time (sec)	Iterations
Image 1:	334.948	290	33.013	19
Image 2:	151.670	153	31.615	20
Image 3:	1406.099	809	42.570	27
Image 4:	622.980	549	37.588	25
Image 5:	186.834	226	24.471	16
Image 6:	1754.254	1062	75.582	45

Table I:
Comparison of Smallest-ID and Random Approaches in Breaking Ties

8.2 Comparison of the Data Parallel and Message Passing Implementations

A comparison of the performance of the various implementations (using the random approach in resolving ties) is presented below. **LP** refers to the Linear Permutation communication scheme and **Async** refers to the asynchronous one.

Image 1: 128×128 image composed of two nested rectangular regions
No. of square regions found at end of split stage = 436
No. of regions found at end of merge stage = 2

	Split Stage		Merge Stage (Random Tie Break)	
	Time (sec)	Iterations	Time (sec)	Iterations
CM Fortran on :				
CM-2 (8K procs)	0.200	4	9.511	19
CM-2 (16K procs)	0.112	4	7.027	20
CM-5 (32 nodes)	0.361	4	33.013	19
F77 + CMMD on :				
CM-5 (32 nodes, LP)	0.022	4	6.914	24
CM-5 (32 nodes, Async)	0.021	4	4.025	20

Image 2: 128×128 image composed of a collection of rectangles
No. of square regions found at end of split stage = 193
No. of regions found at end of merge stage = 7

	Split Stage		Merge Stage (Random Tie Break)	
	Time (sec)	Iterations	Time (sec)	Iterations
CM Fortran on :				
CM-2 (8K procs)	0.200	4	8.184	18
CM-2 (16K procs)	0.112	4	5.345	17
CM-5 (32 nodes)	0.360	4	31.615	20
F77 + CMMD on :				
CM-5 (32 nodes, LP)	0.022	4	9.236	35
CM-5 (32 nodes, Async)	0.021	4	6.441	35

Image 3: 128×128 image composed of a collection of circles
No. of square regions found at end of split stage = 1732
No. of regions found at end of merge stage = 11

	Split Stage		Merge Stage (Random Tie Break)	
	Time (sec)	Iterations	Time (sec)	Iterations
CM Fortran on :				
CM-2 (8K procs)	0.200	4	13.711	24
CM-2 (16K procs)	0.112	4	9.538	25
CM-5 (32 nodes)	0.361	4	42.570	27
F77 + CMMD on :				
CM-5 (32 nodes, LP)	0.022	4	9.454	33
CM-5 (32 nodes, Async)	0.021	4	5.516	28

Image 4: 256×256 image composed of two nested rectangular regions
No. of square regions found at end of split stage = 823
No. of regions found at end of merge stage = 2

	Split Stage		Merge Stage (Random Tie Break)	
	Time (sec)	Iterations	Time (sec)	Iterations
CM Fortran on :				
CM-2 (8K procs)	1.008	5	13.882	26
CM-2 (16K procs)	0.529	5	10.381	28
CM-5 (32 nodes)	2.052	5	37.588	25
F77 + CMMD on :				
CM-5 (32 nodes, LP)	0.097	5	16.512	37
CM-5 (32 nodes, Async)	0.097	5	10.942	29

Image 5: 256×256 image composed of a collection of rectangles
No. of square regions found at end of split stage = 298
No. of regions found at end of merge stage = 7

	Split Stage		Merge Stage (Random Tie Break)	
	Time (sec)	Iterations	Time (sec)	Iterations
CM Fortran on :				
CM-2 (8K procs)	1.008	5	9.287	19
CM-2 (16K procs)	0.529	5	6.633	20
CM-5 (32 nodes)	2.046	5	24.471	16
F77 + CMMD on :				
CM-5 (32 nodes, LP)	0.099	5	14.388	35
CM-5 (32 nodes, Async)	0.098	5	6.640	35

Image 6: 256×256 image of a "tool"
No. of square regions found at end of split stage = 2248
No. of regions found at end of merge stage = 4

	Split Stage		Merge Stage (Random Tie Break)	
	Time (sec)	Iterations	Time (sec)	Iterations
CM Fortran on :				
CM-2 (8K procs)	1.008	5	19.530	34
CM-2 (16K procs)	0.529	5	13.426	33
CM-5 (32 nodes)	2.066	5	75.582	45
F77 + CMMD on :				
CM-5 (32 nodes, LP)	0.098	5	12.192	36
CM-5 (32 nodes, Async)	0.098	5	7.236	38

The bar chart of Figure 4 gives a visual comparison of the times taken by the merge stage in the various implementations. Figure 5 presents the execution time and speedup of the merge stage in the message passing implementation on the CM-5, using asynchronous communication.

8.3 Observations

In examining the performance of the different versions of the split and merge algorithm, we make the following observations:

- The number of merge iterations required to find the regions in an image are not identical in all cases. This is due to the element of randomness that is introduced in selecting a neighbor for merging. The random numbers generated, as well as the order in which messages are delivered affect the actual merges that take place and hence the number of merge iterations required to solve the problem.

- The data parallel (CM Fortran) version runs faster on the CM-2 than on the CM-5. The SIMD hardware of the CM-2 directly supports the data parallel model, while compilers, assemblers, and other system software of the CM-5 have to deal with the many "housekeeping" details such as load balance and synchronization. It is expected, however, that by using a larger number of nodes and/or vector units on the CM-5, the execution time will be significantly reduced.

- As the timing figures and the bar chart of Figure 4 indicate, the message passing implementation (F77 + CMMD) on the CM-5 runs significantly faster than the data parallel (CM Fortran) version on the same machine. The data parallel version relies on the CM Fortran compiler as well as the run-time system to lay out the data and to provide communication among the nodes, while, in the message passing version, the programmer exercises control over synchronization, data partitioning, and load balancing. With the availability of new data distribution directives in High Performance Fortran, the performance of the data parallel implementation should be closer to the hand-coded message passing one.

- The data parallel version is easier to program than the message passing one, precisely because the programmer does not have to be concerned with synchronization, data partitioning, or load balancing. So there is a tradeoff between ease of programming and execution speed. This emphasizes the need for improved compilers, languages, and run-time support, so data parallel languages can be executed more efficiently on distributed machines.

- Of the two communication schemes investigated on the CM-5 (asynchronous and Linear Permutation), the asynchronous scheme is the faster one. In the asynchronous scheme, a node can pursue other computation until messages are ready to be sent or received, while in the Linear Permutation scheme, all the nodes must loop the same number of times until the all required sends and receives are completed.

- The graph modeling the region growing problem constantly evolves during the course of the computation. In the current message passing implementation, the vertices and edges of the graph remain in the same processors throughout the merge stage. This, in general, will lead to load imbalance. A more realistic approach would be to let the active vertices and edges migrate between the processors, so the load is more evenly distributed. Obviously, load balancing will require increased communication and computation. It is expected that load balancing will produce positive results in some test cases, but not in others.

9 Conclusions

We have presented a parallel algorithm for solving the region growing problem based on the split and merge approach. Ties during merging were resolved by selecting a partner at random. The algorithm was implemented on the Connection Machine, models CM-2 and CM-5, in both the data parallel and message passing programming paradigms. Only one and two-dimensional arrays of data were used in the implementations. The performance of the algorithm using the different architectures and programming models was analyzed and compared.

Acknowledgements
We would like to thank Paul Coddington, Pablo Tamayo, and Jhy-Chun Wang for interesting and helpful discussions; Gregor von Laszewski for help in preparing the manuscript; and the referees for their useful and insightful comments.

References

[1] H. Alnuweiri and V. Prasanna, "Parallel Architectures and Algorithms for Image Component Labeling", *IEEE Trans. Patt. Anal. Machine Intell.*, Vol. 14, pp. 1014-1034, 1992.

[2] D. Ballard and C. Brown, *Computer Vision*, Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[3] S. Even, *Graph Algorithms*, Computer Science Press, Potomac, Maryland, 1979.

[4] G. Fox et al, "Software support for irregular and loosely synchronous problems", Technical Report, Northeast Parallel Architectures Center, Syracuse University, May 1992.

[5] S. Hambrusch, X. He, and R. Miller, "Parallel Algorithms for Gray-Scale Digitized Picture Component Labeling on a Mesh-Connected Computer", to appear in *Journal of Parallel and Distributed Computing.*

[6] R. M. Haralick and L. G. Shapiro, "Image Segmentation Techniques", *Computer Vision, Graphics, and Image Processing*, Vol. 29, pp. 100-132, 1985.

[7] S. L. Horowitz and T. Pavlidis, "Picture Segmentation By a Directed Split-and-Merge Procedure", *Proc. 2nd International Joint Conference on Pattern Recognition*, pp. 424-433, August 1974.

[8] C. Leiserson, "The Network Architecture of the Connection Machine CM-5", *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, San Diego, California, June 29-July 1, 1992.

[9] T. Pavlidis, "Image Analysis", *Annual Review of Computer Science*, Vol. 3, pp. 121-146, 1988.

[10] S. Ranka and S. Sahni, *Hypercube Algorithms*. Springer-Verlag, New York, 1990.

[11] S. Ranka, J. Wang, and G. Fox, "Static and runtime algorithms for all-to-many personalized communication on permutation networks", *Proc. International Conference on Parallel and Distributed Systems*, December 1992.

[12] J. C. Tilton, "Image segmentation by iterative parallel region growing with applications to data compression and image analysis", *Proc. 2nd Symposium on the Frontiers of Massively Parallel Computation*, 1988.

[13] M. Willebeek-LeMair and A. Reeves, "Solving non-uniform problems on SIMD computers: Case study on region growing", *Journal of Parallel and Distributed Computing*, Vol. 8, pp. 135-149, 1990.

[14] S. W. Zucker, "Region growing: Childhood and adolescence", *Computer Graphics and Image Processing*, Vol. 5, pp. 382-399, 1976.

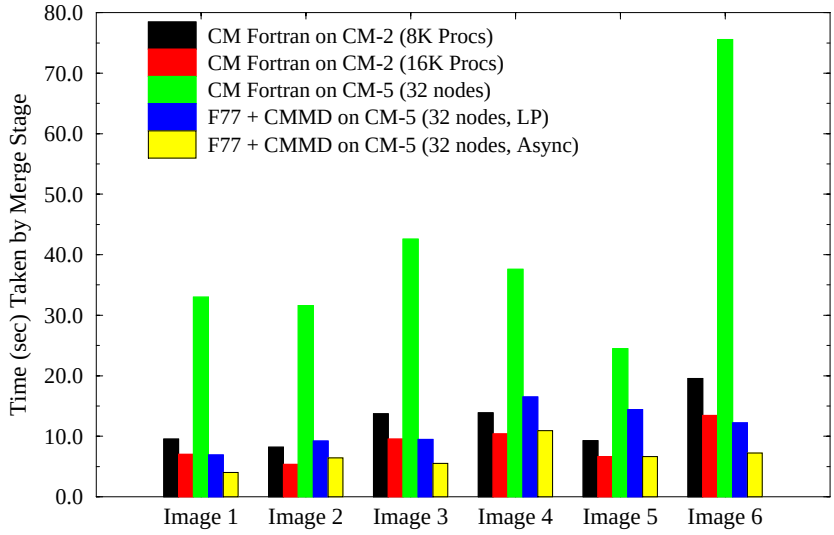


Figure 4: Execution time of the Merge Stage in the Various Implementations

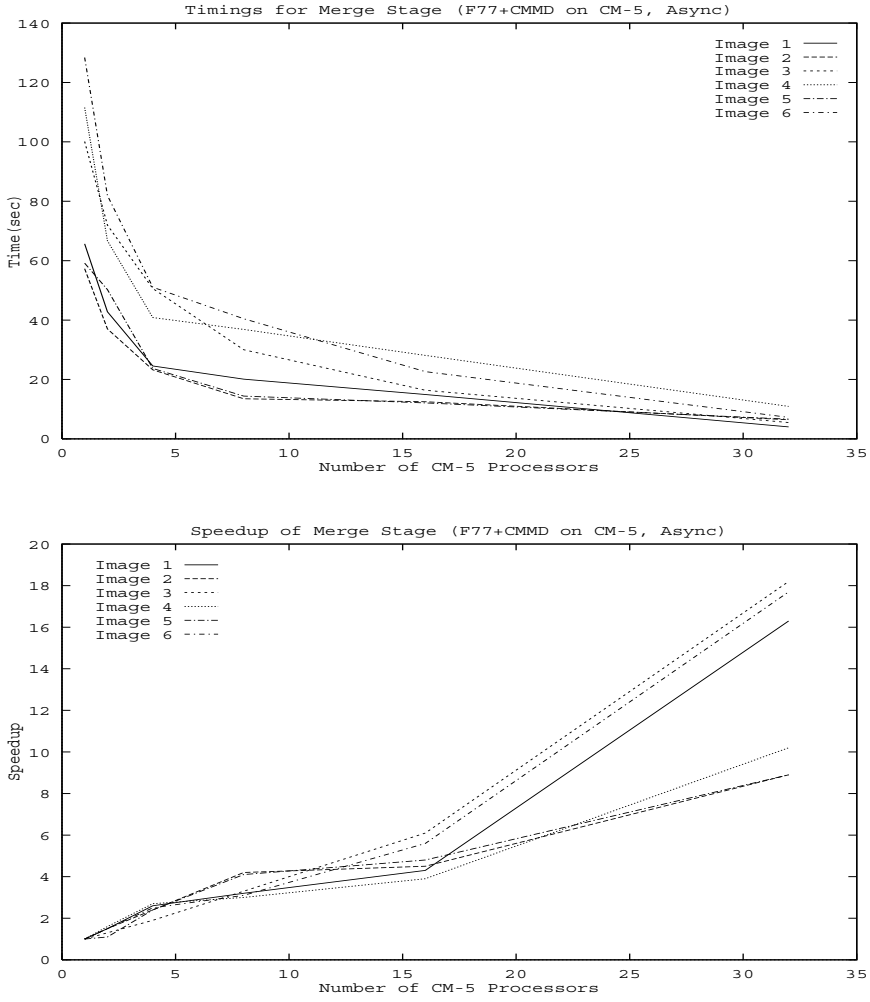


Figure 5: Execution Time and Speedup of the Merge Stage on the CM-5 as a Function of the Number of Processors

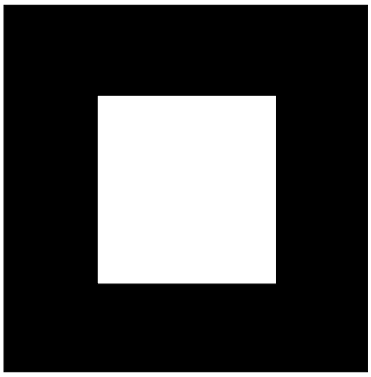


Image 1 (128 x 128 image)

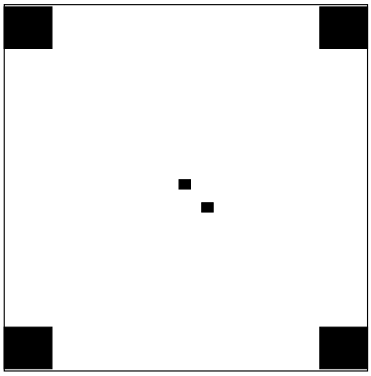


Image 2 (128 x 128 image)

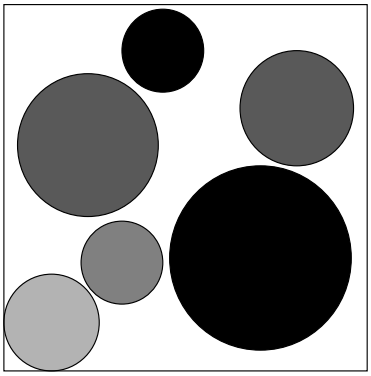


Image 3 (128 x 128 image)

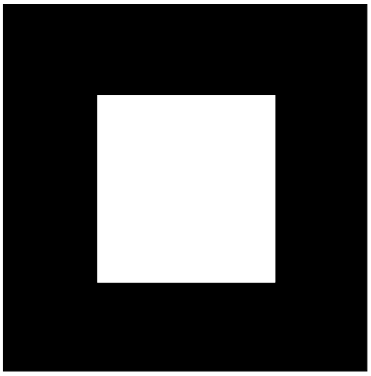


Image 4 (256 x 256 image)

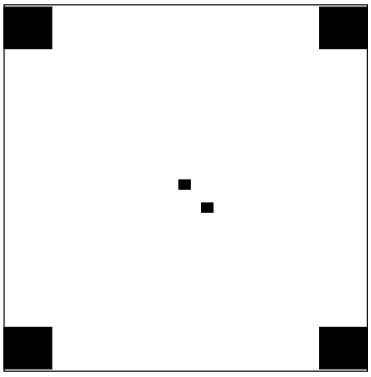


Image 5 (256 x 256 image)

Image 6 (256 x 256 image)

Figure 6: Images 1-6