Solving the Region Growing Problem on the Connection Machine

Nawal Copty, Sanjay Ranka, Geoffrey Fox, and Ravi Shankar

School of Computer and Information Science
Syracuse University, Syracuse, NY 13244
Email: nkcopty, ranka, rshankar@top.cis.syr.edu, gcf@npac.syr.edu

Published in:
Proc. 22nd International Conference on Parallel Processing, 1993, Vol. III, pp. 102-105

Abstract – *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 CM-2 and the CM-5 in the data parallel and message passing models. The performance of these implementations is examined and compared.*

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

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. The *pixel range* homogeneity criterion requires that the difference between the minimum and maximum intensities within a region not exceed a threshold value T.

There are many approaches for solving the region growing problem [1, 2, 10]. This paper presents a parallel algorithm for solving the problem based on the split and merge approach [5]. The algorithm aims to reduce the number of merge steps required to identify the regions in the image by using a preprocessing split stage.

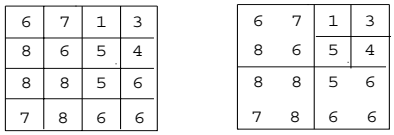
While previous parallel implementations of the split and merge approach have used dynamic or tree structures to represent the regions in the image [8, 9], our implementations use only one and two-dimensional arrays to solve the problem. Moreover, we introduce an element of randomness to the merging of regions. For a detailed presentation, refer to [3].

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

In the split stage, an $N \times N$ image is partitioned into square regions which satisfy 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, 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. Figure 1 shows the square regions produced by the split stage for a 4×4 image, where the threshold value $T = 3$.



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

Figure 1: The Split Stage

The Merge Stage

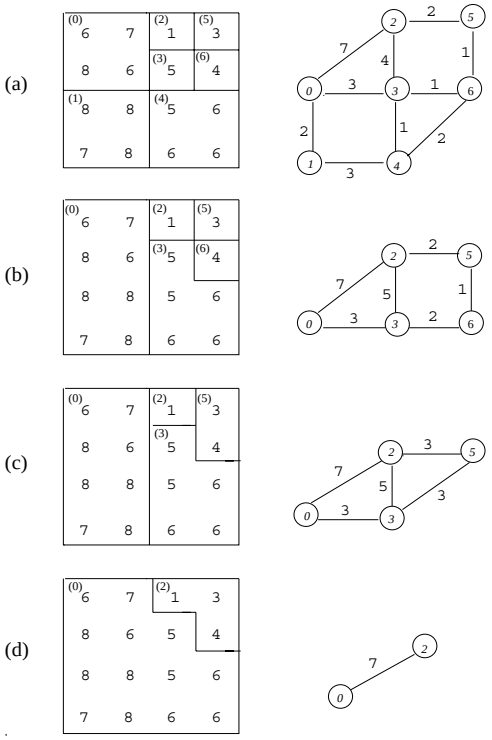
In the merge stage, the square regions are iteratively merged into larger and larger regions which satisfy the homogeneity criterion. The merge continues until no more merges are possible.

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 among 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.

A tie may be broken by selecting the neighbor with the smallest (largest) ID, or by selecting a neighbor at random. Two regions actually merge if they select each other for merging. Once two regions merge, the vertices and edges of the graph are updated to reflect the new regions in the image.

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 appearing in the upper left-hand 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: The Merge Stage

Parallel Implementations

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.

The split and merge algorithm for solving the region growing problem was implemented in both the data parallel and message passing models. In the data parallel model, the CM Fortran programming language was used, and the same program was executed on both the CM-2 and the CM-5. In the message passing model, on the other hand, sequential Fortran 77 supplemented with message passing library routines (CMMD) was used, and the program was executed on the CM-5. Only one and two-dimensional arrays were used to represent the various data items required. Two-dimensional arrays were used to store the intensities and other information pertaining to the pixels, while one-dimensional arrays were used to store information about the vertices and edges of the graph modeling the problem.

Data Parallel Implementation

The data parallel implementation of the split and merge 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. When two regions merge, the region with the smaller ID becomes the representative of the two. 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 active edges, then steps 3 and 4 are repeated. Otherwise, the program terminates.

Message Passing Implementation

The message passing implementation of the split and merge algorithm is a hand-coded translation of the data parallel one and 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 splits independently 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 homogeneous square regions.
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 node programs terminate.

At several points in the message passing implementation, irregular communication is required, where each of the node processors sends zero or more messages to other processors in an irregular fashion.

Two different communication schemes were investigated. The first, called **Linear Permutation (LP)** [7], 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 second communication scheme uses **asynchronous** message passing.

Resolving Ties at Random

In order to achieve a higher degree of parallelism, we introduced an element of randomness in 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, since the latter approach imposes a serialization on the order of the merges. The random approach in breaking ties was shown to be significantly faster than the approach of selecting the neighbor with the smallest (largest) ID, since it generally results in a larger number of merges per merge iteration.

Complexity

Given an $N \times N$ pixel image, the complexity of the parallel split and merge algorithm depends on the number of processors (P) used and the number of iterations required to find the regions in the image.

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.

The time complexity of the split stage in the data parallel implementation on the CM-2 is $O(\frac{N^2}{P} + logP)$, while that of the data parallel and message passing implementations on the CM-5 is $O(\frac{N^2}{P} + (\tau \times logP))$, where τ is the communication set up time for one split iteration.

The Merge Stage: In the best case, a region consisting of R sub-regions will require $logR$ iterations to merge. In the worst case, when only one pair of regions is merged in each iteration, it will require $R-1$ merge iterations.

Let R_i denote the number of homogeneous square regions found in the image at the end of the split stage, and let R_t denote the number of regions found at the end of the merge stage. If we assume that the number of regions is reduced by a factor of k at every step in the merge stage, then the time complexity of the merge stage in the data parallel implementation on the CM-2 is $O(\frac{R_i \times logR_i}{P} + log_k \frac{R_i}{R_t} \times logP)$. For details of the analysis and assumptions made, refer to [3].

The time complexity of the the data parallel and message passing implementations on the CM-5, on the other hand, is difficult to analyze, as the number of messages sent by the processors in each step of the merge stage depends on the image.

Performance

The data parallel implementation (using 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 (using F77 + CMMD) was executed on a 32-node CM-5. A variety of images were used. The performance of the implementations for images of sizes 128×128 and 256×256 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 (secs)	Split Iters	Merge (secs)	Merge Iters
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 (secs)	Split Iters	Merge (secs)	Merge Iters
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 (secs)	Split Iters	Merge (secs)	Merge Iters
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 (secs)	Split Iters	Merge (secs)	Merge Iters
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 (secs)	Split Iters	Merge (secs)	Merge Iters
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 (secs)	Split Iters	Merge (secs)	Merge Iters
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 3 gives a visual comparison of the times taken by the merge stage in the various implementations.

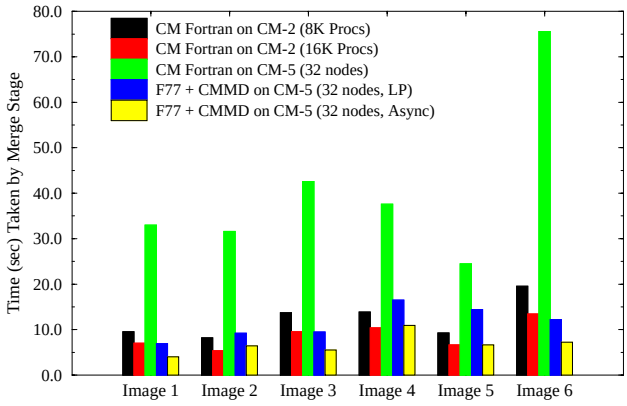


Figure 3: Comparison of Times Taken by the Merge Stage (Images 1-6)

Observations

In examining the performance of the different parallel implementations 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. The random numbers generated, as well as the order in which messages are received affect the actual merges that take place and hence the number of merge iterations required to solve the problem.
- Asynchronous communication on the CM-5 is faster than Linear Permutation, since in Linear Permutation the nodes must loop a larger number of times to complete the required communications.
- The CM Fortran version on the CM-2 runs faster than that 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.
- The message passing implementation runs significantly faster than the data parallel one on the CM-5. The data parallel implementation 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 implementation, the programmer exercises control over synchronization, data partitioning, and load balancing.
- CM Fortran allows only limited ways of distributing data on different processors. With the availability of new data distribution directives in High Performance Fortran, the performance of the data parallel implementation is expected to be closer to the message passing one.

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

References

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

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

[3] N. Copty, S. Ranka, G. Fox, and R. Shankar, "Solving the Region Growing Problem on the Connection Machine", Technical Report, January 1993, Northeast Parallel Architectures Center, Syracuse University.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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, 1990, pp. 135-149.

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