Combining Symbolic and Numeric Computation on the CRAY[1]

Ashok Samal and Tom Henderson

UUCS-86-115

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

November 17, 1986

Abstract

It is now widely accepted that the CRAY supercomputers are very useful for large *numeric* applications, e.g., Finite Element Analysis, Fluid Dynamics, Image Processing, etc. Much work has been done to make them run efficiently on the CRAY. There has also been some effort to do *symbolic* computing (AI applications) on these machines. A dialect of LISP, called PSL (Portable Standard Lisp) has been available on the CRAY for some time. However, there has been no effort to effectively combine the two, which is useful for certain kinds of applications, e.g., image understanding. This work demonstrates the mixed computing capability of the CRAY supercomputer by implementing a rule-based segmentation system. The numeric part consists of the FORTRAN subroutines to segment an image into regions, merge and split regions. A forward-chained production system to guide the segmentation process forms to symbolic component. The process of linking the two components is also described.

[1]This work was done at Cray Research Inc., Mendota Heights during summer 1986.

1 Introduction

It is now widely accepted that the CRAY supercomputers are very useful for large numeric applications, e.g., Finite Element Analysis, Fluid Dynamics, etc. Much work has been done to make them run efficiently on the CRAY. There has also been some effort to do *symbolic* computing (AI applications) on these machines. A dialect of lisp called PSL (for Portable Standard Lisp) has been available on the CRAY-1 and CRAY-XMP since 1984 [3]. It was developed jointly at Los Alamos, University of Utah, and Mendota Heights. It has since been optimized for the XMPs by Cray Germany. However, there has been no effort to effectively combine the two, which is very useful for certain kind of applications.

An application which needs both these capabilities is image understanding. The goal here is to classify/identify different regions in a given image. This is a very complex task and involves the use of several algorithms which are diverse in their computational needs. The algorithms have traditionally been grouped under two main classes: *low level* and *high level*. Low level algorithms work on the image, and produce some symbolic information about the image. These algorithms treat the image as a two dimensional array of numbers. In contrast, high level algorithms work on the symbolic data produced by the low level algorithms to generate a symbolic description of the image. The computation in the low level algorithms is highly numeric. In contrast the high level algorithms are symbolic in nature. Thus it is an application where both numeric and symbolic capabilities are a necessity.

The goal of this project was to credibly demonstrate the combined *symbolic / numeric* capability on the CRAY. The task chosen to show this was the segmentation of satellite imagery. The top level is a rule-based system which guides the segmentation process. This forms the symbolic component of the system. At the bottom there are image processing routines which constitute the numeric component. The numeric routines are in Fortran, while the rule-based system is written in Lisp.

2 Overall System Structure / Control Flow

In this section we briefly describe how the system is organized and how the flow of control takes place. The first thing done after acquiring an image is noise reduction. A 3 by 3 median filter is used to smooth the image. Then the image is partitioned using a coarse, but fast segmentation algorithm. A modified blob coloring algorithm [1] is used for this purpose. An integral part of any region growing algorithm is the merging of small regions [5]. Normally a lot of regions are produced by this method and many of them are extremely small. The next step is to merge the regions of size less than a certain threshold, with the neighbor with the least difference in average intensity. Finally, the properties (e.g., area, aspect ratio, etc.) of the regions are computed and stored in a property table.

Next the working memory to be used by the rule-based system is built. This is done by accessing the property table just mentioned. From now on the rule-based system has total control of the system. Depending on the situation, zero or more rules will fire and correspondingly some actions will be taken. There are only two major actions that are performed: *merge* two regions, and *split* a region into 2 or more regions. Both these operations are done in CFT (Cray Fortran), although the symbolic component also updates the working memory. Both these operations involve updating the segmented image and the property table. The split and merge operations are implemented in CFT since they are very well suited for vector processing. Thus the numeric and the symbolic component work hand in hand to effect the changes in the image.

The system halts when there is no rule that can be fired. At this point the system has the final segmented image in a color map form. Figure 1 gives a pictorial description of structure of the major components of the system.

3 Numeric Component

As mentioned before, it is made out of image processing routines which are all in CFT. CRAY has an image processing package called CSADIE [6]. It is a tape-oriented system and has only very primitive routines, e.g., filtering, OR'ing images, etc. It provides however a nice starting point to develop more sophisticated routines, since it has functions to do I/O cleanly. For our application it was necessary to develop several major routines, since they are not available in CSADIE. They will be briefly described below.

- **Blob Coloring** : This is a simple two pass algorithm to do segmentation. It is described in detail in [1]. It has been modified for gray scale images in an obvious manner.

- **Threshold Finding** : The threshold value for segmentation is adaptively determined from the image under consideration. This is done using an algorithm described in [2]. The first step is to differentiate the image, which also has been implemented, as an independent routine.

- **Computing Region Properties** : Properties of the regions are used by the rule-based system to perform split and merge on them. These are more efficiently computed in CFT. The various properties that are computed are: average intensity, variance, area, aspect ratio, number of adjacent regions, and adjacency values for each adjacent region. Some other properties like xmin, xmax, ymin and ymax are also computed since they help in locating the regions in the image much faster. All these properties have to be recomputed when either merging or splitting takes place.

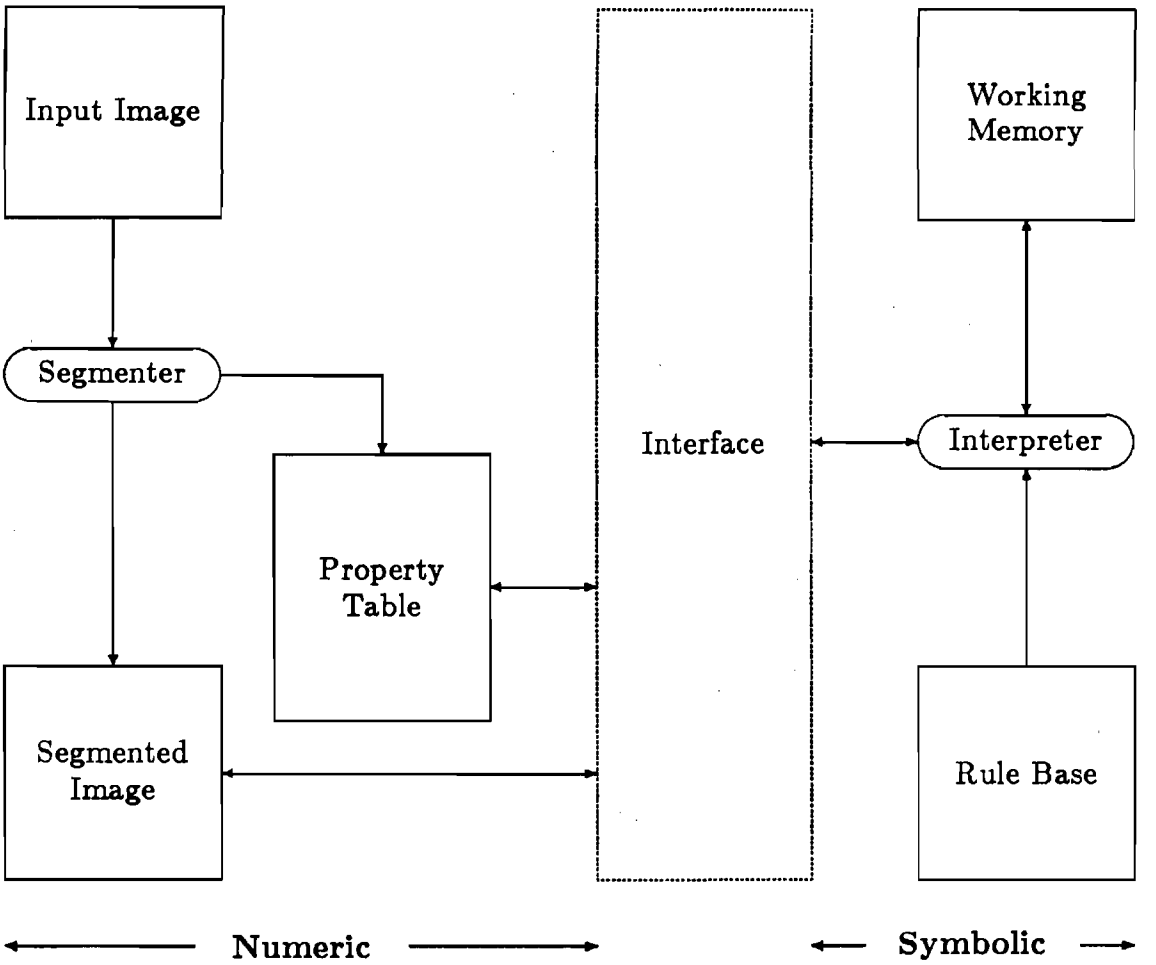


Figure 1: **Block Diagram of the System Showing Major Components**

- **Splitting a Region** : This is one of the actions the rule-based system performs. The region is split into several regions around a certain value obtained from the region itself. The old region is relabelled and the property table is correspondingly updated.
- **Merging two Regions** : This is the other action performed by the rule-based system. Here two adjacent regions are merged and are labelled as the same regions. One of the two regions is removed from the property table and the properties of their adjacent regions are updated.

Some other subroutines, e.g., to check if the histogram of a region is bimodal, etc., have also been implemented.

3.1 Common Blocks

CSADIE has a common block to maintain the parameters of interest to the CSADIE routines. In addition to it, we maintain another common block called PSLC, for storing the items which will be needed by the symbolic component. This includes the ids of the image and the segmented image, number of regions, the property table and the adjacency table (which stores the adjacency information). When the symbolic component needs the value of any property it is looked up in the appropriate table and returned.

4 Symbolic Component

The symbolic component consists of a rule-based system and several lisp functions, which work alongside the rule-based system. All the units in this component are written in PSL.

4.1 Rule-Based System

It is essentially a *forward-chaining* production system. Like any other production system, it has 3 parts: *Rule Base*, *Working Memory*, and *Interpreter*. The working memory here consists of items representing the properties of different regions, e.g., (*Area Region–1 LOW*), (*Aspect–Ratio Region–3 VHIGH*), etc. The details of how the working memory elements are built and maintained will be described later in this section.

4.1.1 Rule Base

The rule base consists of the set of rules which guide the segmentation process. These rules are derived from the knowledge about the segmentation itself and from the domain from which the image is taken, e.g., LANDSAT images, outdoor scenes, etc. Every rule has two

parts: *conditions* and *actions*. A rule is applicable iff all the 'conditions' are satisfied. If the rule is fired, the 'actions' are performed. A typical rule looks as follows:

(Area ?Region VLOW)
(Adjacent ?Region ?Region1)
(Diff Average ?Region ?Region1 VLOW)
$\Longrightarrow$
(Merge ?Region ?Region1)

The items before the $\Longrightarrow$ denote the conditions, while the items after it are the actions. A '?' before a name means it is to be treated as a variable. The rule says that if a region is very small and its average intensity is very close to one of its adjacent regions, then merge the small region with its adjacent region.

Most of the conditions in the rule are all pattern matched with the working memory elements. There are only two kinds of conditions which are specially evaluated, in order to make the testing faster. One of them is the 'DIFF' condition, just used in the previous example. To store the difference of every pair of regions for every property would be tremendously expensive in terms of storage, which would in turn affect the time for pattern matching. So, this is done on demand. The other type of condition which needs special treatment is the 'BIMODAL' condition. There are rules whose conditions check for the bimodality of the histogram of the region. This is a fairly expensive procedure. For the same reason as the 'DIFF' condition, it is evaluated only when necessary and not for all regions.

In our rule base there are about 60 rules which guide the segmention. The rules are fairly simple in nature since the properties computed are simple. Also, all conditions pertain to the properties of the regions. A more sophisticated system would also use the lines present in the image. The action part is very simple in our case. It is one of two actions: *split* a region or *merge* two regions.

4.1.2 Interpreter

This is the component that drives the system. It has three phases: *match*, *conflict resolution*, and *act*. During the matching phase, the conditions of each rule are matched against the working memory elements (except the two special conditions described previously). In the conflict resolution phase, one of the matched rules is selected to be fired. In the 'act' phase, the actions corresponding to the rule are performed. This cycle is repeated until no rule is applicable.

In our implementation, the match and conflict resolution is combined into one phase. The first rule whose conditions are satisfied is fired, and the rules are always selected in the same order. The rationale for this scheme (besides the shortage of time) is that, in

our application, it really doesn't make a lot of difference in the final result. By choosing a different conflict resolution strategy, the order in which regions are split and merged will be different, but the overall result will be very similar.

4.1.3 Working Memory

Working memory contains the properties of the different regions that are currently in the image. Thus it gives a snapshot of the segmentation process. After the initial segmentation and the building of the property table by the numeric component, the working memory is built. The numeric values in the property table are mapped into a symbolic scale of five values, *very low*, *low*, *medium*, *high*, *very high*. A simple technique from [4] is used for this.

Since the matching done by the interpreter is very simple-minded, the matching takes a very long time. To make this more efficient, the working memory was partitioned into several memory banks based on the properties. For example, there are separate working memories for 'aspect-ratio', and 'area'. Thus first step in matching is to see which working memory the condition is to be matched against. As it turned out, it saved a considerable amount of time in the matching process.

Although there are only two kinds of action (split and merge) that are explicitly done, the updating of the working memories is also an integral part of an action. After an action is performed, there are some new regions and some of the old regions which cease to exist. This information has to be added to the working memories. This is done implicitly after every action.

4.2 Utility Lisp Functions

Besides the rule-based system, the symbolic component has lots of lisp functions to perform tasks which complement it, e.g., making the initial working memory, to convert numeric values of the properties of regions to symbolic values, update working memories after split and merge, etc. There are several others which are essentially utility functions.

5 Linking the two Components

As mentioned before, the two components are written in different languages, the numeric component in CFT, and symbolic component in PSL. Hence, it is necessary to build an interface between these two in order for the system to work correctly. This is not a big problem in machines with dynamic loaders. But with a machine like the CRAY which has a static loader, there is no concept of loading an exterior program at runtime.

However, there is a way to get around it, which made linking possible. Since the PSL kernel is written in CFT, the new CFT routines can be linked to the psl startup and a new

lisp can be rebuilt, from which the new CFT routines can be called. This has the same effect as loading the routines at the runtime, but has some major problems. This will be discussed in a later section. The rest of this section will be spent on describing this linking process in detail.

The main steps involved in linking are as follows:

1. Write the new CFT code and debug it thoroughly. Also, compile the code and make a library out of the routines. Although they can be compiled later while building lisp, it is convenient to compile them first and directly load them. This way it takes less time to rebuild lisp.

2. Create a *link* file, which essentially interfaces lisp functions and their corresponding fortran functions. This is briefly explained below.

   All the fortran functions are declared as *foreign function.* For example, if MERGE and SPLIT are CFT functions which need to be called from PSL, then the following declaration has to be made.

   (flag '(MERGE SPLIT) 'foreignfunction)

   This declares 'MERGE' and 'SPLIT' to be foreign–functions which in this case are written in CFT. Now the new PSL functions are defined which can call these CFT functions. All such functions have to be defined before lisp is rebuilt. Thus the following definitions

   (de merge(x y) (MERGE x y)) (de split*(x) (SPLIT x))*

   allows the use of CFT functions SPLIT and MERGE, indirectly through 'merge*' and 'split*', which are lisp functions. It is convenient to put the declarations and the function definitions in one file.

3. Rebuild the *Kernel* module, with these modifications. Four modifications have to be made in the job deck that produces the kernel module. The first two are essentially to access the library of new fortran functions and lisp definition file.

```
*$NEWLIB is the library of new CFT functions
*
ACCESS,DN=$NEWLIB,ID=IMG.
*
* FDEFS.PSL is the link file
*
ACQUIRE,DN=FDEFS,TEXT='$DISK FDEFS.PSL'.
```

Then the *link* file has to be loaded with the other kernel source files. So, the following line should be added to the deck, just after the last kernel file has been loaded.

```
⋮
% AS is the last kernel source file
%
(dskin "AS")
% Now dskin the link file
%
(dskin "FDEFS")
```

Finally, the load line has to be updated, to include the new library, $NEWLIB.

```
LDR,DN=$BLD:BLD,SET=ZERO,MM=10000:5000,MMEPS=100,^
MMLOC=AFTER,LIB=$NEWLIB.
```

If there is more than one library, as is the case with our implementation, they should all be added at this point. Now the job is run to build a new kernel module.

4. The final step is to build a new lisp by using the new kernel. This is done by the 'NEWSYS' job deck, without any modifications.

5.1 A Note About Rebuilding Lisp

Although the procedure described above should produce a new lisp with added links to the CFT routines, there was a major problem during the actual rebuilding process. For some reason the new lisp never got built. The problem was traced to the lisp function *savesystem*, which is responsible for creating a new lisp from the current lisp image. After several unsuccessful attempts to rebuild it, under different environments, we decided to save the binary created while creating the new *Kernel* module and use it. This is done as follows:

```
LDR,DN=$BLD:BLD,SET=ZERO,MM=10000:5000,MMEPS=100,^
MMLOC=AFTER,NX,AB=NPSL,LIB=$NEWLIB.
SAVE,DN=NPSL,ID=PSL.
```

After that, NPSL was used and it didn't create any problems. However, some initializations have to be made before using it. The only major one is to open the UTLIB, before doing anything else. Also, when the new psl is fired, it prints out some COS messages, which should be ignored.

6 Results, Extensions and Discussion

The system was tested and debugged for smaller images (32 by 32). Due to the shortage of time, it was not possible to test the system on larger images. The images that were tried are subimages of a LANDSAT image. Only one band was used for testing purposes. The appendix has a sample run, along with some intermediate images. One trial run with a 128 by 128 image was made. The system took more than 2 minutes, but still could not finish (exceeded time limit). Smaller images take about 30 seconds of CPU time.

Although it was proposed to be a classification system as well, there was no attempt to do that part, due to the lack of time. However, it is a simple extension to add this feature. Essentially, additional CFT code has to be written to measure some additional properties of the regions and after the segmenter has finished the classifier takes over. Correspondingly, the rule-base has to be updated to include rules for classification as well.

To make the system run for bigger images, the array sizes in the common areas have to be enlarged. Also, some of the routines may have to be rewritten for efficiency reasons. For example, in the current system, the small regions are merged after the initial segmentation. Thus each small region has an entry in the property table. For bigger images, the number of initial regions is extremely large, which necessitates the need for a very large property table, even though most of them will be unused after the small regions are merged. A better algorithm would be to dynamically merge the smaller regions, during the process of segmentation.

Despite these drawbacks, we feel the system is complete in terms of what the original goal was: to demonstrate the combined numeric and symbolic capability on the CRAY. This is a credible application to show that mixed computing can be done on the CRAY and it provides a framework for such computation.

Besides the ones mentioned above, there can be several other extensions to the system. The system can be modified to handle multi-channel images, which should be straightforward. The segmentation process can made be more sophisticated by including the edges and lines in the analysis. This would increase the size of the rule base. A line finder has to be written in CFT to do this.

The forward-chaining system can be drastically improved. What has been implemented here is very simple-minded and slow. As the complexity of the rule base and the working memory increases, the current system's performance will go down very fast. A more sophisticated system has to be written to handle bigger applications efficiently.

Although the CFT routines have not been completely optimized, there was some effort to vectorize the code wherever it was straight-forward. However, the routines can be improved by optimizing them further to take advantage of vector processing.

7 Comments and Recommendations

In this section we comment briefly on the various tools that were used to build the system and on the building process itself. The comments pertain to the environment when the system was developed, which has since changed.

7.1 Developing PSL code

There was no support for PSL (or any kind of lisp for that matter) anywhere except on the CRAY. This made the development of PSL code very very difficult and time consuming. Normally one develops lisp code interactively and incrementally using some kind of a lisp editor. There are two ways to run PSL on the cray. In the batch mode, a job is submitted, which runs PSL and the log file returns the results of the run. This has the usual overhead of being queued, etc., which is rather frustrating for testing small functions. Alternatively, one could get into the interactive mode and fire up PSL and then test the functions. Although it might seem to be the faster method to test small functions, it is not always true. When the system is even moderately loaded, this is very slow, and the batch mode is actually faster! It was really frustrating to go through this process to develop PSL code. Obviously, this tremendously reduces the productivity of the programmer.

There is a very simple solution to this problem. Normally, one doesn't need to run lisp on the CRAY to test the code. So, the development can all be done on another machine. After the code is developed and tested to some extent, it can be tested on the CRAY. Thus it is necessary to have a lisp implementation on the host machine for development purposes. It would be of great help for the two lisps to be the same or compatible. Even a different dialect of lisp is better than none. Also, a lisp editor, like NMODE greatly enhances the productivity.

7.2 Rebuilding LISP

The process of linking the CFT code to PSL has been described in detail earlier in this report. The process is actually very straight-forward. However there is another side to it. The process is not flexible. That is, if a change is made in the fortran code, or new code is to be added, the lisp system has to be rebuilt. This process doesn't take much time since it is now done on the CRAY (it took about 25-30 seconds to rebuild our lisp). However, during the development time this is done quite often (we had to rebuild lisp 20-30 times on some days). This further slows down the software development time.

The main reason for this is that the CRAY has a static loader. So, new code can't be linked in at runtime. The scheme used here works, but is not very clean. We can't think of any simple way to get around it. But it would be a great help if one didn't have to rebuild lisp just because the fortran code has been modified.

Another factor in this scheme is the parameter passing. It is very simple to pass parameters from PSL to CFT if the data types are simple, e.g., integers and floats. In our application, only integers were used; hence this was not a factor. However in other applications, it might be necessary to pass complex data types, e.g. lists, or vectors. It is not simple do the parameter passing in these situations. Utility functions have to be added to do these kinds of parameter passing. The knowledge of how the various data types are stored in the two languages is necessary to write the utility functions.

7.3 CSADIE and image processing

It was hoped that CSADIE would have enough subroutines to do all the image processing needs of our system. As it turned out, it has very little which is 'high level' image processing code. Most of the routines are for I/O or for very primitive image processing, e.g., filtering, OR'ing images, etc. Thus a lot of time was spent on developing code for slightly higher level routines.

To do any kind of serious image processing, CSADIE has to be substantially augmented. Another alternative would be to port some already existing image processing package to the CRAY.

Another problem with CSADIE, is that it is tape oriented. One consequence is that an image file can't be opened for reading and writing simultaneously. This prevents in-place computation, which is not uncommon in image processing applications. The only way to do it in the current system is to write into a temp file, close both files, open the original file for writing and the temp file for reading and copy the temp file into the original line by line. Even though it is optimized, it is still very slow.

With bigger memory machines,(CRAY-XMP and CRAY-2), it may be possible to load the whole image into the memory. This will not only overcome the above problem, but also make the system much faster by eliminating the time spent on I/O. However, the size of the image that can be handled will now be dependent on the machine, which is not a positive feature.

A minor comment about the CSADIE manual. It is lots of typos and and some technical inaccuracies.

Acknowledgements

We would like to thank John Aldag and Cal Kirchhof at Cray Research Inc, Mendota Heights, for making this project possible. The help of Dana Dawson and Bill Samayoa at Mendota Heights is also deeply appreciated. Discussions with Dr. Robert Kessler, Department of Computer Science, University of Utah at Salt Lake City helped us enormously during the linking process.

References

[1] Dana H. Ballard and Christopher M. Brown. *Computer Vision.* Prentice-Hall Inc., New Jersey.

[2] Makoto Nagao and Takashi Matsuyama. *A Structural Analysis of Complex Aerial Photographs.* Plenum Press, New York.

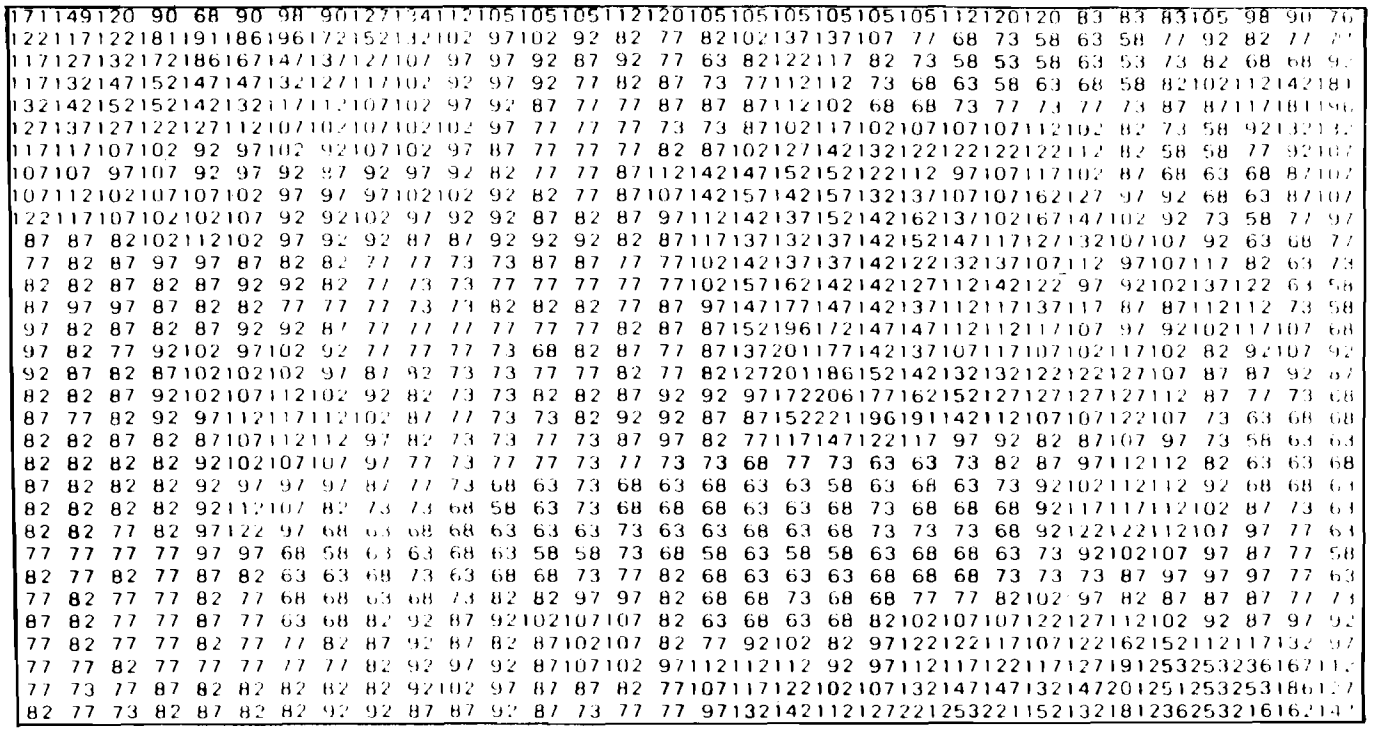
[3] J. Wayne Anderson, Robert R. Kessler and William F. Galway. *The Implementation and Optimization of Portable Standard LISP for the CRAY.*

[4] Ahmad M. Nazif. *A rule-based expert system for image segmentation.* Ph.D. dissertation, Dep. Elec. Engg., McGill University, Montreal, Canada, March 1983.

[5] Steven L. Horowitz and Theodosios Pavlidis. *Picture Segmentaion by a Tree Traversal Algorithm.* JACM, Vol. 23, No. 2, April 1976, pp. 368-388.

[6] William Samayoa. *System Documntation, CSADIE 3.0 for Cray Computers.*

APPENDIX



Original Image

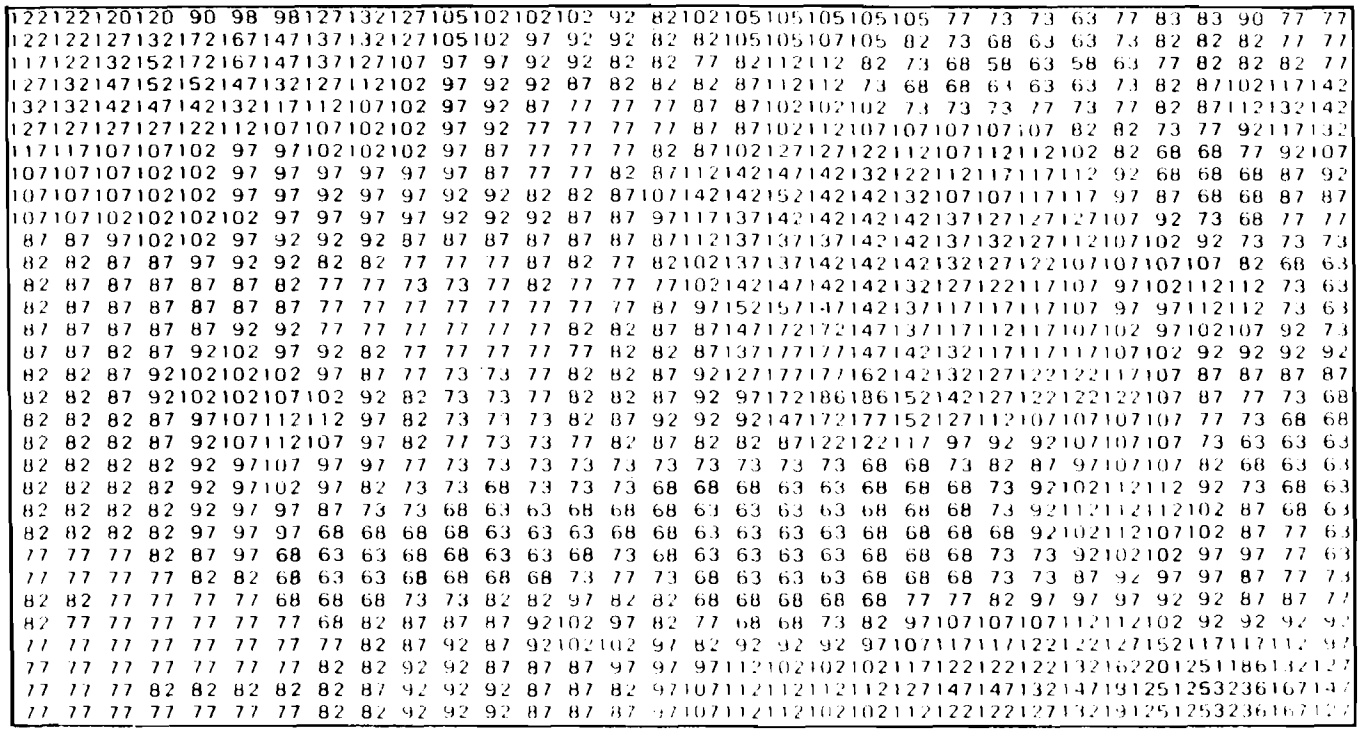
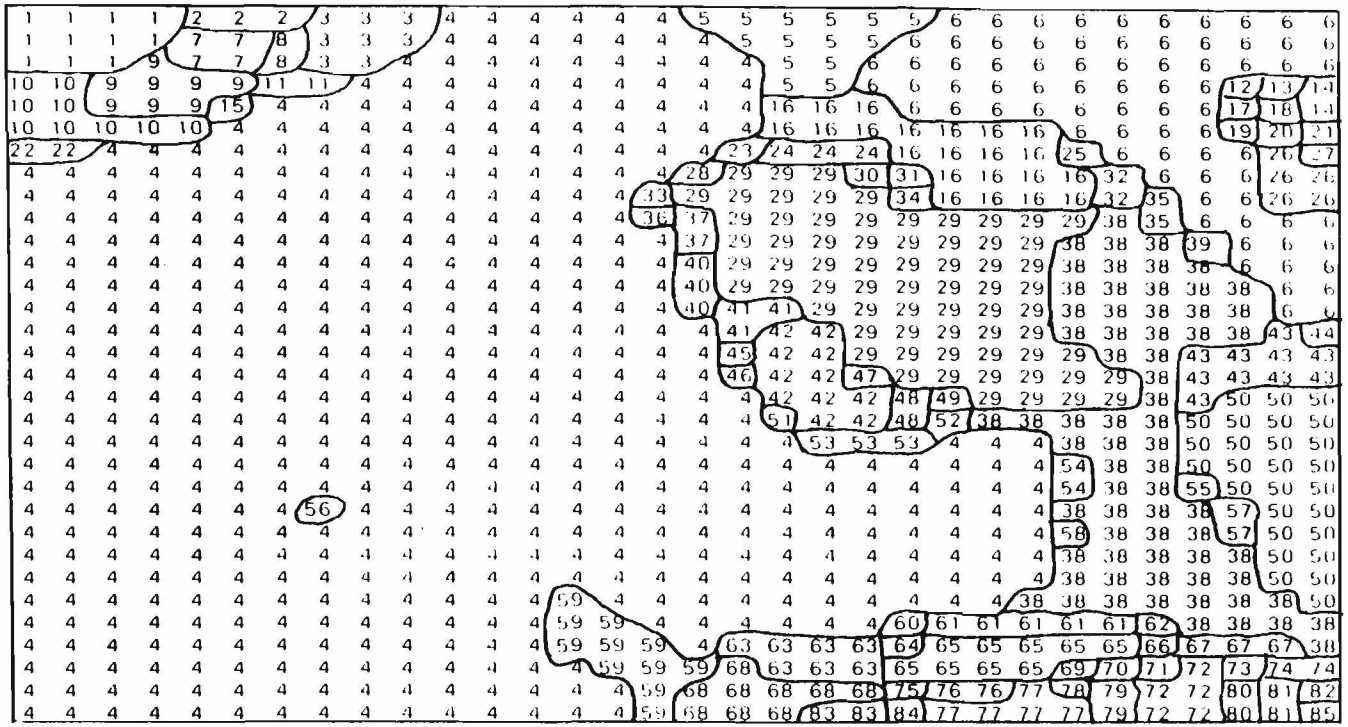
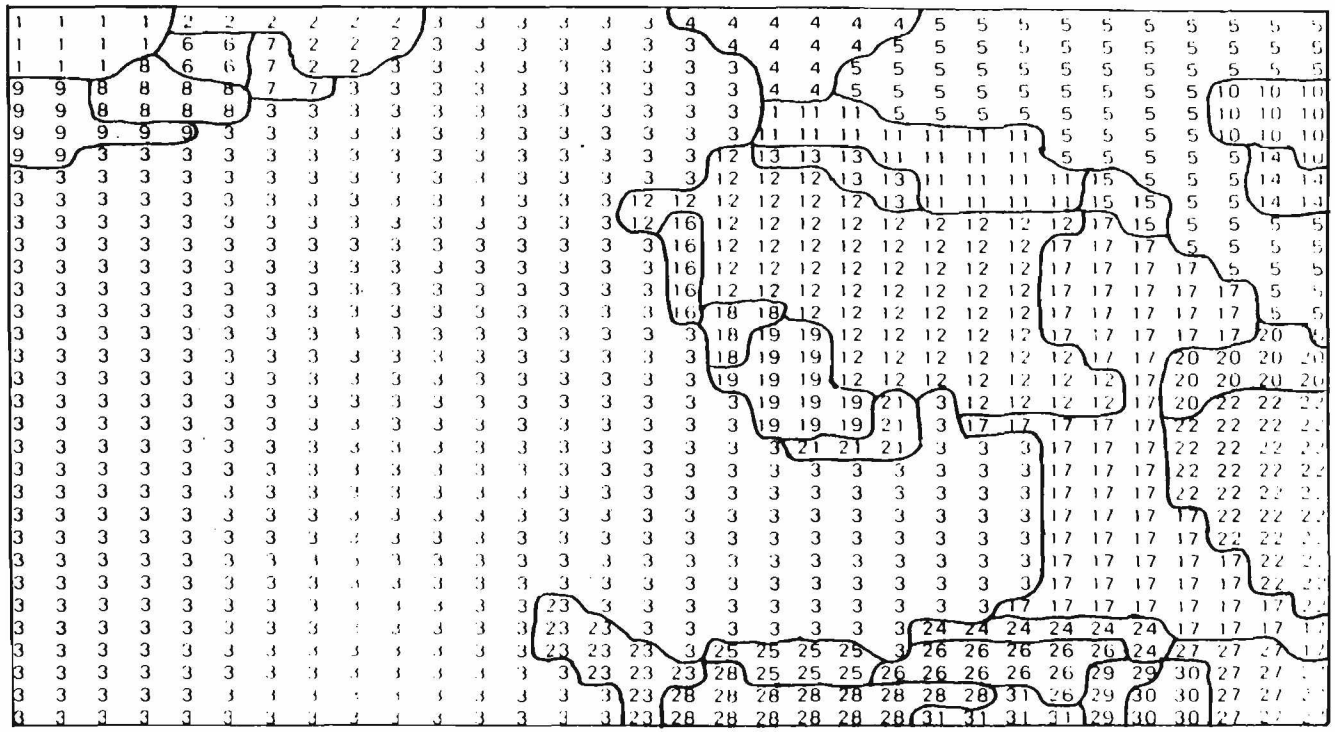


Image after Median Filtering



Initial Segmentation

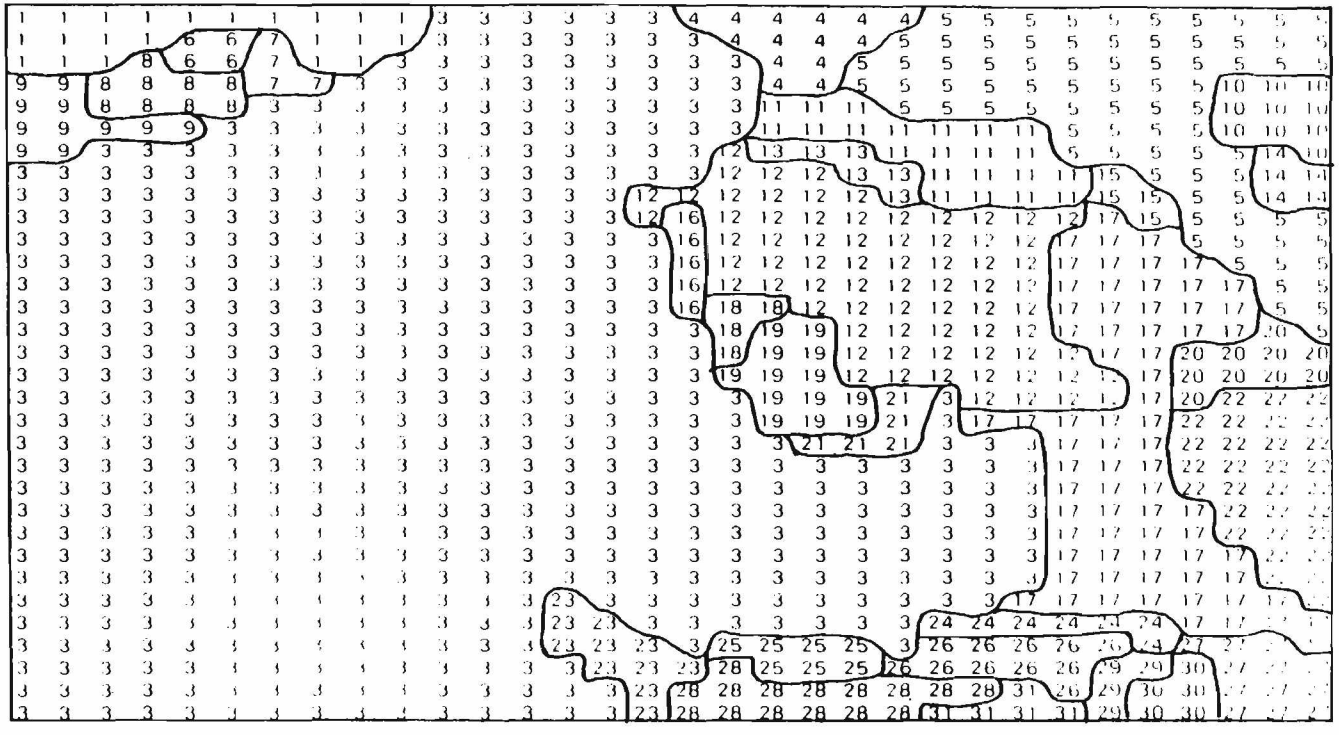
Number of Regions = 85



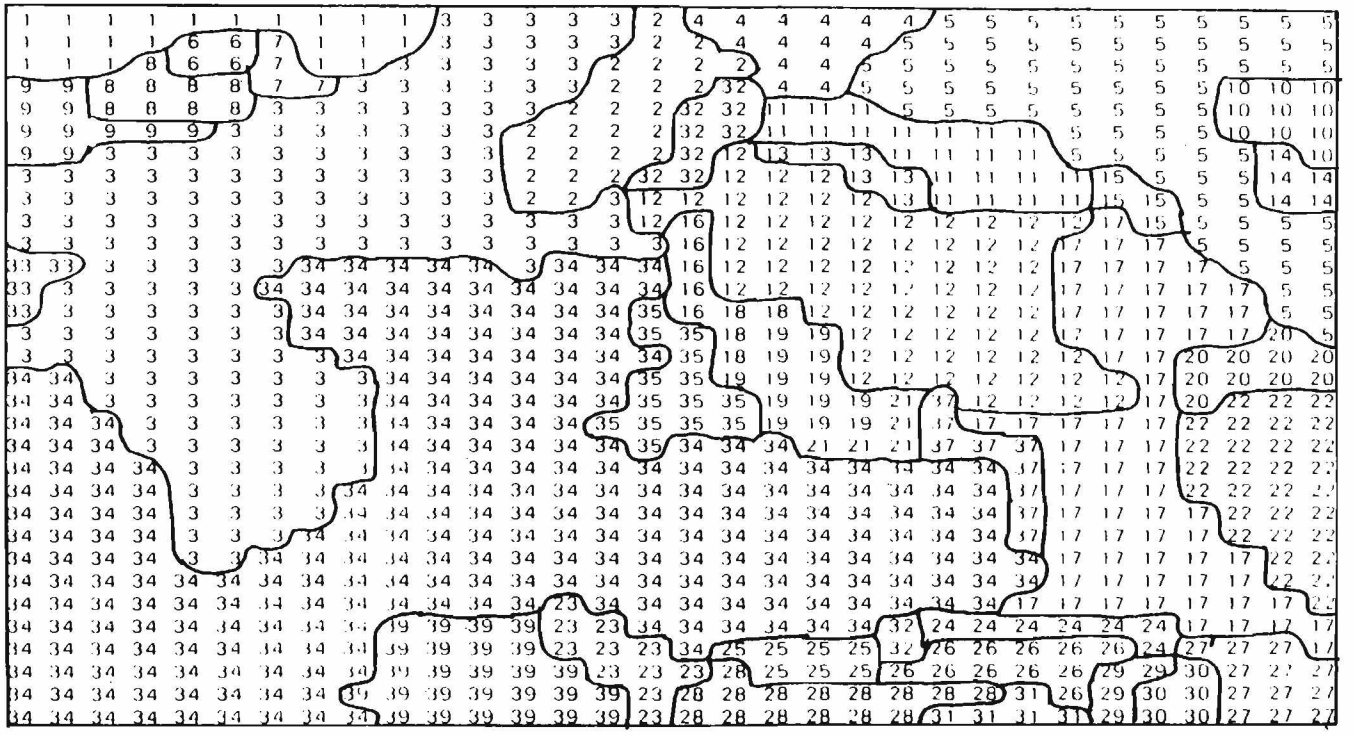
After Merging Small Regions

Minimum Region Size = 4

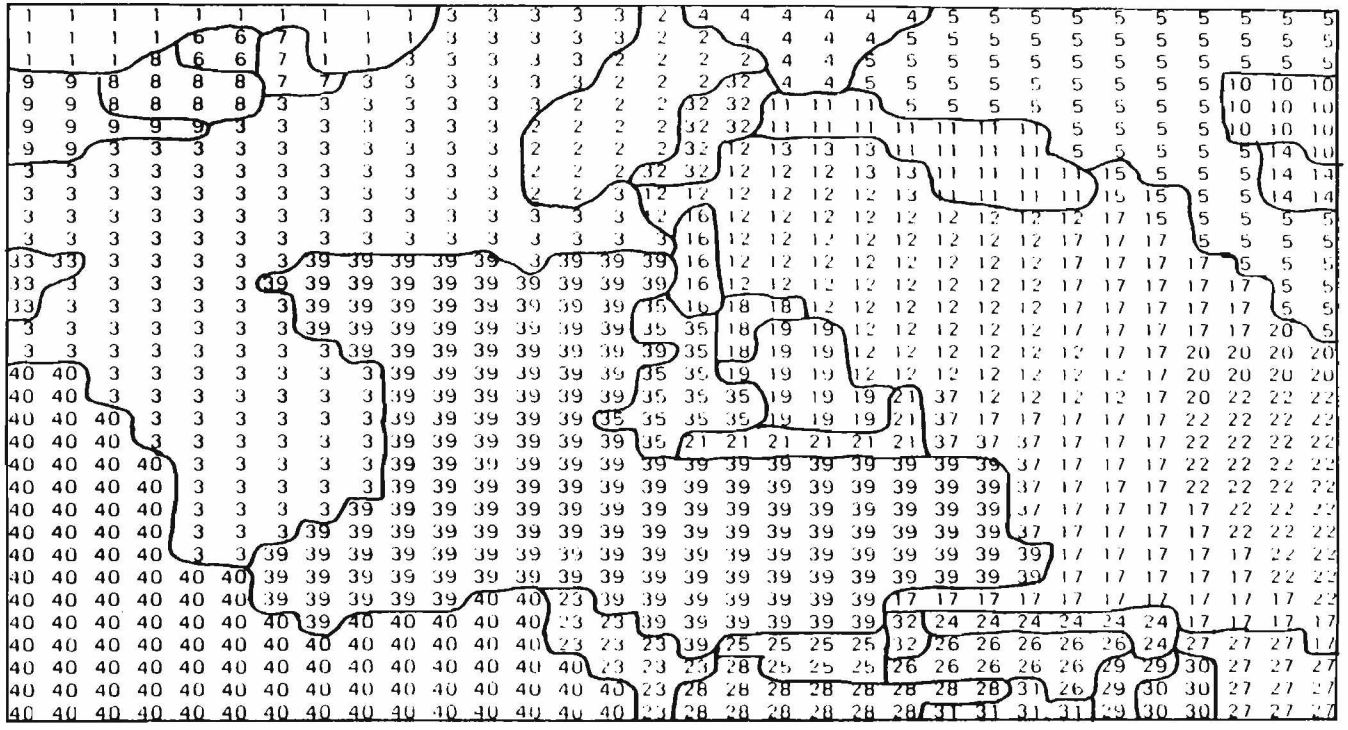
Number of Regions = 31



After Merging Regions 1 and 2



After Splitting Region 3



Final Image

Number of Regions = 37

Total Number of Rules Fired = 6

Time Taken = 37.328 sec