CONNECTED COMPONENTS WITH SPLIT AND MERGE

James J. Kistler
Jon A. Webb

Reprinted from PROCEEDINGS OF THE THE FIFTH INTERNATIONAL PARALLEL PROCESSING SYMPOSIUM, Anaheim, California, April 30-May 2, 1991

Connected Components With Split and Merge

James J. Kistler and Jon A. Webb
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The *split and merge* model is a reasonable method for architecture-independent programming of global image processing operations on parallel architectures. We consider image connected components from the point of view of this programming model, and develop split and merge algorithms that implement various connected components algorithms that have appeared in the literature. The algorithms are implemented in two architectures independent languages we have developed, namely Apply and Adapt. Performance of the algorithms on the **Sun,** the Carnegie Mellon Warp, and the Camegie Mellon Nectar architectures is compared.

1. Introduction

Connected components is an important algorithm at the transition point between iconic and feature-based image processing. It is important because it allows the extraction of geometric information from an image, and geometric information is important in all later stages of processing, regardless of the problem. It is particularly difficult to implement in parallel because:

- The result is global, and depends critically on local events. The presence or absence of a single pixel can affect the labelling of all pixels in the image.
- Processing is irregular. The distribution of processing over the image varies depending on the algorithm, but generally pixels in the middle of homogeneous regions need relatively little processing, while single pixels connecting two homogeneous regions can cause a lot of processing to occur.
- Both the inputs and the outputs are large. In many algorithms at the transition between iconic and feature-based processing, the input is the image, while the output is a small set of features. (For example, the output of connected components may be further processed to extract a list of bounding rectangles of regions.) This can make it easier to partition work among a set of processors.

Since connected components is so important and so hard, it is a natural test case for architecture-independent parallel programming. In order to program the wide array of parallel computers currently available and being developed, it is necessary to develop programming models and languages that allow the programmer to efficiently implement algorithms on parallel computers without being aware of the details of the architectures – i.e., the number of processors, how they are connected, the distribution of data over the processors, and so on. The *split and merge* programming model is one way of doing this. *Split and merge* is based on an extremely common method of programming parallel computers: namely, divide the data into parts according to position, process each part independently on a different processor, and then combine the partial results to create the global result. It has previously been shown that this programming model is powerful: it can be used to compute any function that can be computed in forward or reverse order over a data structure [1].

An implementation of the split and merge model exists in the Adapt programming language [2], and will be used to generate performance figures for the various connected components algorithms on three architectures: the Carnegie Mellon Warp machine [3], the Carnegie Mellon Nectar computer architecture [4], and the **Sun 4.**

2. The Split and Merge Model

In the implementation of the split and merge model in the Adapt programming language, the data is partitioned among processors, each processor computes independently on its portion of the data, and then the results from each processor are combined.

The Adapt language makes some restrictions on this model and embeds it in a specific set of language constructs. The image is always partitioned by rows, and the programmer is encouraged to take advantage of the efficiencies that result from raster-order processing. This is reflected in restrictions placed on the different parts of an Adapt program:

First
: This section can be run only at the beginning of a row. It typically initializes data structures on a processor.

Next
: This section is run once per pixel, and is always preceded by another *Next* or a *First.* Processing "wraps around" the border of the image. The results from the previous *Next* or *First* are available to be reused in the current *Next.*

Combine
: This section is run to combine the results from two adjacent regions of image rows. It has two sets of variables associated with it; a set associated with the top region and a set associated with the bottom region. Its function is to modify the values of the variables associated with the top region so that they have the correct values for the two regions merged together.

TH0363-2/91/0000/0194$01.00 © 1991 IEEE

Last This section is run exactly once, on the final results of all processing. It discards intermediate results and calculates the final values.

For example, in image histogram the *First* section would set the histogram to zero, the Next section would increment a single histogram element, the Combine section would add two histograms, and the Last section would divide the histogram by the total pixel count to create a frequency distribution.

We now consider several algorithms for connected components, starting with the most local methods and ending with the most global.

3. Nearest-Neighbor Propagation

Nearest-neighbor propagation is one of the simplest methods for calculating the connected components of an image. It consists merely of first assigning each significant pixel a unique label, and then repeately propagating to each pixel the minimum of the adjacent labels. This process is repeated until no pixels change.

This algorithm is inherently highly parallel. In fact, as many processors as pixels can be used efficiently, assuming the time to propagate labels between processors is fast. However, it can be extremely slow since the number of iterations, in the worst case (for example, a spiral), is proportional to the area of the image.

The program that assigns the initial label, called `init`, assigns `row*Rows+col+1` to each significant pixel. (The current image position is `(row,col)`, and the image size is `Rows×Cols`.)

The propagation program simply assigns each pixel the minimum of the labels at its neighbors, and sets a flag to true if the pixel value changes. This process is repeated until no pixel changes.

4. Shrink-Expand

Cypher et al. [5] have proposed an algorithm for SIMD mesh-connected machines based on Levialdi's shrinking operation [6]. The algorithm involves $O(\log n)$ iterations of an operation in which each image component is eventually shrunk down to the lower-right pixel of its bounding box (before disappearing). The value at each iteration is saved in a vector for each pixel. Using the vectors, the shrinking process is then inverted, so that after another $O(\log n)$ "expansion" iterations all components have reappeared. When the first pixel of a component reappears it is assigned a unique label; subsequently appearing connected pixels are given the same label.

This algorithm has much better worst-case complexity than nearest-neighbor propagation because the number of iterations is proportional to the "Manhattan diameter" of the largest component rather than its "intrinsic diameter". The former is essentially the perimeter of the bounding box, and is limited to $2\sqrt{n}$; the latter is the maximum of the shortest paths between any two connected pixels, which is bounded by n.

The biggest drawbacks to the algorithm are the relatively large amount of memory it needs to hold the intermediate results vectors and, potentially at least, the I/O requirements. The amount of memory needed is $\sqrt{n}$ bits per pixel[1]. For a worst-case image of moderate size, say 1024x1024, the intermediate vectors will consume 1 gigabit of store. In an Adapt implementation that docs not keep intermediate results on the processors, 100 gigabits of data must be transferred. A realistic implementation must keep intermediate results on the processors.

5. Boundary-Following

The boundary-following algorithm for connected components was originally developed for the Connection Machine[2] by Agrawal et al. [7]. Their algorithm is in two phases. In the first, the boundary pixels of each component are identified and labelled. In the second, interior pixels are labelled by propagating each component's label inwards from its boundary.

Most of the work and complexity of the algorithm is in the first phase, boundary identification and labelling. Assume that the image has been mapped onto the processor array, one processor per pixel. Then, the actions of the first phase are the following:

1. **Mark:** Identify boundary pixels/processors and label them uniquely.
2. **Link:** Link adjacent boundary pixels/processors into boundary "rings."
3. **Merge:** Relabel each ring by propagating around it the label of a "principal" pixel/processor.

Steps 1 and 2 are purely local operations, which require only a 3x3 window around each pixel. The initial label for a pixel is simply the "id" of the processor it is mapped to, and the linking is accomplished through "next" pointers kept in each processor.

The relabelling of rings so that each boundary shares a common label, step 3 above, is done through a technique called distance doubling [8, 9]. Distance doubling allows computation and distribution of the maximum (or minimum) value in a ring of processors in $O(\log n)$ time. At each iteration, each processor communicates with another processor in the ring that is double the distance of that in the previous iteration. Using distance doubling, the principal pixel/processor of a ring is simply that with the largest initial label.

There are two further observations to make about phase

[1] Cypher et al describe a variation of the algorithm which saves only selected intermediate results and recomputes others that it needs during expansion. This reduces the space requirement to $O(\log n)$ bits per pixel, with only constant increase to the time complexity.

[2] Connection Machine is a trademark of Thinking Machines Corporation.

one. First, distance doubling requires rings of processors; therefore, all boundaries must be at least two pixels wide. To ensure this property, the image must be "fattened" prior to phase one and "slimmed" following phase two. The fattened image is called the "dot canvas," and each original pixel corresponds to a 2x2 square in the dot canvas.

The second observation is that a single component may have multiple boundaries. This occurs when there are "holes" in a component. Every component has an outer boundary, and those with holes have additional inner boundaries, one for each hole. A component's boundaries must be unified before the labelling of interior dots can be done. The unification is achieved by having boundary processors scan horizontally in one direction until they encounter another boundary processor. If that processor happens to be a principal, then the next pointers of the two processors are swapped, effectively merging the two rings into one. The scanning/swapping is followed by another distance doubling to relabel rings that have just been merged.

Phase two of the algorithm, interior dot labelling (called `fill`), is straightforward by comparison. Boundary processors simply scan horizontally in one direction, labelling significant pixels until another boundary is encountered. The final step is to slim the dot canvas back to the dimensions of the original image by sampling one out of every quartet of dots.

Much of the boundary following algorithm lends itself to efficient implementation in split and merge. The one operation that does not is distance doubling. which requires arbitrary patterns of memory access. This is handled on the Connection Machine since it provides global shared memory with (nearly) constant access time. However, the split and merge model specifically does not assume the existence of shared memory.

In our implementation distance doubling is performed serially, in the Last section of an Adapt program. Each processor computes a portion of the initial input table, and then forwards its portion on to the Last-processing section. The disadvantage to this approach is obvious: insertion of a serial bottleneck in the computation. Note that the demand on Last-processing by the second approach is not serialization, but shared-memory.

Our split and merge implementation of boundary-following consists of four Adapt programs, `mark`, `link`, `merge`, and `fill`. The first three implement phase one described above, and the fourth implements phase two.

The `mark` maps each input pixel into the dot canvas with boundary dots uniquely labelled. interior dots set to a sentinel value, and other (i.e., insignificant) dots 0. A dense set of labels is assigned by taking advantage of the restriction that we have made on split and merge. In the Combine section we note in an intermediate data structure the number of labels assigned in each region. In the *Last* section we generate a **row** *offset* vector from the intermediate data structure which is output as another parameter of the program. This vector is used in subsequent steps to supply an index to be added to each label.

The `link` program links boundary dots into rings and performs a distance doubling to unify the labels in each ring. The input is the dot canvas and row offset vector from the `mark` program, and a "next-dot" table which is used to initialize the next pointers comprising the rings. The initial label and next pointer of each boundary dot are collected into a boundary table, which is used by the Last section to do the doubling.

The initial `boundary` table is formed by having each processor "forward" `<label, next>` entries generated during Next processing in its region on to the Combine section. The "forwarding" is done via an intermediate data structure. The combining step merges the forwarded entries into a single table, which is then passed to the Last section.

The merge program links inner boundary rings to their outer rings, and performs a second distance doubling to unify the labels in each merged ring. The input is the dot canvas and the (once-doubled) boundary table generated by the `link` program. Rings are merged by "swapping" the next pointers of the principal dots of inner boundary rings with the dots in the outer boundary that are due west of them.

The ***Combine*** and Last sections of merge are identical to those of Link, i.e., preparing for and performing a distance doubling. The output of the program is the (now twice-doubled) `boundary` table.

The `fill` program simply applies the find boundary table to the dot canvas, labelling both boundary and interior dots, and slims the canvas back to the dimensions of the original image. A Next section which notes whether it is "in" or "out" of a component, and what the current label is if the state is "in," is all that is required.

6. Union-Find

Union-find type algorithms make the greatest use of shared knowledge among processors, by assembling an equivalence table that potentially relates any pixel in the image to any other pixel, and then applying this table. This algorithm is commonly used in serial implementations of connected components.

Union-find is actually more powerful than is necessary for computing connected components in images. Suppose we are processing the image row by row, and suppose that image component A lies on a row between two connected image components both labelled B. Then A cannot be unified with any component C that lies outside of B, based on information in the image above this row. This observation has been exploited several times {10, 11, 12}. However, in merging two regions, instead of a region and a row, any two components can be merged. So we cannot use this observation with the split and merge model.

Initial labels are assigned to pixels using the row offset vector method of Section 5. We assign horizontal runs the same label. We must next scan the image and merge labels between rows. We do this by creating a separate equivalence table for each region processed separately a series of application of ***Next.*** The ***Combine*** section then combines these tables.

The separate equivalence tables are created by the `unify` program below:

First Set all entries for labels in the row above the top row in this region to identity.

Next If the current pixel is on, examine the pixel values in the three pixels above, above and to the left, and above and to the right of the current pixel. If any of these are on, perform unions as appropriate between the label for the current pixel and the labels for those pixels.

The ***Combine*** step is simplified by the observation that the only labels that can require merging between two regions are the labels in the top row of the bottom region - these are the only labels that are shared between the two regions. Other labels from the bottom region can be simply copied into the equivalence table for the top region. The ***Combine*** step therefore first copies all labels from the equivalence table for the bottom region into the equivalence table for the top region. Then, for every label in the top row of the bottom region, it works down the chain of equivalences for this label in the equivalence table from the bottom region and sets all these labels equal to the value of the label in the equivalence table for the top region.

The `relabel` step of the union/find algorithm scans the image and assigns each pixel its value in the equivalence table.

7. Implementations of the Split and Merge Model

The Adapt language has been implemented on a variety of architectures; these show the range of possible implementations of the split and merge model. We now describe each of these implementations.

7.1. Serial implementation

On a serial architecture (in the experiments reported here, a Sun 4/330) Adapt can be implemented without use of the ***Combine*** section. All that is necessary is to execute ***First*** once, at the beginning of the image, and then executed ***Next*** once for every pixel in the image, in raster order; finally, ***Lust*** is executed.

7.2. Implementation on the Carnegie Mellon Warp Machine

Adapt was implemented in two different ways on the Carnegie Mellon Warp machine [3]. This computer consists of an input host that feeds data from a large memory to a linear array of ten systolic processors, each of which can communicate only with its left and right neighbors. The last cell is connected to an output host which stores results into a large memory that is shared with the input host. Images are processed by feeding them in raster order through the array. Important limitations and advantages arise from the characteristics of the systolic cells; each has high I/O bandwidth (40 MB/s) and low latency (200 ns) communication, but a relatively small data memory (32 KWords). The entire computer is programmed in a Pascal-level language called W2.

The first Adapt implementation was based on the previous implementation of Apply on Warp [13]: the Adapt compiler was a slightly modified version of the Apply compiler. The image was divided by columns, with each systolic cell taking a contiguous tenth of the image. For each row of the image, cell 0 executed the ***First*** section and then executed ***Next*** for each of its columns. It then transferred all intermediate variables to cell 1, which executed ***Next*** for each of its columns, and so on. In the meantime, cell 0 went on to execute ***First*** and ***Next*** for the next row. A pipeline of processing was thus established with all cells working simultaneously after the tenth row of the image was processed. When the output reached the output host, it performed a ***Combine*** operation between the current row's variables and the variables for the region above the current row. Finally, after the entire image had been processed, the output host performed ***Last.***

This implementation minimizes memory use on the cells; each must provide storage only for one-tenth of a row of the image and the variables used in ***Next;*** they need not store the variables used in ***Combine*** at all. It also involves the output host in the processing, and the ***Combine*** processing is overlapped with the ***Next*** processing on the cells. But the disadvantages of this method are severe. First, the ***First*** and ***Combine*** sections are executed once for every row in the image; if the processing in these sections is expensive, this can seriously add to the computational burden of the program. Second, and more subtly, all intermediate variables must be transferred from one cell to another after the execution of the cell's ***Next's.*** This adds I/O overhead to the processing on a cell, but more importantly, it can actually eliminate parallelism. The systolic queues between cells in Warp have buffer space for only 512 words; if the size of the intermediate variables exceeds this, the sending cell blocks until the receiving cell has read the data. If this happens, no pipeline of processing will be created; instead, only a single cell at a time will execute its ***Next's.***

In the second implementation the image was divided by rows, again with each cell taking a contiguous tenth. Each cell executed ***First*** at the first pixel in its region, and then ***Next*** for all pixels in its region. No 1/0 took place, except for the input and output of images, until the entire image was processed. Then cell 0 transferred its variables to cell 1, which executed ***Combine*** and then transferred its variables to cell 2, and so on. The ***Last*** section was executed on cell 9.

This implementation was far more efficient than the column-partitioned implementation. In fact, the performance of Adapt programs sometimes exceeded handwritten W2 code for the same section. A notable example is image histogram, which was carefully optimized in its handwritten implementation: for a 512x512 image, histogram took 163 ms in the hand-written version but only 119 ms in the Adapt version.

The principal disadvantage of this method is that it requires a lot of memory on the cells. Each cell must store a complete row of the image, as well as two copies the variables used in ***Combine,*** one from the previous cell and one from this one.

7.3. Implementation on the Nectar Architecture

The Nectar architecture [4] consists of a crossbar switch, called a HUB, connected by fiber optics to message passing processors, called CABs, which share bus space with a host processor. In the experiments reported here, all of the host processors are Sun 4/330s, and the CABs are installed in the VME bus.

In the Adapt implementation on Nectar, a special process called a master distributes the images to and collects results from several slave processes. The master and slaves run on the Sun 4/330s; it is also possible to run processes on the CABs, but we did not do this because the CABs do not have hardware floating point. The master divides the image into a number of slices by row, and deals out the slices to the slaves as they request them; each slave gets an initial slice, executes ***First*** and ***Next*** as in the row-partitioned Warp implementation, and then, when it gets halfway through processing its slice, it sends a message to the master requesting more data. The master sends more slices to the slaves that request data. After all the slices have been sent to slaves, the master sends out a map telling each slave where all the slices are.

The map is used by the slaves to execute their ***Combine's*** independently of the master. Each slave keeps track of the slices allocated to it. Slices are combined in a binary tree fashion. Each slave examines its slices, and if it has two slices that can be combined, it does so; if it has any slices for which the upper corresponding slice is on another slave, it sends its slice directly to that slave; and it receives and stores any slices intended for it. The final result of all of these ***Combine's*** is a set of variables on slave 0; this is sent to the master, which executes the ***Last*** section.

This implementation takes advantage of Nectar's crossbar connectivity and the large memories available on the host processors. It uses special characteristics of Adapt to improve efficiency; namely, the knowledge that a slave has that it is halfway through processing by reading the middle row of its slice. It is also automatically load balancing.

There are some problems with the current implementation. The most significant is that the images and other data structures must be sent across the VME bus from the host processors to the CABs and back again. While the CAB to CAB 1/0 bandwidth is fairly high (10 MB/s), the usable VME bandwidth is only 1-4 MB/s.

Another disadvantage of running on the host is that a Unix process must be started for each slave. This takes a substantial amount of time, both because of the process startup time and because the code for the slaves is transmitted over the Ethernet in the current prototype Nectar implementation.

8. Experimental Results

8.1. Results from Sun and Warp

We are comparing different algorithms for the same problem implemented on several difcrent architectures. Several insights can be drawn from this.

First, we can normalize for the different processor architectures by comparing the same algorithm on different architectures. This allows us to combine insights from different architectures.

Second, we can compare the requirements the different algorithms make of the architecture and make general statements, justified by our performance observations, on the performance of different algorithms on parallel architectures in general.

Table 8-1 presents the experimental results for the serial and both Warp implementations of Adapt on all algorithms, for two images: one including many small (several pixel) regions, and the other including a few large **(64x64)** regions.

In many cases the column-partitioned method is orders of magnitude slower than the row-partitioned method. This is because the column-partitioned method requires synchronization between cells at the end of processing every row, while the row-partitioned method synchronizes at the end of processing every ten rows. Also, the innermost loop in the column-partitioned method has a loop bound only one-tenth as large as the loop bound in the row-partitioned method, because it iterates over all pixels in a cell's columns, rather than all the pixels in a row. These factors introduce a significant overhead (as in `init`, `expand`, and `relabel`), which grows larger when there is data to be exchanged between cells (as in `prop` and `shrink`). When the data structures grow large (as in `scan` and `unify`) the limited queue size between cells destroys parallelism, resulting in orders of magnitude differencebetween the two methods.

When we compare the algorithms based on their performance and architectural requirements, we observe:

- Propagate, shrink-expand, and union-find require only local connections between processors. The border-following algorithm is the only one that requires long-distance communication.

Algorithm	Sun 4/330	Warp (Cdunm)	*Wup* (Row)	Sun 4/330	Warp (Column)	Warp (Row)
	Small Region Image			Large Region Image		
Init	0.316	0.416	0.232	0.333	0.471	0.257
Prop	257'	3.14[1]	1.69'	73.3'	102'	59.5[2]
Total	2.89	3.56	1.92	73.6	102.	59.8
Shrink-Expand						
	Small Region Image			Large Region Image		
Shrink	3.28'	2.92'	1.23'	169.[3]	149.'	64.4'
Expand	5 28[1]	3.18[1]	1.96	265.[3]	162'	97 3'
Total	8 56	6.10	3.13	434.	311.	162.
	[illegible]			[illegible]		
Mark	1.68	1.92		1.75	2.39	
Link	6.50	39.4		6.47	38.4	
Merge	3.88	28.3		3.83	28.5	
Fill	3.35	[illegible]		3.35	1.18	
Total	15.4	70.8		15.4	70.5	
Union-Find						
	Small Region Image			Large Region Image		
Scan	0.853	6.50	0.347	0.769	6.50	0.356
Unify	1.22	55.5	0.526	0.783	67.5	0.523
Relabel	0.456	0.528	0.428	0.354	0.549	0.429
Total	2.53	62.5	1.30	1.91	74.5	1.31

Table 8-1: Performance of Connected Components on Sun and Warp.
All times in seconds. Image size: 512x512.
'Five iterations.
[2]One hundred twenty-eight iterations.
[3]Two hundred fifty-five iterations.

- Propagate and shrink-expand require small memories at processors. Boundary following makes moderate memory requirements in our implementation. Union-find is the only algorithm that requires large memories.
- Shrink-expand and border-following make use of only very limited processor facilities; they do not manipulate large integers or do any complex calculations. Propagate uses large integer comparisons, and union-find makes use of several different complex processor features, including **local** addressing.
- The execution times of propagate and shrink-expand depend linearly on the size of the regions in the image. Border-following and union-find are both largely independent of the region size.
- Propagate, shrink-expand, and border-following can all be implemented on very large processor arrays. Propagate and shrink-expand are entirely local in their action (except in the calculation of when to stop repeating), while border-following's global calculations can be done in parallel, given long-distance communication. Union-find can only be implemented on relatively small processor arrays, because in its second step it creates **a** single data structure (the equivalence table). The overhead for creating this table increases with the number of processors.

The last observation on union-find can be quantified by observing the variation in execution time with the number of processors in the `unify` step. The time can be approximated by $t=i+x/n+c\times(n-1)$, where t is the total **ex**ecution time, i is the overhead independent of number of cells (mainly due to I/O of the images to and from the Warp array), x is the execution time on **a** single cell, c is the time for **a** *Combine* operation, and n is the number of cells. With this model and data from runing `unify` with different numbers of cells in the Warp array, we obtain i=161 ms, x=2.76 s, and c=10.0 ms. Given these numbers and different Adapt implementation methods on one-dimensional, two-dimensional, and binary tree-connected processor arrays, we can calculate the maximum number of cells that can be applied to the union-find algorithm: these are 17, 42, and 190 for one-dimensional, two-dimensional, and binary tree-connected arrays, respectively. We can also calculate the most cost-effective array size [14]: these are 10, 17, and 34, respectively. In other words, the maximum array size is limited to a few tens of cells, regardless of the method of implementation of Adapt.

By comparison, the maximum array size for propagate, shrink-expand, and border-following is much larger – perhaps **as** large **as** one processor per pixel, depending on the details of the architecture. So we might expect that these algorithms potentially offer much better speedup than union-find. However, a cost-benefits analysis based on the Sun **4** execution times sheds doubt on this for propagate and shrink-expand. These algorithms are much slower than union-find; assuming **a** maximum region size of 128 pixels, propagate is 38.6 times slower and shrink/expand is 227 times slower. In order for the performance of these algorithms to exceed the performance of union-find, the processor array would have to be this much larger in order to make up for the lost performance: a 17-processor union-find array versus a 656 processor array for propagate, or a 3860 processor array for shrink-expand. Now, the chief advantage of these algorithms when compared with union-find is that they require much smaller per-processor memories than union-find; requiring much larger processor arrays adds additional cost. **As** a result, a the union-find algorithm is much more likely to be cost-effective than the other algorithms, except when very large processor arrays must be used.

The border-following algorithm is only 8.08 times slower on the Sun **4** than union-find: a 17-processor union-find array is equivalent to a 137-processor border-following array. **As** with propagate and shrink-expand, the border-following algorithm requires much less per-processor memory than union-find. This suggests that border-following may be a reasonable alternative to union-find on processor arrays of hundreds of processors or more. If such an array is organized **as** a mesh, and the image **is** divided into blocks (so that each processor takes a rectangle of pixels, with adjacent processor taking ad-

jacent rectangles) then in almost all cases the communication paths formed in the link step of border-following will be short, and will not actually require hardware implementation of a general-purpose switch as in the Connection Machine implementation of union-find.

We conclude the following:

- For small to medium-sized processor arrays (tens of processors) union-find is the best algorithm. It has superior performance overall. In such arrays, it is important not to allocate too many processing nodes to the merge step; the best number depends on how the *Combine* step is implemented, and ranges from ten to a few dozen.
- On medium-sized arrays (hundreds of processors), the border-following algorithm is preferable. It makes smaller memory requirements and offers better performance than propagate or shrink-expand. Union-find is not feasible on such large arrays.
- On large mesh arrays (thousands of processors or more) propagate or shrink-expand is the choice. These algorithms make almost no use of global operations, and depend on local interprocessor communication only. The choice between the two is to be made based on the availability of fast bit-serial operations; if a speedup of six or more is available from such operations, then shrink-expand is likely to be faster than propagate. Note that this observation applies to the Connection Machine, which offers a general-purpose switch communication mechanism as well as mesh communication; local scan operations make it possible to implement propagate much more efficiently than border-following [15].
- The border-following algorithm is the only algorithm presented here that does not fit the split and merge model, as a result of the distance-doubling step.

8.2. Results from Nectar

Even with the automatic load balancing done by the Nectar compiler, and excluding slave startup time from the measurements, the Nectar data exhibits a **great** deal of variability from run to run. A typical performance curve is shown in Figure 8-1. This figure shows a number of outliers in different runs: the actual execution time of the `shrink` program with four slave nodes **is** typically about 730 ms, but times up to 2 s are not uncommon, and a time of over 4 s was recorded.

We have therefore used a combination of outlier rejection and Monte Carlo analysis to analyze this data. Outliers are rejected by sorting the data for each number of slaves, and starting with a small number of the smallest execution times, incrementally add execution times until the variance is observed to increase rapidly. All larger exection times are then rejected.

Monte Carlo analysis is used by fitting a simple model to data with the same mean and variance as the actual

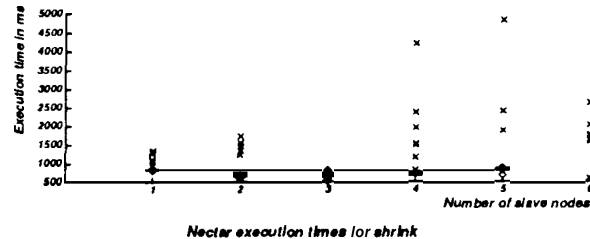

Figure 8-1: Nectar Execution Times for One Iteration of `Shrink`

data, and then predicting execution times with each number of slaves. This gives a predicted mean and variance of execution times as shown in Figure 8-1.

This analysis is not complete: it observably does not account for all aspects of the Nectar data. But we can use it to characterize the current Nectar implementation.

Overall, the current Nectar implementation of Adapt does not compare favorably with the Sun implementation, which used just one Sun 4/330 (as opposed to Nectar's Sun 4/330 master and Sun 4/330 slave nodes). the fundamental reason for the loss of performance is transferring images over the **VME** bus between the **Sun** memory and the Nectar CAB for transmission to the slave nodes.

For example, in the data in Figure 8-1 the best execution time, with three slave nodes, is about 680 ms. The execution time of one iteration of shrink on the Sun 4/330 (Table **8-1**) was about 660 ms. In each execution of the shrink operation, 0.5 MB of data is transferred over the **VME** bus: **a** transfer rate of about 740 KB/s was achieved. This is near the maximum transfer rate of the **VME** bus under **Sun** programmed I/O. Only a higher speed interface, such as HIPPI, to Nectar will improve these results.

We also observed an interesting behavior related to our implementation of the ***Combine*** section on Nectar. The relevant data is shown in Figure **8-2.** Note the "bump" with two slave nodes. We believe this bump is due, at least in part, to the simple method we have chosen for allocation of slices to slaves. In a system with low overall load, the slices will be dealt alternately to slaves; thus, all even-numbered slices end up in slave 0, and all odd-numbered slices in slave 1. During the ***Combine*** step, slave 1 will send all of its slices to slave 0, which will then to ***all*** of the ***Combine*** operations; slave 1 will be completely idle. This and the overhead of having to do more ***Combine*** operations help create the bump. Clearly, random assignment of slices or some other technique that helps to balance load **during** the ***Combine*** phase is needed.

The Adapt **Nectar** implementation is still in an early phase. We expect that further refinement of the implementation and the addition of new hardware to Nectar (including a HIPPI interface expected for 1991) will substantially improve the performance of Adapt here.

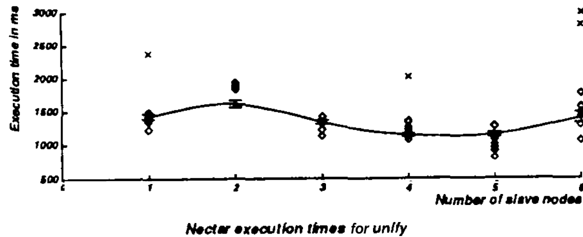

Figure **8-2:** Nectar Execution Times for One Iteration of `Unify`

9. Summary

We have demonstrated that it is possible to implement an architecture-independent programming language for global image processing operations on a variety of computer architectures.

We have shown how various parallel connected components algorithms can be implemented with a common programming model. Only the border-following algorithm, which manages a large distributed data structure through **a** general-purpose switch, does not fit the model.

We have shown how different architectures and **al**gorithms can be compared fairly through the use of such an architecture-independent language.

Acknowledgments

Thanks to George Gusciora, who created the row-partitioned implementation of Adapt on Warp.

This research was supported by the National Science Foundation under grant MP-8920420. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the **Na**tional Science Foundation or the U.S. government.

Bibliography

1. Wcbb, J. A., "The Divide and Conquer Model for Parallel Computation" . Submitted

2. Webb, J. A., "Architecture-Independent Global Image Processing", *Tenth International Conference on Pattern Recognition,* International Assocation for Pattern Recognition, Atlantic City, NJ, June 1990, pp. 623-628.

3. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, **O.** and Webb, J. A., "The Warp Computer: Architecture, Implementation and Performance", *IEEE Transactions on Computers,* Vol. C-36, No. 12, December 1987, pp. 1523-1538.

4. Arnould, E. A., Bitz, F. J., Cooper, E. C., Kung, H. T., Sansom, **R.** and Steenkiste, P. A., "The Dcsign of Nectar: A Network Backplane for Hctcrogcncous Multicomputcrs", *Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III),* ACM, April 1989.

5. Cypher, R., Sanz, J., and Snyder, L., "Algorithms for Image Component Labelling on SIMD Mesh-Connected Computer", *IEEE Transactions on Computers,* Vol. 39, No. 2, February 1990, pp. 276-281.

6. Lcvialdi, **S.,** "On Shrinking Binary Pictures", *Communications of the ACM,* Vol. 15, No. 1, 1972, pp. 7-10.

7. Agrawal, A., Nekludova, L., Lim, W., "**A** Parallcl O(log N) Algorithm for Finding Conncctcd Components in Planar Images", *Proceedings of the 1987 International Conference on Parallel Processing,* 1987.

8. Wyllie, J.C., "The Complexity of Parallcl Computations", Tech. rcport TR 79-387, Cornell Univcrsity Dcpartmcnt of Computcr Scicncc, August 1979.

9. Lim, W., Agrawal, A., Nekludova, L., ''A Fast Parallel Algorithm for Labeling Conncctcd Components in Image Arrays", Tcch. report 15, Thinking Machincs Corporation. 1986.

10. Kung, H.T. and Wcbb, J.A., "Global Operations on the CMU Warp Machine", *Proceedings of 1985 AIAA Computers in Aerospace V Conference,* American Institute of Acronautics and Astronautics, Octobcr 1985, pp. 209-218.

11. Lumia, R., Shaprio, L, and Zuniga, O., "A New Connected Components Algorithm for Virtual Memory Computers", *Computer Vision, Graphics, and Image Processing,* Vol. 22, 1983, pp. 287-300.

12. Schwartz, J., Sharir, M., and Siegcl, **A.**, "An cfficicnt algorithm **for** finding connected components in a binary image", Technical Report 154, New York Univcrsity Dcpartmcnt of Computcr Scicncc, February 1985.

13. Wallace, R. **S.,** Wcbb, J. A. and Wu, I-C., ***"Ar**chitecture Indepcndcnt Imagc Processing: Pcrformance of Apply on Diverse Architccturcs", *Computer Vision, Graphics, and Image Processing,* Vol. 48, 1989, pp. 265-276.

14. Eager, D. L., Zahorjan, J., and Lazowska, E. D., "Speedup versus efficiency in parallel systcms", *IEEE Transactions on Computers,* Vol. 38, No. 3, 1989, pp. 408-423.

15. Blclloch, G.. Personal communication