Performance of Warp on the DARPA Architecture Benchmarks[1]

Jeff Deutch, P C Maulik, Ravi Mosur, Harry Printz, Hudson Ribas, John Senko, P S Tseng, Jon A Webb, I-Chen Wu

3 September 1987

CMU-CS-87-148

Abstract

Warp was a participant in the DARPA Architecture Workshop Benchmark Study, which compared performance of a variety of architectures for image processing on image processing tasks from low- and mid-level vision. We present algorithms and performance figures resulting from this study. These algorithms can performance numbers can be used as a guide to Warp programming at the time of this study. Based on these performance figures, we evaluate the architectural decisions made in the Warp design.

The research was supported in part by Defense Advanced Research Projects Agency (DOD), monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and Naval Electronic Systems Command under Contract N00039-85-C-0134, in part by the US Army Engineer Topographic Laboratories under Contract DACA76-85-C-0002, and in part by the Office of Naval Research under Contracts N00014-80-C-0236, NR 048-659, and N00014-85-K-0152, NR SDRJ-007.

---

[1] A version of this paper will appear in *International Conference on Parallel Processing for Computer Vision and Display*, Leeds, UK, January 1988.

Table of Contents

1 Introduction
2 Warp Status
3 Vision Programming On Warp
    3.1 Input Partitioning
    3.2 Output Partitioning
    3.3 Pipelining
4 Laplacian
5 Zero Crossings Detection
6 Border following
7 Connected components labelling
    7.1 Sketch of the Algorithm
        7.1.1 Vocabulary and Notation
        7.1.2 The Algorithm
    7.2 Asymptotic Running Time
        7.2.1 Parallel-Sequential-Systolic Algorithm
        7.2.2 Parallel-Sequential-Parallel Algorithm
    7.3 Implementation Details
        7.3.1 Warp Architectural Constraints
        7.3.2 Vax Implementation
        7.3.3 Warp Implementations
8 Hough transform
9 Convex Hull.
10 Voronoi Diagram
11 Minimum spanning tree
12 Visibility
13 Graph Matching
14 Minimum-cost Path
15 Warp Benchmarks Summary
16 Evaluation of the Warp Architecture
    16.1 Memory
    16.2 Number of processing elements
    16.3 External host

List of Figures

Figure 1: Folding columns 4
Figure 2: Using results from previous steps 4
Figure 3: Convolving and storing column sums 5
Figure 4: Adding appropriate column sums 5
Figure 5: Input 10
Figure 6: Labels after parallel phase 10
Figure 7: Labels after sequential phase 10

List of Tables

Table 1: Optimized Symmetric Convolution 5
Table 2: Final maps 11
Table 3: Label Computation 11
Table 4: Vax implementation timings 14
Table 5: Estimated WW Warp timings 15
Table 6: Estimated PC Warp timings 15
Table 7: Estimated *i*Warp timings 16
Table 8: Operation counts for Voronoi diagram 18
Table 9: Warp Benchmark Summary 21

1 Introduction

The DARPA Architecture Workshop Benchmark Study was conceived for these reasons:

- To arrive at an initial understanding of the general strengths and weaknesses for image understanding (IU) of the architectures represented.
- To project needs for future development of architectures to support IU.
- To promote communication and collaboration between various groups within the CS community which are expected to contribute to development of real-time IU systems.

The benchmarks chosen represented common image processing operations from low and middle level vision, but did not include high level image processing operations, such as recognition; these operations were felt to be too ill defined at present to properly evaluate machine architectures.

Warp was one of the participants in the study. This paper is a summary of our results, which reflect the performance on Warp on this level of vision, and can also serve as a guide for programming Warp in this area.

The precise definition of the image processing operations as given to the participants was as follows:

1. **Laplacian.** (Edge detection is done by this and the following two tasks. For edge detection, the input is a 8-bit digital image of size 512×512 pixels.) Convolve the image with an 11×11 sampled "Laplacian" operator [10]. (Results within 5 pixels of the image border can be ignored.)
2. **Zero-crossings Detection.** Detect **zero-crossings** of the output of the operation, i.e. pixels at which the output is positive but which have neighbors where the output is negative.
3. **Border Following.** Such pixels lie on the borders of regions where the Laplacian is positive. Output sequences of the coordinates of these pixels that lie along the borders. (On **border following** see [16, Section 11.2.2].)
4. **Connected component labeling.** Here the input is a 1-bit digital image of size 512×512 pixels. The output is a 512×512 array of nonnegative integers in which
   a. pixels that were 0's in the input image have value 0.
   b. pixels that were 1's in the input image have positive values; two such pixels have the same value if and only if they belong to the same connected component of 1's in the input image. (On connected component labeling see [16, Section 11.3.1].)
5. **Hough transform.** The input is a 1-bit digital image of size 512×512. Assume that the origin (0,0) image is at the lower left-hand corner of the image, with the x-axis along the bottom row. The output is a 180×512 array of nonnegative integers constructed as follows: For each pixel (x,y) having value 1 in the input image, and each i, $0<i<180$, add 1 to the output image in position (i,j), where j is the perpendicular distance (rounded to the nearest integer) from $(0,0)$ to the line through (x,y) making angle i-degrees with the x-axis (measured counterclockwise). (This output is a type of Hough transform; if the input image has many collinear 1's, they will give rise to a high-valued peak in the output image. On Hough transforms see [16, Section 10.3.3].)
6. **Convex Hull.** (For this and the following two geometrical constructions tasks the input is a set S of 1000 real coordinate pairs, defining a set of 1000 points in the plane, selected at random, with each coordinate in the range $[0,1000]$. Several outputs are required as follows.) An ordered list of the pairs that lie on the boundary of the convex hull of S, in sequence around the boundary. (On convex hulls see [15, Chapters 3-4].)
7. **Voronoi Diagram.** The Voronoi diagram of S, defined by the set of coordinates of its vertices, the set of pairs of vertices that are joined by edges, and the set of rays emanating from vertices and not terminating at another vertex. (On Voronoi diagrams see [15, Section 5.5].)
8. **Minimal Spanning Tree.** The minimal spanning tree of S, defined by the set of pairs of points of S that are joined by edges of the tree. (On minimal spanning trees see [15, Section 6.1].)
9. **Visibility.** The input is a set of 1000 triples of triples of real coordinates $((r,s,t),(u,v,w),(x,y,z))$,

defining 1000 opaque triangles in three-dimensional space, selected at random with each coordinate in the range [0,1000]. The output is a list of vertices of the triangles that are visible from (0,0,0).

10. **Graph matching.** The input is a graph G having 100 vertices, each joined by an edge to 10 other vertices selected at random, and another graph H having 30 vertices, each joined by an edge to 3 other vertices selected at random. The output is a list of the occurrences of (an isomorphic image of) H as a subgraph of G. As a variation on this task, suppose the vertices (and edges) of G and H have real-valued labels in some bounded range; then the output is that occurrence (if any) of H as a subgraph of G for which the sum of the absolute differences between corresponding pairs of labels is a minimum.

11. **Minimum-cost path.** The input is a graph G having 1000 vertices, each joined by an edge to 100 other vertices selected at random, and where each edge has a nonnegative realvalued weight in some bounded range. Given two vertices P, Q of G, the problem is to find a path from P to Q along which the sum of the weights is minimum.

In what follows, we first describe the current Warp status, and then describe our work on each of the algorithms. We do not review the Warp architecture or programming environment here, since complete reviews are available elsewhere [1, 2, 3, 4, 5].

2 Warp Status

There are three operating Warp machines at Carnegie Mellon. Two of them are prototypes. One was built by General Electric Radar Systems Department (Syracuse) and the other by Honeywell Marine Systems Department (Seattle). Both consist of a linear array of ten cells, each giving 10 MFLOPS, for a total of 100 MFLOPS, and operate in an identical software environment. These machines are referred to as WW Warp, since they are of wirewrap construction. The machines are fed data by MC68020 processors, called the "external host," and the whole system is controlled from a Sun 3/160.

The third machine is a production machine, one of several being constructed by General Electric Corporation. The production machines are built from printed-circuit boards, and are called PC Warp. The baseline power of these machines is also 100 MFLOPS, although they can easily be expanded to 160 MFLOPS by simply adding more cells. (The array can be expanded still further, but this requires a special repeater board and a second rack). The PC Warp is changed in several ways from the WW Warp: cell data and program memories are larger, there is on-cell address generation, and there is a large register overflow file to provide a second memory for scalars. Some of these improvements imply an increased speed on some of the benchmarks, as will be noted. For example, because of on-cell address generation, the cells is able to tolerate an arbitrary skew in computation, which makes it possible to overlap input, computation, and output in many algorithms. Also, improved processor boards in the external host allow improved I/O rates between Warp and the host through DMA, removing the host I/O bottleneck in many cases. Finally, since each cell has more local control, it is possible to make Warp computation more data dependent, by allowing data-dependent I/O between cells, as well as heterogeneous computation (different programs on different cells).

Carnegie Mellon and Intel Corporation are developing the "integrated" version of Warp, called *i*Warp. In this machine, each cell of Warp will be implemented on a single chip. The clock rate will be increased so that each chip will support at least 16 MFLOPS computation, as opposed to 10 MFLOPS in WW and PC Warp. In the baseline machine the cells will be organized into a linear array of 72 cells, giving a total computation of 1.152 GFLOPS. In the following analysis, it has been assumed that each *i*Warp cell can do everything a PC Warp cell can, with an increase of 1.6 in speed (this is a design goal). When I/O bottlenecks have led to a maximum performance time on a benchmark, this has been noted.

All the benchmarks listed below as being implemented on Warp are written in W2, the Warp programming

language. W2 is a procedural language, on about the same level as C or Pascal. Arrays and scalars are suppported, as are `for` loops, and `if` statements. The programmers are aware that they are programming a parallel machine, since each program is duplicated to all cells and then executed locally (with local sequencing) on each cell.

3 Vision Programming On Warp

We have studied vision programming at various levels on Warp for some time now, and developed and documented several different models [8, 14]. In this section we briefly review the various models of Warp programming, for reference in later sections.

All the programs in this paper use the cells in a *homogeneous* programming model: that is, all cells execute the same program, although the program counters on the different cells can differ, and each has its own local data memory. This is a restriction imposed by the hardware of WW Warp. Programs on PC Warp need not follow this restriction.

3.1 Input Partitioning

In this model, which is used for local operations like convolutions, the image is divided into a number of portions by column, and each of the ten cells takes one-tenth of the image. Thus, in 512×512 image processing cell 0 takes columns 0-51 of the image, cell 1 takes columns 52-103, and so on (a border is added to the image to take care of images whose width is not a multiple of ten). The image is divided in this way because it makes it possible to process a row of the image at a time, and because the host need only send the image in raster-order, which is important because the host tends to be a bottleneck in many algorithms.

3.2 Output Partitioning

This model is used for algorithms in which the operation to be peformed is global, so that any output can depend on any input, but can still be computed independently. In this model, each cell sees the complete input image, and processes it to produce part of the output. Generally, the output data set produced by a cell is stored in the cell's local memory until the complete input image is processed. Hough transform is implemented in this way.

3.3 Pipelining

In this model, which is the classic type of "systolic" algorithm, the algorithm is divided into regular steps, and each cell performs one step. This method can be used when the algorithm is regular. (Because the cell code must be homogeneous, this method is of less use on the wire-wrap Warp machine than it usually is in systolic machines). When this method can be used, it is generally more efficient in terms of input and output overlap with computation and local memory use than either of the two models above.

4 Laplacian

Laplacian. Convolve the image with an 11×11 sampled "Laplacian" operator [10]. (Results within 5 pixels of the image border can be ignored.)

The Laplacian given [10] is symmetric, but not separable. (Separable filters can be computed more efficiently, in general, than non-separable filters). In this section we describe a series of optimizations we applied to the Laplacian filter in the Warp implementation, which led to an efficient implementation. These optimizations can be applied to any symmetric filter, and will lead to efficient implementations on many different computer architectures.

Since most filters use masks with an odd number of rows and columns, the rest of this discussion will deal with

this case. Let the size of the mask be represented by $N=2M+1$.

In order to see where the optimizations come from, we first notice that an unoptimized $N \times N$ convolution takes N^2 multiplies and N^2-1 additions per pixel. A separable convolution of the same size would take only $2N$ multiplications and $2(N-1)$ additions.

One way to compute the Laplacian is to compute it as a series of column convolutions. Each column takes N multiplications and $N-1$ additions, and then $N-1$ additions are required to add all of the partial sums. The total number of multiplications is $N \times N = N^2$, and the number of additions is $N \times (N-1) + (N-1) = N^2-1$.

Due to symmetry, we can add the pairs of corresponding pixels within a column before multiplying them by the weights, as shown in Figure 1. Each of the N columns contains M pixels that can be added in this way, and one pixel in the middle which is not part of a pair. We call this column of $M+1$ pixels a "folded" column. After the multiplication, the pixels in each folded column must be added, and then all the columns must be added as before. This saves multiplications, but not additions: the number of multiplications is $N(M+1) = (N^2+N)/2$, while the additions sum to $N \times M + N \times M + (N-1) = N^2-1$.

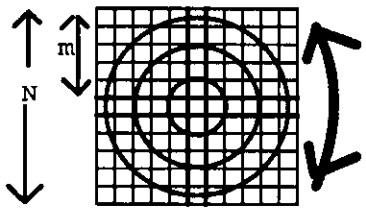

Figure 1: Folding columns

Now note that calculations for a given pixel can share partial results with neighboring calculations in the same row. As we shift the convolution window from the left to the right one step, we can retain all but one of the folded columns from the previous convolution, and sum just one new folded column, as shown in Figure 2. The rest of the algorithm is unchanged. Multiplications are unaffected, but additions are reduced almost by half, to $M + N \times M + (N-1) = (N^2 + 3N - 3)/2$.

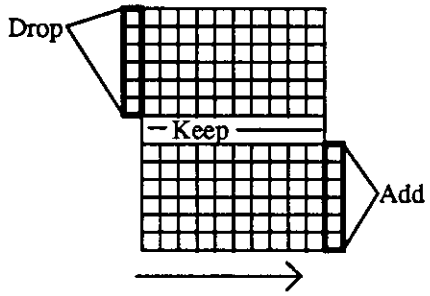

Figure 2: Using results from previous steps

Finally, we notice that the column convolutions are not done with N unique column weights, but rather with $M+1$ unique weights. As we shift the window to the right, we can compute and store the convolution of the new column with all $M+1$ column weights, as shown in Figure 3. Then, as we shift the window up to N pixels to the right, we

will only have to add the appropriate convolved column sums, as shown in Figure 4. Thus again, nearly half of the multiplications and additions can be saved. Thus for each pixel, only $M+1$ partial weighted column sums need be generated, and then $N-1$ additions are required to add the proper partial sums together. The number of multiplications is then $(M+1)\times(M+1)=((N+1)/2)^2$, while the additions come to $(M+1)\times M+M+(N-1)=M^2+4\times M$

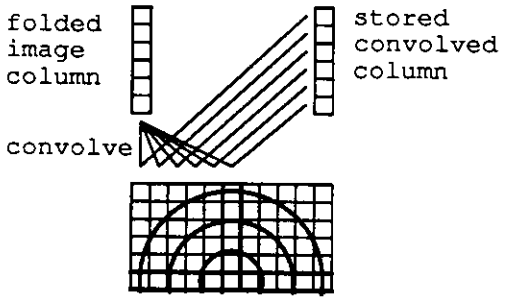

Figure 3: Convolving and storing column sums

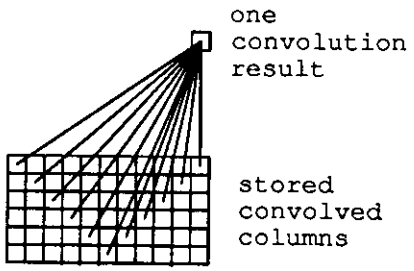

Figure 4: Adding appropriate column sums

Table 1 summarizes our result by comparing the number of multiplications and additions by our method with N^2, the number required for an unoptimized kernel, and $2N$, the number for a separable kernel:

Mask Size	Multiplications	Additions	N^2	$2N$
3×3	4	5	9	6
5×5	9	12	25	10
7×7	16	21	49	14
9×9	25	32	81	18
11×11	36	45	121	22
15×15	64	77	225	30
25×25	144	192	625	50

Table 1: Optimized Symmetric Convolution

An algorithm based on the above model was implemented using input partitioning on the WW Warp, and gave a runtime of 432 milliseconds. The same algorithm was compiled for the PC Warp, and gave a runtime of 350 milliseconds. The change was due to overlap of I/O with computation in PC Warp, which is not possible for this algorithm on the WW Warp. On *i* Warp, assuming a straightforward speedup arising from a 72-cell array with a 16

MHz clock, the time will be 30 milliseconds.

5 Zero Crossings Detection

Zero-crossings Detection. Detect **zero-crossings** of the output of the operation, i.e. pixels at which the output is positive but which have neighbors where the output is negative.

Zero crossing was implemented using the input partitioning model. A three by three window was taken around each pixel. If any elements of the window were negative, but the central pixel was positive, a zero crossing was declared and a "1" was output, otherwise "0" was output. This computation was performed by transforming each 9-element window into a 9-bit integer, with which a table lookup was performed. Input and output were represented as 8-bit pixels. Execution time on the WW Warp was 172 milliseconds; on the PC Warp the time will be approximately 92 milliseconds, due to overlap of I/O with computation. On *i* Warp, the time will be limited by I/O bandwidth to the array to at least 7.8 milliseconds.

In many cases, it is desirable to perform the Laplacian and zero crossing computations in sequence, without saving the results of the Laplacian. In this case, on *i* Warp, the computation can be done more quickly than by performing each individually. We estimate that such a computation will take 31 milliseconds, fast enough for video rate image processing.

6 Border following

Border Following. Output sequences of the coordinates of pixels that lie on the borders of regions where the Laplacian is positive. (On **border following** see [16, Section 11.2.2].)

The algorithm is mapped in two steps. First each Warp cell performs the border following technique on part of the image. Then, these partial results are then combined within the array to produce the complete border trace for the image. The full algorithm is:

- Each cell sends its bottom row to its successor.
- Starting with the bottom row, on the left, each cell inspects the pixels on this row. If the pixel is turned on, the cell begins to trace this connected component. As it traces the component, it builds a list of of its pixels to the next cell. As it visits pixels, it turns them off, so they will not be visited on scans of higher rows.
- Either this component extends to the cell's top row, or it does not. If not, then the list of pixels eventually terminates within the cell's strip; the cell queues the whole list of pixels for output to the next cell, marking the component as complete. But if the component extends to the top row, it may join with a component of the preceding cell. The cell checks its copy of the previous cell's last row to see if this is a possibility. If not, again the list may be passed to the next cell. But if it is, the cell stacks the list it has built so far, and begins processing another component, bottom to top.

This completes the parallel phase of the computation. Each cell now has two lists of borders: those ready for output, and those that must be merged with borders in preceding cells. The cells now run the following merge phase.

- Each cell tries to do two things: (1) empty its ready-for-output queue, and (2) move all the components on its stack to this queue. Operation (1) happens asynchronously, depending upon the next cell's input queue. Operation (2) is performed as follows.
- Eventually, the preceding cell will emit a list of the component touching the stacked component. When this happens, the component may be unstacked, the stacked pixels attached to the proper end of the list received from the previous component (note that this may involve attaching lists to both ends), and pass the now completed list at least, complete in its path through the given cell and its predecessor to the ready-for-output queue.

This algorithm must terminate, since the first cell never has any stacked components. Hence it will eventually

flush all the components on its output queue to the second cell, giving the second cell all the information it needs to move all its stacked components to its output queue. By iterating this argument, it follows that each cell must eventually clear its stack and then its output queue.

Finally, we must provide a time estimate for this algorithm. The first step is essentially a connected components computation. This will take no longer than the parallel step of a UNION-FIND based connected components program below. For PC Warp, with 10 cells, this is 73 milliseconds; for *i*Warp, with 72 cells, this is 6.3 milliseconds. These estimates were obtained by dividing the uniprocessor time of the Hughes HBA [19] implementation of a pure UNION-FIND algorithm by the number of cells, and again by suitable numbers to correct for processor speed.

The second step is a serial merge (in the worst case). We estimate this step will take about 1.02 second for PC Warp, and 690 milliseconds for *i*Warp. These estimates are based on our experience with similar merge steps for the connected components algorithm, and the i/o bandwidth of each machine. Hence our estimates are:

PC Warp: 1.1 seconds
*i*Warp: 690 milliseconds

7 Connected components labelling

Connected component labeling. Here the input is a 1-bit digital image of size 512×512 pixels. The output is a 512×512 array of nonnegative integers in which

1. pixels that were 0's in the input image have value 0.
2. pixels that were 1's in the input image have positive values; two such pixels have the same value if and only if they belong to the same connected component of 1's in the input image. (On connected component labeling see [16, Section 11.3.1].)

In this section we present our parallel-sequential-systolic algorithm for this computation, our timings of a C simulation of the algorithm, and our estimates of its execution time on Warp, PC Warp, and *i*Warp.

Section 7.1 gives the algorithm. Section 7.2 presents the asymptotic running time of the parallel–sequential–systolic algorithm. We also show how to modify this work to get a parallel–sequential–parallel algorithm, and give its running time. Section 7.3 discusses the implementation, covering both our existing C simulation and our planned Warp implementations; here we give the actual execution time of the simulations and the estimated execution times for the Warp implementation, and discuss the constraints imposed by the Warp architecture.

7.1 Sketch of the Algorithm

7.1.1 Vocabulary and Notation

The input to the algorithm is a $N \times N$ array (512×512 in this case) of binary pixels. A 1-valued pixel is called *significant*; all others are *insignificant*. We label the rows and columns consecutively from 0 to N-1, starting in the upper-left-hand corner. The *4-neighbors* of a pixel are the pixels that lie immediately above, below, left and right of it; its *8-neighbors* are the eight pixels that surround it. Two significant pixels x and y lie in the same *connected 4-component (connected 8-component)* of the image iff there is a sequence of significant pixels $p_0, \ldots, p_n$ with $p_0=x$, $p_n=y$, and p_{i-1} a 4-neighbor (8-neighbor) of p_i for each $i=1, \ldots, N$. The algorithm we present here computes connected 4-components. It is straightforward to modify it to compute connected 8-components; the timing estimates we present later are for the connected 8-component version.

Our algorithm executes on a linear systolic array of K processing cells, numbered consecutively from 0 to $K-1$. Each cell processes a set of adjacent rows of the image, called a *slice*. We assume that K divides N, and that the

slices are of uniform size N/K rows. The 0th cell processes the first N/K rows of the image, called slice 0, and so on. When data flows from cell i to cell $i+1$, we will say it crosses the $i, i+1$ *boundary*, or simply, an *inter-cell boundary*. A cell's *label space* is the set of all labels that it may assign to any pixel; cell i's label space is denoted L_i. We choose suitable bounds on the label spaces so that they are guaranteed disjoint.

7.1.2 The Algorithm

The algorithm proceeds in three phases: parallel, sequential, and systolic.

In the parallel phase, each cell computes labels for its slice of the image.

In the sequential phase, computation proceeds serially over each $i-1, i$ boundary, for $i=1, \ldots, K$. The ith stage of this computation effectively passes information about the connectivity of slices 0 through $i-1$ to slice i. The actual computation consists of scanning the $i-1, i$ boundary to construct two maps, which record connectivity information, then applying the second of these maps along the bottom row of slice i to propagate this information downward. Note that after this phase finishes, lower-numbered slices still lack information about higher-numbered slices. We perform this computation in K serial steps because of the limited interconnection topology of Warp.

In the systolic phase, the labels are pumped out of the cell array. As each label crosses into or out of a cell, the cell applies the maps generated in the sequential phase. Since the labels assigned to slice i must pass through cells $i+1, \ldots, K$, this permits higher-numbered cells to modify the labels assigned by lower-numbered cells, completing the computation. Each phase of the algorithm is explained in greater detail below.

Parallel Phase. In this phase, each cell computes preliminary labels for its slice of the image. These labels are drawn from the cell's label space, which are guaranteed not to be used by any other cell. We use a modification of the Schwartz-Sharir-Siegel algorithm [17], which runs in linear time in the size of the slice.

Sequential Phase. In this phase, processing proceeds sequentially in $K-1$ stages over each of the $K-1$ inter-cell boundaries. The function of stage i, when we compute along the $i-1, i$ boundary, is to pass infomation about the connectivity of slices 0 to $i-1$, inclusive, to cell i. This information is recorded in the two maps that are built for each boundary. Cell i builds the maps for the $i-1, i$ boundary. We call the first map σ_i; it is used by cell i to relabel pixels when they enter the cell. We call the second map ϕ_i; it is used by cell i to relabel the pixels when they leave the cell.

The maps have intuitive meanings, as follows. Each ϕ_i tells how to relabel the pixels of slice i to make them consistent with the connections in the $i-1$ preceding slices. Specifically, suppose x and y are two significant pixels of slice i such that there is a path from x to y that passes through slices 0 to i, but no path that lies entirely within slice i. Then after the parallel phase, x and y will bear distinct labels. However, ϕ_i is constructed such that $\phi_i(x) = \phi_i(y)$ iff there is a path from x to y that lies wholly within slices 0 through i. Thus ϕ_i encodes the influence of slices 0 through $i-1$ on slice i.

Similarly, σ_i contains information about connectivity across the $i-1, i$ boundary. Let w and v be significant pixels on the bottom row of slice $i-1$, and let x and y be significant pixels on the top row of slice i, such that w and x are adjacent, and v and y are adjacent. Suppose that x and y are connected by a path that lies wholly within slices 0 through i, but that w and v are not connected by any path that lies wholly within slices 0 through $i-1$. Then after the parallel phase, w and v will bear distinct labels. However, σ_i is constructed such that $\phi_i(x)=\phi_i(\sigma_i(w))=\phi_i(\sigma_i(v))=\phi_i(y)$. Thus σ_i encodes the influence of slice i on slices 0 through $i-1$.

These maps are constructed by the following procedure. We use some special notation. Let $f: M \rightarrow N$; then f is a subset of $M \times N$. We write $f+\langle m, n\rangle$ for the function obtained by deleting the pair $\langle m, f(m)\rangle$ from f and adding the pair

$\langle m,n\rangle$ to the resulting set. For the purposes of the UNION-FIND portion of the algorithm, we assume that each $l \in L_i$ lies in a singleton set $\{l\}$ that bears the name l. We also assume that each map is initialized to the identity map.

```
for i = 1 to K do begin
        get B, the bottom row of slice i-1
        get T, the top row of slice i
        for col = 0 to N-1 do
                if B[col] and T[col] are significant then
                        Call Update(B[col], T[col])
        for col = 0 to N-1 do
                if T[col] is significant then begin
                        φ_i = φ_i + <T[col], FIND(T[col])>
                end
        if i ≠ K then apply φ_i to the bottom boundary of slice i
end

procedure Update(PrevCell, CurrCell)
begin
        if σ_i(PrevCell) = PrevCell then σ_i = σ_i + <PrevCell, CurrCell>
        else UNION(CurrCell, σ_i(PrevCell))
end
```

Note that each of σ_i, ϕ_i may be computed locally by cell i, requiring only the bottom row of slice $i-1$. This is not done in practice, because we want to use path-compression for the UNION-FIND computations, and the cells cannot implement this algorithm efficiently. Because the UNION-FIND operations are performed on the data structures that embody the ϕ_i, we will refer to these operations, when we are accounting for the algorithm's running time, as ϕ lookups and additions, or simply ϕ updates.

The correctness proofs for these algorithms are tedious and are omitted here (a correctness proof for a similar algorithm can be found in Kung and Webb [13]). It remains to show how these maps are used to compute the connected components of the entire image. This is done in the next section.

Systolic Phase. In this phase, the pixel labels are pumped out of the cells. Each significant pixel receives its final label through the following systolic labelling procedure. First, as a label enters cell i, crossing the $i-1, i$ boundary, it is passed through the map σ_i. Note that labels belonging to slice i do not cross this boundary, so are not mapped this way within cell i. Second, as a label leaves cell i, crossing the $i, i+1$ boundary, it is passed through the map ϕ_i. This happens whether the label was received from cell $i-1$, or originated within cell i.

It is not difficult to give an inductive proof that this procedure correctly labels the connected components of the image. However, we believe it is more illuminating to work through an example.

Figure 5 depicts the binary input to the algorithm. Here N=9, K=3. Significant pixels are marked "X." Rows and columns are numbered consecutively from 0, starting in the upper-left-hand corner; we give the coordinates of a pixel as (row, column). Figure 6 shows the labels for the significant pixels after completion of the parallel phase.

Now we work through the computation of ϕ_1 and σ_1. To begin, each map is initialized to the identity map on its domain. When we reach column 1, we note that pixels (2,1) and (3,1) are adjacent and significant. This prompts a call to Update, where we note that σ_1 fixes 1. Hence we add the pair <1,11> to σ_1. Likewise, we add <3,12> to σ_1. But on the call to Update for the pair of labels <3,13>, we note that σ_1 is not the identity on 3. Instead we take the UNION of {13} and {12}; creating the set {12,13} that bears the label 12. At column 7, we add <2,14> to σ_1. Finally, we peform FINDs on the labels that appear on the top row of slice 1 to determine ϕ_1 (it is the identity except

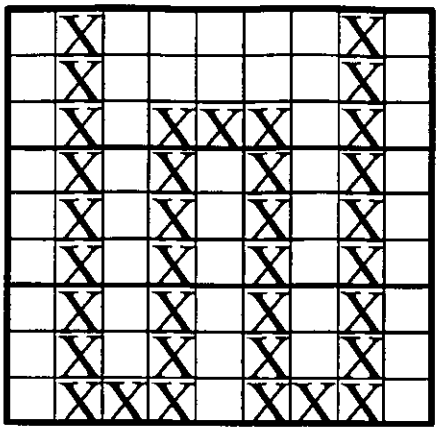

Figure 5: Input

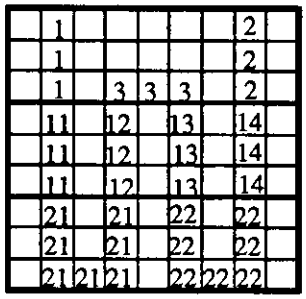

Figure 6: Labels after parallel phase

for the ordered pair <13, 12>), and apply this map to the bottom row of slice 1. The resulting labels appear in Figure 7.

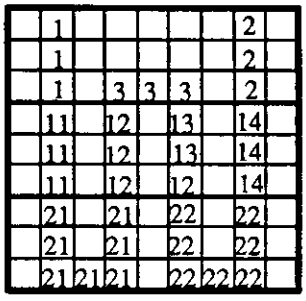

Figure 7: Labels after sequential phase

We perform a similar computation for the 2,3 boundary. The final maps appear in Table 2. (We do not display the ordered pairs associated with the identity portion of each map.)

Now we show how the systolic computation works. Consider pixel (2,4), with label 3. As it leaves cell 0, it

ϕ_1	σ_1	ϕ_2	σ_2
<13, 12>	<1, 11> <3, 12> <2, 14>	<22, 21>	<11, 21> <12, 21> <14, 22>

Table 2: Final maps

passes through ϕ_0, which is the identity map. When it enters cell 1, it passes through σ_1, where it is relabelled 12; when it leaves cell 1, it passes through ϕ_1, which fixes 12. When it enters cell 2, it passes through σ_2, where it is relabelled 21; when it leaves cell 2, it passes through ϕ_2, which fixes 21. Hence the final pixel label is 21. Table 3 summarizes this computation on each of the labels in the example; note that each significant pixel has the same final label.

Labels				Action
1	2	3		enter slice 1, map through σ_1 to
11	14	12	13	leave slice 1, map through ϕ_1 to
11	14	12	12	enter slice 2, map through σ_2 to
21	22	21	21	leave slice 2, map through ϕ_2 to
21	21	21	21	final values

Table 3: Label Computation

7.2 Asymptotic Running Time

This discussion is divided into two parts. In the first, we give the running time of the parallel–sequential–systolic algorithm. In the second, we show how to transform this approach into a parallel–sequential–parallel algorithm, and give its running time.

7.2.1 Parallel-Sequential-Systolic Algorithm

The asymptotic running time of this algorithm is $O(N^2/K + KNG(N) + N^2) = O(KNG(N) + N^2)$, where G is the inverse Ackermann's function. The terms of this expression represent the running times of the parallel phase, the sequential phase, and the systolic phase respectively.

These expressions are obtained as follows. The parallel phase estimate is immediate, since it runs in linear time in the slice size, which is N^2/K.

For the sequential phase, observe that we must perform $K-1$ computations along boundaries. As we process a boundary, we will perform no more than $N/2$ additions to a σ-map, and no more than $N/2-1$ updates to a ϕ-map. Now no ϕ_i or σ_i will map more than $N/2$ elements of their respective domains away from themselves. If the σ_i maps are maintained as linear arrays, each σ_i operation takes constant time. If the ϕ_i maps are maintained as linear arrays with path compression, a sequence of N lookups and additions takes $O(NG(N))$ time. Hence the total time for this phase is $O(KNG(N))$.

For the systolic phase, observe that the last cell must perform a constant-time lookup on each pixel in the image, and all other cells may perform their lookups in parallel. Hence the time for this phase is $O(N^2)$.

Thus a pure systolic implementation has no asymptotic advantage over a linear-time uniprocessor algorithm. This statement is deceptive. Ultimately, any machine must run in $O(N^2)$ time on this problem, since it takes that much

time to pump the data in and out. The advantage of the systolic algorithm is that it performs a useful constant-time computation step during the output, reducing the time spent in the sequential phase.

7.2.2 Parallel-Sequential-Parallel Algorithm

Here we introduce a straightforward modification of the algorithm to obtain an improved asymptotic time bound. It is based on the following simple observation: the systolic phase consists of computing, for each pixel p of slice i, the value of the composed function

$$\lambda_i = (\phi_K \circ \sigma_K \circ \cdots \circ \phi_{i+1} \circ \sigma_{i+1} \circ \phi_i)$$

Hence it suffices to compute each λ_i for $i = K, K-1, \ldots, 0$. But since

$$\lambda_{i-1} = \lambda_i \circ (\sigma_i \circ \phi_{i-1})$$

it is straightforward to compute the λ_i sequentially in $O(KN)$ time. (In fact, this can be done in parallel in $O(N\log(K))$ time.) Once this is done, cell i can obtain the *final* labels for slice i by applying λ_i to each significant pixel. Since this step can be performed in parallel among the cells, the running time of this modified algorithm is $O(N^2/K + KNG(N) + N^2/K) = O(N^2/K + KNG(N))$.

7.3 Implementation Details

In this section we discuss two implementations of this algorithm: a C-language Vax implementation and a Warp implementation. We begin with a sketch of the architectural constraints imposed by the various Warp machines; these constraints motivate some of the design decisions of both the Vax and Warp implementations. Then we discuss the Vax implementation, which was undertaken to learn about the algorithm in a familiar environment. We close with a treatment of the Warp implementation on each of Warp, PC Warp, and *i* Warp.

7.3.1 Warp Architectural Constraints

Two key factors determine most of the design decisions in a Warp implementation of this algorithm: the Warp cell's synchronization requirements and memory size. Let us consider these in turn.

Synchronization. The WW Warp requires compile-time synchronization. Thus an `if-then-else` statement will always take the time required by the slower of the two alternatives, and any loop must run for a fixed (maximal) number of iterations.

As a consequence, the WW Warp runs poorly on algorithms that exhibit good behavior only in the off-line sense. To see this, recall that techniques with good off-line performance – notably path-compression – derive their advantage by performing a few of the operations in a sequence slowly, so that the remaining operations in the sequence will be fast. But Warp forces each step of a procedure to take the time of the slowest possible alternative. Hence an implementation of an off-line algorithm will behave as if the most expensive operation were perfomed at each step of the sequence. Thus the algorithm with the best on-line behavior is always preferred.

This means that we must either abandon the path-compression approach to the sequential stage, or perform the sequential portion of the algorithm on a computation engine that does not have these constraints. Since there is no inherent advantage to performing the sequential phase on the cells (for there is no parallelism to exploit), and since the cost of shipping the necessary data to a suitable processor is low, we choose to do this phase on one of the Warp's MC68020-based cluster processors.

For similar reasons, we cannot improve the execution time by using sophisticated data structures to implement the σ-maps in the systolic phase. Unfortunately, this problem cannot be avoided. The best we can do here is use a data structure with good constant-time performance. This is discussed more fully below.

Memory Constraints. Our formulae for the asymptotic running times of these algorithms are based on the assumption of unit-access time to the data structures that hold the σ and ϕ maps. If memory is not a consideration, this speed can be attained by representing each map by a large array. The PC Warp and *i*Warp machines have enough cell memory to represent the maps this way. The speed estimates below for these machines are based on this assumption.

The WW Warp cell does not have enough memory to do this. Instead we must use an approach that gives good update and access times, with only moderate memory requirements. This is easy to do for the ϕ-maps. If each cell begins the assignment of the initial labels on the top boundary of its slice, the labels of this row will be drawn from the first $N/2$ elements of the slice's label space. Now note that though each ϕ_i is defined on the set $L_0 \cup \cdots \cup L_i$, which contains $(i+1)\lceil N/2 \rceil \lceil N/2K \rceil$ elements, ϕ_i will fix all of $L_0 \cup \cdots \cup L_{i-1}$, and also all of L_i, except possibly those elements of L_i that appear on the top row of slice i. Thus we can maintain ϕ_i as an array of size $N/2$, indexed by offset from the first element of L_i. To compute $\phi_i(r)$, we need only check to see if r lies in the range of interest, then find its offset and look up the value. This approach uses a small amount of memory, with only minor sacrifice of speed. Also, it is efficient for both the sequential phase of the computation, when the algorithm builds each ϕ_i using path-compression techniques, and the systolic phase, when the only operations are look-ups.

The situation is not as nice when we consider the σ maps. It is true that no σ_i will map more than $N/2$ elements away from themselves. This is because only labels that appear on the bottom row of slice $i-1$ may be moved by σ_i. However, these labels are no longer guaranteed to be drawn from some small subset of L_{i-1}. For instance, it is easy to construct an example so that labels drawn from both the first $N/2$ elements and the last $N/2$ elements of L_{i-1} will appear on the bottom row of slice i.

One solution is to maintain each σ_i as an array of ordered pairs, sorted by the first element of each pair. This permits lookup in worst-case $\log(N)$ time, and is well-suited to the systolic stage of the algorithm. In fact, it is the approach we use there. However, it does not permit fast addition to the map, and we must do both lookup and addition operations in the sequential stage. For this reason, we use the self-adjusting binary tree data structure to implement the σ_i in this stage. This data structure exhibits only good off-line performance. However, we use it only during the sequential phase of the calculation, when we build the map. The efficiency and simplicity of this data structure is another reason for doing the sequential calculation elsewhere than the Warp cells.

7.3.2 Vax Implementation

We have implemented the algorithm in C on a Vax 780, simulating the operation of a 10-cell Warp array. Each phase of the algorithm is implemented as one or more procedures, parameterized by cell number. A cell's local memory is represented by several large arrays; systolic communication is simulated by explicit data movement in and out of these arrays.

Note that the value of the σ maps themselves are never needed directly. We are interested only in the ϕ maps, and in the composition maps $\phi \circ \sigma$. For this reason we compute these compositions explicitly ahead of time. This way, each label that traverses a cell is mapped only once, through $\phi \circ \sigma$, rather than through σ and ϕ successively.

The simulation program processes a typical 512×512 image in about 4 1/2 minutes of CPU time. Fortunately, most of this represents the simulation of inter-cell communication.

To learn how the program was spending its time, we used the Unix *prof* [12] performance-monitoring program. The results are summarized in Table 4. The total is less than 4 1/2 minutes because the time for simulating communication is not included.

Phase	Time (seconds)	Time (seconds)
Parallel Phase	33	33
Sequential Phase		
Boundary Scan	.16	
ϕ Update	.10	
σ Update	.53	
$\phi \circ \sigma$ Computation	.06	
Total	.85	.85
Systolic Phase		
ϕ Lookups	3.8	
$\phi \circ \sigma$ Lookups	25	
Total	28	28
Total		62

Table 4: Vax implementation timings

7.3.3 Warp Implementations

We have not yet completed a Warp implementation. In this section we discuss the partial implementation for the WW Warp architecture, and give execution time estimates for the planned PC Warp and *i* Warp implementations. All our estimates are for the parallel–sequential–parallel version of the algorithm, computing connected 8-components.

WW Warp Our implementation for the WW Warp divides the computational burden between the linear systolic array and the cluster processors. The initial and final labellings are done by the systolic array; the sequential step is done by the cluster processors. This permits us to use algorithms with fast amortized time in the sequential step.

After the initial labelling, we would like to retain the initial results in cell memory, transmitting only each cell's boundary rows to the external host for generating the necessary maps. Unfortunately, the WW Warp cell memory is not large enough to hold a labelled slice, and barely large enough to hold the intermediate result required by the initial marking algorithm. This forces us to send the entire contents of each cell's slice to the external host as the labels are generated, then pump these slices back through the array for the final labelling.

We have written, but not yet debugged, all the code for the cell array. We have accurate estimates of the running time of this code, provided by the compiler. We have also estimated the running time of the sequential phase. We derived this estimate from the sequential phase running time of the C implementation, allowing for a slight speed-up of the cluster processors over the Vax, and also for the extra work (computing the λ_i) done in this phase by the parallel–sequential–parallel version of the algorithm. The resulting estimate appears in Table 5.

PC Warp and *i* Warp Architectures. In this section we derive estimated execution times for these architectures. There are three key differences between the design of these cells and those of the WW Warp. The first is that each cell has enough memory to maintain a full slice of labels. This means that we do not need to pump the intermediate labels to an external memory. The second is that the cells are not bound by the synchronization constraints of the WW machine. This means that the sequential phase computation can be perfomed on the cell array. This saves time because we no longer have to do i/o to the cluster processors for this phase, and because the cells run 2.8 times faster than the cluster processors. The third is that each of these machines is more powerful than the WW Warp. Both the

Phase	Time (milliseconds)	Time (milliseconds)
Initial Parallel Phase		
Pump in Image	50	
Initial Labelling	2400	
Total	2450	2400
Sequential Phase		
Boundary Scan	150	
ϕ Update	90	
σ Update	490	
λ Computation	110	
Total	840	840
Final Parallel Phase		
λ Lookups	2200	
Pump Out Labels	50	
Total	2290	2300
Total		5600

Table 5: Estimated WW Warp timings

PC Warp and the *i* Warp can do arithmetic directly on integers; this speeds up any integer arithmetic computation by a factor of 3. Furthermore, the *i*Warp cells run 1.6 times faster than the Warp and PC Warp cells.

The only other salient difference between PC Warp and *i* Warp, for our purposes, is that the *i* Warp contains 72 cells. Thus we can potentially attain more parallelism on *i* Warp. However, because the time taken in the merge phase varies linearly with the number of cells, while the time taken in each parallel phase varies inversely with this number, it is not necessarily best to use the greatest possible number of processors. If the execution time of the algorithm as a function of the number of cells is $T(K)=A/K+BK$, then the best time will be obtained with $K=\sqrt{A/B}$. In the case of *i* Warp, we have $A\approx4.994, B\approx.00812$, so the best K is 25. The estimate below for *i* Warp execution time was made using this value.

The resulting estimates appear in Tables 6 and 7.

Phase	Time (milliseconds)	Time (milliseconds)
Initial Parallel Phase		
Pump In Image	53	
Initial Labelling	710	
Total	760	760
Sequential Phase		
Boundary Scan	53	
ϕ Update	33	
σ Update	3	
λ Computation	41	
Total	130	130
Final Parallel Phase		
λ Lookups	36	
Pump Out Labels	53	
Total	89	89
Total		980

Table 6: Estimated PC Warp timings

Phase	Time (milliseconds)	Time (milliseconds)
Initial Parallel Phase		
Pump In Image	33	
Initial Labelling	191	
Total	224	224
Sequential Phase		
Boundary Scan	83	
ϕ Update	4.7	
σ Update	52	
λ Computation	63	
Total	200	200
Final Parallel Phase		
λ Lookups	9.0	
Pump Out Labels	33	
Total	42	42
Total		470

Table 7: Estimated *i*Warp timings

8 Hough transform

Hough transform. The input is a 1-bit digital image of size 512×512. Assume that the origin (0,0) image is at the lower left-hand corner of the image, with the x-axis along the bottom row. The output is a 180×512 array of nonnegative integers constructed as follows: For each pixel (x,y) having value 1 in the input image, and each i, $0<i<180$, add 1 to the output image in position (i,j), where j is the perpendicular distance (rounded to the nearest integer) from (0,0) to the line through (x,y) making angle i-degrees with the x-axis (measured counterclockwise). (This output is a type of Hough transform; if the input image has many collinear 1's, they will give rise to a high-valued peak in the output image. On Hough transforms see [16, Section 10.3.3].)

The Hough transform algorithm has been previously described [13]. Briefly, each of the ten cells gets one-tenth of the Hough array, partitioned by angle. The input image flows through the Warp array, and each cell increments its portion of the Hough array for all image pixels which are "1". Once the image has been processed, the Hough array is concatenated and output to Warp's external host.

For the particular parameters of this benchmark, which uses an array of 180×512 data, this requires each cell store $18\times512=9$ K words of data. This will not fit on the WW machine, which has a memory of 4K words/cell. But on PC Warp, each cell will have a memory of 32K words, so that the Hough array fits easily. On *i*.Warp 60 cells are used (60 being the largest number less than 72 which evenly divides 180), so that each cell needs to store only $3\times512=1536$ bytes of data.

In order to derive estimates, we implemented a Hough transform program (with a smaller number of angles than in the benchmark) and ran it on the WW machine. The algorithm does not change for more angles, so the estimates given by this method are accurate for the PC Warp with the benchmark parameters.

By derivation from this program, the time per pixel with value "1" is 13 microseconds. Assuming 10% of the image is one, on PC Warp the benchmark will execute in 340 milliseconds. On *i* Warp, the estimated execution time is 60 milliseconds. These times scale linearly with the number of "1"'s in the image.

9 Convex Hull.

Convex Hull. The input is a set S of 1000 real coordinate pairs, defining a set of 1000 points in the plane, selected at random, with each coordinate in the range [0,1000]. The ouput is an ordered list of the pairs that lie on the boundary of the convex hull of S, in sequence around the boundary. (On convex hulls see [15, Chapters 3-4].)

R. A. Jarvis's [11] algorithm was used. This algorithm works as follows:

- Sort the points according to (x,y)-coordinate. The first point is a convex hull point. Call it A_0.
- Let i=0. Repeat the following until $A_{i+1}=A_0$:
  - For each point B in the set, do the following:
    - Calculate the angle from the vector A_i-A_{i-1} to the vector $B-A_i$. (If i=0 we take the second vector to be (−1,0)).
    - The point with smallest angle is a convex hull point. Call it A_{i+1}.

This algorithm obviously has time complexity O(KN), where K is the number of convex hull points, and N is the number of points in the set. The time consuming step in the algorithm is the scan through the set of points to find the next convex hull point.

We implemented the above algorithm on the WW Warp, using C code to program the cluster processors and W2 to program the Warp array. In our implementation, the Warp array performs the inner loop in the algorithm, which finds a new convex hull point by calculation of the angle with all points. This is done in parallel on all cells, by partitioning the set of data points across the array and finding the best point in each cell's dataset individually, then finding the best point of the cell's points. The cluster processors repeatedly accept the new point from the Warp array and pass in this new convex hull point for the next step of the computation.

To test this algorithm, we generated a 1000 node random graph, which had 13 hull points. The measured time on the WW machine was 6.76 milliseconds, with the same execution time on PC Warp. The time for this algorithm scales linearly with the number of hull points.

Assuming a 16 MHz clock time and 72 cells in *i*Warp, each point location will take 26 microseconds, based on an operation count from the Warp implementation. Loading the initial array to the cells will take 250 microseconds, for a total time of 590 microseconds for our sample problem.

10 Voronoi Diagram

Geometrical constructions. The input is a set S of 1000 real coordinate pairs, defining a set of 1000 points in the plane, selected at random, with each coordinate in the range [0,1000]. The output is the Voronoi diagram of S, defined by the set of coordinates of its vertices, the set of pairs of vertices that are joined by edges, and the set of rays emanating from vertices and not terminating at another vertex. (On Voronoi diagrams see [15, Section 5.5].)

We consider the computation of the Voronoi diagram of a set of 1000 real points [9]. The algorithm is:

1. The coordinates of the points are sorted divided equally among the cells so that each cell has 100 points. The sorting is done systolically on the Warp array, using a heapsort algorithm in which each cell builds a heap of 100 points as the data values stream in, passing the rest of the data on to the next cell.
2. Each cell computes the Delauney triangulation of 100 points using a standard sequential algorithm.
3. Cells 1, 3, 7, and 9 receive the Delauney triangulation of their left neighbors. The two Delauney triangulations are then merged to form a single Delauney triangulation in these receiving cells. At the end of this stage we have four Delauney triangulations of 200 points each and two Delauney triangulations of 100 points each in cells 4 and 5. Six cells will be idle during this step.
4. The 200 point triangulations are merged to form 400 point triangulations. At the end of this step we have two triangulations of 400 points each and two triangulations of 100 points each. Eight cells are idle during this step. The mergings are carried out in the in the third and eigth cells.

5. The 400 point and 100 point triangulations are merged to form 500 point triangulations in cells 4 and 6. At the end of this step there are two triangulations of 500 points each. Eight cells are idle during this step.
6. The two 500 point triangulations are merged to give the Delauney triangulation of 1000 points. This operation is carried out in the fifth cell. Nine cells are idle.
7. The dual of the Delauney triangulation thus obtained will give the Voronoi diagram.

Table 8 gives operation counts for each of the steps in the Voronoi diagram algorithm above. These counts were obtained through a C program which computed the Voronoi diagram.

Step	Assignments	Array References	Comparisons	Arithmetic operations	Logical Operations
2	86897	192695	60149	71572	36290
3	89529	198309	60209	74221	36343
4	91754	202898	60264	76401	36388
5	94504	208420	60326	79030	36441
6	97313	214221	60394	81733	36502

Table 8: Operation counts for Voronoi diagram

*i*Warp will have 72 cells instead of 10. Since the time for intermediate data transfers is small we ignore any changes in that and assume linear speedup in the Delauney triangulation computation.

Since the computation of addresses for the array references appears to be the critical path we considered this as the bottleneck in the computation. (PC Warp and *i*Warp will have parallel address computation engines in each cell). Each array reference takes 300ns on PC Warp (100ns for the address computation and 200ns for the memory access) and 100ns on the baseline *i*Warp. The total computation time therefore comes to 64 milliseconds on PC Warp and 8.9 milliseconds on *i*Warp. The initial sort step requires 24 milliseconds on PC Warp and 10 milliseconds on *i*Warp. The number of floating-point data transfers internal to the computation is 3600 (400 in step 3, 800 in step 4, 400 in step 5, and 2000 in step 6). This will take 800 microseconds on PC Warp and 63 microseconds on *i*Warp.

Since the Voronoi diagram computation is taking the dual of the Delauney triangulation, this can be done in parallel. This can be done in pipelined mode (concurrent I/O and computation in a cell) so that the total time of computation will be around the total time for I/O which is around 200 milliseconds on PC Warp, and 120 milliseconds on *i*Warp. The conversion to Voronoi diagram will be part of a pipeline at the end of which Voronoi diagram edges will be transmitted to the host. Hence time for transmission to the host will be included in this.

The total times for the computation are, on PC Warp, 64 milliseconds + 24 milliseconds + 800 microseconds + 200 milliseconds = 290 milliseconds, while on *i*Warp the time is 8.9 milliseconds + 10 milliseconds + 63 microseconds + 120 milliseconds = 140 milliseconds.

11 Minimum spanning tree

Geometrical constructions. The input is a set S of 1000 real coordinate pairs, defining a set of 1000 points in the plane, selected at random, with each coordinate in the range [0,1000]. The output is the minimal spanning tree of S, defined by the set of pairs of points of S that are joined by edges of the tree. (On minimal spanning trees see [15, Section 6.1].)

We use Shamos's algorithm [15], in which we have only to examine edges in the Delauney triangulation to find an incremental edge in the minimum spanning tree. In the worst case 1000 vertices can correspond to 3000 edges,

implying an average of 3 edges per vertex. This means that we have to make a maximum of 2 comparisons to find the edge of minimum length out of a vertex. Since there are 1000 vertices we have to make only 2000 comparisons per stage of the algorithm and, since there are $\log(N)$ stages, we have to make 20000 comparisons in all. Also, as part of the initialization step we have to compute the lengths of all the 3000 edges, which will involve 6000 floating-point multiplications and 3000 floating-point additions. We also have to prepare a data structure which will give the out-degree of a paricular vertex. This will involve 2 comparisons per edge, for a maximum of 6000 comparisons in all. We assume that the minimum spanning tree shall be computed. We also assume that a floating-point multiplication takes 5 microseconds and a floating-point addition takes 2.5 microseconds, and each comparison takes 1 microsecond. Adding up the respective times the total comes to about 65 milliseconds. This time is the worst case since the Delauney triangulation of 1000 points will typically contain much less than 3000 edges.

12 Visibility

Visibility. The input is a set of 1000 triples of triples of real coordinates $((r,s,t),(u,v,w),(x,y,z))$, defining 1000 opaque triangles in three-dimensional space, selected at random with each coordinate in the range [0,1000]. The output is a list of vertices of the triangles that are visible from (0,0,0).

An input partitioning method is used. Each vertex is simply tested to see if it is obscured by any of the triangles. This is done by taking the four planes defined by the triangle vertices and the origin and any two of them, and testing to see if the vertex point lies in the interior of the region defined by the three planes including the origin, but on the far side of the triangle. The mapping onto Warp is to broadcast the set of triangle points to all cells, and then to send to each of the ten cells one-tenth of the vertex set, with each cell testing its portion to see if it is visible. The execution time on the WW Warp is 825 milliseconds (however, the WW Warp machine cannot hold the entire dataset due to memory limitations – this time is a compiler estimated execution time). Some improvement (probably a factor of two to three) is expected on PC Warp, since the algorithm will be able to stop testing a vertex when it is found that a vertex is definitely not obscured by a particular triangle. On *i* Warp, we estimate a speedup of about 10, giving an execution time of 40 milliseconds.

13 Graph Matching

Graph matching. The input is a graph G having 100 vertices, each joined by an edge to 10 other vertices selected at random, and another graph H having 30 vertices, each joined by an edge to 3 other vertices selected at random. The output is a list of the occurrences of (an isomorphic image of) H as a subgraph of G. As a variation on this task, suppose the vertices (and edges) of G and H have real-valued labels in some bounded range; then the output is that occurrence (if any) of H as a subgraph of G for which the sum of the absolute differences between corresponding pairs of labels is a minimum.

This problem includes two subproblems. The first is to find isomorphic embeddings of one the smaller graph in the larger one. Finding one such embedding (or determining the existence of one) is known to be NP-complete [7]. Finding all isomorphisms actually grows exponentially. For example, in one set of randomly generated data, we found about 10^{16} solutions. Because there are too many solutions, no presently existing machine can produce all the solutions in one year.

The second problem is to find the one isomorphism to the graph with the least differences between the corresponding edge and vertex costs. The complexity of the second problem is obviously between finding one and finding all. This problem has not been completed because there were too many solutions to the first problem.

Our parallel algorithm is based on Ullmann's refinement procedure [18] which can prune the search tree by eliminating mappings that are infeasible because of connectivity requirements. The method eliminates mappings as early as possible.

In addition, we developed a more powerful method to cut the search tree as early as possible. The new method uses graph analysis and makes use of some special features of the graph.

We implemented the problem on the Warp host, which is a Sun workstation. Running on a set of randomly generated data for over one hour, we obtained 1188174 solutions, giving 267 solutions/second or about 3.75 milliseconds/solution. At this point, by counting the branching factors of the tree above the portion we had processed, we estimated we had found only about $1.2 \times 10^{-9}\%$ of the solutions, leading to our estimate of 10^{16} solutions for this example.

In the Warp implementation, we parallelize the exploration of the search tree. This is easy to do because the search tree is so large that we can easily assign each subtree to a processor. By straightforward extrapolation of cycle time, we estimate the solution rate in PC Warp to be 2700 solutions/second. Similarly, we estimate the solution rate in *i* Warp to be 19000 solutions/second.

14 Minimum-cost Path

Minimum-cost path. The input is a graph G having 1000 vertices, each joined by an edge to 100 other vertices selected at random, and where each edge has a nonnegative realvalued weight in some bounded range. Given two vertices P, Q of G, the problem is to find a path from P to Q along which the sum of the weights is minimum. (Dynamic programming may be used, if desired.)

The algorithm used here is the best known sequential algorithm, Dijkstra's Single Source Single Destination [6] (SSSD). The algorithm works by repeatedly "expanding" nodes (adding all their neighbords to a list) then finding the next node to expand by choosing the closest unexpanded node to the destination.

The lack of a `while` loop on the WW Warp results in a significant loss of performance, compared to PC Warp and *i* Warp. PC Warp and *i* Warp have very similar mappings:

- **WW Warp.** The WW Warp cannot execute a loop a data dependent number of times, so that the outer loop of SSSD must be mapped into the cluster processors. In this case, the Warp array is used for expanding nodes, and for calculating which node should be expanded next. Node expansion is done by feeding from the cluster processor the descendants of the node to be expanded, and by calculating the distance to the goal of each of these nodes. The computation is extremely simple, and I/O bound on the Warp array. Each node expansion involves the transfer of 200 words of data, which takes 200×1.2 microseconds $=240$ microseconds, since the transfer of a single word takes 1.2 microseconds.

  To find the next node to be expanded, the entire set of nodes must be scanned, and the node nearest the goal is selected. On the WW Warp this means 1000 nodes must be scanned. Again, the computation is I/O bound, so that the execution time is 1000×1.2 microseconds $=1.2$ milliseconds. In the worst case, 1000 nodes must be expanded, for a total time of $1000 \times (1.2$ milliseconds $+240$ microseconds $= 1.44$ s). This number scales linearly with the number of nodes that must be expanded to find the goal.

- **PC Warp.** In PC Warp it is possible to map the outer loop of SSSD into the Warp array, giving a much better time.

  Node expansion is done by prestoring at each cell the costs, giving each cell 100 data. Node expanding is done in parallel in all cells. In the worst case, the slowest cell will have to expand 100 nodes, so that the time for one node expansion is 100×0.25 microseconds $=25$ microseconds.

  The global minimum is calculated in parallel in all cells, and then the minimum among cells is found in one pass through the array. Finding the minimum on each cell takes 0.4 microseconds $\times 100=40$ microseconds. Finding the minimum among cells takes 0.4 microseconds $\times 10=4$ microseconds.

  The total time for one node expansion is therefore 69 microseconds. In the worst case, when 1000 nodes are expanded, the time is 69 milliseconds. This time scales linearly with the number of nodes that must be expanded to find the goal.

- ***i* Warp.** Following the same algorithm partitioning method as for PC Warp, we use 72 cells instead of

10. Now each cell need store only 14 data. The faster cycle time of *i* Warp gives a 10 microsecond time for one node expansion, 7 microseconds to find the global minimum in each cell, and 8 microseconds to find the global minimum across cells. (The minimum across cells is done sequentially from cell to cell, so it takes longer on longer arrays). The total time for one node expansion on *i* Warp is 25 microseconds. In the worst case, the total time for the solution will be 25 milliseconds. This number scales linearly with the number of nodes that must be expanded to find the goal.

15 Warp Benchmarks Summary

In Table 9 we summarize Warp's performance on the IU Architecture benchmarks. With each time, we give its source – from an actual run of WW Warp, from compiled code, or by an estimate (all *i* Warp times are estimated). The times from an actual run are, of course, the most reliable – they are observed times, from an actual run on our WW Warp at Carnegie Mellon, and include I/O. Times marked "compiled code" are just as reliable; the W2 compiler for Warp produces a time estimate, which gives the actual execution time for the algorithm on Warp (we have modified these times as appropriate when the Warp array is not the bottleneck in the execution time of the algorithm). Finally, "estimate" indicates a time which is not based on compiled code, but on some other method, which may not be as reliable. The source of the time is given in the relevant section. We have tried to be as accurate as possible in these estimates, and have tried to err on the side of caution.

Algorithm	WW Warp	PC Warp	*i* Warp
Laplacian	430 ms actual run	350 ms compiled code	7.8 ms
Zero crossing	170 ms actual run	50 ms estimate	7.8 ms
Border following	N/A	1.1 s estimate	690 ms
Connected Components	5.6 s compiled code	980 ms estimate	470 ms
Hough transform	N/A	340 ms compiled code	60 ms
Convex Hull	9 ms actual run	9 ms compiled code	3.2 ms
Voronoi diagram	N/A	290 ms estimate	140 ms
Minimal spanning tree	N/A	160 ms estimate	43 ms
Visibility	830 ms compiled code	400 ms estimate	40 ms
Graph matching	N/A	1800 soln./s estimate	19,000 soln./s
Minimum-cost path	1.4 s estimate	69 ms estimate	25 ms

Table 9: Warp Benchmark Summary

16 Evaluation of the Warp Architecture

In this section we will use the data generated by these benchmarks to evaluate the Warp architecture, by considering the effect of various reasonable design changes. The intent is to explore the design space around the PC Warp. We will consider all of the benchmark algorithms except for minimal spanning tree, which is not performed on the Warp array.

16.1 Memory

In PC Warp, each cell has 32K words of memory, for a total memory in the Warp array of 320K. What is the effect on performance of decreasing the memory size?

Laplacian and zero crossing are input partitioned algorithms. This implies that each cell needs only enough memory to compute the result for the area of the image assigned to that cell – in this case, approximately $11\times52+5\times52=832$ words for the Laplacian, and $3\times52+512=668$ data for zero crossing. If we decrease the memory per cell below this point, the computation can still be done, but only by processing a strip of the image at a time. For example, the Laplacian could process two 512×256 images and need only $11\times26+5\times26=416$ words of memory. (The computation would actually process a slightly wider image, because of the need for overlap at the interior edge. This makes it less efficient.)

Border following and connected components both must store the entire image (distributed through the array) at once to do their processing. This means the total array storage must be at least 256K, plus whatever is needed to store their local tables. If less memory is available than this, the computation becomes exceedingly complex – either the image must be compressed for storage, or several passes must be performed, with a new merge step. This sort of complexity is frustrating for a programmer to deal with.

Hough transform and visibility display the standard behavior of output partitioned algorithms; as memory is reduced, the computation grows proportionately less efficient. For example, for Hough transform the current benchmark requires $180\times512=90K$ words of memory in the array. If only, say, 45K words of memory are available, the computation can be done in two passes, each building half the Hough array; but each pass takes as long as the whole thing on a machine with sufficient memory. Similarly, visibility needs 27K; if less is available than this, multiple passes must be made, each pass deleting some of the points from the visibility set.

The other algorithms (Voronoi diagram, minimal spanning tree, graph matching, and minimum-cost path) all share the characteristics that they require the entire dataset to be stored in the array at once, their computation is fairly complex, and they have small datasets. In a well-designed machine, memory is unlikely to be a problem; but if it is too small to store the complete dataset, programming any of these problems will become very difficult.

16.2 Number of processing elements

PC Warp has ten cells in its array, a fairly small number as parallel machines go. What happens if we increase this number?

The effect on Laplacian, zero crossing, Hough transform, convex hull, and visibility is straightforward; their speed changes approximately linearly, increasing or decreasing as the number of cells is increased or reduced, as long as I/O is not a bottleneck. This bottleneck occurs when the data transfer rate between the external host and the Warp array reaches 12 MB/second, which occurs when the number of Warp cells is 168 for Laplacian, 24 for zero crossing, 180 for Hough transform (since the partitioning is by angle, this is the bottleneck), 130 for convex hull, and 530 for visibility. (Actually, due to the effects of rounding, some of these numbers do not actually represent peaks in performance. For example, we will not observe any change in performance between 128 cells, or four

pixels per cell, and 171 cells, or three pixels per cell.) By this point, effects we ignored in our initial time estimate, such as the cost of overlapping data with an adjacent cell, or the buffer sizes in the interface unit, probably dominate. Except possibly for zero crossing, these limits on the number of cells exceed the practical limits of building and maintaining such a PC Warp array.

Graph matching is similarly partitioned, and it should display the same sort of behavior as the above algorithms. We have not done enough analysis to determine the optimal number of cells.

The case of connected components is quite different. This algorithm consists of two parts, one of which is partitioned like the algorithms above, and the other of which is a merge step. The total time for both steps is $O(A/N+BN)$, where A and B are constants depending on the algorithm for the partitioned and merge steps, respectively, and N is the number of cells. This formula has a minimum when $N=\sqrt{A/B}$. For connected components, this occurs when N=25, as shown in Section .

Similarly remarks apply to Voronoi diagram and border following. We do not have accurate enough estimates to give a definite maximal number of cells in these cases.

16.3 External host

The external host is based on standard MC68020 processors and the VME bus. This is convenient for programming, but may be undesirable for performance. What is the effect of making the external host more powerful?

Naturally, as the external host grows more and more powerful, more and more of the computation can be mapped onto it – in the most extreme case, it can perform the entire computation. We will restrict ourselves to considering the qualitative effects of making the external host more powerful, but still less powerful than the Warp array.

There is no benchmark in which the external host actually creates an I/O bottleneck. However, there are many ways in which a more powerful external host would significantly affect the program mapping. This is most evident in Section . Here, on the WW machine, the external host is used to control the outer loop of the program, while on PC Warp and *i*Warp, the Warp array itself controls this outer loop. In many ways, it is convenient to use the external host for this computation; there is no reason not to split the computation in this way, and it is in some ways easier to program. However, the poor computational abilities of the external host make it advantageous to map as much computation onto the Warp array as possible, even when it is somewhat inconvenient.

Similar remarks apply to border following, connected components, convex hull, and Voronoi diagram. All of these algorithms could use a more powerful external host in the merge phase of their computation.

However, it is interesting to consider alternatives to a more complex external host. It is unlikely that the ratio of power between the external host and the Warp array will shift towards the external host in future versions of Warp or similar systems. Rather, as our ability to build larger Warp arrays grows, it will likely shift in the other direction. We must try to find alternatives to mapping important parts of the computation onto a sequential processor if we are to see further speedup in these algorithms. It seems that a much better alternative to making the external host more powerful is to make the Warp array more flexible, for example by making the communications between the cells more powerful (allowing higher dimensional arrays or logically connecting distant cells).

References

[1] Annaratone, M., Arnould, E., Cohn, R., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., Sarocky, K., Senko, J., and Webb, J.
Warp Architecture: From Prototype to Production.
In *Proceedings of the 1987 National Computer Conference*. AFIPS, 1987.

[2] Annaratone, M., Arnould, E., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., Sarocky, K. and Webb, J.A.
Warp Architecture and Implementation.
In *Conference Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 346-356. June, 1986.

[3] Annaratone, M, Arnould, E., Cohn, R., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., Sarocky, K., Senko, J., and Webb, J.
Architecture of Warp.
In *COMPCON Spring '87*, pages 264-267. IEEE Computer Society, 1987.

[4] Bruegge, B., Chang, C., Cohn, R., Gross, T., Lam, M., Lieu, P., Noaman, A. and Yam, D.
The Warp Programming Environment.
In *Proceedings of the 1987 National Computer Conference*. AFIPS, 1987.

[5] Bruegge, B., Chang, C., Cohn, R., Gross, T., Lam, M., Lieu, P., Noaman, A. and Yam, D.
Programming Warp.
In *COMPCON Spring '87*, pages 268-271. IEEE Computer Society, 1987.

[6] Dijkstra, E.
A note on two problems in connexion with graphs.
Numerische Mathematik 1:269-271, 1959.

[7] Garey, M. R., and D. S. Johnson.
Computers and Intractibility: A guide to the theory of NP-completeness.
W. H. Freeman, 1979.

[8] Gross, T., Kung, H.T., Lam, M. and Webb, J.
Warp as a Machine for Low-level Vision.
In *Proceedings of 1985 IEEE International Conference on Robotics and Automation*, pages 790-800. March, 1985.

[9] Guibas, L. J., and J. Stolfi.
Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams.
ACM Transactions on Graphics 4, 1985.

[10] Haralick, R. M.
Digital Step Edges from Zero Crossings of Second Directional Derivatives.
IEEE Transactions on Pattern Analysis and Machine Intelligence 6:58-68, 1984.

[11] Jarvis, R. A.
On the identification of the convex hull of a finite set of points in the plane.
Information Processing Letters 2:18-21, 1973.

[12] Joy, W. N., Babaoglu, O., Fabry, R. S., Sklower, K.
UNIX Programmer's Manual
4th Berkeley Distribution edition, University of California at Berkeley, 1980.

[13] Kung, H.T. and Webb, J.A.
Global Operations on the CMU Warp Machine.
In *Proceedings of 1985 AIAA Computers in Aerospace V Conference*, pages 209-218. American Institute of Aeronautics and Astronautics, October, 1985.

[14] Kung, H. T. and Webb, J. A.
Mapping Image Processing Operations onto a Linear Systolic Machine.
Distributed Computing 1(4):246-257, 1986.

[15] Preparata, F. P. and Shamos, M. I.
Computational Geometry - An Introduction.
Springer, New York, 1985.

[16] A. Rosenfeld and A. C. Kak.
Digital Picture Processing.
Academic Press, New York, 1982.

[17] Schwartz, J., Sharir, M., and Siegel, A.
An efficient algorithm for finding connected components in a binary image.
Technical Report 154, New York University Department of Computer Science, February, 1985.

[18] Ullman, J. R.
An algorithm for subgraph isomorphism.
Journal of the ACM 23(1):31-42, January, 1976.

[19] Wallace, R. S. and M. D. Howard.
HBA Vision Architecture: Built and Benchmarked.
In *Computer Architecures for Pattern Analysis and Machine Intelligence.* IEEE Computer Society, Seattle, Washington, December, 1987.