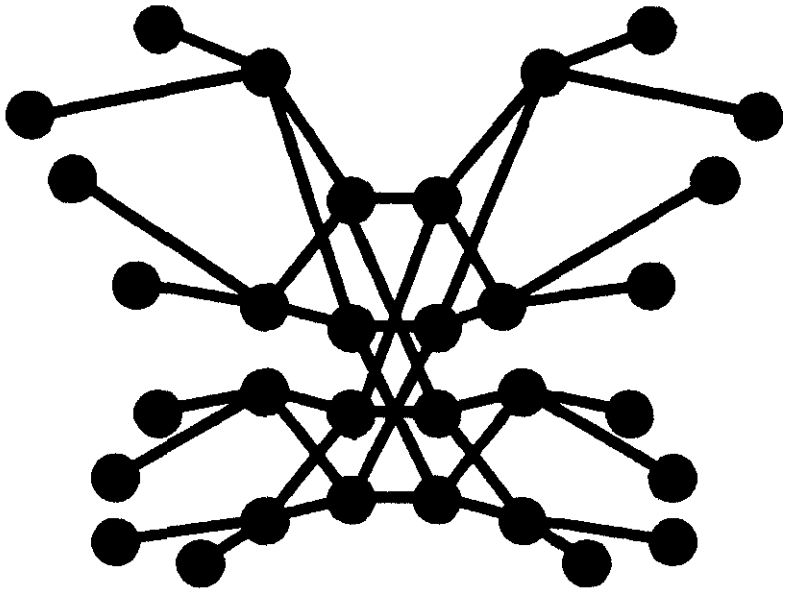


Butterfly Project Report
13

DARPA Parallel Architecture Benchmark Study

C. Brown, R. Fowler, T. LeBlanc, M. Scott, M. Srinivas, L. Bukys,
J. Costanzo, L. Crowl, P. Dibble, N. Gafter, B.March, T. Olson,
L. Sanchis

October 1986

Computer Science Department
University of Rochester
Rochester, NY 14627

DARPA Parallel Architecture Benchmark Study

C. Brown, R. Fowler, T. LeBlanc, M. Scott, M. Srinivas, L. Bukys, J. Costanzo,
L. Crowl, P. Dibble, N. Gafter, B. Marsh, T. Olson, L. Sanchis

October 1986

Abstract

In intensive work over a four-week period in the summer of 1986, seven problems were studied and implemented on the Butterfly. The problems were inspired by various capabilities in computer vision, and were proposed as benchmarks for a DARPA workshop on parallel architectures. They were: convolution and zero-crossing detection for edges, edge tracking, connected component labeling, hough transform, three computational geometry problems (convex hull, voronoi diagram, and minimum spanning tree), three-dimensional visibility calculations, subgraph isomorphism and minimum cost path calculation. BPRs 10, 11, and 14 are detailed reports on three of the problems. BPR13 contains the conclusions of the study and writeups of the work not covered in other BPRs.

This work was supported in part by the Defense Advanced Research Projects Agency U.S. Army Topographic Labs under grant number DACA76-85-C-0001 and in part by the National Science Foundation under grant number DCR-8320136.

Table of Contents

Chapter One: Overview

Overview

Christopher Brown, Tom LeBlanc, Michael Scott
Computer Science Department
29 August 1986

1. Disclaimer

The University of Rochester's response to the DARPA Architecture Workshop Benchmark Study request was a three-week period of activity, commenced from a standing start with the arrival of the problem specifications (Chapter 2). During this time the researchers had to make difficult technical decisions very quickly and to implement and run experiments under severe time pressure. Often sub-optimal methods were chosen for practical reasons. Some of the work has led to internal technical reports, and much of the work is being followed up and will appear in more finished form elsewhere. The contents of this report represent a snapshot of work not currently written up elsewhere, as of our self-imposed deadline of 1 September 1986 (The Architecture Workshop was later rescheduled to mid-November 1986).

The contents of this report represent preliminary work, and should not be considered our best or final answers to the various levels of problems raised by the Benchmark Study.

2. The Study

Rochester's DARPA Architecture Workshop Benchmark Study is made up of several chapters, each written by an individual or a small group. This, Chapter 1, gives an overview of the work and the resulting conclusions. Chapter 2 is a formatted version of the original memo that gave the problem specifications.

The remainder of this document, Chapters 3-9, along with separate Computer Science Department *Butterfly Project Reports*, (numbers 10, 11, and 14) detail technical aspects of our work on individual problems. Generally there is one chapter per problem, except that we used the connected components algorithm (Problem 2, described in *BPR 11*) to do edge-following (Problem 1.c.) as well. Thus Chapter 3 gives results on edge-finding and zero-crossing detection, while Chapter 4 (*BPR 11*) discusses the work on edge-following and connected components. Chapte 5 is equivalent to *BPR 10* and Chapter 8 is equivalent to *BPR 14*.

3. The Effort

Over a three-week period, several students and faculty at the University of Rochester's Computer Science Department worked on the seven architecture

benchmarks proposed by Rosenfeld, Squires, and Simpson (Chapter 2). Because of the short time and limited personnel resources available, the results reported here should not be considered as our last word on any of the problems. We did, however, find the exercise to be stimulating and a good investment. Our report takes the form of this brief summary document and a collection of chapters written by individuals and small groups who addressed individual problems.

Those directly involved in the effort were two staff members, five faculty members, and six graduate students varying from pre-first-year to third year in the areas of artificial intelligence, systems, and theory. The concentration of work was relatively intense, varying from approximately 20% effort to 75% effort per person over the three weeks.

Rochester's place in the Strategic Computing program is to investigate and build programming environments for parallel vision. With this charter, we felt that the more benchmark implementations we could build the better. Further, in the area of programming advanced parallel architectures, often interesting software engineering must be done to improve implementations in the face of performance facts. We believe that theoretical or simulated results, while safer to propound, are of limited interest. Beyond our desire to get programs running, our goals were diverse.

(1) The primary goal is to evaluate the Butterfly Parallel Processor architecture and its existing software resources.

(2) Some of us wanted to use and test utilities we had already developed (e.g. the BIFF utilities used for the edge-finding task and the UNION-FIND package used for connected component labelling.)

(3) Some wanted to code applications in recently-implemented parallel languages and program libraries (e.g. LYNX was used in the triangle visibility task, and the Structured Message Passing library was used in the shortest path problem).

(4) Some wanted to modify and extend existing projects (e.g. the undirected edge-detector extension for the Hough transform task. Another example was an experimental modification of a clustering program to do the minimum spanning tree task -- that work is not reported here.)

(5) Some wanted to explore the mapping of current parallel algorithms from the theoretical literature onto parallel architectures, and to open research avenues in this direction (e.g. the subgraph isomorphism task, which has already generated interesting new scientific results, and the computational geometry tasks).

There was little problem in implementing most of the problems. All told, four programming environments were used:

(1) C and raw Chrysalis (the Butterfly operating system)

(2) The Uniform System of BBN

(3) Structured Message Passing (developed at Rochester)

(4) LYNX (ported to the Butterfly at Rochester).

The programmers ranged from naive first-time users of the Butterfly to highly experienced and sophisticated programmers who could (and did) modify system internals to improve performance.

4. The Problems

The original problem statements appear in the next chapter. Detailed write-ups of our approach to and results on the problems follow in separate chapters. The problem statements were followed as closely as made sense given the scientific goals of the study. For example, in the triangle visibility problem, floating point was not used because the inefficient software implementation of floating point would distort the interesting statistics. (The Butterfly does in fact need the Floating Point Platform upgrade if it is to be useful in serious scientific computing.) In the convex hull problem we went to a larger-than-specified problem size because of results with sequential implementations, and in the graph isomorphism problem we used a smaller problem size than specified for technical reasons. An ambiguity in the shortest path problem statement was interpreted in a way that was *not* advantageous to the Butterfly architecture but seemed to be indicated by the "hint" in the problem statement, and which was more practical given the time constraints. Wherever we have changed a problem specification we have tried to explain why, and tried to indicate responsibly what the consequences of literal interpretation would have been.

We chose the Butterfly several years ago because, among other things, its hardware architecture imposed the *least* constraint on the abstract models of computation it supported. Thus mapping problems onto the Butterfly is a doubly interesting exercise. There is a theoretical phase in which a good algorithm (using one or another abstract computational model) is chosen and the abstract model is matched with a Butterfly programming environment. Then there is an engineering phase in which the implemention is built and made efficient. The best results occur when both these phases are done well. In this set of problems sometimes the first phase went well but the second phase was not attempted (as in the geometry problems we did not implement) or needs more work (as in the triangle visibility problem). Also there were some cases in which the first phase was given short shrift because it looked like a research problem (e.g. the subgraph isomorphism problem), but the second phase was done very stylishly.

The computational domain of the benchmark was not one that could fully take advantage of the Butterfly's MIMD architecture. One computational aspect lacking in the benchmark problems is the case of a cooperating set of independent programs, such as occurs in client-server models. The benchmark tested performance (of programmers, languages, architectures, operating systems, programming environments) on single algorithms solving easily-stated problems. This limitation is worth noting since, in advanced systems, cooperation and

communication between basically independent processes will be important. Also the benchmark problems were small compared to a working AI system. Another set of benchmark problems to illuminate these issues could be proposed and might include construction of a file system, or a system in which results of disparate, asynchronously computed results are merged.

Within its limited perspective, the benchmark did comprise a diverse and challenging set of problems, and allowed us to reach several conclusions. For details on the technical approaches, performance, and individual conclusions, see the following chapters. The next section gives some highlights of our observations.

5. Observations

It is difficult to boil down the diversity of our results into a small set of out-of-context conclusions. Nevertheless, the following observations seem safe.

(1) During the last year, advances made at BBN and at the University of Rochester have made the Butterfly much easier to program under several complementary models of computation. A programmer starting with only a knowledge of standard sequential programming can now produce parallel programs (in the Uniform System or Structured Message Passing) in a day or two. Alternatively, knowing a modern language like Ada would make learning LYNX, and subsequent Butterfly programming, quite easy.

(2) The Butterfly can be efficiently (as well as easily) programmed using several "virtual architectures" (models of parallel computation).

(3) The Butterfly architecture can implement a wide variety of abstract parallel models of computation. Further, the combination of significant local memory and quickly accessible "shared" memory gives the capability for several complementary types of parallelism working together. While programming environments that emphasize one or another parallel model are available now, a single environment that gives the programmer access to a mix of computational models is not. The subgraph isomorphism problem illustrates one case in which a mix would have been useful. At Rochester the PSYCHE project has the goal of providing unified support for a variety of parallel computation models, including both shared memory and message-passsing.

(4) For serious work in the area of scientific computation covered in the benchmark, and probably for general programs, the new Butterfly Floating Point Platform is a necessity. Both floating point operations and integer multiplies are a serious bottleneck (see the Hough Transform Problem).

(5) Microcode support for debugging and performance monitoring would be a significant improvement. There would be considerable payoff in a small microcode fix to provide 32-bit atomic operations. One specific (and easy) upgrade would be microcode hooks to allow logging atomic operations. This facility would allow a reliable record of the order that processes enqueued entries on dual queues.

(6) Memory management is a serious problem. The scarcity of Segment Attribute Registers makes their management a major concern. The inability of the 68000 to do demand paging is also awkward (here again the problem is solved by the Floating Point Platform upgrade). Very large memory objects (larger than one physical memory) are an interesting issue that a few groups are working on -- some benchmark problems (e.g. shortest path) expose the desirability of a clean solution for the large object problem.

(7) A switch that supported simultaneous communication with several destinations would improve the implementation of broadcast or multicast communication used in many algorithms. A combining switch might reduce memory contention, but its efficacy is a research issue.

(8) The Uniform System really provides a global shared name space, not a shared memory. To achieve good speedup of parallel algorithms, local memory must be used to avoid memory contention. Even knowing the standard tricks is not enough to guarantee good performance. The Hough Transform chapter provides an interesting example evolution of program ameliorations. A "shared memory" (as in the planned Monarch) would seem to support Uniform System style programming better. However, it is doubtful that remote memory can ever be made as fast as local memory, and so the local-global question cannot be avoided. A very fast block-transfer capability would improve matters in the current architecture, and would not close off any options in the computational models the Butterfly would support. However, the block-transfer fix does not address the local-global conflict at the conceptual level. Similarly, the fast-switch "shared-memory" does not solve the local-global conflict at the technical level. What is needed perhaps is continued diversification of the abstract models of computation available and in the programming environments that support them.

(9) Amdahl's law is important, and any serial program behavior has serious adverse consequences in speedup performance. Such serialization sometimes hides in the system software and special effort (and talent) are required to avoid or fix it (e.g. the parallel memory allocation modification introduced in the convex hull implementation). The timings shown in Chapter 4 (BPR 11, connected components) and Chapter 7 (triangle visibility) are revealing. Systems code to allocate memory and replicate data dominates times if it is not parallel.

(10) Software to support (efficiently) many more processes than processors on the Butterfly would make implementing a great many important algorithms easier. There are many algorithms in which a process is dynamically allocated to each problem object (e.g. the nodes of a graph), for large numbers of objects. The Uniform System does not answer because it is unable to do process synchronization: processes cannot be blocked, unscheduled, awakened, etc. One reasonable response to this need would be a programming environment such as Concurrent Euclid (Ada would be usable

but not as good), with monitors or a similar means of concurrency control/encapsulation. The actual composition of the environment is a research issue, but it may be necessary to have something like a full blown object-oriented system in which tasks are represented as first class entities encapsulating data, code, and process.

6. Concluding Remarks

The Benchmark Study was a stimulating exercise that served several purposes at Rochester. It has led to new software utilities, to new understandings of Butterfly strengths and weaknesses, to applications for new programming environments developed at Rochester, and to new research avenues in parallel algorithm theory and development. It has encouraged us that our work in building programming environments for the Butterfly has been effective.

Several of the benchmark problems (e.g. the geometry problems) were useful but uncomfortable because they underlined current weak points in the programming systems we have. The graph algorithms need a high degree of cheap (i.e. not SMP or LYNX) parallelism that is independent (i.e. not US-style) -- thus they exposed fruitful areas for future research. We welcome such problems. Our goal is to get the most out of the flexible MIMD architectures of the future, and "counterexamples" to current solutions are always interesting and important. We believe that one of the most promising and important research areas centers around the goal of a single programming environment that can take advantage of the potential for several sorts of parallelism in tightly-coupled MIMD computers, and we are now working actively in in that area.

We believe that much can and will be gained by continuing with the method we have been pursuing at Rochester -- a symbiosis of theoretical, systems, and applications research. We shall continue to build systems for internal and external use that incorporate our theoretical insights and meet the needs of applications. With the basic research underlying the systems, and with the systems as tools, we and others will move forward toward understanding and controlling state-of-the-art parallel programming systems.

Chapter Two: Problem Specifications

DRAFT

MEMO TO: Designers of architectures for image understanding (IU)

FROM: Azriel Rosenfeld, Bob Simpson, Steve Squires

SUBJECT: New architectures for IU

DARPA plans to hold a workshop during the week of Sep- tember 8 in McLean, Virginia to discuss what the next steps should be in developing IU architectures that could be available to researchers by the 1990's.

A lot is known about architectures for low-level vision, but we need to move toward systems that can handle the total vision problem, including both the low- and high- level ends as well as the interface between the two.

Appended to this memo is a set of "benchmark" IU problems. We have tried to define them as precisely as pos- sible, so as to make it possible to predict how a given sys- tem would perform on them. (We have provided some refer- ences to the relevant literature for your convenience.)

You are invited to make such predictions for your (existing or proposed) systems, and to prepare a short paper documenting the results. This paper should be sent to us for distribution to the Workshop attendees by mid August, so everyone will have a chance to evaluate the results and dis- cuss them at the Workshop. If your system is not very effi- cient at some of the tasks, you may wish to indicate how you would improve or augment it to make it more efficient.

We look forward to hearing from you and to seeing you at the Workshop.

Appendix: IU benchmarks

(1) Edge detection
In this task, assume that the input is an 8-bit digital image of size 512 x 512 pixels.

a) Convolve the image with an 11 x 11 sampled "Laplacian" operator [1]. (Results within 5 pixels of the image border can be ignored.)

b) 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.

c) 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 [2], Section 11.2.2.)

(2) Connected component labeling
Here the input is a 1-bit digital image of size 512 x 512 pixels. The output is a 512 x 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 [2], Section 11.3.1.)

(3) Hough transform
The input is a 1-bit digital image of size 512 x 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 x 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 [2], Section 10.3.3.)

(4) 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]. Several outputs are required.

a) 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 [3], Chapters 3-4.)

b) 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 [3], Section 5.5.)

c) 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 [3], Section 6.1.)

(5) Visibility
The input is a set of 1000 triples of triples of real coordinates ((r,s,t),(u,v,w),(x,y,x)), 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).

(6) 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.

(7) 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 real- valued 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.)

References

(1) R.M. Haralick, Digital step edges from zero crossings of second directional derivatives, *IEEE Transactions on Pattern Analysis and Machine Intelligence 6*, 1984, 58-68.

(2) A Rosenfeld and A.C. Kak, *Digital Picture Processing* (second edition), Academic Press, New York, 1982.

(3) F.P. Preparata and M.I. Shamos, *Computational Geometry - An Introduction*, Springer, New York, 1985.

Chapter Three: Edge Finding and Zero-Crossing Detection

Task One : Edge Detection

Thomas J. Olson

1. Introduction

The task is to detect edges in an eight-bit digital image of size 512 x 512 pixels. It is divided into three steps : convolution with an 11 x 11 Laplacian-of-Gaussian operator, zero crossing detection, and chain encoding sequences of connected zero crossings. In these experiments steps a) and b) were handled using image processing utility functions from the Butterfly IFF (BIFF) image processing library [3]. Step c) was performed by a special purpose routine adapted from a connected component labelling function. The test image was a densely textured natural scene in which approximately 25% of the pixels in the zero crossing image were ones. Our conclusions, briefly, are that for the 119-node Butterfly

a) convolution takes 3.48 seconds,

b) zero crossing detection takes 0.16 seconds, and

c) finding chain codes for lines takes 1.47 seconds for this image.

These times are for computational kernels; they do not include time to load the image, allocate memory for the output, et cetera. The sections that follow present implementation details and benchmarks for the first two steps. The third is described in the attached Butterfly Project Report [2].

2. Convolution

The convolution was performed by *convolve()*, the library version of the BIFF utility **iffconvolve**. *Convolve()* uses the Uniform System library [1] to perform a parallel FOR loop over rows of the output image. Each process created in this way does a sequential loop over rows of the mask; for each mask row it makes local copies of the appropriate mask and image rows and then does a linear convolution of the copied rows into a local array of 32-bit accumulators. Finally, it divides each accumulator by the user-supplied divisor and copies it to the output image.

It is easy to see that in order to produce an output, *convolve()* must perform $(502^2)(11^2) = 30{,}492{,}484$ multiplications. (The first term involves 502 rather than 512 because we ignore outputs within five pixels of the border.) Because it does so many multiplications, the execution time of *convolve()* is dominated by the 68000's multiplication time. Unfortunately the current Butterfly C compiler generates a call to an integer (32-bit) multiply routine even when the arguments are both shorts. Figure 1 shows timings and speedup curves for four versions of iffconvolve running an 11 by 11 mask on a 512 by 512 image. The first is coded in standard optimized C. The second replaces the multiplication in the innermost loop with a call to an assembly language

short multiply routine. For the third, we edited the compiler's assembly language output to replace the subroutine call with an in-line short multiply instruction. This is the version normally kept in the BIFF library. In the last version we replaced the multiply instruction with an addition. This gives an incorrect output, but indicates the sort of performance we might expect from a Butterfly with a fast full-parallel multiplier.

The *convolve()* routine is written to be as general as possible and therefor does not take advantage of some features of the problem as stated. First, the 11 by 11 mask to be used is known at compile time. This makes it possible to avoid copying the mask rows into local memory at execution time. The copy operation is quite fast, so we would expect the principal effect of this change to be a reduction in memory contention. The speedup curves of Figure 1 indicate that memory contention is not a serious problem for *convolve()*, so the net effect would be minor. Second, the mask is symmetrical. By factoring appropriately, the number of multiplies that the convolver must do can be cut almost in half. For example, a process working on input rows 0 through 10 would add row 10 to row 0, row 9 to row 1 et cetera, and then convolve rows 0 through 5 with rows 0 through 5 of the mask. Figure 2 shows the effect of these optimizations on the standard and simulated fast multiply versions of *convolve()*.

It should be noted that if we are willing to accept an approximation to the laplacian of a gaussian, we can speed the computation up substantially. Since gaussian masks are x-y separable, we can reduce convolution with an 11x11 mask to two convolutions with 11x1 masks. We can take advantage of symmetry as before, so that for each convolution we do 6 multiplies and 11 adds per pixel. The cost of an 11x11 gaussian convolution then becomes a mere 3,084,288 multiplies and 5,654,528 additions. We can compute the laplacian of the result by convolving with a 3x3 laplacian approximator. However, this method gives a relatively poor approximation to the truth. Better, though slightly more expensive, is to use the Difference of Gaussian (DOG) approximation, which requires two 11x11 convolutions followed by a pointwise difference. We have not benchmarked this method, but expect that it would reduce execution times by at least a factor of three (to about 1.1 seconds) based on the relative numbers of operations.

3. Zero Crossing Detection

For zero crossing detection we use the BIFF utility *zerocr()*. *Zerocr()* is written as a parallel FOR loop over output scan lines. Each row process reads in the corresponding input row and its two neighbors (the top and bottom rows are handled specially). For every positive pixel in the input row it examines the eight neighbors and stores a one in a local array if any of them is negative - otherwise it stores a zero. Finally it copies the local array into the output array. Timings are shown in Figure 3.

References

1. BBN Laboratories, The Uniform System Approach To Programming the Butterfly Parallel Processor, Version 1, Oct 1985.
2. L. Bukys, Connected Component Labelling and Border Following on the BBN Butterfly Parallel Processor, Butterfly Project Report 11, University of Rochester, Computer Science Department, Aug 1986.
3. T. J. Olson, An Image Processing Package for the BBN Butterfly Parallel Processor, Butterfly Project Report 9, University of Rochester, Computer Science Department, Aug 1986.

Figure 1 : Four Versions of lffconvolve
convolving 11x11 delsqg with 512x512 natural image
run times in seconds

procs	standard C	short mpy subroutine	short mpy in line	simulated fast mpy
4	617.52	318.68	153.53	121.11
8	313.66	159.35	78.00	61.53
16	156.84	80.95	39.00	30.77
32	78.44	40.49	19.52	15.40
64	39.23	20.26	9.77	7.71
119	24.52	12.66	6.11	4.82

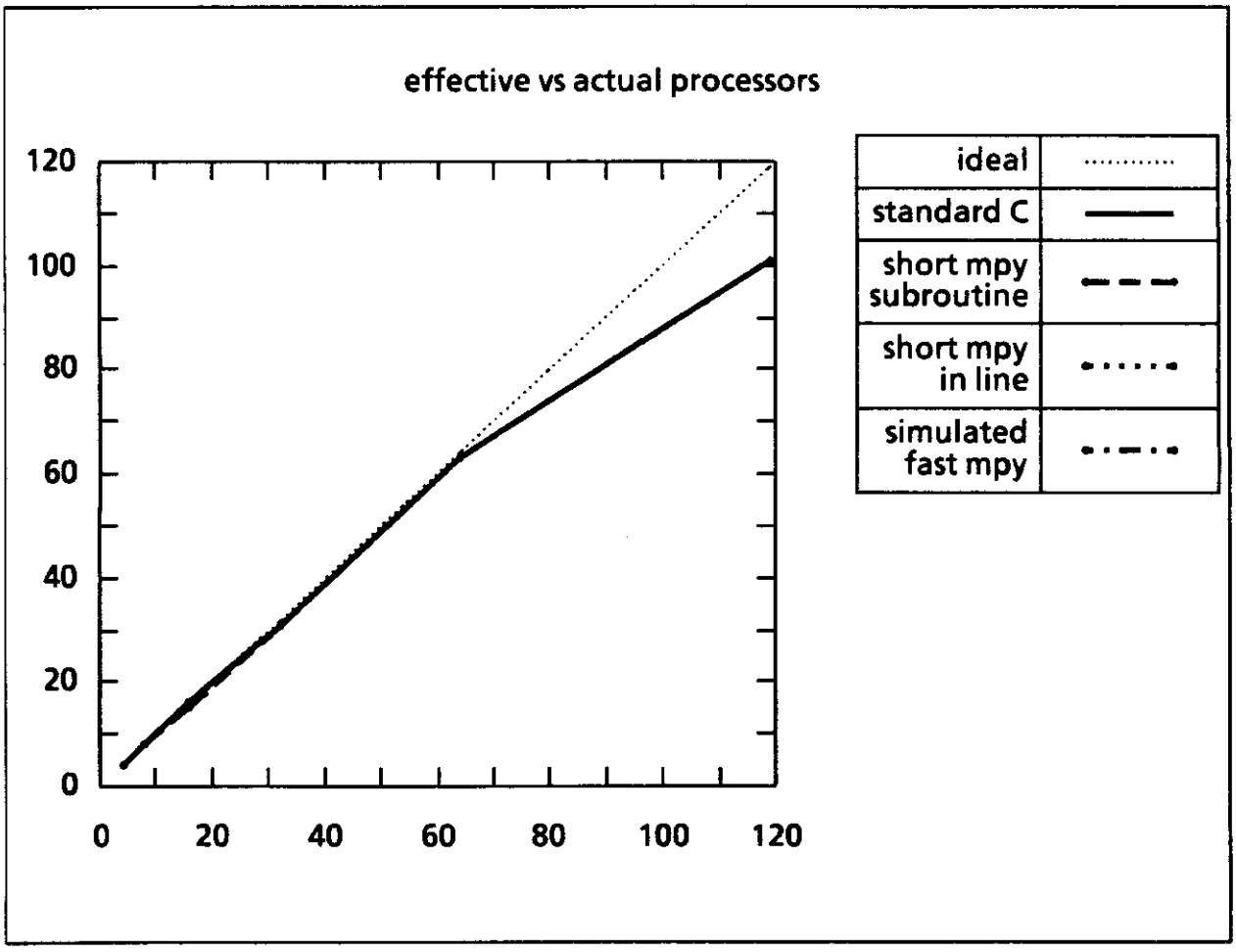


Figure 2 : Optimized Iffconvolve
convolving 11x11 delsqg with 512x512 natural image
taking advantage of mask properties
run times in seconds

procs	standard short mpy in line	standard simulated fast mpy	optimized short mpy in line	optimized simulated fast mpy
4	153.53	121.11	87.27	68.25
8	78.00	61.53	43.58	34.04
16	39.00	30.77	22.14	17.30
32	19.52	15.40	11.08	8.66
64	9.77	7.71	5.56	4.35
119	6.11	4.82	3.48	2.72

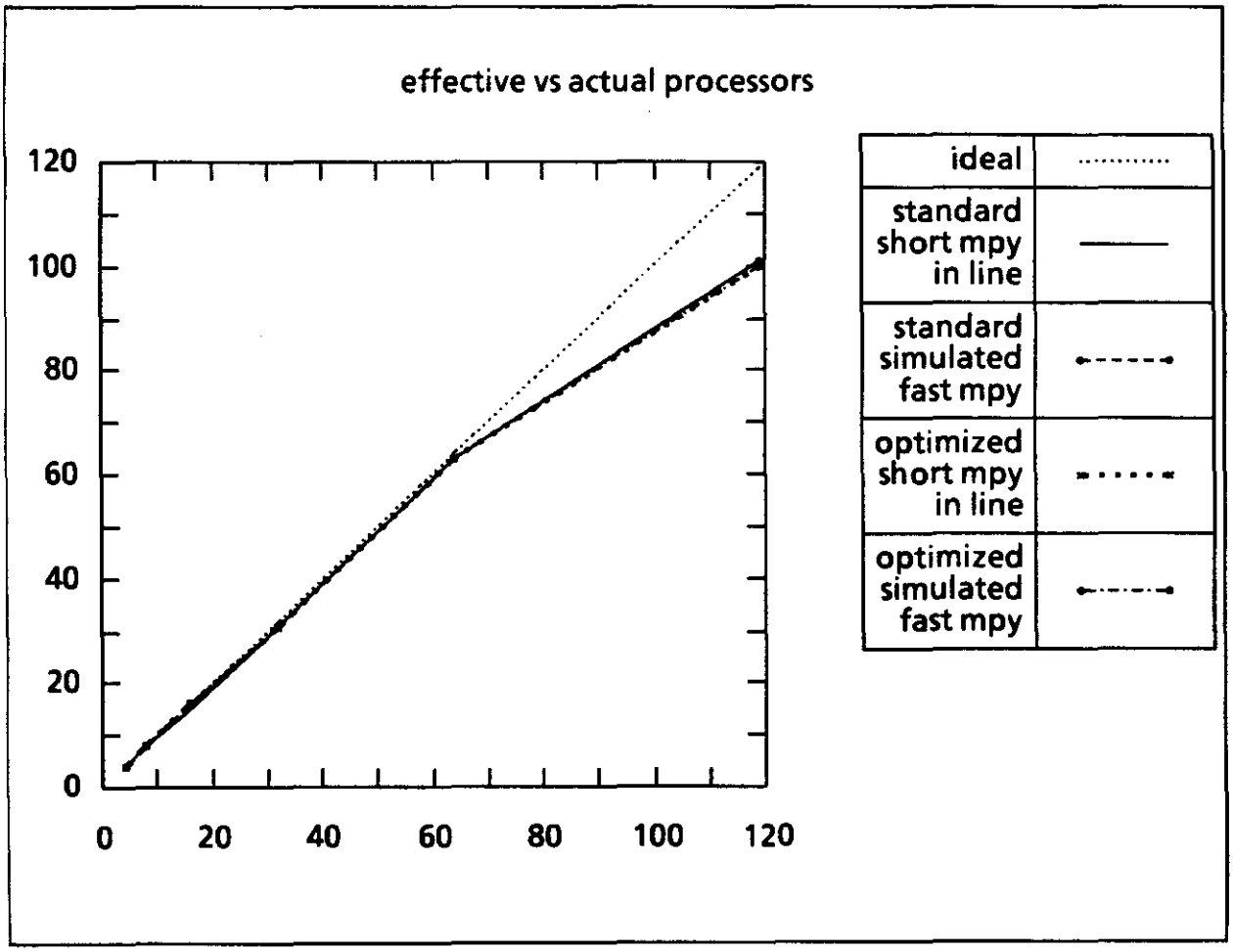
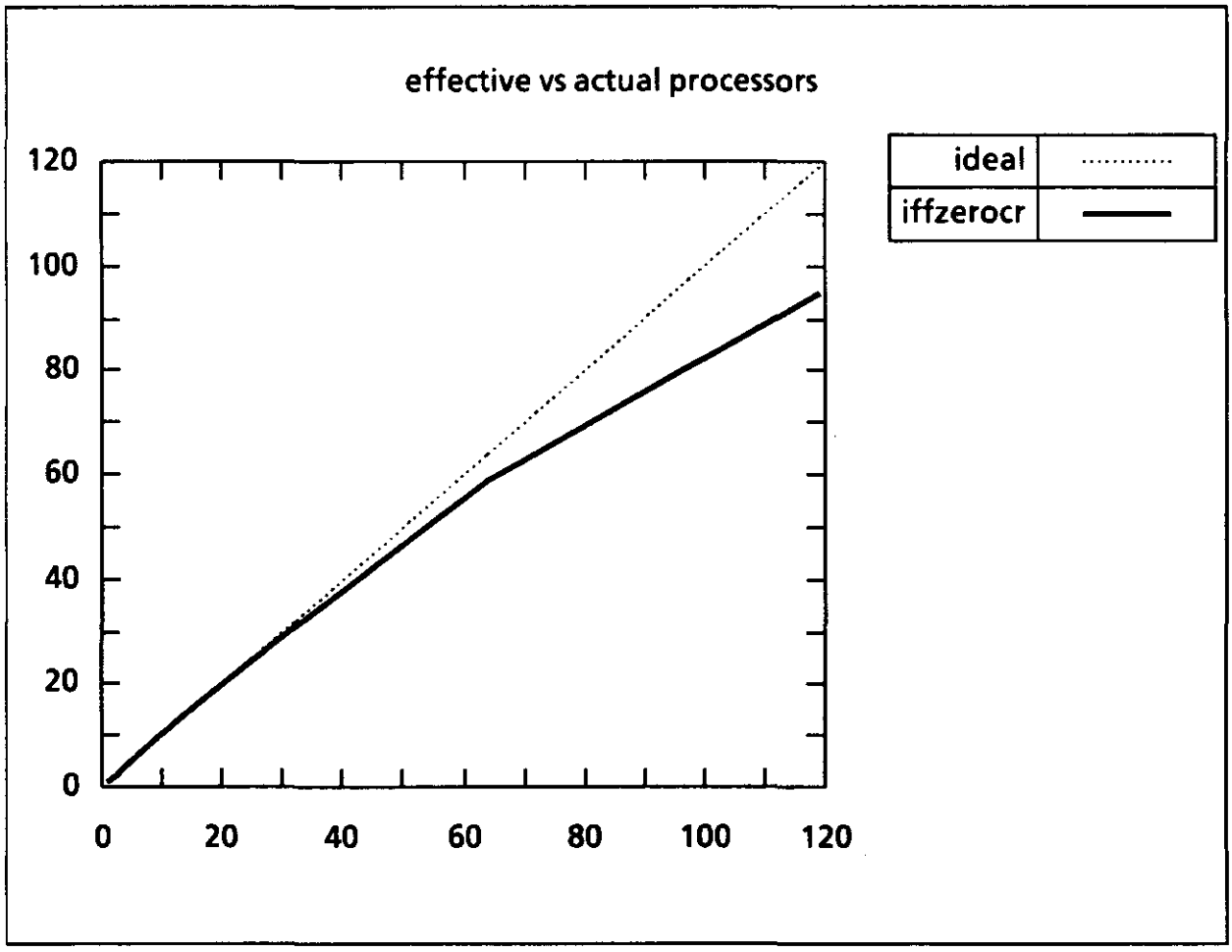


Figure 3 : zero crossing detection

run times in seconds

procs	iffzerocr
1	15.77
2	7.85
4	3.92
8	1.96
16	0.99
32	0.51
64	0.26
119	0.16



Efficient Convolution with Symmetric Masks

Tom Olson

The following amelioration to the inner loop of the 11x11 Laplacian convolution approximately halved the time needed for this portion of the benchmark, from 6.11 seconds to 3.48 seconds.

In the basic convolution multiply and add stage, every point (x, y) in the output can be computed by the expression

```
for row = 0 to 10
  for col = 0 to 10
    tmp = tmp + mask(row, col)*image(y+row, x+col)
```

which requires 121 adds and multiplies. We can use the symmetry properties of the mask to reduce the work. First, rewriting the above gives

```
for col = 0 to 10 tmp = tmp + mask(5, col)*image(y+5, x+col)
 for row = 0 to 4
  for col = 0 to 10
    tmp = tmp +
          mask(row, col)*image(y+row, x+col) +
          mask(10-row, col)*image(y+10-row, x+col);
```

Since mask(10-row, col) = = mask(row, col), we can write this as

```
 for col = 0 to 10 tmp = tmp + mask(5, col)*image(y+5, x+col)
 for row = 0 to 4
  for col = 0 to 10
    tmp = tmp +
          mask(row, col)*(image(y+row, x+col) + image(y+10-row, x+col));
```

which takes the same number of adds but only 66 multiplies. We can do even better by realizing that we're going to do this at every point along a row. That means we can precompute the image row sums once and for all. That is, to compute row y in output, sum rows y and y+10, y+1 and y+9, ..., y+4 and y+6, and include row y+5 to get a 6 row by n column matrix. Then simply convolve with the bottom 6 rows of the mask.

In summary, the standard convolution for 512x512 image and 11x11 mask, ignoring dropping edge outputs is
502x502x11x11 = 30,492,484 mpys and adds.

Using the symmetry of the mask, the code above reduces the counts to
502x512x5 = 1,285,120 adds to make the 502 6x512 matrices, plus

502x502x11x6 = 16,632,264 mpys and adds to do the individual convolutions.

So multiplies are reduced by almost half.

We can take further advantage of the available symmetries by folding the 6x11 mask we use in the implementation above around the middle column. This cuts the number of multiplies to 502x502x6x6 = 9,072,144. Similarly, by folding the 6x6 mask around its diagonal we can reduce the number of multiplies to 21 per mask, giving 502x502x21 = 5,292,084 for the total. Unfortunately the number of additions stays constant at about 16M, and the loops and indexing become complex, so it is not clear that these refinements will actually improve execution times. These techniques are applicable to any rotationally symmetric mask, so if they do prove worthwhile we will probably put a special convolution routine into the BIFF library for rotationally symmetric masks.

Chapter Four: Connected Component Labeling

see: Butterfly Project Report 11

Chapter Five: Hough Transformation

see: Butterfly Project Report 10

Chapter Six: Geometrical Constructions

Geometry Problems

Robert J. Fowler and Neal Gafter
August 21, 1986

1 Introduction.

The approach that we took in investigating the suitability of the Butterfly architecture for the geometric problems was to attempt the parallelization of good sequential algorithms for those same problems. Thorough discussions of such sequential algorithms for computational geometry may be found in [Mel84] and [PS86]. In [ACG*85] Aggarwal *et al* sketched some parallel algorithms for computational geometry derived from these sequential algorithms. Problems they addressed included the computation of two–dimensional convex hulls, Voronoi diagrams, and other problems. The model of computation that they used is the concurrent–read, exclusive–write (CREW) variant of the PRAM model of computation. The methods used by Aggarwal *et al* (at least for convex hull and the Voronoi diagram) are the parallelization of optimal sequential algorithms. We use a similar approach, but directed not towards the theoretically interesting questions of optimality in the asymptotic sense and of membership in well–known complexity classes (e.g. NC), rather towards achieving good performance when implemented on the Butterfly. In particular, we are using these problems as an opportunity to explore how to map algorithms designed for abstract models of parallel computation onto a physically realized parallel architecture.

2 Salient Aspects of the Butterfly Architecture.

Each node on the Butterfly consists of a processor and a megabyte of memory local to that node. In addition, the nodes are connected with a "butterfly" (hence the name) interconnection network that allows each node to access the memory of the others. The interconnection network resolves contending attempts to access each memory module by serializing those attempts. Because of the large granularity of the memory modules this "hidden" serialization of contending access attempts can be a major problem with the approach of attempting to adapt PRAM algorithms that assume the possibility of very fine grained parallelism. We believe that the investigation of data structures that can be shared with low contention among a reasonable number of processors to achieve medium scale parallelism on such a machine is an area for potentially fruitful research.

The architecture seen by an application programmer is not determined solely by the underlying hardware, rather by a combination of hardware and the software architecture of the operating system cum programming environment. The latter can have as much or more of an effect as the former on the successful implementation of an algorithm on a particular machine. The quality of the programming environment affects both the ease of implementation as well as how well the underlying machine is used [Sny86].

Of the programming environments currently available on the Butterfly we chose the Uniform System because it most closely resembles the PRAM model. In the course of this exerecise we encountered the following specific problematic aspects of the Uniform System:

- The Uniform System appears to have been designed to be used in a style in which memory is statically allocated when the application is initialized and in which there is a small number of generators that spawn a large number of tasks. In contrast, the geometric problems naturally seem to fit into a style that uses dynamic memory allocation and in which tasks are spawned dynamically a few at a time as a program executes its recursive algorithm. There appear to be substantial penalties for using this latter style in the Uniform System.

- The geometric problems all involve the construction of a graph with some specified properties from a set of input points. An efficient parallel program to solve such problems must have efficient parallel implementations of the abstract data types *set* and *graph*. A natural representation of a graph is as some form of dynamic list structure. One consequence of this is that either the system or the application program should provide efficient parallel memory management. The global memory management provided by the Uniform System is in fact sequential. Thus, even a program that appears to be parallel can in fact be serialized by system code. We provided our own parallel memory management, but this illustrates how easy it is for implicit serialization to be introduced by the programming environment.

- Another consequence of using dynamic list data structures is the need to provide concurrency control for the elements. It is possible to do concurrency control in the Uniform System but it is awkward. The natural way of doing this is to incorporate the concurrency control mechanism in the programming language.

- The assignment of processors to tasks must be made more efficient and flexible. Task startup can introduce a substantial amount of overhead that can wipe out the benefits of fine and medium grain parallelism. In addition, we discovered that the implementation of task scheduling task allocation scheme can force a processor to be idle when it is not logically required by the application to be so and there is useful work it could do.

These factors contribute to the difficulty of using the Butterfly hardware architecture effectively. This illustrates the need for improved parallel programming environments as well as the need for those environments to provide the programmer with an accurate and detailed enough model of computation to guide intelligent choices in implementation.

Because our interest in these exercises is the investigation of the problem of mapping abstract algorithms onto the Butterfly we emphasized general implementations rather than attempting to tune the programs to exploit specific details of the problem statement(s). Thus, we are at least as interested in very large numbers of points distributed in arbitrary regions as we are in small numbers of points distributed uniformly in a square.

3 An Abstract Convex Hull Algorithm.

We concentrated our efforts on understanding the design and implementation of a parallel two–dimensional convex hull program. Most of the issues that would arise in the implementation of programs to solve the other problems appear in the convex hull problem and, in the limited time available, we deemed it more important to understand one implementation than to dilute our efforts by attempting "quick and dirty" implementations of all of the geometric problems.

Our approach is similar to that proposed by Aggarwal *et al*, but is an attempt to exploit the bounded, medium–grained parallelism found on a Butterfly. It is a parallelization of the Quickhull [PS86] algorithm. Our parallel implementation has the following steps:

1. Select the maximum and minimum elements along some direction using as much parallelism as is effectively available. We assume that there are N points and that we can use P procesors effectively. Each processor is given a subset of approximately N/P elements of which it finds the maximum and minimum elements. The global maximum and minimum is computed from the subset values using the usual PRAM trick of combining the subset extrema using binary trees. The useful (non–overhead) part of this step requires time proportional to $N/P + logP$.

2. If the initial points are A and B, then the initial approximation to the hull is the ordered list of the directed line segments AB and BA. The remaining points are partitioned into two sets. one above AB and the other above BA. This is done in parallel, with each processor working on a subset of the input. To allow for parallel partitioning, a set of points can be represented as a tree of variable length arrays (sub-buckets) of points. The partitioning is done by having each processor copy its part of the input local memory using a block transfer. It then scans it's input sequentially while locally building its part of the output set consisting of those points above the specified line. The other points are discarded from the set because they can not contribute to the hull at this location. The sub–sets from each process are merged in parallel into the output trees. The reason for using a tree rather than a simple linked list is to allow for

efficient parallel access and to allow each internal node of the tree to keep a count of the points that it contains. As each sub–set is constructed the point furthest from the line is found. The local extrema are also combined in the binary tree to find the global extremem. This point is on the convex hull. The time for one of these steps is proportional to $N/P + \log P$.

3. At any time the current approximation to the hull is kept as a doubly–linked list of points. All of the as yet unknown points on the hull are outside this approximation so the points within it can be discarded. Furthermore, any point that can possibly be added to the hull between a pair of points in the current approximation must be above the line of support through them. When a newly found hull point is added to the list it replaces one of the line segments on the approximation with two others. A new task is created for each of these. Each task takes as its input a new line segment and the set of points above the old segment. It selects the set of points above its line segment and finds the extremal point with respect to that segment. This point is guaranteed to be on the convex and when added to the approximation initiates the next level of recursion. Each branch of the recursion terminates when its input is the empty set.

   These sub–problems generated by the recursion are solved in parallel. As above, selection and maximum are done in parallel if the size of the input set is large enough. If the largest sub–problem of each step in the recursion is a bounded fraction of the size of its parent problem then the total depth of the tree will be proportional to $\log H$ where H is the number of points on the hull. Since the expected number of hull points will be proportional to $\log N$ [PS86] the total expected time to execute the algorithm should be proportional to $\log\log N(N/P + \log P)$.

Note that the problem statement says that the points are distributed uniformly in a square and that there are only 1000 of them. By [PS86] this means that the expected number of points on the hull will be approximately twenty. Given the granularity of parallelism available on the Butterfly this is a very small problem instance and it is difficult to justify a parallel solution for it. We have therefore taken the licence to solve much larger problem instances and to look at other distributions of the points.

Although we have not attempted to tune the program to take advantage of the details of the problem statement, we are taking advantage of the square region by using the line $x = y$ to determine the direction in which to search for the initial extremal points.

Note also that our initial implementation does not use the above mentioned "tree of arrays" representation of a set. As a result there may be contention for adding points to the set. This contention may be contributing a linear time component to the running times. Once we have had the time to run the experiments needed to understand the current implementation better we can experiment with changing the representation of a set.

4 Evaluation of the Convex Hull Program.

It is difficult to evaluate the effectiveness of parallelism in geometry problems because the sequential Quickhull algorithm is so good. Craig McGowan provided the following set of single processor Quickhull timings (in seconds) obtained on several varieties of Sun workstation.

Points	2/50, 4 Meg	2/120 2 Meg	3/160C, 4 Meg
100	0.02	0.02	0.01
200	0.04	0.04	0.02
500	0.08	0.08	0.03
1000	0.16	0.16	0.05
2000	0.34	0.34	0.09
5000	0.84	0.84	0.24
10000	1.66	1.66	0.48
20000	3.34	3.34	0.98
50000	8.16	8.16	2.42
100000	16.30	16.30	4.87
200000	32.64	50.74	9.72
500000	182.95	308.51	37.23

These are Stuart Friedberg's comments on these experiments:

1. This Quickhull program is a CPU-bound task that carefully avoids 32-bit multiplies and floating point operations. The Sun-3's are roughly 4 times faster than Sun-2's. For CPU-bound tasks with int or long multiplies or with floating point, the Sun-3's should do even better. A simple program profiler for the Butterfly indicates that some programs spend more than 95 percent of their time in Chrysalis doing the software integer multiply necessary to compute array indicies. The 68020 processor, unlike the 68000 has a hardware 32-bit multiply. Thus it appears that a processor upgrade could have a significant impact upon execution speeds. The addition of a 68881 floating point coprocessor could have an even greater effect on speed in computations in which floating point and trigonometric functions are common.

2. The Sun 2/120's are Multibus-based, while 2/50's don't even have a bus. This makes IO comparisons hard between them and a Sun-3/160C, which is VMEbus-based. However, we can see that when both the 2/50 and 3/160C with the same amount of memory are thrashing, the Sun-3 still runs 6 times faster. It would be interesting to see a comparison between a /120 and a /160 with the same amount of memory and the same processor type.

Despite the excellent performance of the sequential algorithm the parallel version was able to use some parallelism effectively. Given our initial implementation using the sequential memory allocator, a Butterfly computes the convex hull of 10000 points in the following times:

Processors	Time	Speedup
1	7.60	1.00
2	4.31	1.76
3	3.22	2.35
4	2.54	2.99
5	2.06	3.68
6	1.81	4.18
7	1.73	4.38
8	1.54	4.91
9	1.48	5.13
10	1.42	5.32
11	1.20	6.32
12	1.24	6.11
13	1.37	5.52
14	1.17	6.44
15	1.20	6.33
16	1.15	6.60

These times (in seconds) reflect the actual computation time, excluding the time to load the program and the input data. As expected. the high overhead of managing the parallel implementation limits the amount of effective parallelism obtainable. Furthermore, the execution times do not decrease monotonically as processors are added. The source of this is likely to be some kind of scheduling or concurrency control artifact introducing serialization in a way that is very sensitive to the number of processors.

Note that a single Butterfly node executes the parallel implementation at about one sixth of the speed of a Sun 2 executing the straightforward sequential implementation. Part of the difference is due to hardware differences and part is due to overhead in accomodating potential parallelism. When 8 nodes are allocated to the problem the Butterfly outperforms the Sun 2. but at no point can it compete with the Sun 3. It is difficult to overcome the handicaps of lower single processor and memory speeds combined with the disadvantage of not having powerful parallel arithmetic in hardware.

To reduce the total amount of overhead and to eliminate a known significant source of "hidden" serialization a second version of the program was written that incorporated its own parallel memory management package. This second implementation performed as follows:

	Number of Points					
	1000		5000		10000	
Processors	Time	Speedup	Time	Speedup	Time	Speedup
1	1.08	0.99	4.29	1.0	8.01	0.99
2	.60	1.78	2.4	1.78	4.33	1.84
3	.46	2.30	1.8	2.38	3.27	2.44
4	.32	3.28	1.37	3.12	2.46	3.25
5	.29	3.70	1.16	3.68	2.00	3.99
6	.25	4.25	1.02	4.18	1.81	4.42
7	.26	4.09	.91	4.7	1.61	4.96
8	.24	4.36	.83	5.14	1.46	5.47
9	.22	4.87	.77	5.53	1.35	5.89
10	.22	4.75	.74	5.8	1.28	6.25
11	.20	5.20	.7	6.12	1.20	6.62
12	.20	5.38	.67	6.37	1.14	7.00
13	.19	5.48	.63	6.71	1.11	7.20
14	.20	5.31	.62	6.89	1.08	7.40
15	.19	5.46	.61	6.93	1.04	7.65
16	.19	5.42	.62	6.82	1.03	7.73
17	.18	5.83	.62	6.88	1.02	7.79
18	.19	5.54	.60	7.05	.99	8.06
19	.17	6.14	.6	7.05	.98	8.11
20	.17	6.00	.6	7.10	.97	8.19

The improved program is faster than the original, is able to use more processors effectively on average, and as processors are added the running time decreases.

5 The Voronoi Diagram.

Rather than compute the Voronoi diagram directly we would compute its dual, the Delaunay triangulation. We expect that a straightforward recursive parallelization can be performed upon the divide and conquer Delaunay triangulation program of Lee and Schacter [LS80]. We believe that this is the approach appropriate for the Butterfly.

The problem with this is that the final merge step will take time proportional to $\sqrt{N}$ on average. If all partitions are made so as to keep the merges proportional to boundary length, such as by alternating in the X and Y directions then the expected time could be $O(\log N\sqrt{N})$. The question is whether or not this can be improved on the Butterfly. The Aggarwal *et al* paper sketched a parallel merge step using $O(logN)$ time and $O(N)$ processors. Thus, it is clear that the merge can be sped up aymptotically on a PRAM, but at the moment it is not clear how to program it and how to design the data structures on the Butterfly so as to simulate the fine granularity implied by their paper.

The "divide" step of the algorithm requires partitioning the points into linearly separable subsets. In sequential implementations this is done by sorting the points along one of the coordinate axes. There is at the moment no general purpose sorting package for the Butterfly. Sorting is one of the most studied and fundamental computational problems, the fact that we still do not have a good, implemented solution on the Butterfly is indicative of the lack of maturity of the software environment on the machine.

6 The Euclidian Minimum Spanning Tree.

The EMST can be easily derived in sequential time proportional to N from the Voronoi diagram (Delaunay triangulation) [PS86] since the edges of the tree will be a subset of the edges of the triangulation.

Kwan and Ruzzo [KR84] survey "edge-adaptive" parallel algorithms for computing minimum spanning trees of arbitrary graphs. These have running times in the CREW PRAM model of

$O(E \log N/P$. Since the edges that need to be considered are a sub-set of the edges of the Delaunay triangulation the cost for the Euclidian minimum spanning tree once the triangulation is found will be $O(N \log N)$. As with the other two problems, this presupposes that we will be able to program efficiently shareable data structures representing dynamic graphs.

An alternative to using a PRAM-style algorithm would be to use Bentley's [Ben80] optimal algorithm that uses $N/\log N$ processors in a fixed interconnection network. Since $\log 1000 \approx 10$ the a program that finds the EMST for 1000 nodes would potentially map very well onto a Butterfly of 100 nodes. In contrast to the PRAM algorithms mentioned above, Bentley's algorithm is designed for a set of simple processing elements that communicate over a fixed interconnection network. In particular, it is suitable for a VLSI implementation. While this avoids the problem of designing shareable dynamic data structures for graphs, the algorithm assumes a fine grained parallelism that depends upon very efficient inter-processor communication. As mentioned elsewhere in this collection of reports the SMP programming environment provides interprocessor communication in approximately two milliseconds. This is still too large in comparison to the amount of computation to be done at each node per message. The effect of communication overhead can be reduced by blocking several logical messages per physical message, but this increases the complexity of the programming effort. What seems to be needed here is some form of inter-processor *streams* interface.

References

[ACG*85] Alok Aggarwal, Bernard Chazelle, Leo Guibas, Colm O'Dunlaing, and Chee Yap. Parallel computational geometry (extended abstract). In *Proceedings 26th IEEE FOCS*, pages 468–477, Tucson AZ, October 1985.

[Ben80] Jon Louis Bentley. A parallel algorithm for constructing minimum spanning trees. *Journal of Algorithms*, 1:51–59, 1980.

[KR84] S.C. Kwan and W.L. Ruzzo. Adaptive parallel algorithms for finding minimum spanning trees. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 439–443, Bellaire, Mich., 1984.

[LS80] D.T. Lee and B.J. Schachter. Two algorithms for constructiong a delaunay triangulation. *Int. J. Comput. Inf. Sci.*, (3):219–242. 1980. Also appeared as GE Technical Report 79ASD007, July 1979.

[Mel84] Kurt Melhorn. *Data Structures and Algorithms, Volume 3: Multi-Dimensional Searching and Computational Geometry. EATCS Monographs on Theoretical Computer Science*, Springer–Verlag, New York, 1984.

[PS86] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry, An Introduction.* Springer–Verlag, New York, 1986.

[Sny86] Lawrence Snyder. *Type Architectures, Shared Memory, and the Corollary of Modest Potential.* Technical Report TR 86-03-04, Department of Computer Science, University of Washington, Seattle, WA, 1986. To appear in Annual Review of Computer Science, Vol. 1, 1986.

Chapter Seven: Visibility Calculations

Triangle Visibility

Christopher Brown, Liudvikas Bukys, Michael Scott

16 October 1986

1. The Problem

The problem as stated is ambiguous. We take it to mean "report visible *vertices*". The size of the problem was well-chosen, providing a reasonable exercise that exposed limitations in algorithms and data structures. The problem specifies floating point, but we use integers. The lack of hardware 32-bit arithmetic in the 68000 is enough to confuse the architectural issues, and the lack of floating point is such an obvious and important one that it should not be further folded into the problem. There is evidence that even the integer multiplication in array index calculations on the 68000 is inefficient enough to distort the architectural picture. Since there is an easy fix to this problem on the Butterfly, issues such as contention, the number of processes supported, and so forth are more interesting.

2. The Approach

A shared memory, SIMD-like, Uniform System virtual architecture fits with the algorithm we chose to implement, which is a quadratic all-against-all comparision of points against triangles for visibility. Below we discuss variations on this theme, and give the justification for the approach that we ultimately implemented. There is of course substantial room for more work on this algorithm, and there are other approaches as well.

3. Three Algorithms

We describe two algorithms, PointTri() and TriTri(), and a hybrid variant. PointTri() is basic.

```
PointTri(Points, Triangles)
{
for_each Point
        for_each Triangle
                if Occludes(Triangle, Point) mark Point "Hidden";
}
```

PointTri() can be enhanced in obvious ways to prune the full $3N^2$ search: In Occludes(), quit early and continue the loop as soon as it is determined that a triangle cannot hide a point. As soon as a point is found to be occluded, break the inner loop. Empirically, it seems this pruning gives a factor of two speedup (random inputs) over the full search. This speedup motivates TriTri(), which removes (a subset of) occluded triangles as well as occluded points from consideration, thus cutting down on the length of both inner and outer loops.

3.1. Point against Triangle

For PointTri(), computation falls into two stages, called 1 and 3 for consistency with TriTri().

(1) Stage 1 is a linear setup stage in which four planes are calculated for each triangle: the plane in which the triangle lies and the plane through the origin and each triangle side. These planes are kept together as a row in a Triangle array, and each point is a row in a Point array.

(2) Stage 3 is the quadratic (doubly-nested for_loop) comparison of points with triangles referred to above. Occluded points are marked "Hidden."

3.2. Triangle against Triangle

In TriTri(), Stage 1 has more to do, there is a Stage 2, and Stage 3 is more complicated. The idea is to sort triangles by order of their likelihood of obscuring other triangles, and to consider them in this order, getting the maximum pruning advantage. The right quantity to sort on is the amount of volume in the (1000 x 1000 x 1000) cube of space shadowed by a triangle (hidden by it from the origin). A quick approximation to this volume is quite good enough (details below).

(1) Stage 1 computes the triangle's approximate shadowed volume as well as its planes.

(2) Stage 2 sorts triangles by their approximate shadowed volume.

(3) Stage 3 calculates hidden points and a subset of hidden triangles: triangles and points each have a "Hidden" mark. Without solving the full hidden line problem, it is safe to mark a triangle "Hidden" if it is hidden by another single triangle. The control structure of the nested loops is slightly more complex because of the extra break condition (a triangle is hidden). The same Occluded(Point,Triangle) function is still the inner-loop workhorse.

3.3. Hybrid -- Point against Sorted Triangles

The idea here is add TriTri()'s Stage 2 to PointTri(), to sort triangles by shadowed volume, again hoping the extra work pays for itself with an increased pruning factor.

4. Some Geometric Details

Points are represented by three integers (x,y,z), planes by four integers (A,B,C,D) from the plane equation $Ax+By+Cz+D$. For Stage 1, if **u** and **v** are "triangle edge" vectors (the difference between two vertex points) then **u**×**v** is a vector (A,B,C), giving three plane coordinates. The fourth coordinate is given by $D = -(x,y,z)\cdot(A,B,C)$. A,B,C,D need not be scaled to make (A,B,C) a unit vector for the purposes of this work, and integer arithmetic is sufficient to hold all significant digits. Further, for the edge plane calculations the origin is a vertex, so **u** and **v** are just triangle vertices and $D=0$.

For Stage 2, the triple product $V = \mathbf{x} \times \mathbf{y} \cdot \mathbf{z}$ gives a volume proportional to that enclosed between the origin and the triangle. The strange quantity e, simply the sum of all the nine x, y, and z components of the three vertex points, is taken approximately to vary monotonically with the distance from the origin to the centroid of the triangle. $(V/e^3)-V$ is the final approximation of trucated shadowed volume, up to some scaling constants. The cost of the whole approximation is 17 multiplies and 14 adds.

This approximation was compared with a much more elaborate one that projects the triangle onto the unit sphere, computes the area of the resulting triangle, computes the centroid exactly, and then computes the shadowed volume fairly precisely truncated by a sphere of radius 1.42. PointTri was modified to do a non-pruned triangle-point computation and to report how many points were occluded by each triangle. This information was used to establish the "correct" order for the triangles -- increasing number of occluded points. The sort by both the shadowed-volume criteria was quite successful and yielded a (surprisingly) good approximation to the "correct" sort. The availability of a relatively cheap and effective sorting criterion paved the way for a fair experimental investigation of the sort's utility, which was easier than a responsible theoretical analysis.

For Stage 3, the central visibility calculation for point x and triangle (A,B,C,D) is $\mathbf{d} = \mathbf{x} \cdot (A,B,C) + D$. If the d for any of the four planes is negative (with my sign conventions) the point is on the unshadowed side of the plane. Thus in the worst (point hidden) case there are three multiplies, three adds and a comparison for one plane (with nonzero D) and three multiplies, two adds, and a comparision for each of three planes (with D zero). Any negative result terminates the calculation with a "Not Hidden By This Triangle" result.

5. Early Experiments

Uniprocessor implementations of the three algorithms established that the pruning accomplished by TriTri() and the Hybrid PointTri() was not worth the effort. Sorting was done by the UNIX qsort() utility. With TriTri() in the worst case, three times the number of points must be checked as in PointTri(), and the number of triangles that are wholly hidden by other single triangles is not very large. The Hybrid algorithm produced times comparable with PointTri(), but up to 1300 points no clear dominance was established, so it appears that sorting just pays for itself in the Hybrid PointTri(). Of course a fast parallel sort could change the results on the Butterfly. The linear Stage 1 (setting up the geometry) is, as expected, extremely fast compared to the quadratic Stage 3. The pruning provided by quitting early in the Stage 3 of PointTry() yields about a factor of two in speed.

6. Initial Uniform System Implementations

The algorithm PointTri() lends itself naturally to a Uniform System implementation. The Uniform System gives parallel for-loop capability. The implementation simply parallelized the main loops in Stages 1 and 3. The

resulting code came to 450 lines for Stage 1 and 185 lines for Stage 3. It was run in several versions on the three Butterfly Parallel Processors at the University of Rochester. Representative code appears in the last section.

Version 1 scattered the (point, visibility) and triangle arrays as usual. Version 2 locally cached the row pointers to these arrays. Version 3 locally stored the point coordinates and cached the row pointers to the triangle and visibility arrays.

7. Times

Comparative timing shows that the VAX 750 is approximately 10 times as fast on this job as a single node in our (not floating-point platform) Butterfly computer.

1000 Triangles VAX and Butterfly (Version 1) Times	
Configuration	Time in Seconds
1 VAX 11/750	97
1 Bfly Node	1035
2 Bfly Nodes	520
4 Bfly Nodes	261
8 Bfly Nodes	131
16 Bfly Nodes	67
32 Bfly Nodes	35
64 Bfly Nodes	25

1000 Triangles on Butterfly (8 Nodes) Effect of Caching (Versions 1, 2, 3)	
Caching Version	Time in Seconds
8 Nodes, Version 1	131
8 Nodes, Version 2 (row ptrs)	79
8 Nodes, Version 3 (Vers. 2 + points)	67

8. Further Uniform System Implementations

Two revised versions of the PointTri() algorithm were implemented by Bukys with improved results. Some of the improvements are due to the release of the new Butterfly compiler; others are due to some tuning of the implementation.

The major difference between this implementation and the previous ones is the memory-sharing strategy. Since the algorithm uses a brute-force $O(n^2)$ strategy, each point-processing step may access every triangle data structure. These computations will clearly run fastest when every processor has its own local copy of the data structures describing triangle geometry. Such sharing is possible because the data strctures are computed only once and can be treated as read-only and static thereafter. Unfortunately, it takes time to replicate the data structures.

This program illustrates the resulting tradeoff dramatically: Replicating read-only data takes some time, but makes the computation run fast; sharing some data reduces replication overhead but increases computation time due to remote references and (perhaps) memory contention.

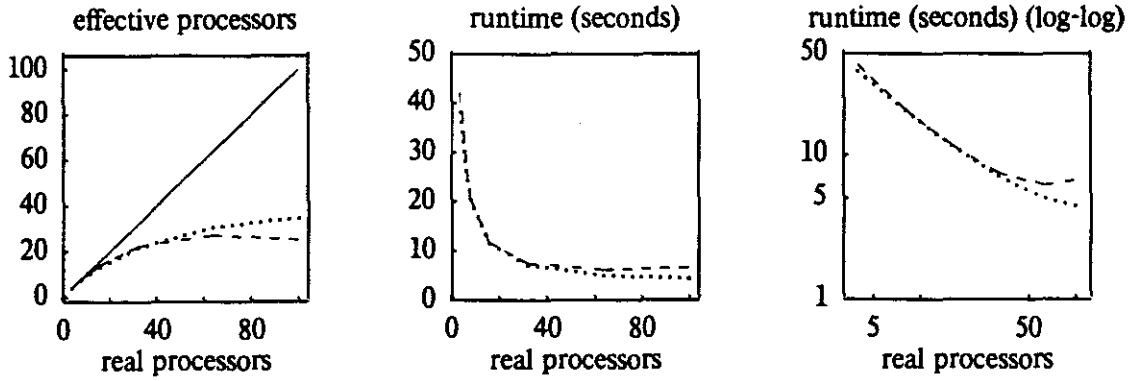
Further, the method of replication has a significant impact on runtime. The Uniform System implements two mechanisms for automatic replication: SharePtrAndBlk, which makes all copies from a single master, and ShareSM, which makes copies from previous copies, distributing the replication load among memories in a tree-like fashion with resulting logarithmic speedup. While the two procedures implement essentially the same function, their performance varies drastically. In the table below, compare the times in the rows "replicate triangle points" and "replicate planes" for the two implementations. Experiments have shown that the simple SharePtrAndBlk procedure works well for small pieces of data (under 2200 bytes), while the fancier ShareSM begins paying for itself for pieces of data larger than that. Unfortunately, the current Uniform System package provides the ShareSM procedure in a form suitable only for sharing row pointers of matrices. It would be a good idea to make both Share procedures use a data-size criterion for choosing replication method.

The following table breaks down the time spent in different phases of the computation for a 100-processor run of the algorithm. The final times were 6.5 seconds and 4.1 seconds, with the difference mainly accounted for by the different system calls implementing replication (shown in the "replicate planes" row. A constant 1.4 seconds is spent in generating the data (serially for replicability). The table illustrates that in the two computational steps (compute triangle parameters and detemine obscuration of points by triangles) typical speedups were almost linear (note the processor efficiencies of between 69% and 86% in the rows "make triangle and edge planes" and "visibility"), even with 100 processors running. However, the cost of replication is significant, and actually slows down the computation in the SharePtrAndBlk implementation for large numbers of processors. See the listing of times and graphs below. An obvious further tuning is to explore the tradeoff and find the amount of maximally efficient sharing.

Times for 100 Processors, 1000 Triangles

step	SharePtrAndBlk()		ShareSM()	
	effcy	*time(secs)*	*effcy*	*time(secs)*
initialize benchmark (100 procs)	-	4.186	-	4.200
allocate triangle pts	3.9%	.000	3.9%	.000
make 1000 random triangles	3.9%	1.473	4.0%	1.463
replicate triangle points	.1%	.669	1.1%	.124
allocate planes, replicate ptrs	1.1%	.026	1.0%	.030
make triangle & edge planes	74.1%	.035	68.7%	.033
replicate planes	.1%	2.368	.6%	.590
visibility: 256 points visible	86.4%	1.839	77.0%	1.852
FreeAll	2.3%	.048	.6%	.062
TOTAL (w/o initialization)	25.2%	6.458	34.6%	4.154

Speedup Graphs for Triangle Visibility



The graphs above were produced from the following raw data.

Raw Data for SharePtrAndBlk version:

[4] time = 676665 ticks = 42.29 sec; ep = 3.9; eff = 0.9999
[8] time = 332146 ticks = 20.75 sec; ep = 8.1; eff = 1.0186
[16] time = 188039 ticks = 11.75 sec; ep = 14.3; eff = .8996
[32] time = 120625 ticks = 7.53 sec; ep = 22.4; eff = .7012
[64] time = 99205 ticks = 6.20 sec; ep = 27.2; eff = .4263
[100] time = 107235 ticks = 6.70 sec; ep = 25.2; eff = .2524

Raw Data for ShareSM version:

[4] time = 610096 ticks = 38.13 sec; ep = 3.9; eff = 0.9999
[8] time = 324462 ticks = 20.27 sec; ep = 7.5; eff = .9401
[16] time = 184055 ticks = 11.50 sec; ep = 13.2; eff = .8286
[32] time = 113449 ticks = 7.09 sec; ep = 21.5; eff = .6722
[64] time = 79820 ticks = 4.98 sec; ep = 30.5; eff = .4777
[100] time = 70453 ticks = 4.40 sec; ep = 34.6; eff = .3463

9. A Pipeline Algorithm in LYNX

A systolic approach to solving the triangles problem was suggested by Peter Dibble and refined and implemented by Michael Scott. Triangles are fed into one end of a process pipeline. When they emerge at the other end, their vertices are marked "visible" or "hidden." In the simplest version of the algorithm, there are an equal number of triangles and processes. A special process feeds the head of the pipeline with triangles whose vertices are all marked "visible." An additional, marker triangle is fed through last. Before the pipeline begins operation, a preliminary phase of the algorithm precomputes, in parallel, the coefficients of plane equations that will be needed to determine if a point is obscured.

Each pipeline process keeps the first triangle that reaches it. It passes subsequent triangles on to its successor, marking as hidden any previously-visible vertices that are obscured by the original triangle it kept. When the marker triangle arrives, the process passes its personal triangle on to its successor, followed by the marker. Triangles emerging from the end of the pipeline have been compared against every other triangle.

An optimized version of the algorithm attempts to hasten comparisons that are most likely to find obscured points. In addition to computing plane equations, the initialization phase also computes the approximate volume of space shaded by each triangle. Each pipeline process compares the shaded volume of each newly-received triangle against the shaded volume of its personal triangle. If the new triangle is "bigger," it swaps them, keeping the new triangle and passing the old on to its successor.

The optimization is particularly important in practice, as there are many fewer processors than triangles. If each of the early stages of the pipeline is run on a different processor, and if each of the triangles in those early stages shadows a large volume of space, then the odds are good that relatively few obscuration tests will be needed in later stages of the pipeline.

Scott coded the pipeline algorithm in LYNX, a message-based language available on the Butterfly here at Rochester. The original version, with one process per triangle, does not accommodate large problem instances, because of operating-system limitations on the number of processes per processor. Scott then made a second implementation in which the pipeline, having exhausted processors, doubles back *through existing processes* as many times as necessary to provide one stage per triangle. If there are K processors, then processor I contains stages I, $2K-I+1$, $2K+I$, $4K-I+1$,

The multiple-stages-per-process implementation is significantly more complicated that the original version. It has uncovered a bug in the Chrysalis operating system which, in the limited time available to us, we have not yet been able to correct. For 200 triangles (the largest problem size that does not trigger the Chrysalis bug), the algorithm completes in about 15 seconds with a 100-processor pipeline.

10. Architectural Implications

Floating point processing (and hardware integer processing) is necessary. BBN currently provides an upgrade package (M68020-68881 daughter board) that we hope to acquire.

The Butterfly can present many abstract architectures to the user. For the Uniform System algorithm, a high-level and fairly superficial list of observations follows. In the US, memory allocation causes dramatic serialization: Parallel allocation would help. Carla Ellis and Tom Olson at Rochester are studying that problem. A geometric coprocessor or preprocessor for fast computation of trigonometric, roots, vector and matrix operations would be useful (the WARP comes to mind here). Better debugging support, from symbolic tools down to certain microcode enhancements would speed the development cycle. A combining switch would reduce memory contention, which may be a bottleneck in this computation.

The ability to share (copy) data quickly between processors would make a significant difference in this implementation, since in the final versions much data copying was done to increase the locality of computations on individual nodes. There is clearly a tradeoff in the current architecture between memory contention and the cost of massive data copying.

Serialization within the system is costly. Some of it can be avoided by the clever user, but some of it (such as memory allocation) should be parallelized by the system.

In the LYNX algorithm, we believe that the inefficiency of the pipeline is due primarily to the relative expense of sending a message (on the order of 2 or 3 milliseconds) compared to the amount of real work performed per message. To amortize the cost of message-passing, we would need in a practical implementation to pass more than one triangle in each message. Like the need to package more than one pipeline stage in a process, the need to package more than one triangle in a message complicates the implementation considerably, and suggests that the parallelism of the pipeline is too fine-grained for effective expression in LYNX. Unfortunately, there is no software package currently available on the Butterfly that supports orders of magnitude more processes than processors. The Uniform System does not count in this regard because its tasks cannot be suspended and are therefore able to synchronize only by busy-waiting.

We have begun to realize that a large and important class of problems can be solved by devoting a process to each of a very large number of objects. Many parallel algorithms in the current literature are of this flavor: the geometric problems in the benchmark provide more examples. To aid in mapping algorithms from the literature onto the Butterfly, a language akin to Mesa or Concurrent Euclid would be a very useful tool. Ada would also work, though probably not as well.

11. Stage 3, Version 3 Of PointTri() and the Uniform System

Besides making the inner loop of the computation explicit, this code segment illustrates several points. First, it shows that the Uniform System is easy to use: Both the US and SMP libraries give the new user very rapid startup. Second, it reveals that the Butterfly architecture is not actually a shared memory machine. There are several standard practices to reduce memory contention, the most common being local caching or copying of data. These practices acknowledge local memory. Below, local copies are made in the initializing routines GenericInit() and TInit(), and in the inner loop routine TriHidesPt(). Also the point array ShrPts[] has been copied to every node. Further, US has some hidden serializations: the storage allocator works sequentially, using global locks. The Allocate() call in ForAllTriangles() is natural but can (and should) be eliminated. Implicit in this example is that the ease of Butterfly programming and the flexibility of the architecture place a burden on the designer to come up with an efficient algorithm and data structures -- the architecture does not dictate them.

```
/*************STAGE 1 NOT SHOWN HERE ******************************/
/*************STAGE 3 -- CHECK  POINTS AGAINST TRIANGLES***********/

CheckPoints() /*outer parallel for loop -- for all points */
        {
        GenOnIndex(GenericInit, ForAllTriangles, p, 3*(p->N));
        }

ForAllTriangles(Arg, PointNdx) /* inner loop -- for all triangles */
```

```
        Problem *Arg;
        int PointNdx;
        {
        Problem *t;
        int i;

        t = (Problem *) Allocate(sizeof(Problem));
         /*this Allocate should be avoided: allocation is done serially */

        t-> N = myproblem.N;
        t->ThisPointNdx = PointNdx;
        t-> Vis = myproblem.Vis;
        t-> Tris = myproblem.Tris;
        t->ThisVis = -1;              /*create problem structure */

        GenOnIndex(TInit, TriHidesPt, t, t->N);  /*parallel for loop */
        }

TInit(Arg)    /* Standard practice. make local copies of
              global scattered data to avoid contention. In this case
              copy row pointers and problem structure */
Problem *Arg;
{
static int *vis[POINTS];
static int *tris[TRIANGLES];
block_copy(Arg, &MyTProb, sizeof(Problem));
block_copy(MyTProb.Tris, tris,(MyTProb.N)*sizeof( int *));
block_copy(MyTProb.Vis, vis,(MyTProb.N)*3*sizeof( int *));
MyTProb.Tris = tris;
MyTProb.Vis = vis;
}


TriHidesPt(Arg, TriNdx)  /* inner loop computation: does triangle hide pt? */
        Problem *Arg;
        int TriNdx;

        {
        int offset, MyX, MyY, MyZ, PlaneNdx;

        if( MyTProb.ThisVis == 0) return; /*is point already invisible? */

        offset = (MyTProb.ThisPointNdx)*POINTCOLS;
        MyX = ShrPts[offset];
        MyY = ShrPts[offset+Y];
```

```
        MyZ = ShrPts[offset+Z]; /*get point x, y, z */


        block_copy(MyTProb.Tris[TriNdx],MyTriangle,(TRICOLS)*sizeof(int));
                /* make local copy of scattered data */

        if( (MyX * Coord(TRIPLANE,A) /* dotproduct with plane of triangle */
                + MyY * Coord(TRIPLANE,B)
                + MyZ * Coord(TRIPLANE,C)
                + Coord(TRIPLANE,D)) <= 0)
                return;         /*not hidden -- quit */

        for (PlaneNdx = EDPLANE1; PlaneNdx <= EDPLANE3; PlaneNdx++)
                /*dot with 3 planes of edges */
                {
                if( (MyX * Coord(PlaneNdx,A)
                + MyY * Coord(PlaneNdx,B)
                + MyZ * Coord(PlaneNdx,C)
                <= 0) )
                return;                 /*quit early if not hidden*/
                }
        /*point hidden if get to here*/
        MyTProb.ThisVis = 0;
        MyTProb.Vis[MyTProb.ThisPointNdx][0] = 0;
                /*set local and global visibility */

        } /* end TriHidesPts */

/*********************MAIN ROUTINES ****************/

FullJob()
        {
        MakeTriangles(); /*do Stage 1 */
        CheckPoints();  /* do Stage 3 -- see above */
        FreeAll();      /* clean up */
        }

main()
        {
        InitializeUs(); /*Uniform System Initialize*/
        MakeShrPts();           /* generate random 3-D triangle vertices */
        TimeTest(SetUp, FullJob, MyTestPrint);
         /*run algorithm, get times on different numbers of processors */
        }
```

Chapter Eight: Graph Matching

see: Butterfly Project Report 14

Chapter Nine: Minimum-Cost Path

Minimum-Cost Path

Brian D. Marsh and Thomas J. LeBlanc
August 1986

1. Introduction

We describe an implementation of the *minimum-cost path problem* on the BBN Butterfly using the SMP message-passing library developed at the University of Rochester. The problem statement for finding the minimum-cost path is as follows:

> *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 real-valued 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.)*

Given this problem statement, it is ambiguous as to whether we are required to solve the *all-pairs-shortest-path problem*, which then allows the user to query the result regarding individual pairs of nodes, or whether we are to solve the *single-source-shortest-path problem* for a particular pair of nodes. Given that dynamic programming was specifically mentioned in the problem statement and is normally used to solve the *all-pairs-shortest-path problem*, we felt constrained to implement that problem, despite the fact that we believe the *single-source-shortest-path problem* has a more interesting parallel solution and would better exhibit the flexibility of the BBN Butterfly architecture. In the following sections we describe our parallelization of Floyd's algorithm for the *all-pairs-shortest-path problem*, an implementation of the algorithm on the Butterfly using the SMP message-passing library package, and our performance results.

2. A Parallelization of Floyd's Algorithm

We chose to implement a parallel version of Floyd's dynamic programming solution to the *all-pairs-shortest-path problem* [1]. The input graph is represented by an adjacency matrix. An entry, [i,j], corresponds to the weight of the edge from vertex i to vertex j. Nonexistent edges are denoted by a symbol representing infinite distance.

During execution each entry of the matrix corresponds to the cost of the minimum-cost path between two vertices. Initially, only those vertices that share an edge have a path of finite cost. Floyd's algorithm iterates over each row of the matrix, finding successively lower cost paths. During the k'th iteration, the algorithm computes the cost of the minimum-cost path between all pairs of nodes, i and j, that pass through no vertex numbered greater than k. For a graph with N vertices, N iterations are necessary. Therefore, the algorithm is $O(N^3)$. The code for the algorithm is as follows:

```
for k := 1 to N do
  for i := 1 to N do
    for j := 1 to N do
      if A[i, k] + A[k, j] < A[i, j] then
        A[i, j] := A[i, k] + A[k, j]
      end if
    end for
  end for
end for
```

An obvious parallelization of this algorithm results from treating each **for** loop as a parallel **for** loop. However, the granularity of the innermost loop is not large enough to justify the overhead of process allocation in the Butterfly. For this reason we chose to use the processing of an entire row as the unit of granularity for parallelism. We divided the problem matrix uniformly among the available processors, so that each processor has some subset of rows in the matrix. Since the size of the input graph is defined to be on the order of 1000 vertices, each processor must iterate over approximately 10 rows. The code for each process is:

```
for k := 1 to N do
  if row k is local then
    broadcast row k
  else
    receive row k
  end if
  for each local row i do
    for j := 1 to N do
      if A[i, k] + A[k, j] < A[i, j] then
        A[i, j] := A[i, k] + A[k, j]
      end if
    end for
  end for
end for
```

The primary data dependency in this algorithm is that all processes need a specific row at the same time, a row whose values are dependent on past computation. This synchronization constraint forces the processes in the algorithm to run in lockstep. On the k'th iteration, each process computes the optimal paths for its local rows using the values stored in row k. Computation cannot proceed until these values are known. The implementation, therefore, must have an efficient broadcast mechanism. For this reason, among others, we chose to implement the algorithm using the SMP library package.

3. An SMP Implementation of Floyd's Algorithm

An implementation of the *all-pairs-shortest-path problem* was done in C using the SMP library package developed at the University of Rochester [3]. SMP is a message-based programming environment for the Butterfly. Processes are dynamically created within SMP *families*. Interprocess communication within a family is based on asynchronous message-passing (send/receive) according to a fixed communication topology. When using SMP the programmer sees a small set of procedure calls for creating processes, specifying interconnection topologies, and sending messages. The details of the Chrysalis operating system needed to implement processes and communication are hidden. The programmer is free to concentrate on the issues pertaining to the application, rather than the underlying primitives.

There were several reasons for choosing SMP for this application. The most important reason is that our experience with a similar application [4] had shown that exploiting data locality could lead to significant performance advantages when compared with the shared memory approach of the Uniform System [2]. That is, storing a subset of the rows in memory local to the process that will modify those rows and exchanging rows in messages requires less communication than storing the rows in a globally shared memory. Another reason for using SMP is that broadcast communication, which is used in our algorithm, is directly supported. Finally, we were able to use this application to gain additional experience with SMP.

Our parallel version of Floyd's algorithm does not make full use of the tree of dynamic process structures available in SMP. In our implementation, a single parent process is responsible for creating a child process on each processor. Each child process is given some subset of the rows in the initial adjacency matrix. On the k'th iteration, each child process receives a message containing row k and computes new values for its local rows. The process containing row $k+1$ then broadcasts that row to all its siblings to start the next iteration.

The *send* primitive of SMP accepts a list of destination processes, therefore, both broadcast and multicast can be done directly in SMP. The SMP implementation of *send* is such that the cost of sending to one sibling (or to the parent) is the same as sending to 100 siblings. In each case, the message is copied to a buffer local to the sending process and flags are set indicating the intended recipients. Using the SMP *receive* primitive, destination processes can inspect the shared buffer to determine if there is a message directed to them. If so, the message is copied into the local memory of the receiving process.

One of the problems with broadcasting in SMP is that the Butterfly provides no hardware support for simultaneous communication with multiple destinations. In SMP each potential recipient of a message must map the message buffer into its local address

space to check for a message. Since each process in our algorithm is expecting to receive rows from every other process, the source list of each *receive* operation is very long. All the processes listed in the source list will have their message buffers mapped into the local address space during each iteration. This turns out to be extremely time consuming when the list is very long and, in an early implementation of our algorithm, was a dominating factor. Fortunately, we were able to exploit the inherent synchronization in our algorithm to reduce the overhead of broadcasting by minimizing the number of buffers examined on each iteration.

On each iteration, every process expects to receive a particular row. Despite the fact that rows are broadcast, the source for each row is known. Hence, in our implementation, we invoke the *receive* operation on the k'th iteration with a source list of size 1, namely, the process containing row k. This way, only one message buffer is mapped into the local address space on each iteration. We were able to improve performance by 50% using this approach. The performance of the resulting implementation is summarized in the next section.

4. Performance Results

The program to solve the *all-pairs-shortest-path problem* was developed on a host Vax 11/750 and downloaded to the Butterfly for execution. A sequential version was also implemented on a SUN workstation for comparison purposes. Coding and debugging the application program required about one week of effort by a graduate student; some additional time was spent debugging the SMP library.

For the purposes of the benchmark experiments, random graphs of various sizes were generated. We performed detailed experiments using two graphs: G1, a random graph containing 100 vertices with 10 edges per vertex, and G2, a random graph containing 300 vertices with 30 edges per vertex. We did not perform any experiments with the graph size given in the problem statement, 1000 vertices with 100 edges per vertex, for two reasons:

a) In order to demonstrate how well our implementation scales to multiple processors, we needed to run the algorithm with a varying number of processors and compare it to the single processor case. G2 requires 33 minutes of execution time on a single processor. By running significantly larger problems, we would be consuming a limited resource (Butterfly availability) and learn very little in return.

b) The cost matrix for a graph with 1000 vertices requires 4MB. While our Butterfly does have 1MB on each node, Chrysalis does not have good tools for creating and manipulating large objects that span multiple processors. The extra programming effort necessary to run such a large problem was not warranted.

In each of our test runs, only 100 processors were used, even though 120 processors were available. We did this so that all of our graphs would be uniformly distributed among the available processors. In this way, we eliminated the "tail end" effects that might otherwise distort our measurements.

Our performance results for finding the *all-pairs-shortest-path* solution for G1 and G2 on the Butterfly are shown in Figures 1-4. We have not included the initialization overhead in the results; **only actual computation time was measured**. The parent process in the SMP family was responsible for maintaining the timing results. All children synchronize with the parent, the clock is initialized, and all processes then begin computing. The results show the elapsed time between clock initialization and the final response from child processes.

These same graphs were also run on a SUN 2/50 workstation with 4MB of memory and a Vax 11/750 with 2MB of memory. G1 took 44.5 seconds on the SUN, 158 seconds on the Vax, and 69 seconds on a single Butterfly processor. G2 took 1205 seconds on the SUN, 2787 seconds on the Vax, and 1907 seconds on a single Butterfly processor. As can be seen in Figure 1, a small graph of 100 vertices can efficiently use 25 processors on the Butterfly (19 *effective processors*); additional processors do not provide much improvement in performance. The larger graph, G2, can make use of all 100 processors. In either case, only 2 Butterfly nodes are needed to significantly improve upon the sequential version on both the SUN and Vax.

5. Conclusions

To summarize the results of our experience with the *all-pairs-shortest-path problem*: a parallel version of Floyd's algorithm was easily implemented using SMP on the Butterfly and the resulting performance demonstrated nearly linear speedup using up to 100 processors. What follows are some comments about the choice of algorithm, software, and architecture.

The dynamic programming approach to the *all-pairs-shortest-path problem* is ideally suited to a vector machine; the Butterfly Parallel Processor has no special capability in this regard. Nevertheless, we felt this would be the easiest solution to implement in the limited time available. The fact that we were able to implement a solution to this problem on the Butterfly in a short period of time, a solution that demonstrated nearly linear speedup over the sequential version for large graphs, gives some measure of the flexibility of the Butterfly architecture. It would have been interesting to compare our experiences on this problem with similar experiences on the *single-source-shortest-path problem*, a similar problem with a more interesting parallel solution. Time did not permit this comparison.

Our experiences with the SMP system were very positive. A new graduate student was able to implement Floyd's algorithm in about one week of effort. The SMP library

dramatically reduces the learning curve for the Butterfly. However, the SMP library was only recently released, and we did encounter a few system bugs. All of the bugs were repaired in the same week. This effort did point out the need for some optimizations when handling source and destination lists in an SMP broadcast. We expect this will lead to slight modifications in the way SMP treats such lists. We also plan to add some additional routines that help in performing timing tests.

Our biggest problems with the Butterfly architecture continue to be related to memory management, in particular, the lack of segment attribute registers (SARs). SAR management was the source of most of the SMP bugs and is also the main difficulty in manipulating large objects. However, as we have gained more experience with the Butterfly, we have accumulated tools and techniques for solving most of the problems associated with SAR management. (For example, SMP incorporates a SAR cache for message buffers.) We expect that continued experimentation will yield additional solutions.

References

1. A. Aho, J. E. Hopcroft and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley Publishing Company, 1983.

2. BBN Laboratories, The Uniform System Approach To Programming the Butterfly Parallel Processor, Version 1, Oct 1985.

3. T. J. LeBlanc, N. M. Gafter and T. Ohkami, SMP: A Message-Based Programming Environment for the BBN Butterfly, Butterfly Project Report 8, Computer Science Department, University of Rochester, July 1986.

4. T. J. LeBlanc, Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study, *Proceedings of 1986 International Conference on Parallel Processing*, (to appear) August 1986.

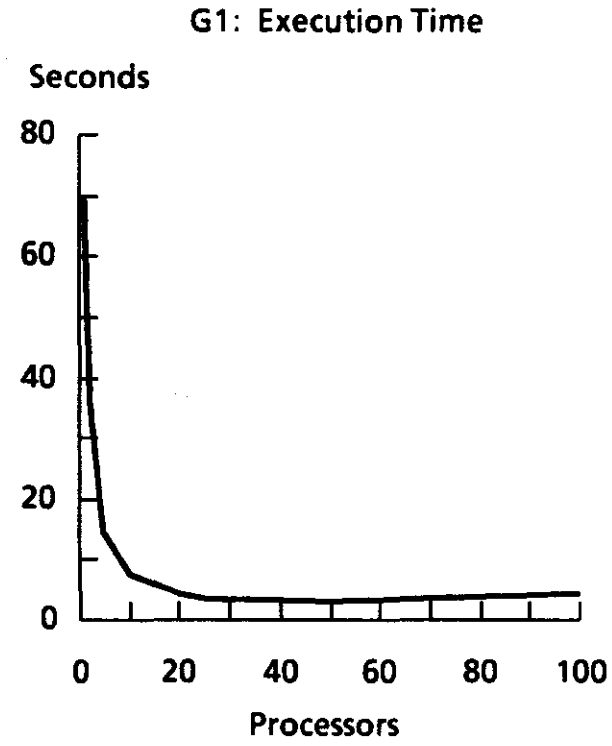


Figure 1

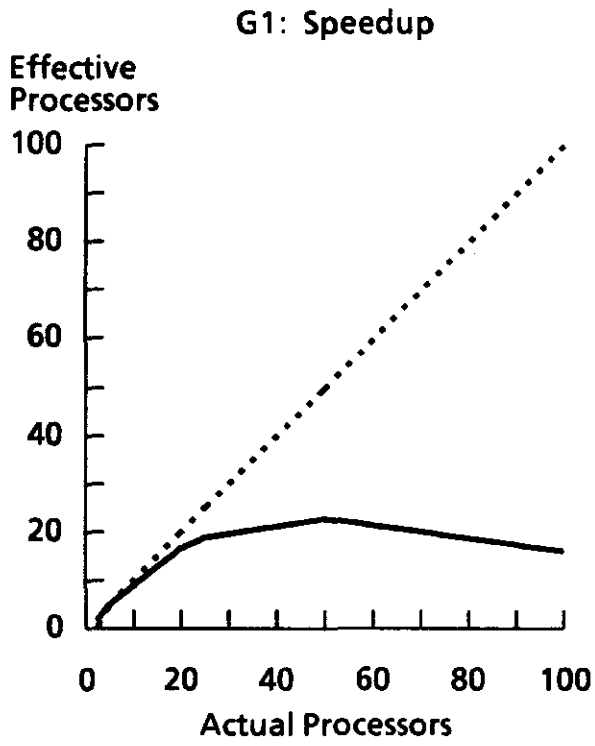


Figure 2

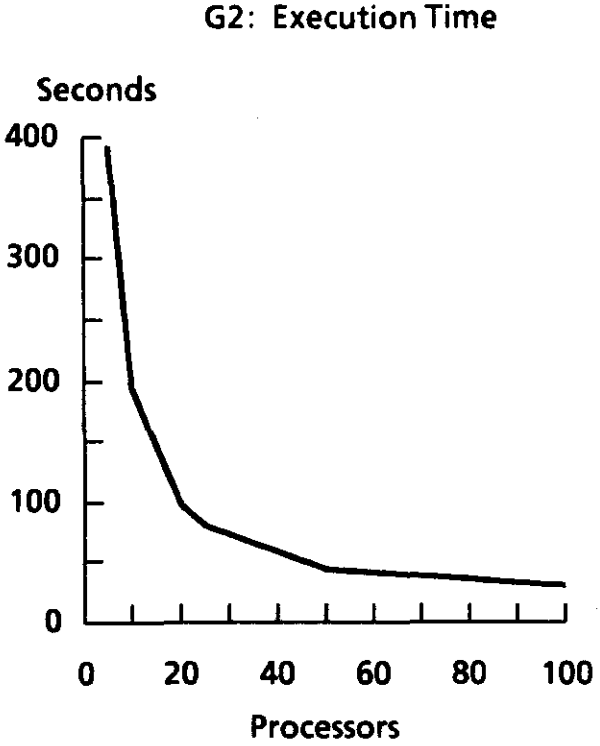


Figure 3

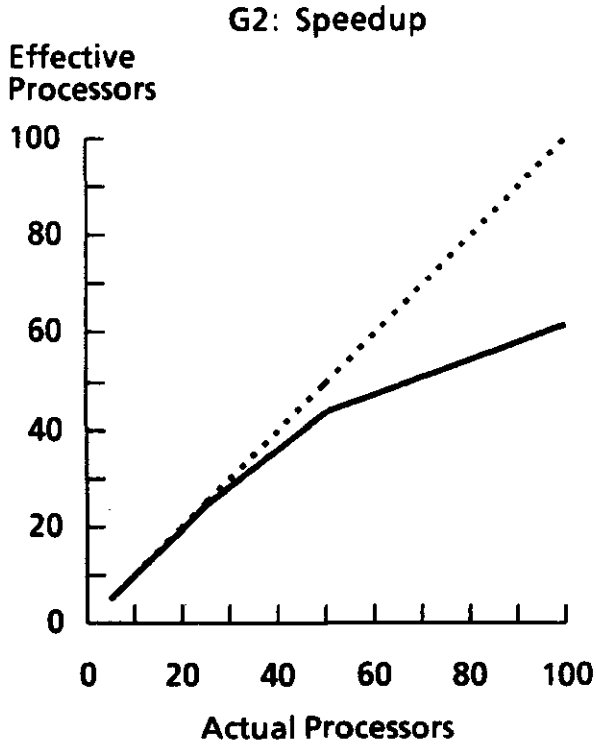


Figure 4