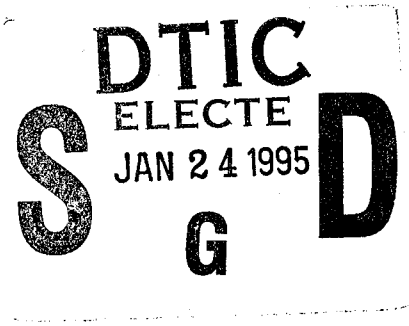
ARMY RESEARCH LABORATORY

Exploiting Parallelism in a Monte Carlo Image-Matching Algorithm

Dale Shires

ARL-TR-667 January 1995



NOTICES

Destroy this report when it is no longer needed. DO NOT return it to the originator.

Additional copies of this report may be obtained from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161.

The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The use of trade names or manufacturers' names in this report does not constitute endorsement of any commercial product.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 1995	Final, March 1994–June 1994

4. TITLE AND SUBTITLE

Exploiting Parallelism in a Monte Carlo Image-Matching Algorithm

5. FUNDING NUMBERS

PR: 1L162618AH80

6. AUTHOR(S)

Dale Shires

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

U.S. Army Research Laboratory
ATTN: AMSRL-CI-S
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Army Research Laboratory
ATTN: AMSRL-OP-AP-L
Aberdeen Proving Ground, MD 21005-5066

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

ARL-TR-667

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Recovering three-dimensional information from a visual scene requires producing the disparity map of matched stereo images. The matching approach discussed in this paper involves the Metropolis algorithm, a simulated annealing technique. Although simple in principle, this stochastic optimization technique requires immense computational resources. Adapting the algorithm for parallel computing is discussed and demonstrated for several computer architectures.

14. SUBJECT TERMS

stereoscopic range finding; parallel processing; Monte Carlo method

15. NUMBER OF PAGES
53

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

INTENTIONALLY LEFT BLANK.

Contents

1 Introduction **1**

2 Image Matching **4**

3 Parallelism **8**

4 Implementation **10**
4.1 Networked Sun IPCs and C-Linda 10
4.2 Sun 2000 with Solaris Threads 11

5 Measurements **12**

6 Analysis and Interpretation **15**

7 Conclusions **21**

A Linear Stereo Matching Algorithm **24**

B C–Linda Parallel Stereo Matching Algorithm **33**

C .Solaris Threads Stereo Matching Algorithm **41**

INTENTIONALLY LEFT BLANK.

List of Figures

1 Stereo imaging camera system. 2
2 Disparity indicates distance in the scene. 3
3 Simulated annealing algorithm in pseudo-code 5
4 Three-dimensional wedding cake structure. 12
5 Random dot stereogram. The left image is formed by displacing points in the right image. 13
6 Stereo matching results. (Disparity maps are encoded as gray scale values. The left image was generated sequentially, the right image in parallel.) 15
7 Three-dimensional representation of a four-level random dot stereogram. 16
8 Sequential time on Sun IPCs and Sun 2000. 16
9 C-Linda performance on Sun IPC network. 17
10 Threads performance on Sun 2000. 17
11 Speedup achieved using C-Linda on Sun IPC network. 18
12 Speedup using C-Linda plotted against ideal linear speedup. . 18
13 Efficiency of Sun IPC network. 19
14 Speedup achieved using threads on Sun 2000. 20
15 Speedup using threads plotted against ideal linear speedup. . . 20
16 Efficiency of threads on Sun 2000. 21

INTENTIONALLY LEFT BLANK.

1 Introduction

The eye-brain system achieves three-dimensional depth perception by taking advantage of two separate and distinct images captured by each eye. The image of an object projects on the eye's retina. The image of the left eye will differ from that of the right. This is called *retinal disparity* [1]. The brain fuses the two images and interprets the disparity into distances of the object from the eyes. These binocular cues produce perception of depth not available with monocular vision. The speed with which the human brain can integrate the continuous sensor information streaming from the optic nerve dwarfs the capabilities of current computing hardware.

Computational stereo describes the process of synthesizing the mechanics of the binocular vision. Stereo pairs obtained from digital imaging are compared to extract three-dimensional characteristics of a photographed scene. The mathematics involve only trignometry; however, the number of transformations for high resolution images becomes staggering. For this technique to receive more attention, the solution algorithms must execute quickly and efficiently. Parallel computing offers the greatest hope of mimicking the feats of the brain.

Basically, a stereo camera pair is used to take pictures of a scene where range information is required. Because of binocular parallax, these cameras (referred to as the left and right cameras) will acquire slightly different images of the scene since they are at different locations. This effect can be seen in Figure 1. The camera origins are aligned along the y and z axes and are only displayed along the x axis. I_L and I_R are the left and right images of the stereo camera pair. The point P is in the three-dimensional scene and is projected onto the left and right camera photosensitive plates at P_L and P_R, respectively. F_L and F_R represent the focal points of the camera systems. The disparity is the difference in the locations of P_L and P_R on the photosensitive plates, which results from the cameras being in different horizontal locations. This disparity and the projection lines are used to determine P's position in the world.

For example, a point very distant from the cameras will appear to be at the same vertical and horizontal position on monitors connected to the left and right cameras. However, a point close to the camera pair will not be in the same position on the two monitors. Instead, the point on the left monitor will be displaced (disparity) to the right from the point on the right monitor.

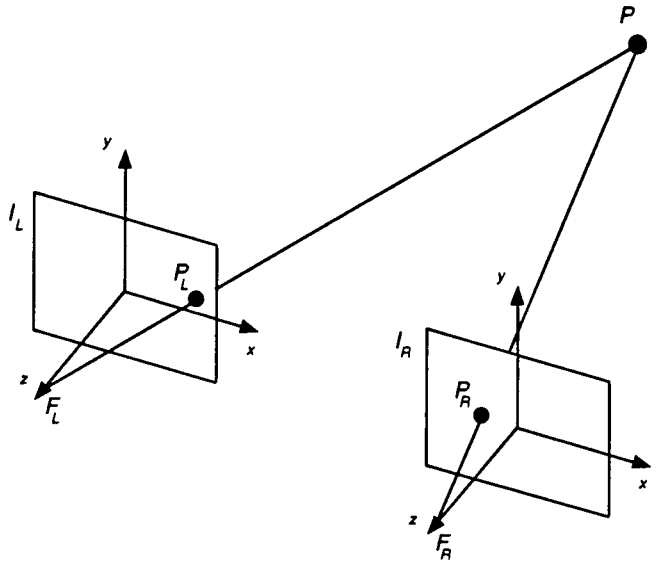


Figure 1: Stereo imaging camera system.

One can see this phenomenon in Figure 2. Notice that points distant in the picture are at roughly the same location in both left and right images. However, points close to the camera, such as the white spot on the highway, have a much greater disparity in the two images.

Once the matching points are found in an image, an inverse perspective transform or simple triangulation can be used to derive the two lines where the projection of the world point strikes the photosensitive plate of the camera. When these lines are intersected, the three-dimensional characteristics of the scene can be recovered [2].

The ability to properly match stereo camera pair images is important to any application in which distance or range information must be extracted from an image. One such application is plotting terrain contours from images shot by camera pairs mounted on helicopters. Once the images are matched, a contour plot is created of the terrain, which shows elevations and depressions by computing how far away the ground points are from the cameras. The computer vision and robotics fields use this technique to allow machines, both moving and stationary, to compute information about their environments [3]. This technique should also find favor with the military since it does not

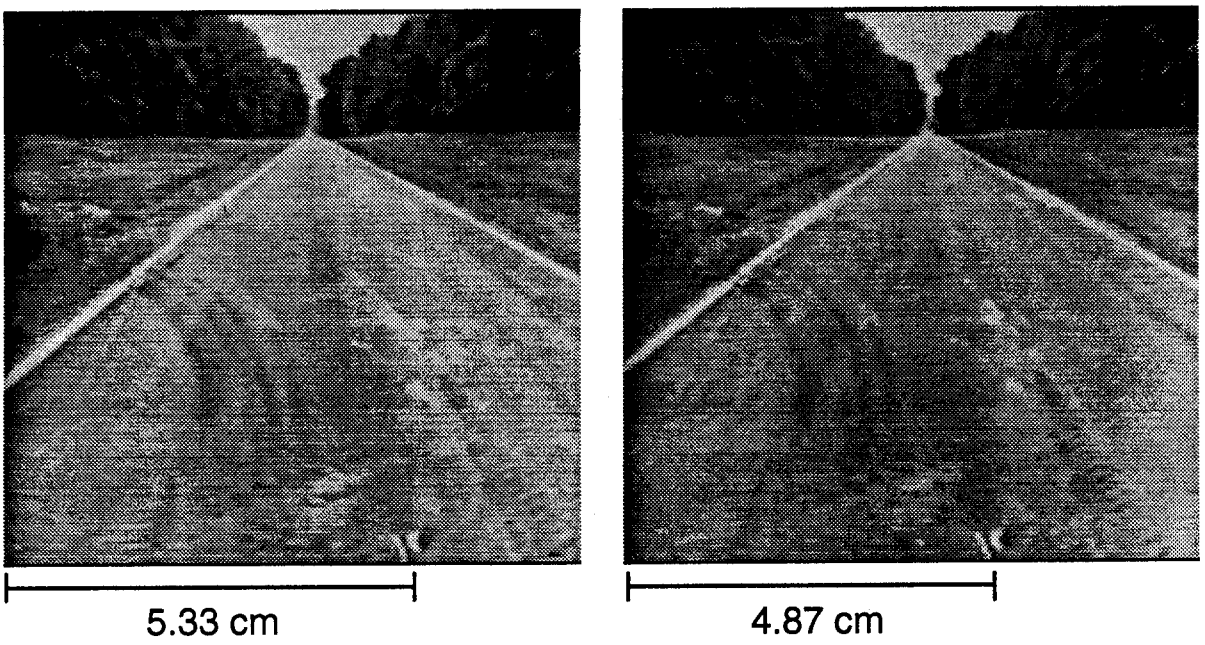


Figure 2: Disparity indicates distance in the scene.

employ active sensing devices. Active devices, such as radar and lasers, are easily detected and hinder stealth operations.

The algorithm described in this paper is stochastic. It is an undirected Monte Carlo search through the image space which produces a very fine, globally optimized disparity map where every pixel in one image is matched with its corresponding pixel in the stereo pair. Just as with other Monte Carlo algorithms, this approach requires a significant number of floating point operations. However, the process of matching pixels typically requires only local interactions. On the computer, this translates into local references to memory. Furthermore, the amount of processing for each pixel remains uniform. These two properties, locality of memory reference and uniform computational loading, make the algorithm appear ideal for parallel processing. A perfectly parallel algorithm should exhibit a linear speedup as a function of the number of processors employed. This simple measure will serve as

a benchmark for measuring the level of parallelization in the application of stereo imaging.

The paper has two goals. The first is to present a method for performing stereo image matching and describe how it was modified to exploit parallelization. The second goal is to discuss the timing behavior of the algorithm in sequential versus parallel modes to include a comparison of different computer architectures. The sequential and parallel versions of the algorithm along with actual timing results are discussed in more detail in later sections.

2 Image Matching

Stereo matching requires global optimization. Since the digital image data maps pixel intensities to a relatively low resolution (typically eight bits, implying 256 discrete levels), there are many possible matches in the local sense. That is to say, swatches of one image may appear to map other portions of the stereo pair. To perform stereoscopic ranging, the whole image must be taken into account. Hence, the requirement for global optimization.

The most popular optimization technique to locate a global optimum is called *simulated annealing* [4]. As the name implies, the approach imitates a natural process. Annealing involves heating a solid to the extent that the molecules may randomly rearrange themselves and then cool gradually. Slowly lowering the temperature allows the molecules to settle into the lowest energy state, commonly described as thermal equilibrium. If the temperature rate declines too fast, defects may become frozen into the end state. If thermal equilibrium is maintained throughout the cooling cycle, the final system should be a globally optimized structure. For example, perfect crystals are grown in this manner.

Basically, the simulated annealing algorithm mimics the physical process via an undirected Monte Carlo search through the image space. This procedure, known as the Metropolis algorithm, samples states in a system at equilibrium. Since the system is kept in thermal equilibrium, its states have a Boltzman distribution

$$P(E) = \exp\left[\frac{-E}{T}\right] \tag{1}$$

in which E is energy, T is the temperature of the system, and $P(E)$ is the probability of a state having energy E. Random, local state transitions are

```
read I_R, I_L
D(row, col) = random number in [0 ... D_max]
T = T_max
/* loop according to fixed annealing schedule */
while T ≥ T_min
        iterate a fix number of times
            S' ⇐ random state change S
            ΔE = E(S') − E(S)
            /* accept lower energy states */
            if ΔE < 0 then S = S'
            else
                P = exp [−ΔE/T]
                j = random number in [0 ... 1]
                /* accept higher energy only with Boltzman probability */
                if j < P then S = S'
        reduce T by predefined percentage
end while
```

Figure 3: Simulated annealing algorithm in pseudo-code.

performed by varying the disparity only slightly. The change in energy that would result from the new disparity is determined. If the new state takes the system to a lower energy level, it is accepted. If the new state takes the system to a higher energy level, the new state is accepted only with probability $P = \exp[\frac{-\Delta E}{T}]$.

The simulated annealing technique is outlined in Figure 3. The system is taken to equilibrium by the Metropolis algorithm by considering random, local state transitions on the basis of the change in energy that they imply. Since the system is stochastic, these local state changes can take the system away from convergence as well as toward it. This helps to prevent the system from sinking into local minima. The processing is complete when the system is in equilibrium at the lowest energy state achievable.

The rate of temperature reduction during annealing is determined based upon the changes in energy that occur. In the original algorithm, energy distributions are constantly monitored to determine when equilibrium is reached

and temperature should be lowered. Enough iterations of the algorithm should be performed at each temperature to bring the system to thermal equilibrium. Temperature should only be lowered once there is no longer any significant decrease in energy. In this version, a fixed annealing schedule is used to limit global accumulators and synchronization that would disrupt parallelization. Fixed annealing schedules have been shown to be effective for these problems. Furthermore, global accumulators are not required since optimizing local energies has the same overall effect as optimizing global energy without loss of correctness [5].

Several parameters must be set when using simulated annealing with fixed cooling schedules. The first value that must be determined is the starting temperature. The initial value of T must be chosen in such a way that virtually all transitions are accepted. This means that $\exp\left[\frac{-\Delta E}{T}\right] \simeq 1.$ for all points on the lattice. A general procedure for picking the initial value of T is as follows. Start with a high value and perform a number of iterations. If the acceptance ratio, which is the number of transitions accepted divided by the number of transitions proposed, is less than a certain cutoff value, double the value of T_0. An acceptance ratio of 0.8 is frequently used [4].

Determining the appropriate number of state transitions needed at each temperature represents an important consideration for accurate simulations. This value is highly dependent on the size of the problem space. For example, stereo matching an image for which the maximum disparity is 20 will require more iterations than stereo matching an image that has a maximum disparity of 10. Several techniques have been proposed to help determine this number. One widely employed rule is that the average number of iterations of the algorithm for each temperature should be roughly equal to that of the number of variables of the problem being solved [4]. More experimental results may be used by monitoring the number of iterations required to bring the system to thermal equilibrium for the given temperature [3].

Another concern involves decreasing the control variable T. Only small decrements to T should be allowed to make sure the system can re-adjust to equilibrium based on the number of iterations that are to be employed for each value of T. A frequently used rule is

$$T_{k+1} = \alpha \cdot T_k, \ k = 0, 1, 2 \tag{2}$$

in which α is a constant smaller than but close to 1 [4].

The last major concern is when to terminate the algorithm. The algorithm may be terminated when the state space of the problem no longer fluctuates. Such a point usually arrives when the value of T is sufficiently low and $\exp\left[\frac{-\Delta E}{T}\right] \simeq 0$.

For this implementation, a similar cooling schedule as described by Barnard was used [5]. Temperature is initially 100 which easily allowed for transition occurrences of greater than 80%. Ten passes through the problem space were performed for each temperature. This value is very close to the maximum disparity of 8 for each test case. Temperature was decreased by 10% after the 10 loop passes. This uses the decrement rule with an α value of 0.9. The algorithm is terminated when T drops below 1.

Annealing may be simulated in any problem requiring a globally optimized solution. The process is most useful in problems that are very complex, have high degrees of interaction between the elements, and do not have absolute solution spaces. Stereo matching fits all these criteria. Since every point must be matched, the process must match anywhere from roughly 16,000 to 260,000 points (image sizes usually range from 128×128 to 512×512). Matching one point requires analyzing how its neighbor point matched, and how that neighbor's neighbor matched, and so forth. Furthermore, there are often areas of occlusion or wide areas of homogeneous intensity in the images that do not allow for precise matching. As one can see, the problem quickly becomes one of optimizing over a broad spectrum of possibilities.

To use simulated annealing, one must model the problem as an analog to an actual physical system. This will enable the algorithm to determine when one state is closer to being optimized than another state. A function is constructed which analyzes the current state of the process and assigns a scalar value to it. This function is known as the *energy function* for the system. Simulated annealing attempts to find a global state that has a minimum energy.

The function used in this case has two sources of energy contribution [6]. For the image-matching problem, the associated energy at each pixel is

$$E(r,c) = \mid I_L(r,c) - I_R(r,c+D(r,c)) \mid + \lambda \mid \nabla D(r,c) \mid \qquad (3)$$

in which I_L and I_R represent the brightness at row r and column c of the left and right images, respectively; D is the disparity; ∇D is the sum of the absolute differences between D and its eight nearest neighbors; and λ is a weighting factor.

The first term in the equation is known as the brightness constraint. It basically states that matching points, or pixels, on the left and right images should be of approximately the same intensity value. For example, if the left and right images are digitized into eight bit gray scale brightnesses, a pixel on the left image with a brightness value 68 should be matched to a pixel on the right image with approximately the same value. Identical cameras and settings should be used to limit variations in brightness, contrast, and so forth.

The second term is sometimes referred to as the smoothness constraint. It asserts that the horizontal shift of an element should be the same as that of its neighbors. This smoothness condition is necessary since the first contraint is strictly local, and stereo correspondences are locally ambiguous. In other words, without this constraint, surfaces would not be spatially coherent.

For example, suppose the left and right images are composed of solid black backgrounds with a white square 50×50 pixels roughly in the center of the image. The square in the left image has a disparity value of 5 (all points in the square have been shifted 5 places to the right). If only the brightness constraint were used, a point directly inside the left edge of the square could have a disparity value of anywhere from 0 to $\simeq 50$ since its brightness matches all other pixels in the square. However, because of the smoothness constraint, the left and right edge disparity value of 5 will be propagated to the rest of the interior of the square.

3 Parallelism

It is the rule, rather than the exception, that algorithms have some aspect that can benefit from parallelism and thus decrease overall computation time. Because only local interactions are considered between points on the image lattices, the Metropolis algorithm can benefit greatly from parallelism.

As one searches for the most efficient way to exploit parallelism, a key point to keep in mind is the structure of the data in the problem. This algorithm's fundamental parallelism is one of a result-based nature. Ideally, one processor for each point on the disparity map would be active. Each processor would be responsible for computing its assigned pixel's disparity and that disparity alone. This is a dynamic, 2-D data structure; trapped inside each array element is a process that computes $D(r, c)$ for each pixel.

At the conclusion of the algorithm, the processes would vanish and leave a resultant disparity for the (r, c) in question.

Result-based computations perform very well on finely-grained architectures. These are computers with many processors that are governed by one master processor. An example of this type of architecture is available on the CM-2 computer, a connection machine built by Thinking Machines, Inc., having 64,936 simple one bit processors [7]. Each processor in this model would run the same algorithm but on differing data sets. The processors would be assigned a group of pixels on which to work. All processors would perform in lock-step governed by a master process. Since the disparity map is only updated after all points have been analyzed, each processor can work independently of its neighbor. However, finely grained architectures are not very common and good speedup can still be achieved by switching to an agenda-based parallelism using a master-worker program [8].

Agenda parallelism is very suitable for the more common coarsely-grained architectures. These are architectures that typically use shared memory and usually have fewer processors than fine grain machines. A master-worker type of agenda parallelism is used in this instance. In this computational stereo algorithm, a master process starts a series of workers to perform the image-matching task. Basically, each worker executes the code described by the text and flowchart in the previous section of this paper. All workers execute the same code; the only difference is which part of the image the workers get. This is also often referred to as SIMD (single instruction, multiple data) computing. Each processor runs the same program in this model. Each processor, however, works only on its own data set.

It was initially planned to split the $N \times N$ image into blocks of size $m \times m$ where $m = (N/numberworkers)$. For example, if the image were of size 128×128 and we wanted to use four workers, each worker would get a block of the image of size 64×64. This would have created overhead, since each worker would have had to communicate disparity information along its edge to its neighboring workers. Instead, since there is almost no vertical disparity in the image, the workers each get a block of the image of size $m \times N$. With the 128×128 and four workers example, each worker would get a block of size 32×128. Each worker can now compute without synchronization problems since vertical disparity does not need to be broadcast and received. Since the array blocks are homogeneous, a static scheduling method is used. Each processor gets one contiguous block of the overall image. No interleaving of data nor

dynamic scheduling would be warranted with this data set. Relatively good speedup should be achievable since there is no interprocess communication.

4 Implementation

Two different coding environments and architectures were used for running the sequential and parallel versions of the simulated annealing algorithm. The same sequential code was run on both architectures and it was written in C. This code is listed in Appendix A.

4.1 Networked Sun IPCs and C-Linda

The first environment was a distributed network of Sun workstations using IPC processors operating at 25 MHz. The parallel algorithm was written in C-Linda [8]. This code development environment is a superset of the C language with special functionally added to support Linda operations in a distributed programming environment. The Linda environment allows machines on local or wide area networks to spawn processes on other machines and also allows these processes to communicate data. Appendix B lists the code for C-Linda.

Linda uses a memory model called tuple space. Since each computer on the network is distinct, they do not share a common memory area. Tuple space allows for a virtual shared memory area between computers that may contain both active and passive tuples. The `eval` statement is used to start active tuples. These active tuples will start parallel processes on the computers on the network. At run time, the user specifies the number of computers that should be employed to work on the specific problem. A list of available computers with Linda installed is kept in local files on each computer. Linda will then ask the computers listed in the valid file to assist. If enough computers are available, the code will execute. When an `eval` statement is executed, Linda will spawn the process to a cooperating computer. Linda will attempt some load balancing. The operating system will not allow one computer in the link to receive and process all the `eval` statements.

Data tuples are created and consumed by the statements `out` and `in`, respectively. Pattern matching is employed for these operations, which uses both formals and actuals. Formals are variables that become instantiated and

are matched according to variable type. The statement `out(1, "test")` will create a 2-D tuple in tuple space. It consists of two actuals. The statement `in(? i, ? text)` consists of two formals and will consume the data and give `i` the value `1` and `text` the string `"test"`. This is only true if `i` is a variable of type integer and `text` is an array of characters.

Semaphores are available in Linda to prevent deadlock and race conditions which may occur in certain `in` and `out` data space operations. If a process is waiting to consume tuples and none are available, it will block.

The following listing shows a small code section of Linda tuple space operations.

```
main {
   eval(Compute(1))
   out(1, 2) /* adds (1, 2) tuple to tuple space */
   /* in may block if no matching tuples are available */
   in(1, ?result) /* consumes (1, 3) tuple, result = 3 */
   }
Compute(i) { /* i = 1 */
   in(i, ?data) /* consumes (1, 2) tuple, data = 2 */
   out(i, ++data) /* out (1, 3) tuple */
   }
```

4.2 Sun 2000 with Solaris Threads

The second architecture used was a Sun 2000 having eight processors operating at 50 MHz each. The operating system for this computer was Sun OS 5.3 having Solaris threads which allows for multiple processes in a shared memory environment. The code for this implementation is listed in Appendix C. Each thread can operate independently and asynchronously. Basically, a thread is a single sequence of steps performed by one or more programs. A thread runs through a program, and there is only one point of execution in a thread at any instant [9]. The new threads are created by a C library call and are dispatched to the processors in the computer by the operating system.

The system call to create new threads requires a function to be specified to which the new thread will branch. Any number of threads may be created, but it is generally inefficient to create more threads than processors. Doing so will cause one or more processors to run the remaining threads which will correspondingly result in their performance degradation. Parallelizing

the code required source level modifications of the sequential code by adding new procedure calls and library loads during linking. Appendix C lists the code for the program using threads.

5 Measurements

The stereo matching algorithm was tested with several computer-generated random dot stereograms. These stereograms represent synthetic three-dimensional objects. The side of the object facing the camera system is texture mapped with solid black. The solid black is then speckled with randomly placed white pixels. Figure 4 shows the four-tiered "wedding cake" structure used for testing this algorithm.

Figure 4: Three-dimensional wedding cake structure.

The number of white dots is limited to 10% of the total image to test the robustness of the algorithm. The right image is assigned the random dot stereomap. Since the object is tiered, a stereo camera system above and facing the object perceives the dots to be at different locations in the right and left images. This effect is simulated by creating the left image of the stereo pair by shifting pixels in the right image to the right. Pixels around

the outer edge were not offset, the next level in was offset by 2 pixels, the next level 4 pixels, and the center was offset by 6 pixels. Pixels with high movement represent areas that would be close to a camera looking down from above the wedding cake whereas pixels with no disparity would be distant from the camera. Figure 5 shows the random dot stereo pair.

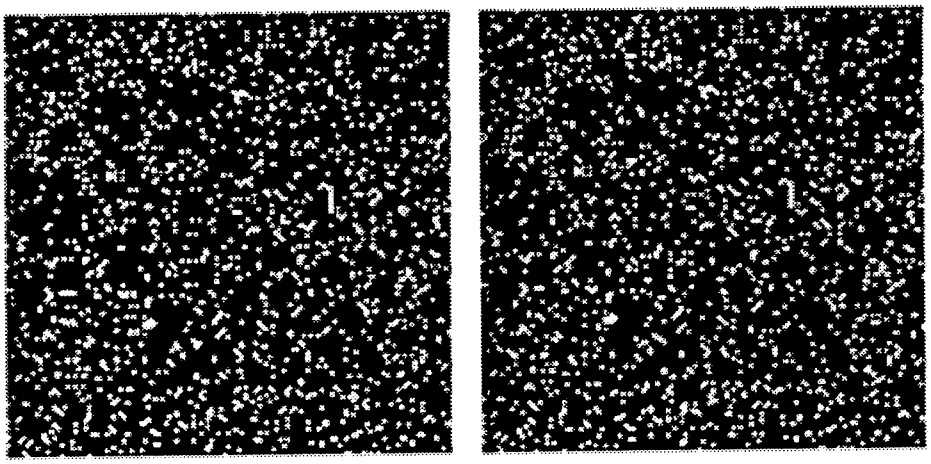


Figure 5: Random dot stereogram. The left image is formed by displacing points in the right image.

Because of the well-defined disparity maps, these random dot images represent ideal cases for evaluating stereo matching algorithms. That is, we know how the result should look, whereas in a real-world image, there would be some doubt as to what an exact map should look like. There are still some areas of ambiguity such as sections devoid of white pixels; however, the overall structure of the map remains clear. Image sizes of 128×128, 256×256, and 512×512 were used. These sizes were used since most frame grabbers and digitizers, as well as some image-processing routines, historically use image sizes of 2^n.

The primary goal of parallelization is to achieve speedup. Several measurement techniques are used to judge algorithm parallelization and architectural efficiency. The first metric is speedup. In an ideal environment, n processors working on a problem should be able to solve it n times faster.

This ideal is rarely achieved. The actual speedup achieved is defined as

$$S_p = \frac{T_l}{T_p} \tag{4}$$

in which T_l is the linear (one processor) completion time for the algorithm and T_p is the parallel completion time using p processors. The second metric used is efficiency. Efficiency is defined as

$$E_p = \frac{T_l}{p \cdot T_p}. \tag{5}$$

This number indicates the overall efficiency of the p processors working on the problem. Ideally, this number should be as close to 100% as possible but will suffer because of conditions of load imbalancing, communication costs, and various other parallelization overheads. As an example of these two metrics, consider an algorithm A that takes 45 time units to run and a parallelized version of A with 4 processors that takes 16 time units to run. The above metrics for this situation equate to

$$S_4 = \frac{45}{16} = 2.81, \; E_4 = \frac{45}{16 \cdot 4} = 0.70$$

or a speedup of 2.81 and a processor efficiency rating of 70%.

Figure 6 shows both the sequential and parallel results from matching the random dot stereogram shown in Figure 5. These results are from the Sun 2000 but are representative of a solution from any architecture. The parallel result was generated using four processors. The disparity maps produced by the algorithm, which are actually two-dimensional arrays with integer disparity values, are encoded as gray scale values for visual representation. Pixels with higher disparity values are closer in stereo (i.e., at the top of the structure) and are displayed as brighter shades of gray. Since the algorithm is non-deterministic and employs random number generators, some small differences can be seen in the resultant images. The three hard horizontal edges in the parallel result case stems from the fact that the edge falls directly on a processor's border.

The three-dimensional representation of the disparity map is shown in Figure 7. Notice that the stereo matching algorithm has very closely recovered the wedding cake structure shown in Figure 4.

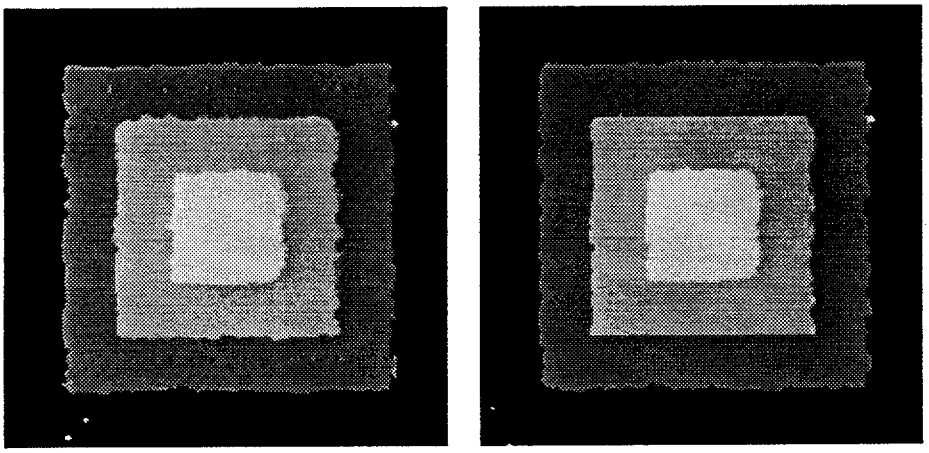


Figure 6: Stereo matching results. (Disparity maps are encoded as gray scale values. The left image was generated sequentially, the right image in parallel.)

6 Analysis and Interpretation

All times are based on averages of approximately five trials per case. Figure 8 shows the sequential time that the algorithm took on both the Sun IPC and 2000 machines.

As expected, because of faster processor speeds, the 2000 outperformed the IPC by about a factor of 4 in all cases. It can also be seen that the curve in both cases follows the same shape and that the time it takes to complete the algorithm grows proportionally with problem size.

The next charts (Figures 9 and 10) show the time the algorithm took for multiprocessors in both hardware environments.

Eight workers (or threads) maximum were used for the Sun 2000 since eight processors were installed on the test machine and 16 workers were used for the IPC network since 16 workstations could easily be acquired. From the shape of the bars, one can see that there are no anomalies in the graph. In all cases, as expected, the more processors working on the image-matching problem produced smaller computation times.

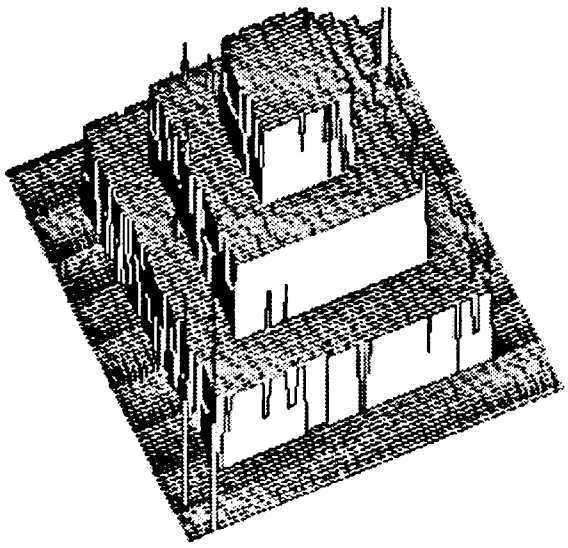


Figure 7: Three-dimensional representation of a four-level random dot stereogram.

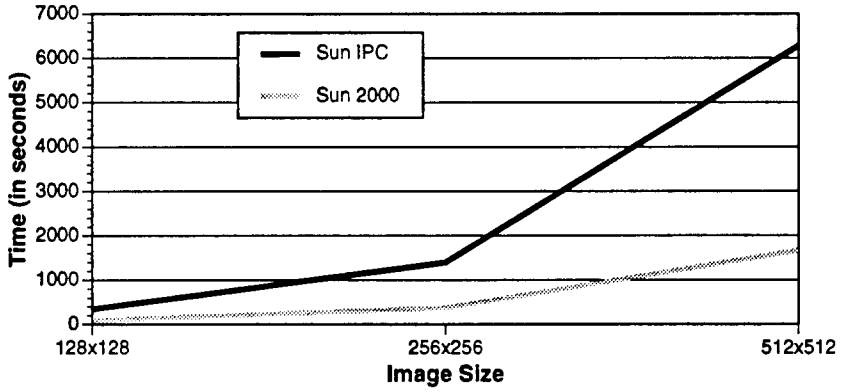


Figure 8: Sequential time on Sun IPCs and Sun 2000.

Figures 11 and 12 show the speedup achieved on the IPC network. Notice that the speedup per the number of processors is almost the same regardless of the image size on which the matching was performed. For example, the speedup for two processors is almost the same for the 128×128, 256×256, and 512×512 images. Image size does not seem to play a major role in

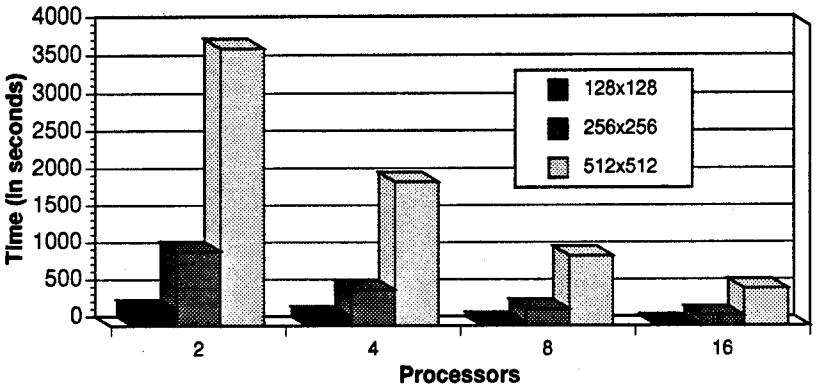


Figure 9: C-Linda performance on Sun IPC network.

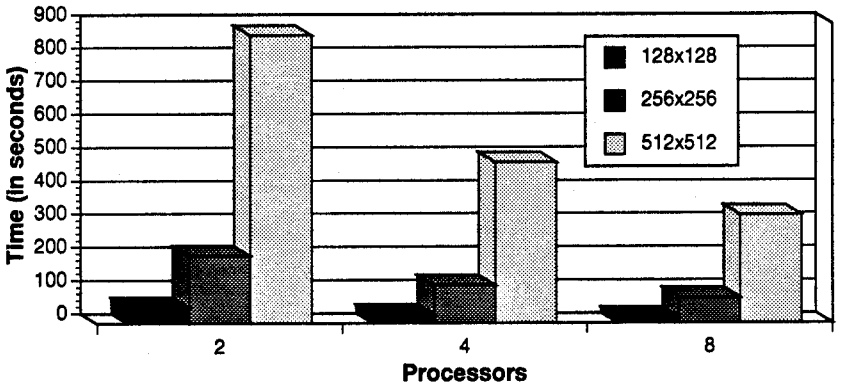


Figure 10: Threads performance on Sun 2000.

defining speedup. The case at 256×256 with 16 processors is an outlier, but this would undoubtedly have stabilized with more testing.

Speedup is achieved in all cases but does not quite reach its theoretical maximum in any case. Notice in Figure 12 that the speedup moves farther away from the line representing ideal speedup in the case of more processors. After some testing, it was determined that the operations to start workers on distributed machines, the master outing data to these machines, and these machines consuming the data, were very time-inexpensive operations. For example, in a typical case with the maximum number of workers (16), it only took roughly 0.2 second to start all 16 workers. The master outing data and workers consuming it usually only took about 2 seconds in these cases.

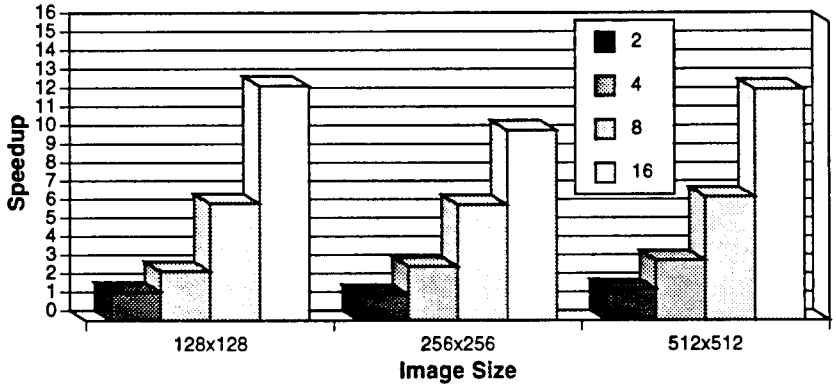


Figure 11: Speedup achieved using C-Linda on Sun IPC network.

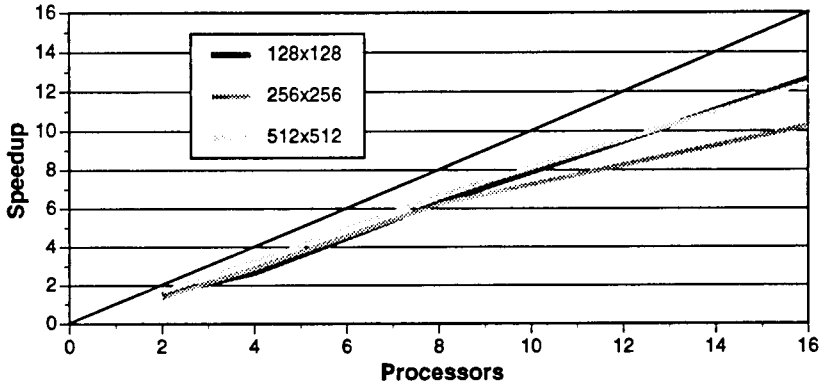


Figure 12: Speedup using C-Linda plotted against ideal linear speedup.

However, time varied greatly in the amount of time it took the workers to complete. In the 16-worker example, the time interval from when the first worker completed to when the last worker completed was anywhere from 30 to almost 50 seconds. In the 256 $\times$ 256 image size case, if all workers completed at the time the first worker finished, this would reduce the time by close to 40 seconds and increase speedup to almost 15. This is also why the speedup is moving away from the ideal speedup line on the graph. It only takes one slow worker to slow speedup and with more workers on the problem, a "weak link" is more likely. Speedup is very dependent on load averages for the machines working on the problem.

The efficiency for the IPC network is shown in Figure 13. The data in this chart is not as orderly, and there does not appear to be any emerging pattern. The efficiency is in the range from $\simeq 0.65$ to 0.8 for all cases. Eight processors seem to be the most efficient for all image sizes. The wide variations in the graph are attributable to the wide ranges of computational loads on the machines in the network. With the 256×256 image, 16-worker example as stated previously, if all processors finished as quickly as the first, the efficiency would be close to 95%. This graph is very dependent on the "weak" processor as well.

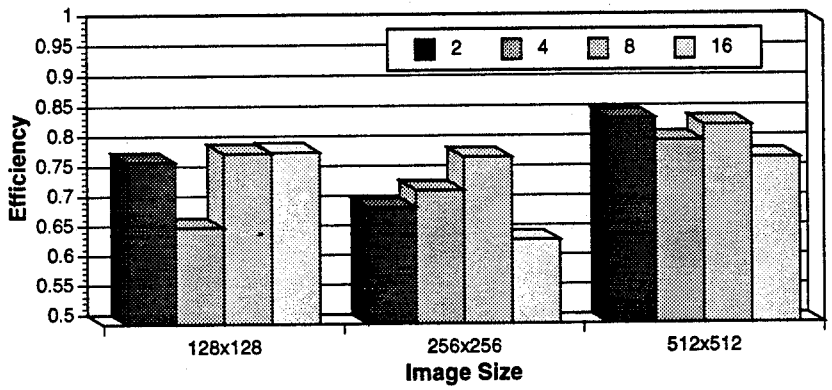


Figure 13: Efficiency of Sun IPC network.

Speedup achieved using threads on the Sun 2000 computer is shown in Figures 14 and 15. Just as with the Sun IPCs, it can be seen that speedup grows relatively consistently. In this case as well, image size does not seem to play a major role in influencing speedup. The speedup for two processors is roughly the same for the 128×128, 256×256, and 512×512 image size cases.

The speedup for two and four processors is good. The speedup for eight processors is much less than expected. The efficiency of the processors is also very good for the two and four processor (threads) cases. This can be seen in Figure 16.

The efficiency drops quickly and uniformly as more processors are used and is quite poor for the eight processor case. Just as with the IPCs, there were large gaps in the time interval it took for threads to complete. For example, in the 256×256 image case with eight workers, the average time

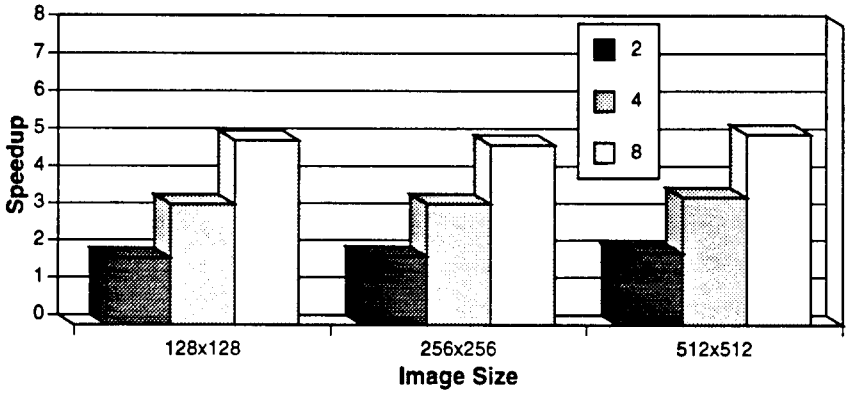


Figure 14: Speedup achieved using threads on Sun 2000.

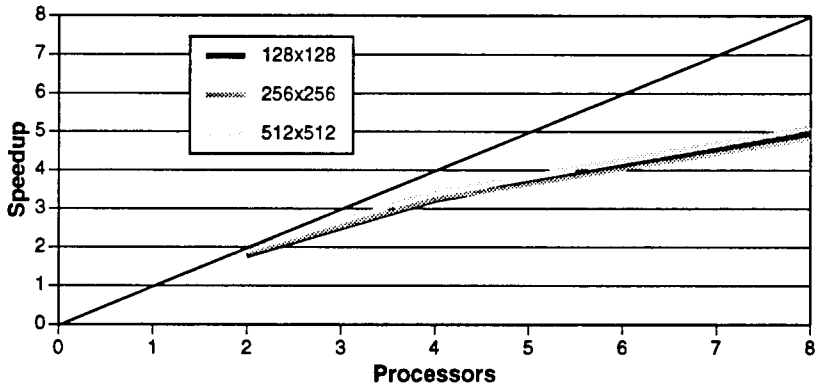


Figure 15: Speedup using threads plotted against ideal linear speedup.

interval between when the first thread completed to when the last thread completed was approximately 17 seconds. Adjusting for this delay, if all threads completed as early as the first one did, speedup would increase to 6.5 and efficiency would be close to 81%.

There are a few possible explanations for this poor performance with threads. The first reason is machine load. The Sun 2000 used routinely had load averages of 5.0 to 10.0 with an average of 60 to 80 users when the test cases were computed. The speedup will decrease as more of the machine's resources are tapped so the efficiency will be higher in the two-processor case than the eight-processor case.

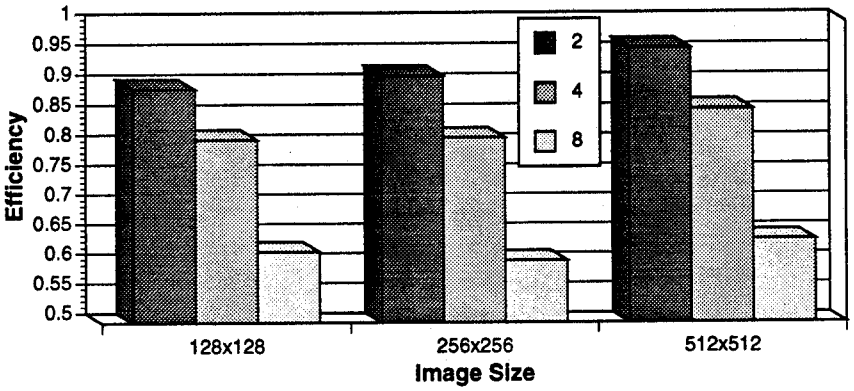


Figure 16: Efficiency of threads on Sun 2000.

Another issue that could not be fully resolved was whether all eight threads that were created were actually mapped to eight different processors. One processor sharing threads will slow the speedup dramatically. During some of the testing, 16 threads were created. In this case, two threads should have been mapped to each processor. When the speedup was computed, it was identical to the eight thread case with an efficiency of roughly 30%. If the threads could not be scheduled on separate processors, this would also account for the poor speedup and poor efficiency with eight threads.

One other concern in a shared memory environment like this one is subpage thrashing. If two or more threads are trying to write to the same page or sub page, some operating system coordination and overhead are introduced. The 2-D arrays used to store the image gray scale values varied in size and there was no way to ensure that the arrays were subpage aligned. This seems to be part of the problem and explains why there are slower times with more processors. Four processors trying for the same subpage will generate more overhead than two processors also competing for the page.

7 Conclusions

Simulated annealing offers an attractive method to solve large combinatorial problems. It is conceptually simple and relatively easy to implement. Once a proper energy function or heuristic is defined for the system, simulated annealing can become an effective solution technique in many problem areas.

Its one main drawback is completion time since so many iterations must be performed to take the system to its lowest energy configurations. This time can be substantially reduced by using parallel computers and taking advantage of the parallel nature of the image-matching algorithm.

Several conclusions may be drawn regarding the parallel architectures that were used for this algorithm. For limited numbers of processors, the Sun 2000 achieves greater speedup and is more efficient that the IPC network. If this algorithm were to use inter-process communications, the 2000 would probably be much more efficient. For the two and four processor case, the shared memory of the Sun 2000 outperforms the IPC network for all image sizes with very high efficiency (higher than 80% in all cases). The IPC network with C Linda start to show better speedup factors and much better efficiency than shared memory threads in cases when more processors are used and would probably start to outperform (in time required for algorithm completion) the faster Sun 2000 with some extension to the graph. For example, in all image size cases, the time 16 IPC processors took was very close to the time the eight processor Sun 2000 took to perform the stereo match. The shared environment of threads requires more operating system coordination to which the IPC network is immune. This is readily visible by comparing the efficiency bar graphs.

The amount of speedup achievable in the IPC network was good but was slightly lower than was expected. The deviation is understandable, however, since one slow worker will cause movement away from ideal speedup. It is usually the case that at least one non-dedicated machine on a network will have a high load average. The efficiency was good, however, across most cases with the majority of readings above 70%.

The threads environment on the Sun 2000 was more of a disappointment and will require more research into which of the factors mentioned previously are degrading performance. Except for the two and four processor case, the speedup achieved was poor with only a factor of five speedup with eight processors and an efficiency of less than 64%. More advanced techniques will have to be employed to determine how load averages and page thrashing played a role in this disappointing performance.

In terms of cost versus performance ratios, the network of Sun IPCs was quite high. A network of these low-cost machines began to perform comparable to the more expensive Sun 2000 in cases of large problem sizes.

References

[1] D.F. Rodgers and J.A. Adams. *Mathematical Elements for Computer Graphics.* McGraw-Hill, New York, 1990.

[2] Dana H. Ballard and Christopher M. Brown. *Computer Vision.* Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[3] Stephen T. Barnard. Stochastic stereo matching over scale. *International Journal of Computer Vision*, 3:17–32, 1989.

[4] P. J. M. van Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications.* Kluwer Academic Publishers, Boston, 1987.

[5] Stephen T. Barnard. A stochastic approach to stereovision. In *Readings in Computer Vision.* Addison-Wesley, New York, 1987.

[6] Stephen T. Barnard. Stereo matching by hierarchical microcanonical annealing. In *DARPA Image Understanding Workshop*, 1987.

[7] Gene Golub and James M. Ortega. *Scientific Computing: An Introduction with Parallel Computing.* Academic Press, Inc., Boston, 1993.

[8] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs.* The MIT Press, Cambridge, Massachusetts, 1992.

[9] Robert P. Weaver. *Sun OS 5.3 Guide to Multithread Programming.* Sun Microsystems, Inc., Mountain View, California, 1993.

A Linear Stereo Matching Algorithm

```
/*

Stereo matching algorithm based on simulated annealing.

Concept attributable to Stephen Barnard, "A Stochastic Approach to
Stereo Vision"  Readings in Computer Vision, Addison-Wesley, New York,
1987

Author: Dale R. Shires

Revision Date: 5/19/94

Notes:

See the article for more details on the Monte-Carlo search method.

Limited error checking is employed.

A fixed annealing schedule is currently used.  Global accumulators
could also be used on the energy to help determine temperature
reduction.

The procedure uses many defines and reads two images, one named
LeftImage### and RightImage### where ### is the number of rows used.
This implementation only uses 2^n image sizes and rows and columns must
be the same.

The program currently does not output the actual disparity values.  Rather
it outputs a gray scale map representing these values.  The resultant
map name is STEREO_RESULTS

*/

#include <math.h>
#include <stdio.h>
#include <sys/time.h>

#define ROWS 512    /* Image height */
#define COLS 512    /* Image width */
#define LAMBDA_IMPORTANCE 5 /* weighting value */
#define D_MAX 8     /* Maximum disparity was 18 */
#define D_MIN 0 /* Minimum disparity */
#define LEFT_IMAGE 1 /* Left image identifier */
#define RIGHT_IMAGE 2 /* Right image identifier */
#define STOP_COL (COLS - D_MAX) /* limit disparity */
#define RANDOM_DISPARITY (rand() % D_MAX)
#define RANDOM_PROBABILITY (rand() % 32767 / 32767.0) /* [0..1] */
#define STARTING_TEMP 100.0 /* starting temp of the system */
#define ENDING_TEMP 1.0 /* stop annealing at this temp */
#define TEMP_REDUCTION 0.1 /* reduce temp this much after loop */
#define LATTICE_SCANS 10 /* loop this many times for each temp */
```

```
/* Define types: */

typedef char String255[256];
typedef unsigned char Pixel;

/* Global arrays */

Pixel leftImage[ROWS][COLS];
Pixel rightImage[ROWS][COLS];
int disparities[ROWS][COLS];
int tempDisparities[ROWS][COLS];


/* Read the gray scale values of a digital image stored in ASCII format */

void ReadImageIntensityValues(theImage, which)
Pixel theImage[ROWS][COLS];
int which;
{
register int rowCounter, colCounter;
register Pixel *ptr;
String255 fileName;
char pixelValue;
FILE *theFile;

if (which == LEFT_IMAGE)
sprintf(fileName, "LeftImage%d", ROWS);
else
sprintf(fileName, "RightImage%d", ROWS);


theFile = fopen(fileName, "r");

if (theFile == NULL)
printf("Could not open file %s\n", fileName);

for (rowCounter = 0; rowCounter < ROWS; rowCounter++)
{
ptr = theImage[rowCounter];
for (colCounter = 0; colCounter < COLS; colCounter++)
{
fscanf(theFile, "%c", &pixelValue);
*ptr = (unsigned char)pixelValue;
++ptr;
}
}

fclose(theFile);

} /* end ReadImageIntensityValues */


/* Generate a new state +- 1 unit from the previous state.  No overflow or
underflow is allowed */
```

```
int RandomNewState(oldState)
register int oldState;
{
register int randomNumber;

if (RANDOM_DISPARITY > ((D_MAX / 2)-1))
randomNumber = -1;
else
randomNumber = 1;

oldState = oldState + randomNumber;

if (oldState < D_MIN)
oldState = D_MIN;
else if (oldState > D_MAX)
oldState = D_MAX;

return(oldState);

} /* end RandomNewState */


/* Randomized the system.  The disparity map is initialized in the range
from [D_MIN .. D_MAX] */

void RandomizeSystem()
{
register int row, col;

for (row = 0; row < ROWS; row++)
for (col = 0; col < COLS; col++)
disparities[row][col] = RANDOM_DISPARITY;

} /* end RandomizeSystem */


/* Output the resultant disparity map in a gray-scale format.  0 indicates
areas on no disparity and 255 indicates areas of maximum disparity.
The disparity range is divided into steps in a 256 brightness range. */

void CreateGrayScaleMap()
{
register int row, col;
FILE *theFile;
  char fileName[250];

theFile = fopen("STEREO_RESULTS", "w");

for (row = 0; row < ROWS; row++)
for (col = 0; col < COLS; col++)
fprintf(theFile, "%c", disparities[row][col] * (255 / D_MAX));

fclose(theFile);
```

```
} /* end CreateGrayScaleMap */


/* Compute the energy of a pixel based on photometric and disparity
parameters. */

int Energy(row, col, disparity)
int row, col, disparity;
{
register int leftIntensity, rightIntensity;
register int photo;
int del;

del = (abs(disparity - disparities[row-1][col-1]) +
   abs(disparity - disparities[row-1][col]) +
   abs(disparity - disparities[row-1][col+1]) +
   abs(disparity - disparities[row][col-1]) +
   abs(disparity - disparities[row][col+1]) +
   abs(disparity - disparities[row+1][col-1]) +
   abs(disparity - disparities[row+1][col]) +
   abs(disparity - disparities[row+1][col+1]));


leftIntensity = leftImage[row][col];
rightIntensity = rightImage[row][col+disparity];
photo = abs(leftIntensity - rightIntensity);

return(photo + (LAMBDA_IMPORTANCE * del));

} /* end Energy */



/* The following energy functions are similar to the function Energy but
   cover the special cases along the boarders where 8 neighbors are not
present. */

int TopEnergy(col, disparity)
int col, disparity;
{
register int leftIntensity, rightIntensity;
register int photo;
int del;

del = (abs(disparity - disparities[0][col-1]) +
   abs(disparity - disparities[0][col+1]) +
   abs(disparity - disparities[1][col-1]) +
   abs(disparity - disparities[1][col]) +
   abs(disparity - disparities[1][col+1]));


leftIntensity = leftImage[0][col];
rightIntensity = rightImage[0][col+disparity];
photo = abs(leftIntensity - rightIntensity);
```

```
return(photo + (LAMBDA_IMPORTANCE * del));

} /* end TopEnergy */


int LeftEnergy(row, disparity)
int row, disparity;
{
register int leftIntensity, rightIntensity;
register int photo;
int del;

del = (abs(disparity - disparities[row-1][0]) +
   abs(disparity - disparities[row-1][1]) +
   abs(disparity - disparities[row][1]) +
   abs(disparity - disparities[row+1][0]) +
   abs(disparity - disparities[row+1][1]));


leftIntensity = leftImage[row][0];
rightIntensity = rightImage[row][disparity];
photo = abs(leftIntensity - rightIntensity);

return(photo + (LAMBDA_IMPORTANCE * del));

} /* end LeftEnergy */


int BottomEnergy(col, disparity)
int col, disparity;
{
register int leftIntensity, rightIntensity;
register int photo;
int del;

del = (abs(disparity - disparities[ROWS-1][col-1]) +
   abs(disparity - disparities[ROWS-1][col+1]) +
   abs(disparity - disparities[(ROWS-1)-1][col-1]) +
   abs(disparity - disparities[(ROWS-1)-1][col]) +
   abs(disparity - disparities[(ROWS-1)-1][col+1]));


leftIntensity = leftImage[ROWS-1][col];
rightIntensity = rightImage[ROWS-1][col+disparity];
photo = abs(leftIntensity - rightIntensity);

return(photo + (LAMBDA_IMPORTANCE * del));

} /* end BottomEnergy */



int TopLeftEnergy(disparity)
int disparity;
```

```
{
register int leftIntensity, rightIntensity;
register int photo;
int del;

del = (abs(disparity - disparities[0][1]) +
   abs(disparity - disparities[1][0]) +
   abs(disparity - disparities[1][1]));

leftIntensity = leftImage[0][0];
rightIntensity = rightImage[0][disparity];
photo = abs(leftIntensity - rightIntensity);

return(photo + (LAMBDA_IMPORTANCE * del));

} /* end TopLeftEnergy */


int BottomLeftEnergy(disparity)
int disparity;
{
register int leftIntensity, rightIntensity;
register int photo;
int del;

del = (abs(disparity - disparities[(ROWS-1)-1][0]) +
   abs(disparity - disparities[(ROWS-1)-1][1]) +
   abs(disparity - disparities[ROWS-1][1]));

leftIntensity = leftImage[ROWS-1][0];
rightIntensity = rightImage[ROWS-1][disparity];
photo = abs(leftIntensity - rightIntensity);

return(photo + (LAMBDA_IMPORTANCE * del));

} /* end BottomLeftEnergy */


/* Perform the simulated annealing. */

void Anneal(temperature)
double temperature;
{
register int row, col;
register int deltaEnergy, newEnergy;
register int *tempPtr1, *tempPtr2;
int oldEnergy;
int newState;

/* compute top left energy */

newState = RandomNewState(disparities[0][0]);
oldEnergy = TopLeftEnergy(disparities[0][0]);
newEnergy = TopLeftEnergy(newState);
deltaEnergy = newEnergy - oldEnergy;
```

```
if (deltaEnergy <= 0)
tempDisparities[0][0] = newState;
else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/temperature))
tempDisparities[0][0] = newState;
else
tempDisparities[0][0] = disparities[0][0];

/* compute top row energy */

for (col = 1; col < STOP_COL; col++)
{
newState = RandomNewState(disparities[0][col]);
oldEnergy = TopEnergy(col, disparities[0][col]);
newEnergy = TopEnergy(col, newState);
deltaEnergy = newEnergy - oldEnergy;
if (deltaEnergy <= 0)
tempDisparities[0][col] = newState;
else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/temperature))
tempDisparities[0][col] = newState;
else
tempDisparities[0][col] = disparities[0][col];
}

/* compute left column energy */

for (row = 1; row < (ROWS-1)-1; row++)
{
newState = RandomNewState(disparities[row][0]);
oldEnergy = LeftEnergy(row, disparities[row][0]);
newEnergy = LeftEnergy(row, newState);
deltaEnergy = newEnergy - oldEnergy;
if (deltaEnergy <= 0)
tempDisparities[row][0] = newState;
else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/temperature))
tempDisparities[row][0] = newState;
else
tempDisparities[row][0] = disparities[row][0];
}

/* compute bottom left energy */

newState = RandomNewState(disparities[(ROWS-1)-1][0]);
oldEnergy = BottomLeftEnergy(disparities[(ROWS-1)-1][0]);
newEnergy = BottomLeftEnergy(newState);
deltaEnergy = newEnergy - oldEnergy;
if (deltaEnergy <= 0)
tempDisparities[(ROWS-1)-1][0] = newState;
else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/temperature))
tempDisparities[(ROWS-1)-1][0] = newState;
else
tempDisparities[(ROWS-1)-1][0] = disparities[(ROWS-1)-1][0];


/* compute bottom row energy */
```

```
for (col = 1; col < STOP_COL; col++)
{
newState = RandomNewState(disparities[ROWS-1][col]);
oldEnergy = BottomEnergy(col, disparities[ROWS-1][col]);
newEnergy = BottomEnergy(col, newState);
deltaEnergy = newEnergy - oldEnergy;
if (deltaEnergy <= 0)
tempDisparities[ROWS-1][col] = newState;
else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/temperature))
tempDisparities[ROWS-1][col] = newState;
else
tempDisparities[ROWS-1][col] = disparities[ROWS-1][col];
}

/* compute main grid energy */

for (row = 1; row < (ROWS-1)-1; row++)
for (col = 1; col < STOP_COL; col++)
{
newState = RandomNewState(disparities[row][col]);
oldEnergy = Energy(row, col, disparities[row][col]);
newEnergy = Energy(row, col, newState);
deltaEnergy = newEnergy - oldEnergy;
if (deltaEnergy <= 0)
tempDisparities[row][col] = newState;
else if (RANDOM_PROBABILITY <
exp((double)-deltaEnergy/temperature))
tempDisparities[row][col] = newState;
else
tempDisparities[row][col] = disparities[row][col];
}


/* copy the temp disparity array into the main disparity array */

for (row = 0; row < ROWS; row++)
{
tempPtr1 = disparities[row];
tempPtr2 = tempDisparities[row];
for (col = 0; col < COLS; col++)
{
*tempPtr1 = *tempPtr2;
++tempPtr1;
++tempPtr2;
}
}

} /* end Anneal */



void main(argc, argv)
int argc;
char **argv;
{
```

```
double currentTemp;
int scanCounter;
int temp;
struct timeval tp1, tp2;
struct timezone tzp1, tzp2;

printf("Executable = %s\n", argv[0]);
printf("Rows = %d, Cols = %d\n\n", ROWS, COLS);

/* read the left and right images */

ReadImageIntensityValues(leftImage, LEFT_IMAGE);
ReadImageIntensityValues(rightImage, RIGHT_IMAGE);

/* randomize the disparity map of the system */

gettimeofday(&tp1, &tzp1);

RandomizeSystem();

currentTemp = STARTING_TEMP;

while (currentTemp >= ENDING_TEMP)
{
for (scanCounter = 0; scanCounter < LATTICE_SCANS; scanCounter++)
{
Anneal(currentTemp);
temp = currentTemp;
}
currentTemp -= (currentTemp * TEMP_REDUCTION);
}

gettimeofday(&tp2, &tzp2);

printf("%ld, %ld\n", tp1.tv_sec, tp1.tv_usec);
printf("%ld, %ld\n", tp2.tv_sec, tp2.tv_usec);

CreateGrayScaleMap();

} /* end main */
```

B C–Linda Parallel Stereo Matching Algorithm

```
/*

Parallel stereo matching algorithm based on simulated annealing.

Concept attributable to Stephen Barnard, "A Stochastic Approach to
Stereo Vision"  Readins in Computer Vision, Addison-Wesley, New York,
1987

Author: Dale R. Shires

Revision Date: 5/18/94

Notes:

This code has been written for parallel processing with C-Linda.
Appropriate source level annotations to the code have been made.

See the article for more details on the Monte-Carlo search method.

Limited error checking is employed.

The procedure requires two files, one named LeftImage and one
named RightImage to be present in the directory.  This implementation
only uses 2^n image sizes and the rows and columns must be the same.

The program currently does not output the actual disparity values.
Rather, it outputs a gray scale map representing these values.
The resultant map name is STEREO_RESULTS

*/


#include <stdio.h>
#include <sys/time.h>
#include <math.h>

#define ROWS 256  /* the number of rows in the image */
#define COLS 256 /* the number of columns in the image */
#define D_MAX 8 /* maximum horizontal disparity */
#define D_MIN 0  /* minimum disparity */
#define LAMBDA 5 /* the lambda weighting factor */
#define WORKERS 16  /* number of workers to use */
#define LEFT_IMAGE_NAME "LeftImage"
#define RIGHT_IMAGE_NAME "RightImage"
#define START_TEMP 100.0  /* starting temp of the system */
#define END_TEMP 1.0 /* stop annealing at this temp */
#define TEMP_REDUCTION 0.1 /* reduce temp this much after loop */
#define LOOPS 10 /* number of loops per temperature */
#define RANDOM_DISPARITY (rand() % D_MAX)
#define RANDOM_PROBABILITY (rand() % 32767 / 32767.0) /* [0..1] */
```

```
#define STEPSIZE (ROWS / WORKERS) /* number of rows per worker */

typedef char String[255];

int RandomNewState(oldState)
int oldState;
{

(RANDOM_DISPARITY > ((D_MAX / 2) - 1)) ? --oldState : ++oldState;

(oldState < D_MIN) ? oldState = D_MIN : (oldState > D_MAX) ? oldState = D_MAX : 1;

return(oldState);

}  /* end RandomNewState */


void ReadImage(theImage, fileName)
unsigned char theImage[ROWS][COLS];
String fileName;
{
int row, col;
FILE *theFile;

theFile = fopen((char *)fileName, "r");

if (theFile == NULL)
printf("Error opening file %s\n", fileName);

for (row = 0; row < ROWS; row++)
for (col = 0; col < COLS; col++)
theImage[row][col] = getc(theFile);

fclose(theFile);

} /* end ReadImage */


void CreateGrayScaleMap(disparities)
int disparities[ROWS][COLS];
{
int row, col;
FILE *theFile;

theFile = fopen("STEREO_RESULTS", "w");

for (row = 0; row < ROWS; row++)
for (col = 0; col < COLS; col++)
putc((255 / D_MAX) * disparities[row][col], theFile);

fclose(theFile);

} /* end CreateGrayScaleMap */
```

```
int RandomizeMap(theMap)
int theMap[STEPSIZE][COLS];
{
int i, j;

for (i = 0; i < STEPSIZE; i++)
for (j = 0; j < COLS; j++)
theMap[i][j] = RANDOM_DISPARITY;

}  /* end RandomizeMap */


/* This is the function that each worker gets.  The function is somewhat
long but this was done to limit shared memory and numbers of calls to
make.  There are several special cases that must be dealt with.  These
are areas where a point does not have 8 neighbors, such as the top row,
side column, bottom row, and topLeft and bottomLeft points in the
block */

int ComputeDisparities(workerNumber)
int workerNumber;
{
unsigned char leftImage[STEPSIZE][COLS], rightImage[STEPSIZE][COLS];
int disparities[STEPSIZE][COLS];
int tempDisparities[STEPSIZE][COLS];
double temperature;
int i, m, n, j, k;
int counter;
double currentTemp = START_TEMP;
int row, col, newState, oldState;
int oldEnergy, newEnergy, deltaEnergy;

/* randomize my local disparity map */

RandomizeMap(disparities);

/* read in my chunk of the images */

for (i = 0; i < STEPSIZE; i++)
in(workerNumber, i, ? leftImage[i]:n, ? rightImage[i]:m);

out("done in process");

/* start the processing */

while(currentTemp >= END_TEMP)
{
for (counter = 0; counter < LOOPS; counter++)
{
/* compute the top left energy */

newState = RandomNewState(disparities[0][0]);
oldState = disparities[0][0];
oldEnergy = ((abs(disparities[0][1] - oldState) +
abs(disparities[1][0] - oldState) +
```

```
abs(disparities[1][1] - oldState)) * LAMBDA) +
abs(leftImage[0][0] - rightImage[0][oldState]);
newEnergy = ((abs(disparities[0][1] - newState) +
abs(disparities[1][0] - newState) +
abs(disparities[1][1] - newState)) * LAMBDA) +
abs(leftImage[0][0] - rightImage[0][newState]);

deltaEnergy = newEnergy - oldEnergy;
if (deltaEnergy <= 0)
tempDisparities[0][0] = newState;
else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/currentTemp))
tempDisparities[0][0] = newState;
else tempDisparities[0][0] = disparities[0][0];

/* compute the bottom left energy */

newState = RandomNewState(disparities[STEPSIZE-1][0]);
oldState = disparities[STEPSIZE-1][0];
oldEnergy = ((abs(disparities[STEPSIZE-1][1] - oldState) +
abs(disparities[STEPSIZE-2][0] - oldState) +
abs(disparities[STEPSIZE-2][1] - oldState)) * LAMBDA) +
abs(leftImage[STEPSIZE-1][0] - rightImage[STEPSIZE-1][oldState]);
newEnergy = ((abs(disparities[STEPSIZE-1][1] - newState) +
abs(disparities[STEPSIZE-2][0] - newState) +
abs(disparities[STEPSIZE-2][1] - newState)) * LAMBDA) +
abs(leftImage[STEPSIZE-1][0] - rightImage[STEPSIZE-1][newState]);

deltaEnergy = newEnergy - oldEnergy;
if (deltaEnergy <= 0)
tempDisparities[STEPSIZE-1][0] = newState;
else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/currentTemp))
tempDisparities[STEPSIZE-1][0] = newState;
else tempDisparities[STEPSIZE-1][0] = disparities[STEPSIZE-1][0];

/* compute left side energy */

for (row = 1; row < STEPSIZE-2; row++)
{
newState = RandomNewState(disparities[row][0]);
oldState = disparities[row][0];
oldEnergy = ((abs(disparities[row-1][0] - oldState) +
abs(disparities[row-1][1] - oldState) +
abs(disparities[row][1] - oldState) +
abs(disparities[row+1][1] - oldState) +
abs(disparities[row+1][0] - oldState)) * LAMBDA) +
abs(leftImage[row][0] - rightImage[row][oldState]);

newEnergy = ((abs(disparities[row-1][0] - newState) +
abs(disparities[row-1][1] - newState) +
abs(disparities[row][1] - newState) +
abs(disparities[row+1][1] - newState) +
abs(disparities[row+1][0] - newState)) * LAMBDA) +
abs(leftImage[row][0] - rightImage[row][newState]);

deltaEnergy = newEnergy - oldEnergy;
```

```
if (deltaEnergy <= 0)
tempDisparities[row][0] = newState;
else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/currentTemp))
tempDisparities[row][0] = newState;
else tempDisparities[row][0] = disparities[row][0];

}

/* compute top row energy */

for (col = 1; col < COLS - D_MAX; col++)
{
newState = RandomNewState(disparities[0][col]);
oldState = disparities[0][col];
oldEnergy = ((abs(disparities[0][col-1] - oldState) +
  abs(disparities[1][col-1] - oldState) +
abs(disparities[1][col] - oldState) +
abs(disparities[1][col+1] - oldState) +
abs(disparities[0][col+1] - oldState)) * LAMBDA) +
abs(leftImage[0][col] - rightImage[0][col+oldState]);

newEnergy = ((abs(disparities[0][col-1] - newState) +
  abs(disparities[1][col-1] - newState) +
abs(disparities[1][col] - newState) +
abs(disparities[1][col+1] - newState) +
abs(disparities[0][col+1] - newState)) * LAMBDA) +
abs(leftImage[0][col] - rightImage[0][col+newState]);

deltaEnergy = newEnergy - oldEnergy;
if (deltaEnergy <= 0)
tempDisparities[0][col] = newState;
else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/currentTemp))
tempDisparities[0][col] = newState;
else tempDisparities[0][col] = disparities[0][col];

}

/* compute bottom row energy */

for (col = 1; col < COLS - D_MAX; col++)
{
newState = RandomNewState(disparities[STEPSIZE-1][col]);
oldState = disparities[STEPSIZE-1][col];
oldEnergy = ((abs(disparities[STEPSIZE-1][col-1] - oldState) +
  abs(disparities[STEPSIZE-2][col-1] - oldState) +
abs(disparities[STEPSIZE-2][col] - oldState) +
abs(disparities[STEPSIZE-2][col+1] - oldState) +
abs(disparities[STEPSIZE-1][col+1] - oldState)) * LAMBDA) +
abs(leftImage[STEPSIZE-1][col] - rightImage[STEPSIZE-1][col+oldState]);

newEnergy = ((abs(disparities[STEPSIZE-1][col-1] - newState) +
  abs(disparities[STEPSIZE-2][col-1] - newState) +
abs(disparities[STEPSIZE-2][col] - newState) +
abs(disparities[STEPSIZE-2][col+1] - newState) +
abs(disparities[STEPSIZE-1][col+1] - newState)) * LAMBDA) +
```

```
abs(leftImage[STEPSIZE-1][col] - rightImage[STEPSIZE-1][col+newState]);

deltaEnergy = newEnergy - oldEnergy;
if (deltaEnergy <= 0)
tempDisparities[STEPSIZE-1][col] = newState;
else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/currentTemp))
tempDisparities[STEPSIZE-1][col] = newState;
else tempDisparities[STEPSIZE-1][col] = disparities[STEPSIZE-1][col];

}

/* compute the main grid energy */

for (row = 1; row < STEPSIZE-1; row++)
for (col = 1; col < COLS - D_MAX; col++)
{
newState = RandomNewState(disparities[row][col]);
oldState = disparities[row][col];
oldEnergy = ((abs(disparities[row-1][col-1] - oldState) +
  abs(disparities[row-1][col] - oldState) +
abs(disparities[row-1][col+1] - oldState) +
abs(disparities[row][col-1] - oldState) +
abs(disparities[row][col+1] - oldState) +
abs(disparities[row+1][col-1] - oldState) +
abs(disparities[row+1][col] - oldState) +
abs(disparities[row+1][col+1] - oldState)) * LAMBDA) +
abs(leftImage[row][col] - rightImage[row][col+oldState]);

newEnergy = ((abs(disparities[row-1][col-1] - newState) +
  abs(disparities[row-1][col] - newState) +
abs(disparities[row-1][col+1] - newState) +
abs(disparities[row][col-1] - newState) +
abs(disparities[row][col+1] - newState) +
abs(disparities[row+1][col-1] - newState) +
abs(disparities[row+1][col] - newState) +
abs(disparities[row+1][col+1] - newState)) * LAMBDA) +
abs(leftImage[row][col] - rightImage[row][col+newState]);

deltaEnergy = newEnergy - oldEnergy;
if (deltaEnergy <= 0)
tempDisparities[row][col] = newState;
else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/currentTemp))
tempDisparities[row][col] = newState;
else tempDisparities[row][col] = disparities[row][col];

}

/* move the temp disparities into the main array */

for (j = 0; j < STEPSIZE; j++)
for (k = 0; k < COLS; k++)
disparities[j][k] = tempDisparities[j][k];

}
currentTemp -= (currentTemp * TEMP_REDUCTION);
```

```
}

/* output results */

for (j = 0; j < STEPSIZE; j++)
out(workerNumber, j, disparities[j]:COLS);

}  /* end ComputeDisparities */


real_main(argc, argv)
int argc;
char **argv;
{
unsigned char leftImage[ROWS][COLS], rightImage[ROWS][COLS];
int disparities[ROWS][COLS];
int result[COLS];
int i, j, count;
int workerID, row, cols;
struct timeval tp1, tp2;
struct timezone tzp1, tzp2;

printf("Executable = %s\n", argv[0]);
printf("Rows = %d, Cols = %d\n", ROWS, COLS);
printf("Workers = %d\n\n", WORKERS);

/* set the random seed to make sure we get different results each time */

gettimeofday(&tp1, &tzp1);
srand(tp1.tv_sec);

ReadImage(rightImage, RIGHT_IMAGE_NAME);

ReadImage(leftImage, LEFT_IMAGE_NAME);

start_timer();
timer_split("Start");

/* start up the workers */

for (i = 0; i < WORKERS; i++)
eval(ComputeDisparities(i));

timer_split("Workers started");

/* out the data to tuplespace for the workers to consume */

count = 0;
for (i = 0; i < WORKERS; i++)
for (j = 0; j < STEPSIZE; j++)
{
out(i, j, leftImage[count]:COLS, rightImage[count]:COLS);
++count;
}
```

```
timer_split("Data outed");

for (i=0; i< WORKERS; i++)
in("done in process");

timer_split("workers ined data");

/* collect results from the workers */

for (i = 0; i < ROWS; i++)
{
in(? workerID, ? row, ? result:cols);
for (j = 0; j < COLS; j++)
disparities[(workerID * STEPSIZE)+row][j] = result[j];
}

timer_split("Results Collected");

/* create a visual representation of the disparity map */

CreateGrayScaleMap(disparities);

print_times();

}  /* end real_main */
```

C Solaris Threads Stereo Matching Algorithm

```
/*

Parallel stereo matching algorithm based on simulated annealing.

Concept attributable to Stephen Barnard, "A Stochastic Approach to
Stereo Vision"  Readins in Computer Vision, Addison-Wesley, New York,
1987

Author: Dale R. Shires

Revision Date: 5/18/94

Notes:

This code has been written for parallel processing with Solaris threads.
Appropriate source level annotations to the code have been made.

See the article for more details on the Monte-Carlo search method.

Limited error checking is employed.

The procedure requires two files, one named LeftImage and one
named RightImage to be present in the directory.  This implementation
only uses 2^n image sizes and the rows and columns must be the same.

The program currently does not output the actual disparity values.
Rather, it outputs a gray scale map representing these values.
The resultant map name is STEREO_RESULTS

*/


#include <stdio.h>
#include <sys/time.h>
#include <math.h>
#include <thread.h>
#include <synch.h>
#include <errno.h>

#define ROWS 256  /* the number of rows in the image */
#define COLS 256 /* the number of columns in the image */
#define D_MAX 6 /* maximum horizontal disparity */
#define D_MIN 0 /* minimum horizontal disparity */
#define LAMBDA 5 /* the lambda weighting factor */
#define WORKERS 8 /* number of workers to use */
#define LEFT_IMAGE_NAME "LeftImage"
#define RIGHT_IMAGE_NAME "RightImage"
#define START_TEMP 100.0 /* starting temperature of the algorithm */
#define END_TEMP 1.0 /* ending temperature of the algorithm */
#define TEMP_REDUCTION 0.1 /* reduce temp this much after loop */
```

```
#define LOOPS 10 /* number of loops per temperature */
#define RANDOM_DISPARITY (rand() % D_MAX)
#define RANDOM_PROBABILITY (rand() % 32767 / 32767.0)
#define STEPSIZE (ROWS / WORKERS)

typedef char String[255];

unsigned char leftImage[ROWS][COLS], rightImage[ROWS][COLS];
int disparities[ROWS][COLS], tempDisparities[ROWS][COLS];


int RandomNewState(oldState)
int oldState;
{

(RANDOM_DISPARITY > ((D_MAX / 2) - 1)) ? --oldState : ++oldState;

(oldState < D_MIN) ? oldState = D_MIN : (oldState > D_MAX) ? oldState = D_MAX : 1;

return(oldState);

}  /* end RandomNewState */



void WriteImage(theImage, fileName)
unsigned char theImage[ROWS][COLS];
String fileName;
{
int row, col;
FILE *theFile;

theFile = fopen((char *)fileName, "w");

if (theFile == NULL)
printf("Error opening file %s\n", fileName);

for (row = 0; row < ROWS; row++)
for (col = 0; col < COLS; col++)
putc(theImage[row][col], theFile);

fclose(theFile);

} /* end WriteImage */



void ReadImage(theImage, fileName)
unsigned char theImage[ROWS][COLS];
String fileName;
{
int row, col;
FILE *theFile;

theFile = fopen((char *)fileName, "r");
```

```
if (theFile == NULL)
printf("Error opening file %s\n", fileName);

for (row = 0; row < ROWS; row++)
for (col = 0; col < COLS; col++)
theImage[row][col] = getc(theFile);

fclose(theFile);

} /* end ReadImage */


void CreateGrayScaleMap(disparities)
int disparities[ROWS][COLS];
{
int row, col;
FILE *theFile;

theFile = fopen("STEREO_RESULTS", "w");

for (row = 0; row < ROWS; row++)
for (col = 0; col < COLS; col++)
putc((255 / D_MAX) * disparities[row][col], theFile);

fclose(theFile);

} /* end CreateGrayScaleMap */


int RandomizeMap(theMap)
int theMap[ROWS][COLS];
{
int i, j;

for (i = 0; i < ROWS; i++)
for (j = 0; j < COLS; j++)
theMap[i][j] = RANDOM_DISPARITY;

}  /* end RandomizeMap */


/* This is the function that each worker gets.  The function is somewhat
   long but this was done to limit shared memory and numbers of calls to
   make.  There are several special cases that must be dealt with.  These
   are areas where a point does not have 8 neighbors, such as the top row,
   side column, bottom row, and topLeft and bottomLeft points in the
   block */

void ComputeDisparities(workerNumber)
int workerNumber;
{
double temperature;
int i;
int n, m;
```

```
int j, k;
int counter;
double currentTemp = START_TEMP;
int row, col, newState, oldState;
int oldEnergy, newEnergy;
int deltaEnergy;
int startRow, stopRow;
int theWorker;
struct timeval tp;
struct timezone tzp;

/* set the worker number */

theWorker = workerNumber;

/* determine start and stop positions for this worker */

startRow = theWorker * STEPSIZE;
stopRow = startRow + STEPSIZE;

gettimeofday(&tp, &tzp);
printf("Start %d %ld\n", theWorker, tp.tv_sec);

/* start the processing */

while(currentTemp >= END_TEMP)
{
for (counter = 0; counter < LOOPS; counter++)
{

/* compute the top left energy */

        newState = RandomNewState(disparities[startRow][0]);
        oldState = disparities[startRow][0];
        oldEnergy = ((abs(disparities[startRow][1] - oldState) +
           abs(disparities[startRow+1][0] - oldState) +
           abs(disparities[startRow+1][1] - oldState)) * LAMBDA) +
           abs(leftImage[startRow][0] - rightImage[startRow][oldState]);
        newEnergy = ((abs(disparities[startRow][1] - newState) +
           abs(disparities[startRow+1][0] - newState) +
           abs(disparities[startRow+1][1] - newState)) * LAMBDA) +
           abs(leftImage[startRow][0] - rightImage[startRow][newState]);

        deltaEnergy = newEnergy - oldEnergy;
           if (deltaEnergy <= 0)
              tempDisparities[startRow][0] = newState;
           else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/currentTemp))
              tempDisparities[startRow][0] = newState;
           else tempDisparities[startRow][0] = disparities[startRow][0];

/* compute the bottom left energy */

        newState = RandomNewState(disparities[stopRow-1][0]);
        oldState = disparities[stopRow-1][0];
        oldEnergy = ((abs(disparities[stopRow-1][1] - oldState) +
```

```
            abs(disparities[stopRow-2][0] - oldState) +
            abs(disparities[stopRow-2][1] - oldState)) * LAMBDA) +
            abs(leftImage[stopRow-1][0] - rightImage[stopRow-1][oldState]);
         newEnergy = ((abs(disparities[stopRow-1][1] - newState) +
            abs(disparities[stopRow-2][0] - newState) +
            abs(disparities[stopRow-2][1] - newState)) * LAMBDA) +
            abs(leftImage[stopRow-1][0] - rightImage[stopRow-1][newState]);

         deltaEnergy = newEnergy - oldEnergy;
            if (deltaEnergy <= 0)
               tempDisparities[stopRow][0] = newState;
            else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/currentTemp))
               tempDisparities[stopRow-1][0] = newState;
            else tempDisparities[stopRow-1][0] = disparities[stopRow-1][0];

         /* compute left side energy */

         for (row = 1; row < stopRow-2; row++)
            {
            newState = RandomNewState(disparities[row][0]);
            oldState = disparities[row][0];
            oldEnergy = ((abs(disparities[row-1][0] - oldState) +
               abs(disparities[row-1][1] - oldState) +
               abs(disparities[row][1] - oldState) +
               abs(disparities[row+1][1] - oldState) +
               abs(disparities[row+1][0] - oldState)) * LAMBDA) +
               abs(leftImage[row][0] - rightImage[row][oldState]);

            newEnergy = ((abs(disparities[row-1][0] - newState) +
               abs(disparities[row-1][1] - newState) +
               abs(disparities[row][1] - newState) +
               abs(disparities[row+1][1] - newState) +
               abs(disparities[row+1][0] - newState)) * LAMBDA) +
               abs(leftImage[row][0] - rightImage[row][newState]);

            deltaEnergy = newEnergy - oldEnergy;
            if (deltaEnergy <= 0)
               tempDisparities[row][0] = newState;
            else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/currentTemp))
               tempDisparities[row][0] = newState;
            else tempDisparities[row][0] = disparities[row][0];

            }

      /* compute top row energy */

      for (col = 1; col < COLS - D_MAX; col++)
      {
      newState = RandomNewState(disparities[startRow][col]);
      oldState = disparities[startRow][col];
      oldEnergy = ((abs(disparities[startRow][col-1] - oldState) +
        abs(disparities[startRow + 1][col-1] - oldState) +
      abs(disparities[startRow + 1][col] - oldState) +
      abs(disparities[startRow + 1][col+1] - oldState) +
      abs(disparities[startRow][col+1] - oldState)) * LAMBDA) +
```

```
abs(leftImage[startRow][col] - rightImage[startRow][col+oldState]);

newEnergy = ((abs(disparities[startRow][col-1] - newState) +
  abs(disparities[startRow + 1][col-1] - newState) +
abs(disparities[startRow + 1][col] - newState) +
abs(disparities[startRow + 1][col+1] - newState) +
abs(disparities[startRow][col+1] - newState)) * LAMBDA) +
abs(leftImage[startRow][col] - rightImage[startRow][col+newState]);

deltaEnergy = newEnergy - oldEnergy;
if (deltaEnergy <= 0)
tempDisparities[startRow][col] = newState;
else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/currentTemp))
tempDisparities[startRow][col] = newState;
else tempDisparities[startRow][col] = disparities[startRow][col];

}

/* compute bottom row energy */

for (col = 1; col < COLS - D_MAX; col++)
{
newState = RandomNewState(disparities[stopRow-1][col]);
oldState = disparities[stopRow-1][col];
oldEnergy = ((abs(disparities[stopRow-1][col-1] - oldState) +
  abs(disparities[stopRow-2][col-1] - oldState) +
abs(disparities[stopRow-2][col] - oldState) +
abs(disparities[stopRow-2][col+1] - oldState) +
abs(disparities[stopRow-1][col+1] - oldState)) * LAMBDA) +
abs(leftImage[stopRow-1][col] - rightImage[stopRow-1][col+oldState]);

newEnergy = ((abs(disparities[stopRow-1][col-1] - newState) +
  abs(disparities[stopRow-2][col-1] - newState) +
abs(disparities[stopRow-2][col] - newState) +
abs(disparities[stopRow-2][col+1] - newState) +
abs(disparities[stopRow-1][col+1] - newState)) * LAMBDA) +
abs(leftImage[stopRow-1][col] - rightImage[stopRow-1][col+newState]);

deltaEnergy = newEnergy - oldEnergy;
if (deltaEnergy <= 0)
tempDisparities[stopRow-1][col] = newState;
else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/currentTemp))
tempDisparities[stopRow-1][col] = newState;
else tempDisparities[stopRow-1][col] = disparities[stopRow-1][col];

}

/* compute the main grid energy */

for (row = startRow+1; row < stopRow-1; row++)
for (col = 1; col < COLS - D_MAX; col++)
{
newState = RandomNewState(disparities[row][col]);
oldState = disparities[row][col];
oldEnergy = ((abs(disparities[row-1][col-1] - oldState) +
```

```
  abs(disparities[row-1][col] - oldState) +
abs(disparities[row-1][col+1] - oldState) +
abs(disparities[row][col-1] - oldState) +
abs(disparities[row][col+1] - oldState) +
abs(disparities[row+1][col-1] - oldState) +
abs(disparities[row+1][col] - oldState) +
abs(disparities[row+1][col+1] - oldState)) * LAMBDA) +
abs(leftImage[row][col] - rightImage[row][col+oldState]);

newEnergy = ((abs(disparities[row-1][col-1] - newState) +
  abs(disparities[row-1][col] - newState) +
abs(disparities[row-1][col+1] - newState) +
abs(disparities[row][col-1] - newState) +
abs(disparities[row][col+1] - newState) +
abs(disparities[row+1][col-1] - newState) +
abs(disparities[row+1][col] - newState) +
abs(disparities[row+1][col+1] - newState)) * LAMBDA) +
abs(leftImage[row][col] - rightImage[row][col+newState]);

deltaEnergy = newEnergy - oldEnergy;
if (deltaEnergy <= 0)
tempDisparities[row][col] = newState;
else if (RANDOM_PROBABILITY < exp((double)-deltaEnergy/currentTemp))
tempDisparities[row][col] = newState;
else tempDisparities[row][col] = disparities[row][col];

}

/* move the temp disparities into the main array */

for (j = startRow; j < stopRow; j++)
for (k = 0; k < COLS; k++)
disparities[j][k] = tempDisparities[j][k];

}
currentTemp -= (currentTemp * TEMP_REDUCTION);
}

gettimeofday(&tp, &tzp);
printf("Stop %d %ld\n", theWorker, tp.tv_sec);

}  /* end ComputeDisparities */


int main(argc, argv)
int argc;
char **argv;
{
int i, j, count;
int workerID, row, cols;
struct timeval tp1, tp2, tp3;
struct timezone tzp1, tzp2, tzp3;
void *stat;
thread_t thr;
```

```
printf("Executable = %s\n", argv[0]);
printf("Rows = %d, Cols = %d\n\n", ROWS, COLS);

RandomizeMap(disparities);

gettimeofday(&tp1, &tzp1);
srand(tp1.tv_sec);

ReadImage(rightImage, RIGHT_IMAGE_NAME);

ReadImage(leftImage, LEFT_IMAGE_NAME);

gettimeofday(&tp1, &tzp1);

for (workerID = 0; workerID < WORKERS; workerID++)
thr_create(NULL, 0, ComputeDisparities, workerID, THR_NEW_LWP, NULL);

gettimeofday(&tp2, &tzp2);

for (workerID = 0; workerID < WORKERS; workerID++)
thr_join(0, &thr, &stat);

gettimeofday(&tp3, &tzp3);

printf("Start             %ld,    %ld\n", tp1.tv_sec, tp1.tv_usec);
printf("Threads created   %ld,    %ld\n", tp2.tv_sec, tp2.tv_usec);
printf("Done              %ld,    %ld\n", tp3.tv_sec, tp3.tv_usec);

CreateGrayScaleMap(disparities);

}  /* end real_main */
```

NO. OF COPIES	ORGANIZATION
2	ADMINISTRATOR DTIC ATTN DTIC DDA CAMERON STATION ALEXANDRIA VA 22304-6145
1	CDR USAMC ATTN AMCAM 5001 EISENHOWER AVE ALEXANDRIA VA 22333-0001
1	DIR USARL ATTN AMSRL OP SD TA 2800 POWDER MILL RD ADELPHI MD 20783-1145
3	DIR USARL ATTN AMSRL OP SD TL 2800 POWDER MILL RD ADELPHI MD 20783-1145
1	DIR USARL ATTN AMSRL OP SD TP 2800 POWDER MILL RD ADELPHI MD 20783-1145
2	CDR US ARMY ARDEC ATTN SMCAR TDC PCTNY ARSNL NJ 07806-5000
1	DIR BENET LABS ATTN SMCAR CCB TL WATERVLIET NY 12189-4050
1	DIR USA ADVANCED SYSTEMS R&A OFC ATTN AMSAT R NR MS 219 1 AMES RESEARCH CENTER MOFFETT FLD CA 94035-1000
1	CDR US ARMY MICOM ATTN AMSMI RD CS R DOC RDSTN ARSNL AL 35898-5010
1	CDR US ARMY TACOM ATTN AMSTA JSK ARMOR ENG BR WARREN MI 48397-5000

NO. OF COPIES	ORGANIZATION
1	DIR USA TRADOC ANALYSIS CMD ATTN ATRC WSR WSMR NM 88002-5502
1	CMDT US ARMY INFANTRY SCHOOL ATTN ATSH CD SECURITY MGR FT BENNING GA 31905-5660
	ABERDEEN PROVING GROUND
2	DIR USAMSAA ATTN AMXSY D AMXSY MP H COHEN
1	CDR USATECOM ATTN AMSTE TC
1	DIR USAERDEC ATTN SCBRD RT
1	CDR USACBDCOM ATTN AMSCB CII
1	DIR USARL ATTN AMSRL SL I
5	DIR USARL ATTN AMSRL OP AP L

NO. OF COPIES	ORGANIZATION
	ABERDEEN PROVING GROUND
46	AMSRL CI
	W MERMAGEN
	J BLAKE
	R LODER
	W STUREK
	N BOYER
	AMSRL CI A
	H BREAUX
	M A ORTWEIN
	AMSRL CI AA D TOWSON
	AMSRL CI AC
	D PRESSEL
	D ZOLTANI
	AMSRL CI AD
	P DYKSTRA
	J GROSH
	T KENDALL
	AMSRL CI C
	B BROOME
	J DUMER
	AMSRL CI CA A CELMINS
	AMSRL CI CB M HIRSCHBERG
	AMSRL CI CC
	D GWYN
	G HARTWIG
	R KASTE
	AMSRL CI CD J GANTT
	AMSRL CI CI N PATEL
	AMSRL CI S
	A MARK
	M BIEGA
	B BODT
	B CUMMINGS
	K FICKIE
	C HANSEN
	P HARNDEN
	E HEILMAN
	R HELFMAN
	V KASTE
	M MARKOWSKI
	M S ORTWEIN
	T PURNELL
	T ROHALY
	D SHIRES
	K SMITH
	M TAYLOR
	M THOMAS
	J WALL
	AMSRL CI SA
	S CHAMBERLAIN
	B COOPER
	A DOWNS
	AMSRL CI T R ROSEN
	AMSRL WT PB E BAUR

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number ARL-TR-667 Date of Report January 1995

2. Date Report Received _____

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) _____

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.) _____

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. _____

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) _____

CURRENT ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

OLD ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

(Remove this sheet, fold as indicated, tape closed, and mail.)
(DO NOT STAPLE)

DEPARTMENT OF THE ARMY

OFFICIAL BUSINESS

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO 0001, APG, MD

Postage will be paid by addressee

Director
U.S. Army Research Laboratory
ATTN: AMSRL-OP-AP-L
Aberdeen Proving Ground, MD 21005-5066