SIMULATION OF A CONNECTIONIST STEREO ALGORITHM ON A SHARED-MEMORY MULTIPROCESSOR

by

Charles V. Stewart
and
Charles R. Dyer

Computer Sciences Technical Report #760

March 1988

Simulation of a Connectionist Stereo Algorithm on a Shared-Memory Multiprocessor

Charles V. Stewart

Charles R. Dyer

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract

This paper presents results on the simulation of a connectionist network for stereo matching on a shared-memory multiprocessor. The nodes in the network represent possible matches between points in each image. Because we only consider matches between intensity edges, only a few of these nodes actually represent candidate matches for a given pair of images; consequently, only the active part of the network is ever constructed by the simulation. This includes the nodes representing candidate matches and the connections between these candidate matches as defined by a variety of constraints. Thus, the simulation involves constructing a list of matches and a list of connections, and simulating the iterations of the network. Each of these phases can be partitioned among a number of processors with very little interference between the processors due to synchronization or mutual exclusion. The resulting parallel implementation produced nearly linear speed-ups for up to the nine processors in our system.

The support of the National Science Foundation under Grant No. DCR-8520870 and NASA under Grant No. NAGW-975 is gratefully acknowledged.

1. Introduction

Connectionist networks are a significant new model of computation for computer vision research. Motivated by the computing organization of the brain, connectionist networks are characterized by a large number of simple computing elements (nodes) and a massive interconnection network between the nodes. At present, these networks must be simulated in order to test them empirically. On a serial computer these simulations are cumbersome since each update to the activation of one node requires gathering outputs from neighboring nodes, scaling each output, and summing the scaled values. This process is repeated for each node in the network at each iteration. Thus, because of their simple, regular computational requirements, these simulations are obvious candidates for parallel implementation.

In this paper we investigate the parallel simulation of a connectionist network that is based on the General Support Algorithm for stereo vision.[7] The Sequent Symmetry S81® was used to implement and test the simulation. In stereo vision there are two cameras at different positions recording images of a scene. In the stereo matching problem, pairs of points must be identified in the two images that correspond to the same feature in the scene. The difference between the positions of these points may then be used to obtain depth information about the underlying scene feature. The General Support Algorithm solves the stereo matching problem using a connectionist network model of computation. Each node in the network represents a possible match between pairs of points in the images. Through repeated updates to the activations of the nodes in the network, the correct matches are identified. These are the nodes with relatively large activations when the iterations have been completed. The motivation, design and sequential simulation results of this algorithm are described in Stewart.[7]

The Sequent Symmetry is a true shared-memory multiprocessor. It contains a number of identical 32-bit microprocessors, a high speed data bus, and a single common memory.

Symmetry and DYNIX are registered trademarks of Sequent Computer Systems, Inc. UNIX is a registered trademark of AT&T

Contention for the bus is reduced by 16 kilobyte caches that are local to each processor. The caches are write-through, i.e. when one processor writes to a shared variable stored in a cache, that write is echoed to main memory. This invalidates all other copies of the variable that might be stored in other caches. More precisely, cache locations are allocated in small memory blocks so that writes to any part of a block invalidates all copies of the block in other caches.

The software environment of the Symmetry system is controlled by the DYNIX operating system, a distributed version of UNIX. There are two forms of parallelism available on the system, function partitioning and data partitioning. Function partitioning involves simultaneous activation of a set of distinct procedures; data partitioning involves multiple activations of the same procedure. The most common instance of data partitioning occurs when a group of identical procedures divides up the iterations of a large loop. In doing so there are two methods for controlling the division of work. First, in static scheduling the iterations of the loop are partitioned *a priori*. Second, in dynamic scheduling each process works on an iteration (or group of iterations) and, upon completion, allocates more iterations from a central list. Data partitioning requires more overhead, but it may be preferable for two main reasons: it accommodates variations in the processing times of the individual iterations and it automatically adjusts the workload when there is contention for the processors from other programs. As a final feature of the software environment, synchronization and mutual exclusion in the DYNIX system are realized using system primitives built on top of semaphores.

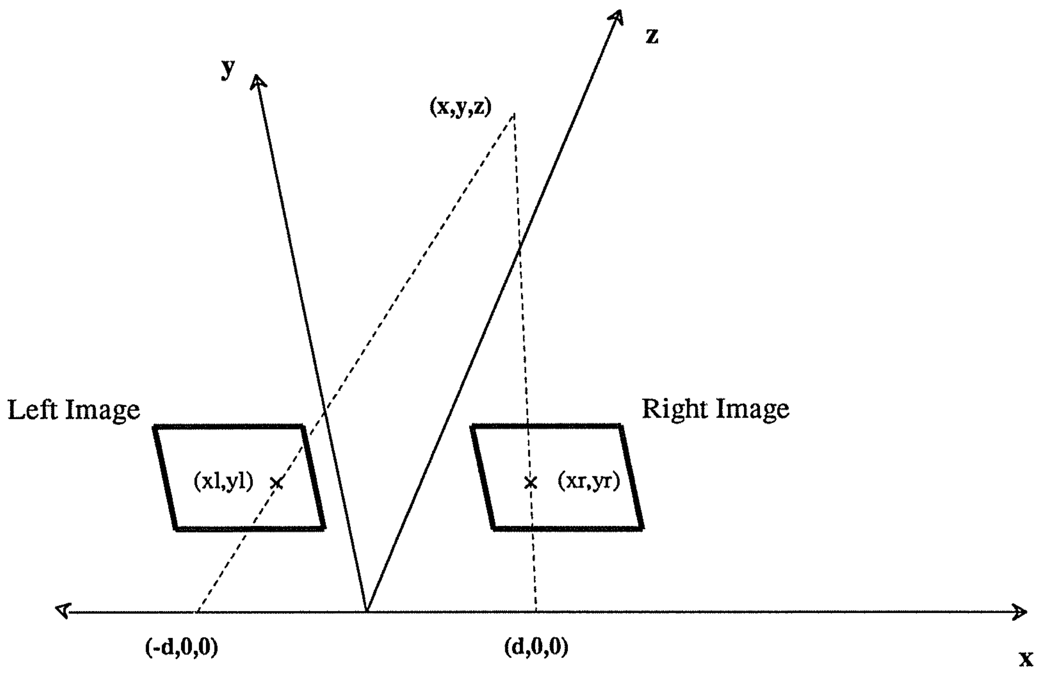
In the remainder of this paper we describe the parallel simulation of the General Support Algorithm (GSA) in detail. Section 2 outlines the connectionist network design of the GSA. Section 3 describes a simulation based on this network. Section 4 analyzes the parallel implementation of the simulation. Finally, Section 5 discusses the results of testing the parallel simulation on a Sequent Symmetry system with 10 processors.

2. The General Support Algorithm

In this section we define the stereo matching problem and outline the General Support Algorithm, including the details of its connectionist implementation.

2.1. Stereo Matching Problem

There are several stages to the binocular stereo problem. These include: (1) determining the imaging geometry (we assume that this is known), (2) detecting features in each image for matching, (3) finding the matches between these features (the *correspondence problem),* and (4) interpreting the results. The focus here is the correspondence problem. Figure 1 shows the typical model of stereo imaging geometry. The cameras are assumed to have point lenses and the images are formed through perspective projection. The lines of sight of the cameras are normal to the image plane, and are parallel to the z-axis in the $x-z$ plane. This model provides an important simplification that is used in many stereo algorithms: a given point (x_0, y_0) in one image can only match points (x_1, y_0) in the other image. That is, the search for matches for a



given point is limited to *the same row* in the other image. This is the *horizontal epipolar scanlines* assumption.

As described above, the matching process involves finding features in each image and matching them with the other image. The most common matching features, and the ones used in the General Support Algorithm, are oriented edges. These edges are often detected and matched at multiple resolutions. At coarse resolutions there are fewer edges, so matching is often less ambiguous. At fine resolutions there are more edges and hence more possible matches, but the position information at fine resolutions is more accurate.

The *disparity* of a match is the difference between the positions of the features in the two images. Matches with large disparities indicate scene features closer to the cameras.

The selection of the valid matches from among the candidates has been studied by numerous researchers. Most frequently they have proposed *constraints* to assist in this process. These constraints depend on assumptions about the scene and its appearance in the two images. Groups of matches meeting the requirements of the constraints are favored over other matches. The GSA integrates the influence of a number of constraints. In the algorithm, the constraints are defined as local interactions between pairs of matches. In addition, with one exception, these constraints define only *positive* interactions between matches.

2.2. Connectionist Implementation of the GSA

The General Support Algorithm is implemented in a hierarchical connectionist network. The lower levels of the network perform multiresolution edge detection and determine candidate matches. The highest level is the matching network. In this network each node represents a distinct possible match between a pair of image coordinate points. For a given pair of input images, only a small percentage of the match nodes will be activated because edge detection produces a sparse set of results. Connections between pairs of nodes in the matching network are defined by the various constraints used.

A number of constraints are implemented in the network. The only one having negative influence on a match node is the **uniqueness** constraint.[3] Generally, uniqueness states that there can be at most one valid match for each edge. In our formulation of it, there are two uniqueness influences on a given match between a pair of edges, one from each image. One is the maximum of the other matches for the edge from the left image. The other is the maximum of the other matches for the right image edge.

The **detailed match**[1] constraint provides additional initial activation to those matches where the areas around the edges in the two images have similar intensities. It does not require any connections between candidate matches.

The remainder of the constraints define connections between pairs of candidate matches. **Multiresolution**[2] specifies relations between matches at adjacent resolution levels. In our usage, there are two forms of multiresolution support. In *coarse-to-fine* multiresolution, a match at one resolution level can support a match at the next finer resolution when the two matches have nearly the same position and disparity. In *fine_to_coarse* multiresolution, support occurs in the opposite direction. The weights of these connections are defined by the parameters *BASE_COARSE_TO_FINE* and *BASE_FINE_TO_COARSE*, respectively.

Figural continuity[4] defines support between pairs of nearby matches at the same resolution that might be along the same contour in the two images. The weight of support is given by *BASE_FC / distance*, where *distance* is the distance between the matches.

The **disparity gradient**[5, 6] defines support between a pair of matches as follows. Let p_1 and p_2 be two candidate matches with disparities $d(p_1)$ and $d(p_2)$, respectively. Then these matches meet the disparity gradient requirement if

$$\frac{|\,d(p_1) - d(p_2)\,|}{D(p_1,p_2)} \leq 1$$

where $D(p_1,p_2)$ is the distance between the matches, $D(p_1,p_2) \leq K$, for constant $K \approx 10$. To determine the connection weight for the disparity gradient, let $d_{1,2} = |\,d(p_1) - d(p_2)\,|$. Then the weight is given by:

$$\frac{BASE\ DG}{D\,(p_1,p_2)} * \frac{c}{(d_{1,2}+c)}$$

where $c \geq 1$ is a constant.

The connections define relations between nodes in the matching network. The nodes have real-valued activations and outputs. The computation in the network involves iterative changes to these values. The equations defining the node activation and output functions in the connectionist network are as follows. The activation of a node depends on the combined influences of the supporting constraints, uniqueness, decay and the node's prior activation. The supporting input to a match is a weighted, linear combination of the constraint input, given by:

$$I_i = \sum_{j=1}^{N_i} O_j\, w_{ji}$$

where the O_j are outputs from other match nodes, and the w_{ji} are the connection weights. Uniqueness input is given by:

$$B_i = \beta \max(O_j \mid j \in Left_i) + \beta \max(O_k \mid k \in Right_i)$$

where $Left_i$ is the set of competing matches for the left edge of match i, and $Right_i$ is the set of competing matches for the right edge. The new activation is:

$$A_i = (1-\delta)\, A_i + I_i + B_i$$

where δ is a decay parameter. Decay contributes an additional negative influence that forces a match to have continued support throughout the network's iterations. The output, O_i is simply a threshold function of the activation:

$$O_i = \begin{cases} A_i, & A_i \geq \phi \\ 0, & A_i < \phi \end{cases}$$

where ϕ is a threshold parameter. The activation is limited to the range [-1..1], and the output is allowed to be in the range [0..1]. When the activation of a node reaches 1.0 it is said to be *saturated*, and it remains at 1.0 for the duration of the matching procedure.

3. Simulation of the General Support Algorithm

The connectionist network described above is extremely large and therefore quite expensive to simulate directly. Our encoding scheme[8] requires $L\, k\, N^2$ nodes, where L is the number of resolution levels, k is the number of discrete disparity values, and the images are each of size N by N. In addition, each node has a large number of connections. However, for a given pair of images, the nodes that actually represent candidate matches are sparsely distributed. We use this observation to structure the simulation as follows. For each pair of images we construct only the active part of the network. This contains only those nodes representing candidate matches and only those connections between pairs of candidate matches. The result is subnetwork that is functionally isomorphic to the entire network for the given pair of images.

The simulation involves two phases of processing: the first phase builds the three main data structures, and the second phase simulates the iterations of the network. The data structures consist of the edge images (note that we do not consider edge detection as part of the simulation), the list of candidate matches (i.e. the match nodes), and a list of connections between these matches. During the iteration phase of the simulation, the network alternately and synchronously updates all nodes' activations and outputs. This process repeats either for a fixed number of iterations or until a small percentage of the nodes have an intermediate output value (e.g. $0.25 \leq O_i \leq 0.75$).

The three main data structures are shown in Figure 2. The edge images are represented as two-dimensional arrays. Those positions having no edges are represented as null entries. Each entry contains a pointer to a list of candidate matches for that edge (the "Match ptrs" in Figure 2).

The match nodes list actually consists of three separate lists: the node activations ("A" in Figure 2), the node outputs ("O" in Figure 2), and the descriptions of the nodes. Each description includes the image coordinates for the edges in the match and a pointer to the list of input

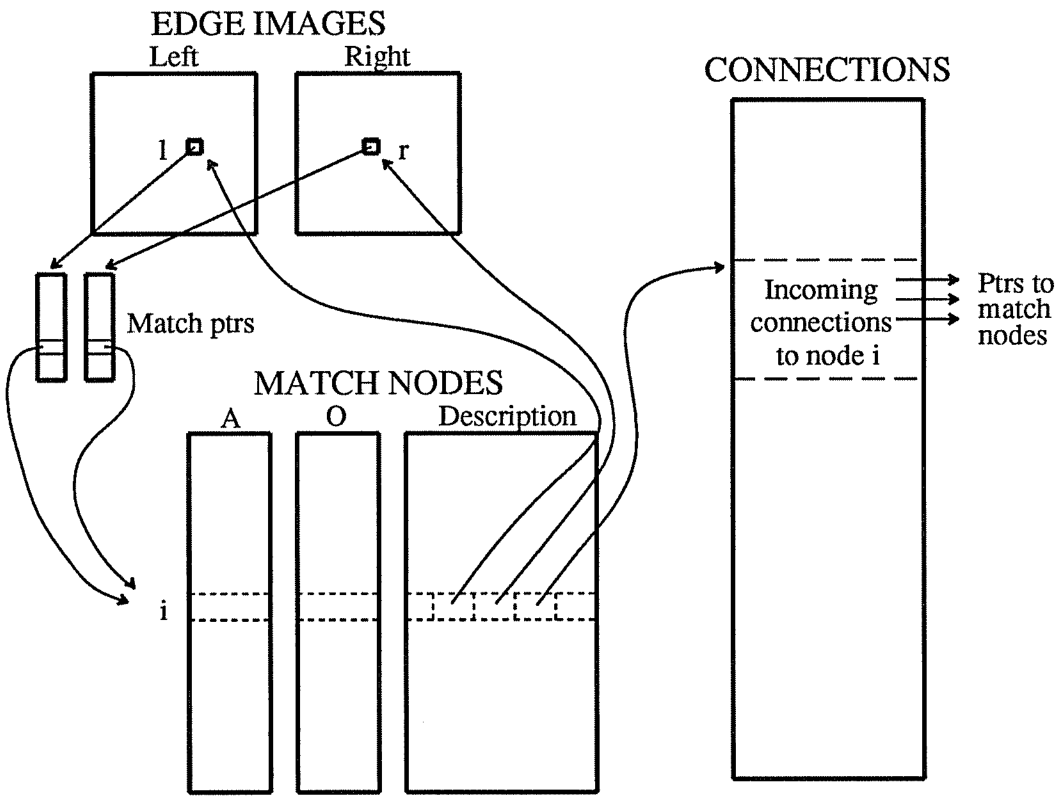


Figure 2. Diagram of the three main data structures used in the simulation. See the text for discussion.

connections to the match. Candidate matches are determined by searching the edge images. When a candidate is identified, a location in the match node list is allocated and the initial activation and output of the match are computed. In addition, the match is added to the linked list of matches for both the left and right edges. This list facilitates the uniqueness computation.

The third data structure, the connections list, requires the most space. Each entry in this list contains the weight of the connection and the index of the output match node. Empirically, we found that there are approximately 150 active connections per match. This varies, of course, with the density of edges in the images. Determining the active incoming connections to a given match requires examining both the edge images and match list for other matches that meet the requirements of a given constraint. Fortunately, many of the time consuming computations, such as the distance functions and the weight values in the disparity gradient, can be precomputed and stored in a look-up table. When the list of input connections to a match is complete, space for the connections is allocated in a contiguous block in the connections list.

After the three data structures are built, the simulation of the network iterations begins. At each iteration the new activations of all of the nodes are computed. Once this completes these activations are used to compute the new outputs from the nodes. During the activation computation for a given match, the previous activation for the match, the list of its connections in the connections array, and the list of competing matches for both the left and right edges are needed. The list of connections is used to access various locations in the output list to compute the weighted input to a match. During the output computation of an iteration, the activation of a node is read and then its new output value is stored. Other details of the match description are not needed during the output computation.

4. Parallel Implementation

The simulation described in the previous section was designed so that it could be implemented in parallel. In this section we describe the details of this implementation including: synchronization and mutual exclusion among the parallel loop iterations, static vs. dynamic allocation of the iterations, the number of consecutive loop indices allocated by each processor at once, and some additional hardware contention issues. In making the choice between static and dynamic allocation for a given loop of the simulation, and in choosing the number of iterations allocated at once, we used simulation results for a test random-dot stereogram. After the best choices were identified we measured the speed-up of the parallel implementation compared to the sequential version of the simulation. In the remainder of this section we consider each of the above issues beginning with the parallel structuring of the individual parts of the simulation.

During the input of the edge image, a separate process is created for each resolution level for each of the left and right images. The edge data for each of these images reside in separate files. Thus, there are $2L$ different processes created, where L is the number of resolution levels.

The duration of these processes will vary according to the number of edges in a given input file, but since input is sequential, no further division of work is possible. When there are fewer than $2L$ processors available, each processor must handle multiple input images sequentially.

The construction of the match list is not as straightforward as the image input. As noted above, space to store the matches is allocated from a shared array. In a parallel implementation, multiple processes need to allocate matches simultaneously. In addition, the linked list of matches for a given edge must be updated for each match node. Both of these issues can potentially cause significant problems because each requires some form of mutual exclusion. For the most part, however, these problems can be avoided by making use of the epipolar scanline assumption discussed in Section 2. That is, because all matches for the edges in a row must be in the same row in the other image, one processor will locate all of the matches for the row. The processor finds the candidate matches and stores them in its local data area. When all of the matches have been found within a row, the processor allocates enough contiguous space in the shared match list and then copies the local data into this list. Note that while copying, mutual exclusion is unnecessary. This allocation method reduces contention for access to the match list because large blocks are allocated at once. It also completely eliminates the need for mutual exclusion in updating the linked list of matches for each edge because all of the matches for an edge are found and stored sequentially by a single process.

The parallel construction of the connections list works in much the same way as the construction of the match list. One process finds all of the connections for a single match node. In doing so it temporarily stores these connections in a local buffer. After it finds all of the connections for the node, it allocates space in the global connections list and copies the local data into this list. This method has the additional advantage that the connections for a node are always stored consecutively in the global list.

Once the data structures are initialized in shared memory, the simulation of network iterations begins. During the iterations there is no need for mutual exclusion among the parallel processors. In the activation computation for a node, a processor must reference the outputs of a group of other nodes. However, during the activation computation, the node outputs are *read-only*, so mutual exclusion in referencing them is unnecessary. During the output computation for a node, only the activation and output values of the node are referenced, so mutual exclusion is unnecessary here as well.

Synchronization does not cause significant delays at any part of the simulation. It must occur after each phase of building the data structures, between each iteration of the network, and between the activation and output computations of each iteration. The synchronization between activation and output computations is realized using a DYNIX program primitive called a "barrier" that all processes must reach before any of them proceeds to the output computation. In all other cases synchronization is handled implicitly. This occurs because processes initiated in parallel from a DYNIX procedure must all end before that procedure may continue.

Next, consider the issues of dynamic vs. static allocation of loop iterations and loop allocation size. Generally, in the tests we ran, there was little difference between the different options. (The results below show the average time using nine processors when there were no other users active on the system.) Clearly, there is no need for dynamic allocation of the output update computation during the network iterations. In each of the other parts of the computation the time required for each iteration varied, but this did not greatly affect the performance of the simulation. For the match list construction, dynamic allocation in small blocks (2 rows at a time) was nearly the same as static allocation (1.58 seconds vs 1.53 seconds). For the construction of the connections list, dynamic allocation of single iterations was slightly faster than static allocation (36.83 seconds vs. 37.35 seconds). The performance using dynamic allocation was improved by increasing the number of loop iterations allocated at once each processor to 8 (36.40 seconds). Finally, the activation computation was only marginally improved by dynamic allocation of larger blocks, i.e. static allocation required 25.02 seconds whereas 24.85 seconds were required for dynamic allocation of blocks of 4 iterations. Thus, in our experiments, although dynamic allocation improved the performance of the algorithm slightly, the differences were not very significant.

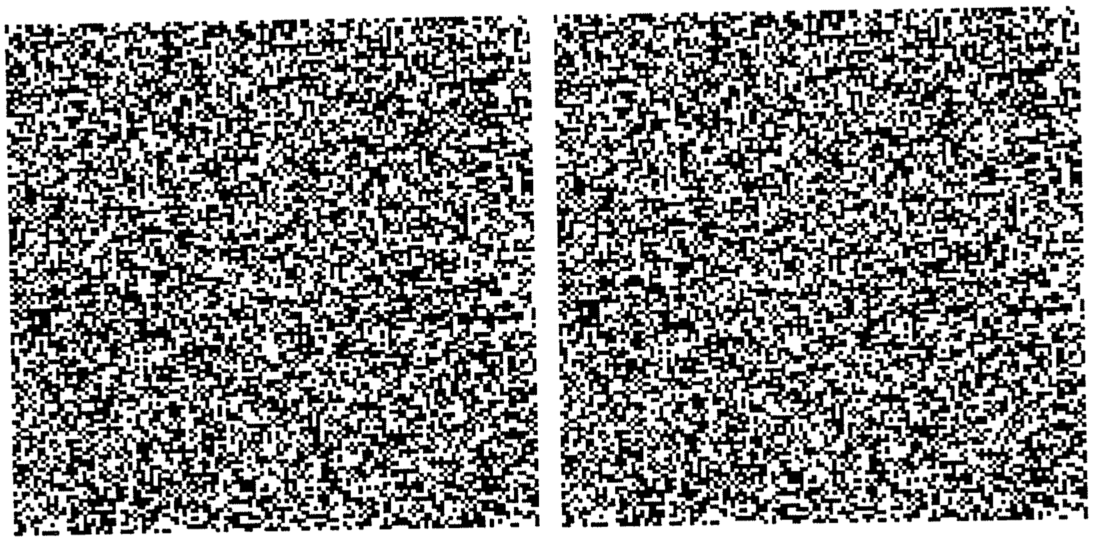
As a final consideration in the parallel simulation, the major potential hardware problem involves contention for the main memory bus. (When there are multiple users, main memory allocation and paging may be issues, but we do not consider them here.) The caches local to each processor are designed to alleviate this somewhat. However, the simulation of the GSA is significantly larger than the combined size of the caches (the example used here required approximately 16M bytes). This implies that there is a potential for bus contention when a large number of processors is used. As will be shown in the next section, this problem did not occur for our experiments. With larger numbers of processors this may become a problem.

5. Results

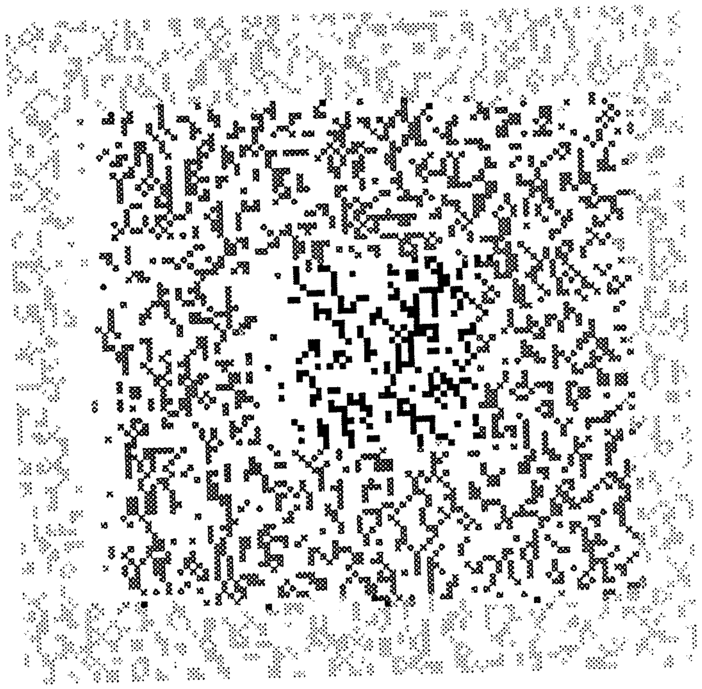
This section presents the results of the parallel simulation on the random-dot stereogram shown in Figure 3. Each image was 128 by 128 pixels and represented three different surfaces at disparities 0, 6 and 12. (Disparity 12 is closest to the viewer, and is represented by the darkest points in Figure 3b.) There were 11,090 active nodes and 1,637,570 active connections for this stereogram. The algorithm produced 97% correct matching decisions.

Our version of the Sequent Symmetry configuration used 10 Intell 80386 processors linked via a bus and sharing common memory. The caches local to each processor contain 16 kilobytes, and the main memory was large enough to hold the entire simulation (> 16 megabytes). All but one of the processors can be allocated for parallel execution.

Our results show the speed-ups achieved in executing the simulation using different numbers of processors, ranging from 1 to 9. For each number of processors, the simulation was run three times with no other active users on the system, and the results were averaged. The times for the simulation were separated into times for input, match list building, connections list building, and the time to run 10 iterations of the network. These are shown in Table 1. The



(a) Input stereogram



(b) Disparity image results for the finest resolution and the left image (disparity is encoded as intensity).

Figure 3. Random-dot stereogram with three layers. The darkest points in the disparity image (b) represent points that are closest to the viewer.

Simulation Times for N Processors					
Num	Phases of the Simulation				Total
Processors	Input	Match	Connections	Iterations	Time
1	17.43	4.54	297.3	213.9	533.1
2	11.87	2.67	150.8	105.6	270.9
3	9.23	1.98	101.6	71.2	184.0
4	7.67	1.68	76.8	54.2	140.5
5	6.56	1.54	62.1	44.6	114.8
6	5.13	1.48	52.3	36.3	95.2
7	5.14	1.47	45.4	31.5	83.5
8	5.21	1.49	40.6	27.9	75.2
9	5.15	1.53	36.4	25.0	67.0

Table 1. Timing results for the parallel simulation.

results are graphed for building the match list, building the connections list, and simulating the network iterations in Figures 4, 5 and 6, respectively. Figure 7 shows the overall speed-up obtained. In each figure, the dark curve shows the actual times and the broken lines shows the optimal speed-ups. Speed-ups are measured in terms of the effective number of processors. That is, for N processors, the effective number of processors is $T(1)/T(N)$, where $T(i)$ is the execution time when using i processors.

As shown in Figure 4, the match list building did not approach the optimum speed-up, and performance actually degraded slightly after 7 processors. Since the loops of match list building were relatively independent, the only likely explanation is that the speed-up was limited by the non-trivial time required to initiate processes. With images requiring more active match nodes, the allocation will require more time and so speed-ups will probably continue to be obtained beyond 7 processors. Due to current memory size restrictions we have been unable to test larger images.

The speed-ups for building the connection list and for the iterations of the network were both almost optimal. This is encouraging since these steps required by far the greatest percentage of the total time. Building the connections list required 56% of the sequential processing time, and 54% when 9 processors were used. Ten iterations required 40% and 37% of the time respectively.

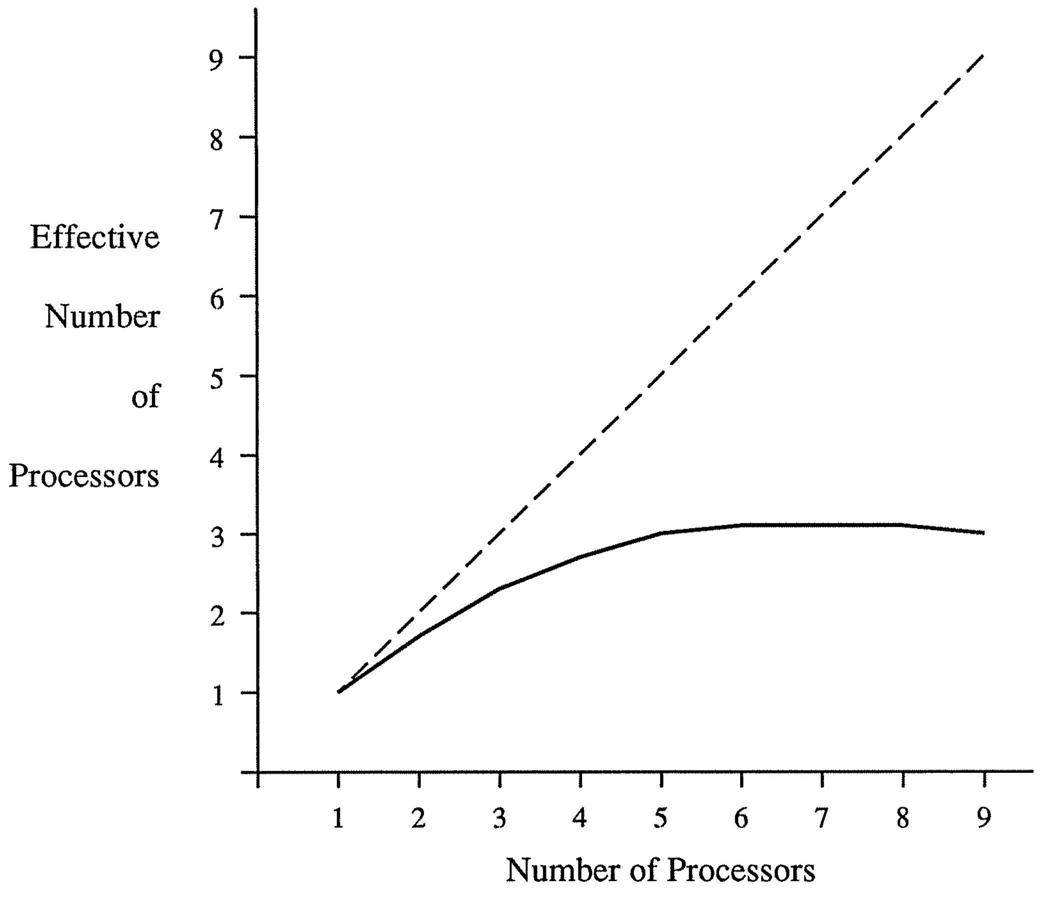


Figure 4. Speed-ups obtained for building the list of candidate matches. The dark curve shows the actual speed-ups and the broken lines shows the optimal speed-ups. Speed-up is measured in terms of the effective number of processors. That is, for N processors, the effective number of processors is $T(1) / T(N)$, where $T(i)$ is the execution time when using i processors. Note that the best processing time is obtained for 7 processors. See the text for the discussion of this.

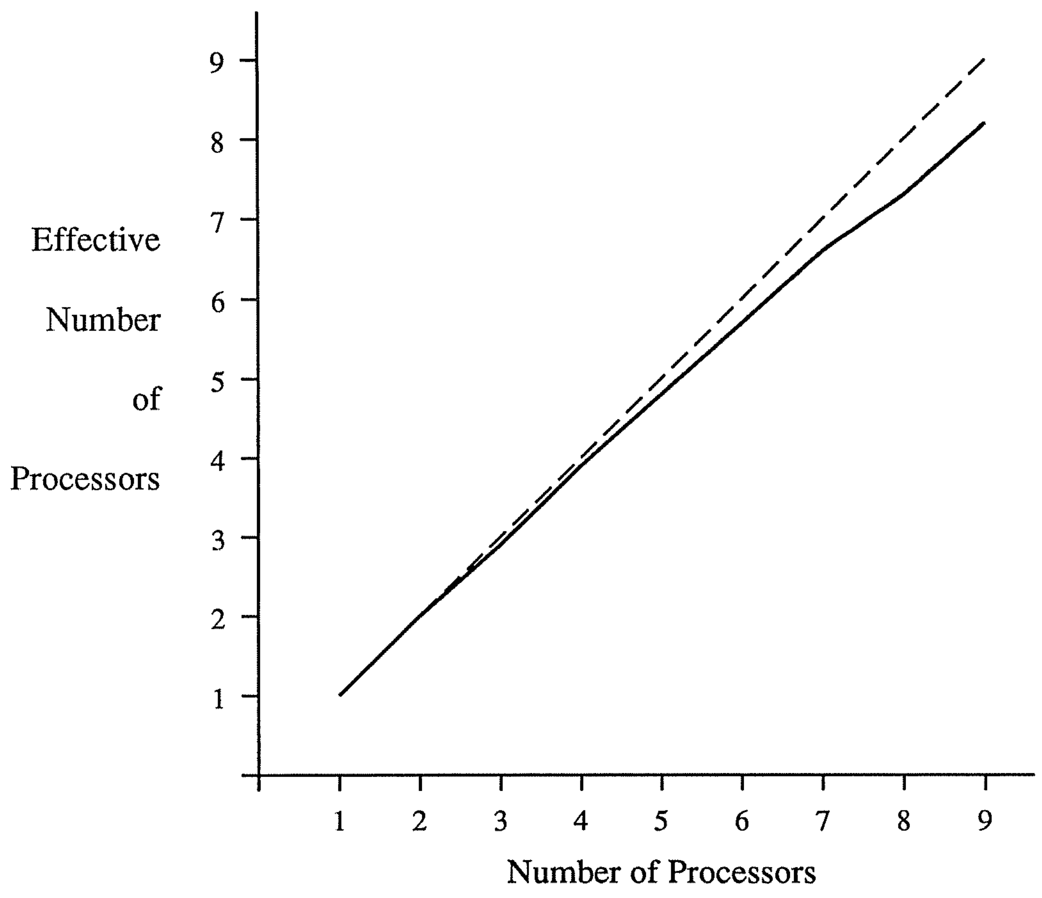


Figure 5. Speed-ups obtained for building the connections list.

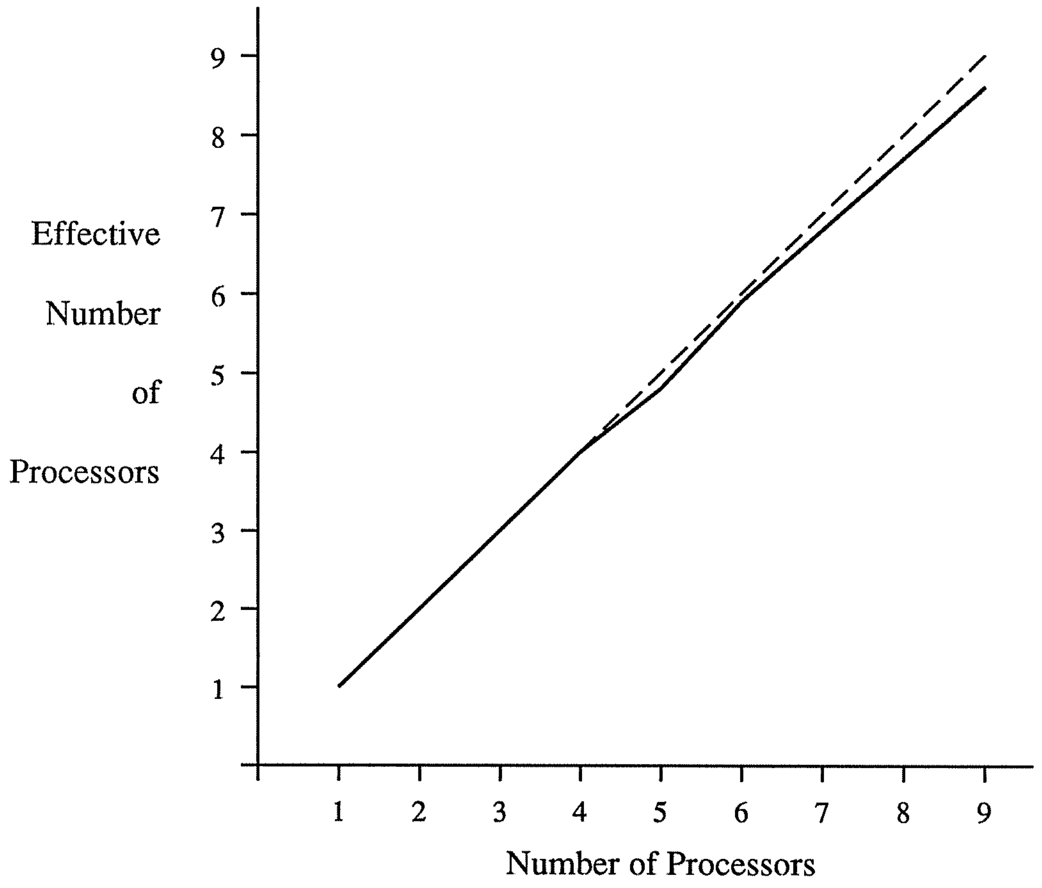


Figure 6. Speed-ups obtained for the iterations of the network.

Because the connection list building and the iterations dominate the processing time, the parallel speed-up for 9 processors was nearly optimal. Increasing the number of processors is likely to continue improving the performance. The decrease in time due to increases in the number of processors may flatten for large *N* because of either: (1) contention for the memory bus, or (2) mutual exclusion in allocating space in the shared memory. Based on our data, it is not possible to predict the point at which this will occur.

In summary, the results presented here have shown the effectiveness of a parallel implementation of the simulation of the General Support Algorithm, a connectionist network for stereo vision, on a Sequent Symmetry S81 shared-memory multiprocessor. This was realized by careful development of the simulation so that there was little contention in terms of mutual

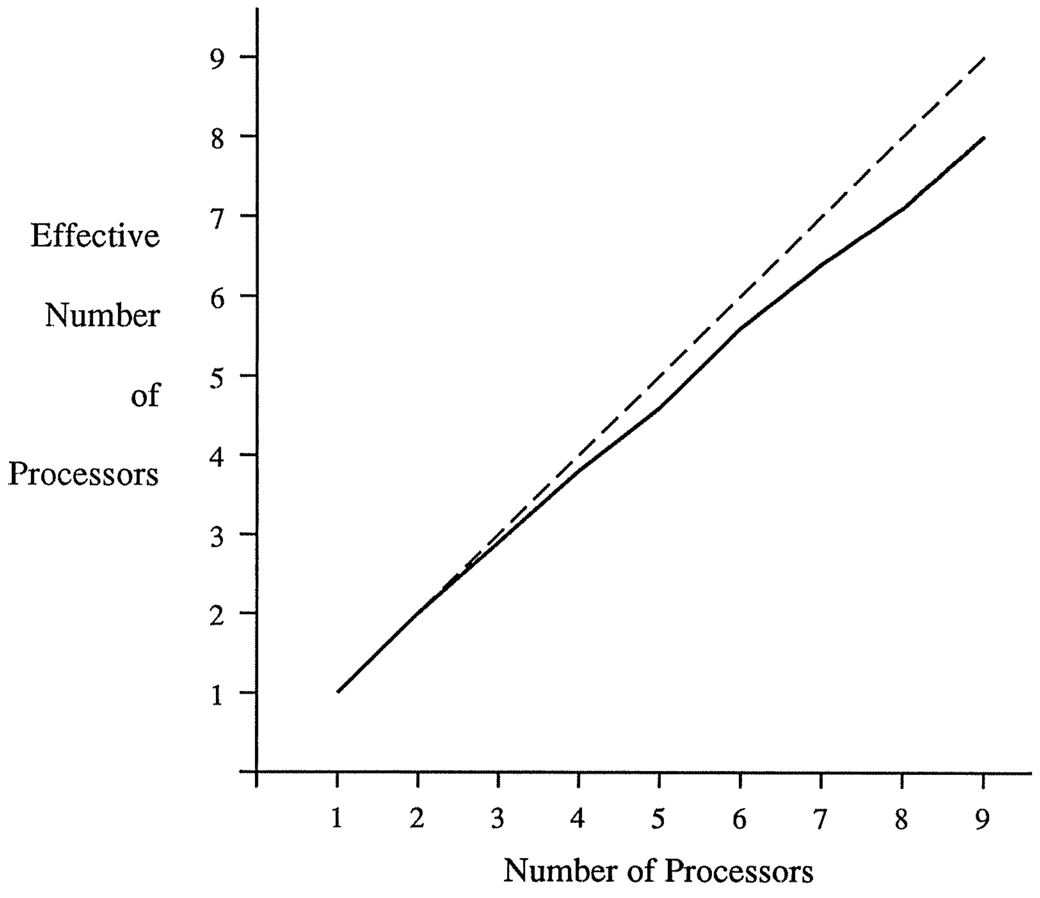


Figure 7. Speed-ups obtained for the entire simulation.

exclusion and synchronization. This performance shows that the study of connectionist network vision algorithms can be greatly facilitated through parallel simulation on shared-memory multiprocessors.

References

1. H. H. Baker and T. O. Binford, ''Depth from edge and intensity based stereo,'' *Proc. Seventh Int. Joint Conf. on Artificial Intelligence*, pp. 631-636 (1981).
2. D. Marr and T. Poggio, ''A computational theory of human stereo vision,'' *Proc. Royal Society of London, B.* **204** pp. 301-328 (1979).
3. D. Marr, *Vision,* W.H. Freeman and Company, New York (1982).
4. J. E.W. Mayhew and J. P. Frisby, ''Psychophysical and computation studies towards a theory of human stereopsis,'' *Artificial Intelligence* **17** pp. 349-385 (1981).
5. S. B. Pollard, J. E.W. Mayhew, and J. P. Frisby, ''PMF: a stereo correspondence algorithm using a disparity gradient limit,'' *Perception* **14** pp. 449-470 (1985).
6. K. Prazdny, ''Detection of binocular disparities,'' *Biological Cybernetics* **52** pp. 93-99 (1985).
7. C. V. Stewart and C. R. Dyer, ''Local constraint integration in a connectionist model of stereo vision,'' *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, (1988).
8. C.V. Stewart, ''Connectionist models of stereo vision,'' PhD Disseration, University of Wisconsin, Madison, WI (1988).