A HIERARCHICAL, PARTITIONABLE, KNOWLEDGE BASED PARALLEL PROCESSING SYSTEM

N. Alexandridis
H-A Choi
B. Narahari
S. Rotenstreich
A. Youssef
Department of Electrical Engineering and Computer Science
The George Washington University
Washington, DC 20052

Abstract
The performance of a parallel processing system is governed by both the task characteristics and the architecture characteristics. Since many tasks share common characteristics it is feasable to extract and integrate the knowledge of task and architecture characteristics, thus improving the performance of the system. This is one of the primary motivations behind this research. The objective is twofold: to ease the burden of parallel programming on the user and, to use the knowledge of task characteristics towards improving mapping, partitioning and scheduling of the tasks onto a parallel machine. To this end, we use the concepts of parallel high level primitives and a *primitives table* that serves as the knowledge base for the system. We propose a model of a hierarchical partitionable architecture that would efficiently support our system. The run-time characteristics stored in the knowledge base can be used by the system for mapping and scheduling the algorithm, and partitioning the architecture. We discuss how the knowledge base is used for mapping, scheduling, and execution of parallel algorithms on the parallel processing system while reducing the burden on the user.

§1. Introduction

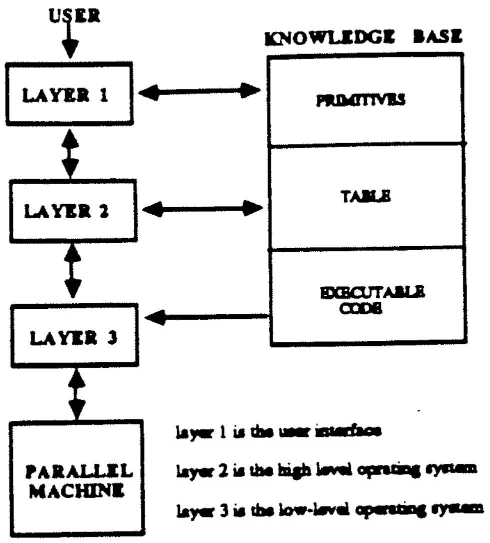
Large scale Parallel processing systems provide significant speed-ups for many applications, particularly for many image processing and scientific applications which are typically computation intensive. While parallel processing reduces the execution time of many algorithms, the performance depends on many factors that do not arise in sequential programming. These include configuration of the underlying parallel architecture, parallelization of the algorithm, matching algorithms to the architecture, and partitioning the system. These issues, and therefore the performance of the system, are governed by both the task characteristics and the architecture characteristics.

Task characteristics include the number of processors required, the complexity of the basic operations, communication pattern, data structure, and type of parallelism. Architecture characteristics include the number and type of processors in the system, the topology of the interconnection networks, memory organization, and synchronous (SIMD) or asynchronous (MIMD) mode of operation. Therefore, as noted in [8], there is a need to match the task and architecture characteristics in order to improve the performance of the system. For example, if the algorithm exhibits parallelism at the data level (i.e., a large amount of data requiring similar and simple computation) then an SIMD architecture with a large number of simple processing elements would be an efficient architecture for the problem. Furthermore, if the algorithm uses a two-dimensional array as its data structure and the communication is between adjacent array points, then a two-dimensional Mesh would be an efficient interconnection network for such an algorithm. Examples of algorithms with these characteristics are image smoothing and two-dimensional convolution of an image [16]. As illustrated by the example, an integration of task and architecture characteristics would improve the performance of a parallel processing system. Such an integration is feasible because a close examination of specific application domains, such as image processing and scientific computing, would show that many tasks share common characteristics. This is one of the primary motivations of our research.

Some high level tasks may be decomposed into several parallel sub-tasks, where each sub-task may have a different processing requirement and can be executed in parallel. Many image understanding tasks exhibit this property. Tasks with such a property can be efficiently implemented on partitionable architectures. A partitionable parallel processing system consists of a pool of processors, controllers and common system resources, and can be configured into a number of simultaneous partitions (subsets of the system resources) each of which executes a sub-task. Examples of such systems include REPLICA [10], PASM [13], NETRA [14], the Cosmic Cube [11] and the Butterfly [4]. Partitioning can be viewed as a method of reconfiguring the system to match the resources to the requirements of the algorithm used to execute the task. Partitioning, however, also introduces the problem of (a) deciding the partition size, and (b) scheduling the partitions. If one had a knowledge of the task characteristics, such as the time and number of processors required, it simplies the problem of deciding the partition sizes and scheduling. Having the user specify the parallel subtasks, the partition sizes, and the mapping onto the partitions, would place a tremendous burden on the user. This research investigates an approach for defining and using a knowledge base of task characteristics, and a partitionable hierarchical architecture that would interact with the knowledge base in order to efficiently execute image processing and scientific computation tasks. The objective is twofold: to ease the burden of parallel programming on the user and to use the knowledge of task characteristics towards improving mapping, partitioning and scheduling algorithms.

To integrate into the architecture the knowledge pertaining to task characteristics, we use the concept of parallel *primitives* and a *primitives table*. The primitives table is the knowledge base for the arhchitecture. The primitives are frequently used basic SIMD operations such as vector addition, vector inner product, computing the sum, and basic image processing operations. The user would write an algorithm as a composition of primitive operations. For example, multiplication of two matrices can be expressed as a sequence of vector inner products. The primitives table has an entry for each primitive storing the specification of the primitive and information needed by the system for efficient execution of the primitives. This includes the number and type of processors required, expected execution time, alternative network topologies, and the executable code. The executable code is different for each alternative network topology, and thus we have a set of algorithms corresponding to each primitive. The knowledge base consists of three components: (1) the primitives definition, (2) information on the run-time characteristics and (3) the executable codes of the algorithms for the primitive.

Our proposed system, illustrated in Figure 1.1, is organized as three layers. The first layer receives programs written using the primitives and it serves as the user interface. Layer 1 accesses the first component of the knowledge base to check if the primitives are well defined (i.e., there is a matching of formal and actual parameters). It then generates a list of primitives, which can be executed in parallel, and sends this list of primitives to be executed to Layer 2. The second layer is a high level operating system, which could be viewed as an *intelligent operating system* [5]. The high level operating system serves as the interface between the user and the architecture, and it uses the knowledge base of algorithm characteristics to select an efficient algorithm for the primitive, and schedule a partition of processors to execute this algorithm. In our system the partitioning and scheduling are initiated by the high level operating system and are driven by the primitives table. The high level operating system supplies the low level operating system, which is layer 3, with the set of routines to be run, the schedule, and the configuration (i.e., partition). The low


USER
KNOWLEDGE BASE
LAYER 1
PRIMITIVES
LAYER 2
TABLE
EXECUTABLE CODE
LAYER 3
PARALLEL MACHINE
layer 1 is the user interface
layer 2 is the high level oprating system
layer 3 is the low-level operating system

System Overview
Figure 1.1

level operating system places the processors in the required configuration and downloads the executable code from the knowledge base to the parallel processors. The low level operating system controls and monitors the parallel machine, and performs all other functions of a traditional operating system. It supplies the high level operating system with the run time information such as the number of processors available and the status of a task. The layered structure of our system allows us to employ scheduling, partitioning and reconfiguration methods used by other systems added on top of our own.

To map the algorithms onto the parallel processor we need a flexible partitionable architecture that could efficiently implement all the alternative algorithms. In this paper we investigate a model of an architecture that would effectively support our knowledge based system. It must be noted that our knowledge based approach, namely the components at layer 1 and layer 2, can be implemented on any paritionable parallel architecture and is therefore independent of the actual machine one chooses.

The next section discusses the concept of primitives and examples of primitives and their corresponding table entries. To illustrate the effectiveness of a knowledge based approach, this paper focuses on the domains of image processing and vector processing. In Section 3 we discuss some aspects, such as scheduling and partitioning, of the layer 2 operating system using the knowledge base. In section 4 we outline our proposed architecture that would serve as the underlying architecture for our system. The architecture is a multiple SIMD system and we propose a Banyan-Hypercube [17] network as the underlying network of the system. In the last section, we give a summary and concluding remarks and discuss future research being investigated.

§2. The Knowledge Base

A computational task may be decomposed into many sub-tasks each of which computes a specific function. Formally, one can view a task as computing an output set $\{Y_1, \ldots Y_m\}$ from the set of inputs $\{X_1, \ldots X_n\}$, where $\{Y_1, \ldots Y_m\} = f(X_, \ldots X_n\})$. The function f may be a composition of many simpler functions. For example, multiplication of two matrices A and B can expressed as a sequence of vector inner products. This can be written as $C[i,j] = A(i) \bullet B(j)$, where $C[i,j]$ is the entry in row i and column j of the final matrix, and $A(i) \bullet B(j)$ is the inner product of the i-th row of matrix A and the j-th column of matrix B (thus, $\bullet$ is the vector inner product function). If the inner product was considered as a basic operation, then the user could write the task of matrix multiplication as a composition of basic operations.

Furthermore, if the parallel code for the inner product algorithm was stored in the system database then the user does not have to specify the entire parallel code for the multiplication nor the system configuration on which to run the algorithm. This is the underlying motivation behind the concept of *primitives*. Primitives are frequently used basic operations (in the application domain). Operations such as sum, average, and maximum of n numbers are frequently used operations in many application domains, and thus are primitives in our system. As discussed earlier, the primitives form the first component of our knowledge base and their characteristics, the primitives table, form a second component. In what follows we discuss the concept of primitives, a sample of primitives and their table.

In image understanding systems the set of tasks that must be computed are known in advance and, in addition they employ a set of frequently used *low level* image processing operations [16] [3]. These low level tasks include image smoothing, thresholding, convolution, connected component labelling, Fourier transforms, etc.. For example, in the domain of stereo computer vision, the task of obtaining 2-D surface information from 2-D images consists of sub-tasks such as edge detection, feature matching, hough transform, surface fitting, and surface interpolation [3]. The edge detection task may involve image smoothing, gradient magnitude computation (using a sobel operator), and thresholding [16]. The task also consists of high-level vision (image understanding) algorithms such as surface interpolation and parameter computation which are MIMD algorithms. Similarly, many scientific computations involve frequently used operations such as matrix multiplication, vector inner product, sum of vectors, vector cross product, eigenvalue computation, and Monte Carlo simulations, to name a few.

The concept of using a database of task characteristics in a parallel processing system was studied by Delp et.al. in [5]. They investigated an Image Understanding System, that stores the code and heuristics of appropriate algorithms, that has an intelligent operating system for scheduling and selecting algorithms. However, theirs is a special purpose system that can only call and execute image understanding tasks stored in the image database. Thus, their primitives are entire tasks and do not provide the user with a means of defining new programs using a set of primitives (which is our intent). Furthermore, they do not address the network configuration required as a task characteristic while our table includes this parameter (as discussed in the following subsections). Our long term goal is to provide a general purpose environment wherein the user does not experience the burden of parallel programming.

2.1 Specification of Primitives

In this paper we attempt to identify the low-level SIMD operations, which we refer to as low-level primitives, and in the future we plan to include high-level primitives of MIMD operations. In Table 1 we give a sample of primitives for our system. These primitives are stored in the knowledge base of our proposed system.

The primitive definition consists of the name of the primitive and the input and output parameters. For example the inner product is listed as Inner-Product(input,input,output) to specify that the primitive requires two inputs (two vectors) and produces one output (in this case a scalar quantity). The user programs are input to Layer 1, the user interface, which then accesses the knowledge base to check if the user program is well defined (i.e., the primitives are defined and the formal and actual parameters match correctly). Layer 1 then sends the list of primitives to be executed, for the task at hand, to Layer 2. The user has no knowledge of the particular algorithms selected for execution, their schedule or the configuration of the architecture. This approach clearly lessens the burden of parallel programming on the user. To extend the knowledge base we allow the user to define new primitives (along with their definitions and table specifications discussed in the next subsection). This allows for a more flexible system in which the application domain can be extended according to the capability and demands of the user.

Table 1
Image-Smooth(Input,Output)
Threshhold(Input,Input,Output)
Convolution(Input,Output)
Summing (Input,Ouput)
Inner-Product (Input,Input,Output)
Fourier-Transform(Input,Output)
Connected-Components(Input,Output)
Gradient-Computation(Input,Output)
Image-K-Curvature(Input,Output)
Hough-Transform(Input,Output)
Matrix-Transpose(Input,Output)
Get-Window(Input,window size)
Broadcast(data)

The primitives can be basic computations or may be pure communication instructions such as broadcasting or matrix transpose. The user may write a program using the communication primitives to fetch the data needed by his program. Window operations on images are very frequently used by many algorithms such as smoothing, and convolution. The smoothing operation involves replacing a value of a pixel with the average value of pixels in a $k \times k$ neighborhood of the pixel. Therefore the smoothing algorithm itself can be written as fetching a $k \times k$ window of values and then computing the average. This implies that primitives themselves may be expressed as a composition of other primitives. At the current time we are interested in first obtaining a set of primitives that are frequently used. Under this framework, the smoothing algorithm would itself be a primitive written without using the window primitive. In the future we plan to provide a hierarchy of primitives. As an example of a high-level primitive using a set of low level primitives, consider the 2-dimensional Fourier transform of an image, FFT-2D, expressed as a sequence of 1-dimensional fourier transforms, FFT-1D.

2.1. Example. The 2-D Fourier transform (FFT-2D) is computed from the 1-dimensional transform of rows and columns. If the 1-dimensional transform (FFT-1D) was a primitive then the FFT-2D algorithm can be written as:

```
Input: $N \times N$ Image
for i:= 1 to $N$ do FFT-1D(N);
Matrix-Transpose(N×N);
for i:= 1 to $N$ do FFT-1D(N);
```

2.2. Example. As an example of a complete task that could be desired by a user, we consider an example of a benchmark image understanding task defined by Weems et.al. in [16]. In this task, there are two inputs, the intensity image and the depth image, and the task involves recognizing an object composed of rectangles. The task requires both bottom-up and top-down processing. It performs low level operations, steps 1 through 5 and step 8, to extract rectangles from the two input image data. The rectangles extracted from the image are matched with the model rectangles, for the object, that are stored in its memory (i.e., an image database). After finding a rectangle in the intensity data (i.e., after step 7) it performs a top-down probe (of the intensity and depth images) for confirmation or veto of the initial match found at step 7. The entire task itself is specified as a composition of low level primitives and steps 6,7,9,10 are high level tasks which require data from the low level steps. The task may be written as:

1 Connected-Component-Label(X_I, Y_1);
2 Image-K-Curvature(Y_1, Y_2);
3 Image-Smooth(X_D, Y_3);
4 Gradient-Computation(Y_3, Y_4);
5 Threshold(Y_4,threshold level,Y_5);
6 Rectangle-Generation using Y_2 - Output Y_6;
7 Graph-Matching using Y_6 and Y_3 - Output Y_7;
8 Hough-Transform(Y_5, Y_8);
9 Rectangle-Search using Y_8;
10 Top-Down-Probe using output of step9.

Steps 1 through 5, and step 8 call primitives from our knowledge base. The high-level tasks, at steps 6,7,9, and 10, may themselves be decomposed into simpler functions and expressed as a composition of higher level primitives. As part of our future research we plan to define such primitives.

2.2 The Primitives Table

The set of primitives (with just their definition) in the knowledge base would clearly reduce the burden on the user but in themselves do not provide any information that can be used by the system for mapping and scheduling the task onto the machine. We store run-time information in the primitives table which forms the second component of our knowledge base. This information allows intelligent and efficient use of the parallel machine resources by the layer 2 operating system. For each primitive we store alternative algorithms (and their performance characteristics) that correspond to an implementation on different interconnection topologies. For each alternative network topology we store information such as the optimal number of processors required as a function of the input size, the expected execution time as a function of the number of processors and input size. For example, the Image-Smooth primitive can be implemented on a Mesh or a Ring. Each implementation is a different algorithm. The run-time characteristics of each implementation dictates the priority of the algorithm that should be selected based on the available configuration and number of the processors. The executable code for each alternative algorithm is stored as the third component of the knowledge base. Once a selection of an particular algorithm is made, the low level operating system at layer 3 downloads the code into the parallel machine. The system at layer 2 would attempt to allocate the optimal number of processors required (configured as the best alternative network), but when this number is not available it allocates the best possible number and hence we store the execution time as a function of the number of processors and input size. The details of this process are discussed in the next section. In what follows we discuss the organization of the primitives table and sample entries.

The entry to the table is made using the primitive. When the primitive is Image-Smooth the system can access the entries corresponding to that primitive. The table has upto three (this number can be changed in the future) alternatives, i.e., choices, of implementations for the primitive. Each alternative is stored in a priority order and corresponds to a particular interconnection topology (note that each implementation is an algorithm whose code is also stored as the third component of the knowledge base). The priority order is based on the execution time on the three network topologies (note that this implies we pick the network with the least communication overhead since the computation time is the same when an identical number of processors are used in each network). For each choice the table stores the four parameters of expected execution time (as the sum of computation and communication time), optimal number of processors required as a function of the input size, the Input data structure, and the Output data structure. The organization of the table is shown in Figure 2.2. The prioritized choices are listed as Net-1, Net-2, and Net-3 to mean the network topologies of the first,second and third priority. The first choice for the Image-Smooth primitive is the Mesh topology, the second choice is the cube (i.e., hypercube) and the third choice is a ring. The Mesh is the first choice since for a given number, P, of processors the execution time (i.e., the sum of computation and communication time) is the least among the three networks. The input and output data structure is a 2 dimensional array (i.e., the image data), and the execution time and number of processors is specified as a function of the input image size.

The knowledge base can be treated as a 3 dimensional array, where the dimensions are the primitive, the run-time information parameters and the three choices. In the future we plan to add more parameters such as the type of processors required, the mode of operation (SIMD or MIMD), and the data allocation (i.e., how is the data mapped to the local memories). Another aspect we need to investigate is the generation of a list of parallel primitives (from the user program) by the Layer 1 system (the user interface). For example, the FFT-2D example involved a *for* loop that called the primitives. To recognize the existence of parallel FFT-1D primitives we must use code parallelization methods from techniques employed by parallel compilers.

§3. Scheduling and Partitioning

The function of the high level operating system (layer 2) is to select a set of algorithms to be used, for computing the users task, to get a good execution time. The goal is to find a mapping of the necessary algorithms onto the machine's resources in order to achieve a good performance and make an efficient use of these resources. The system accesses the knowledge base to determine a good selection. The final selection is based on the information in the knowledge base and the current configuration of the system.

When Layer 2 receives the list of primitives to be executed from Layer 1, it looks up the table entries corresponding to that primitive and uses the information towards the selection of an efficient algorithm to be executed on the available machine resources. It calls the *Allocate_Partition* function, and the input parameters are the primitive and the optimal number of processors required for each alternative network. The partition allocation algorithm first checks if the first priority algorithm (i.e., on Net-1) with the optimal number of processors can be scheduled. If this is not possible then the expected execution time is used to decide which algorithm is to be implemented and the number of processors to be used. For example, if the primitive was Image-Smooth, with 100 as the size of the input image, then the system first checks if a Mesh of 100 processors can be allocated to the algorithm, failing which it looks at the other choices. To determine the number of processors available and if a partition of a specific size and configuration is available, the algorithm calls two Layer 3 services. The *Layer3_Allocate(Net i,P)* routine asks the layer 3 system to allocate a partition of P processors configured as network Net i, and layer 2 sends the address of the code (for the selected algorithm) to be downloaded by layer 3 into the machine. The *Layer3_AvailableP(Net i)* routine checks the status of the system and returns the number of processors available configured as Net$i$. These two routines serve as the communication channel between layer 3 and layer 2. We also provide the *Allocate_Partition* algorithm with a set of functions which access the knowledge base (the primitives table component). The function Net(Primitive,i) reads the table and returns the network of the i-th choice for the primitive. The function Time(Primitive,i,P) looks in the table for the execution time formula and computes the expected time for the algorithm using P processors configured as Network i. For example, Net(Image-Smooth,1) returns the first choice which is a Mesh network and Time(Image-Smooth,1,100) returns the expected run-time for the algorithm using a Mesh of 100 processors. We now present the *Allocate_Partition* algorithm.

Function *Allocate_Partition* performs the main task of allocating a partition for the execution of a primitive. This function resides in layer 2 and uses the services provided by the abstract data type *Primitives_Table* and the services of layer 3 operating system calls.

The following operations are exported by the *Primitives_table* abstract data type:

Primitive	Network	Execution time	Processors required	Input	Output	Code
<primitive>	<choice 1>	<time on choice 1>	<Proc. reqd as Net 1>	<input on Net 1>	<output on Net 1>	<code1>
	<choice 2>	<time on choice 2>	<Proc. reqd as Net 2>	<input on Net 2>	<output on Net 2>	<code2>
	<choice 3>	<time on choice 3>	<Proc. reqd as Net 3>	<input on Net 3>	<output on Net 3>	<code3>
Image-Smooth	Mesh	$kN^2/P + c\,4N/P$	N^2	2-D Array	2-D Array	<code for mesh>
	Hypercube	$kN^2/P + c\,8N/P$	N^2	2-D Array	2-D Array	<code for cube>
	Ring	$kN^2/P + c\,4N/\,P$	N	2-D Array	2-D Array	<code for ring>
Sum	Tree	$k(\log P+N/P) + c \log P$	N	Array	one point	<code for tree>
	Hypercube	$k(\log P+N/P) + 2\,c \log P$	N	Array	one point	<code for cube>
	Mesh	$k(\log P+N/P) + c\,P$	N	Array	one point	<code for mesh>
Inner-Product	Tree	$k(\log P+N/P) + c \log P$	N	1-D array	one point	<code for tree>
	Hypercube	$k(\log P+N/P) + c\,2.\log P$	N	1-D array	one point	<code for cube>
	Mesh	$k(\log P+N/P) + c\,P$	N	1-D array	one point	<code for mesh>

N is the size of the input, P is the number of processors used in the algorithm

k is the time for one arithmetic operation, c is the time for one communication step

<code i> is a pointer to the executable code stored in the third component of the kowledge base

Organization of the Primitives Table

Figure 2.2

Alternatives(Primitive) - number of alternative network connections available for this primitive.

Net(Primitive,i) - network type for the *i*-th alternative of Primitive.

Time(Primitive,i,P) - execution time of the *i*-th alternative of Primitive given P processors.

Two services of layer 3 are used by the allocation function:

Layer3_AvailableP(Net) - number of processors available configured as Net connection type.

Layer3_Allocate(Net,P) - allocate a network of type Net with P processors.

We assume the existence of two types: *Primitives* is a type defining the range of all possible primitives. *ProcessorN* is a type defining the range of the possible number of processors in the system. In function *Allocate_partition* access to operations exported by *Primitives_table* are preceded by the characters PT in a typical qualifying way.

function *Allocate_Partition*(Primitive:Primitives;N:ProcessorN) **return** boolean;

```
         P,PreP:ProcessorN;
         Start:boolean:=false;
         i,Previ:integer;
begin
1.       for i:=1 to PT.Alternatives(Primitive) do
2.           P := Layer3_AvailableP(PT.Net(Primitive,i));
3.           if P ≠ 0 then
4.               if not Start then
5.                   Start:=true;
6.                   if P ≥ N then
7.                       Layer3_Allocate(PT.Net(Primitive,i),N);
8.                       return true;
9.                   else
10.                      PrevP:=P;
11.                      Previ:=i;
12.                  endif;
13.              elseif PT.Time(Primitive,Previ,PrevP)≥ PT.Time(Primitive,i,P);
14.                  then
15.                      Layer3_Allocate(PT.Net(Primitive,i),P);
16.                      return true;
17.              endif;
18.          endif;
19.      endfor;
20.      if Start then
21.          Layer3_Allocate(PT.Net(Primitive,Previ),PrevP);
22.          return true;
23.      else
24.          return false;
25.      endif;
         end Allocate_partition;
```

Since the table entry contains the formula for the execution time (PT.Time(Primitive,i,P), the function is able to calculate the approximate execution time for an algorithm using P processors connected as network type PT.Net(Primitive,i). This way the function can consider the alternatives it can schedule. For instance, if the primitive is Image Smoothing with an image size of 512×512, we may have a choice between a 16 processor Mesh as one priority and a 100 processor Ring with a lower priority. The function will schedule the alternative with the lowest return from *PT.Net(...)* (in this case the Ring). Steps 15 and 21 accomplish this function. Therefore, the system may allocate an available partition when the optimal partition configuration is not available. We call this process a *folding* of the algorithm since we are using a reduced number of processors and/or a network configuration that may not be the first alternative. In the simulation runs shown in Figure 3.3 it can be seen that by using the folding steps leads to a better system performance (both in utilization and the total execution time).

When *Allocate_partition* returns a false, layer 2 queues the request. The queuing policy reflects the scheduling function used by this layer; it can be queued according to the smallest number of processors requested or the largest such number dependening on the optimization sought.

When the function returns true, layer 2 has to update its task tables to indicate that this primitive was scheduled and then initiate the start of its execution by sending the address of the executable code to layer

3. Layer 3 then proceeds to download the code and commence execution on the processors in the partition selected for the task.

To demonstrate the advantage of this partition allocation scheme let us consider the case when the task is the Image Smoothing task on a 512×512 array of pixels. If we have a choice between a 16 processor Mesh (priority 1 network) and a 64 processor ring (priority 3 network), our system would assign the 64 processor ring system to compute the algorithm instead of assigning the best possible mesh configuration as is done in other systems where the user specifies the network type. By using the knowledge in the table our system is able to identify this communication characteristic of the smoothing algorithm and assign the more efficient machine configuration. If on the other hand the primitive was a matrix transpose where the matrix was stored row-wise then, since there is only communication time and no computation, the mesh (which has time $\sqrt{P}N$ for $N \times N$ matrix, $N > P$) is assigned instead of the ring (which has time PN).

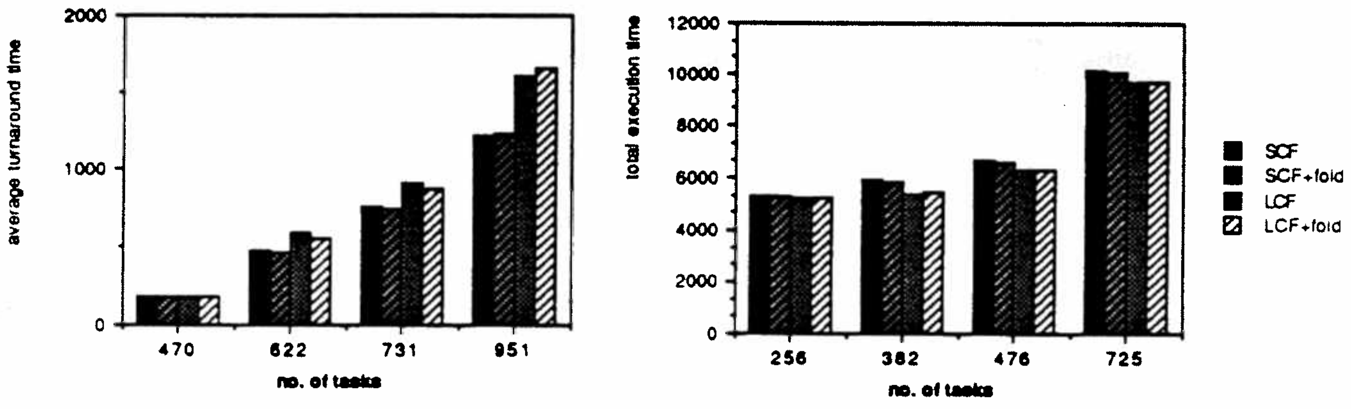


Figure 3.3

The two histograms, in Figure 3.3, provide results of simulating four layer 2 scheduling strategies on a hypercube of 512 processors. Similar results were attained for other cube sizes. The four strategies are: smallest cube first (SCF), smallest cube first with folding (SCF+fold), largest cube first (LCF), and largest cube first with folding (LCF+fold). Four identical randomly generated groups of tasks were run with each strategy. SCF gives a better average turnaround time, while LCF gives a better utilization of the hypercube. In both cases, folding improves on the results of the underlying strategy except where the load is extremely high (in the average turnaround time histogram with an input of 951 tasks). The hypercube partitioning, i.e., *Layer2_Allocate* was based on a buddy system [2].

If we have the approximate run-time information and the current status of the tasks executing on the system, then the scheduler can 'predict' when the tasks (currently running on the system) will complete. This information could be used by the scheduler to decide whether to allocate a currently available partition (which is not of the optimal size) or to wait till the first task finishes and assign a different configuration. This would introduce more complexity into our scheduling algorithms. We plan to investigate such scheduling algorithms which will certainly improve throughput and execution time and would require the same knowledge base.

§4. System Architecture

This section proposes a a Hierarchical Partitionable Knowledge based Architecture that serves as a model for our parallel processing system, and outlines some salient features of our proposed architecture.

Applications in the area of low level image processing and scientific computation exhibit what is known as *data parallelism*, where the same operation may be applied to many data elements. A candidate architecture for exploiting such parallelism is the SIMD architecture. The SIMD architectures consist of a large number of processing elements executing the same instruction simultaneously on different data elements. Much like parallelism at the data level, parallelism can also exist at the task level where several sub-tasks can execute in parallel to solve the main task. This is particularly true in high level vision algorithms. Therefore partitionable architectures are ideal for application areas which have tasks that exhibit both types of parallelism. When each sub-task exhibits parallelism at the data level, we allocate an SIMD partition to execute the sub-task. Thus we have a multiple SIMD (MSIMD) machine configuration. When each sub-task exhibits parallelism at the task level then a MIMD partition is allocated to execute the sub-task.

In addition to the type of parallelism, the complexity of the task (in terms of the complexity of the basic operations) would also dictate the performance of the algorithm. In Computer Vision systems the computational complexity increases as one proceeds from low-level image processing operations to high level operations such as symbolic matching [16] [14]. Hierarchical architectures with non-homogeneous processors are therefore suitable for such systems. Examples of hierarchical systems include pyramidal systems such as the PAPIA [1] and the Pyramid Machine for cellular logic [15], and hierarchical non-homogenous architectures such as the NETRA [14].

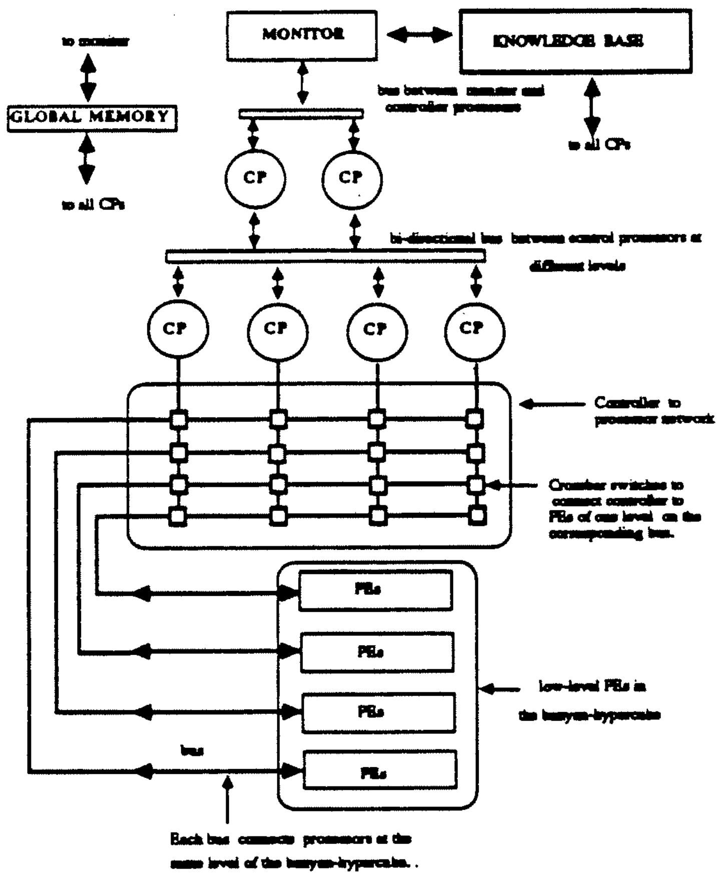
4.1 System Architecture

Our model architecture is intended to be a large integrated multiprocessor architecture for image analysis and understanding and scientific processing. The system is partitionable using a hierarchy of controllers that themselves operate as an MIMD sub-system. The conceptual system is illustrated in Figure 4.4. The system consists of the following components:

(1) A large number ($\simeq 10^4$) of *Processing Elements* (PEs) that perform computations on large data sizes.
(2) A *Banyan-Hypercube* Interconnection network [17] that interconnects the PEs.
(3) A hierarchy of *control processors* (CPs) that control the partitions of PEs.
(4) An interconnection network that connects CPs to PEs.
(5) A Monitor that is the centralized control of the entire system and controls scheduling, allocation and the flow of control in the multiprocessor and serves as the user interface.

The bottom level CPs serve purely as the controllers of a partition of PEs. The intermediate layers of CPs are more powerful and control the CPs at the layer below, so as to access output information after a partition has completed its task. In addition, they schedule high-level tasks in a MIMD mode. For example, they could provide code to lower level CPs along with partition parameters, and thus have lower level CPs coordinate partitions (of PEs) to execute for example graph matching algorithms on the features extracted after initial image processing by the PEs. Furthermore, such MIMD tasks would require very little communciation between processes and therefore a shared bus suffices as the interconnection between the levels of CPs.

The monitor interacts with the knowledge base to determine the scheduling and partitioning of system resources. Thus Layer 1 and Layer 2 of the knowledge based system are carried out in the monitor. Once the schedule and partition sizes are determined the monitor downloads the schedule, partition allocation information and address of the code to the CP hierarchy. The user task is sent (from the monitor) by sending the address of the primitive code (in the knowledge base) to be executed by the partition. This requires that the CPs must have access to the knowledge base. The controller (for the partition) need only receive (from the higher level) information about the processors in its partition and the address of the code to


MONITOR
KNOWLEDGE BASE
to monitor
GLOBAL MEMORY
to all CPs
bus between monitor and controller processors
to all CPs
CP
bi-directional bus between control processors at different levels
Controller to processor network
Crossbar switches to connect controller to PEs of one level on the corresponding bus.
PEs
low-level PEs in the banyan-hypercube
bus
Each bus connects processors at the same level of the banyan-hypercube.

System Architecture
Figure 4.4

be executed by the PEs. By sending only information such as the address of the code instead of downloading the entire code (as is done in REPLICA, and PASM) for the task, we can reduce the load on the set of buses between the CPs (and the monitor). The NETRA system uses a tree of controllers and therefore involves functionally the same process of downloading. Although we have used buses, we note that since the amount of traffic on the buses is small the performance is comparable to the tree structure while adding more flexibility in terms the PEs that may be chosen by each partition. A future extension would be to allow for the set of CPs to share some of the burden of Layer 2 scheduling and partitioning algorithms.

The operations performed by the PEs are simple low-level operations such as arithmetic operations, intermediate level processing such as computing the Hough transform [16], and more sophisticated tasks such as graph matching. In our model, these PEs are general purpose microprocessors with their own local memory (on the order of 16K -64K). The non-homogenous nature is in terms of the different processing capabilities of the Monitor, CPs, and the PEs. One could envision each level of the network we propose to consist of different types of processing elements. We plan to provide this extension to our architecture after we have incorporated more parameters (mentioned in Section 2) into the knowledge base. In such a system, we could assign partitions of different sizes and types of processors and therefore include the processor type as another table entry.

4.2 The Interconnection Networks

The various levels of control processors are interconnected by shared buses between levels. As noted earlier, this startegy would suffice since there is little communication needed between CPs scheduling an MIMD task. However, the same is not true for the set of PEs and the network that connects lowest level controllers with the PEs. We first present the Banyan-hypercube network as the interconnection between PEs, and then the crossbar switch between lowest level CPs and PEs.

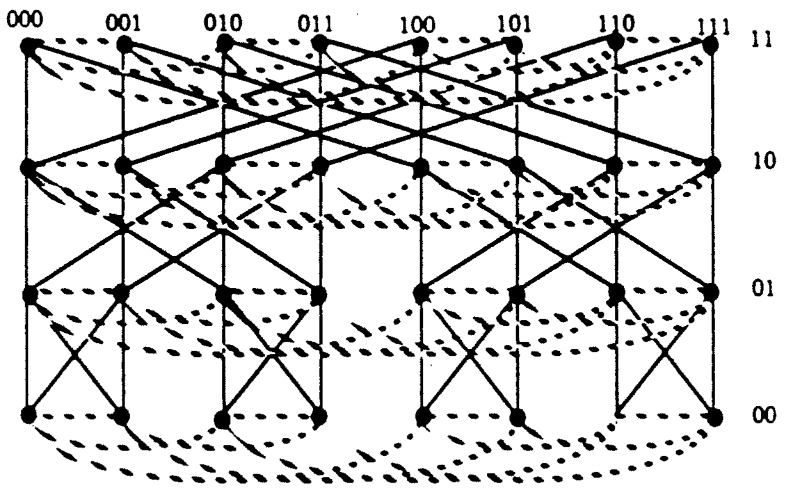
The Banyan-Hypercube network, proposed in [17], is a synthesis of Banyan and Hypercube networks. Hypercubes [11] and regular rectangular Banyans [7], [9], are among the networks that have been proposed and studied in the past. The synthesis of the two networks gives us a hierarchical network that combines advantageous features of both networks. We first recall (from [7]) the definition and properties of a banyan and then define the banyan-hypercube.

A banyan is a Hasse diagram of a partial ordering such that for all s_i source nodes and for all s_j sink nodes there is exactly one path from s_i to s_j. The source nodes are called *base* nodes and the sink nodes are called as the *apex* nodes of the banyan graph. The outdegree of a node is called the spread and the indegree is called the fanout. A banyan is said to have ℓ levels if every path from a base node to an apex node is of length ℓ. A banyan is said to be a regular banyan [9] if all nodes (except the base and apex) have the same inuegree and outdegree. Furthermore, if the indegree is equal to the outdegree then it is said to be a (ℓ, s) rectangular banyan of spread s. The number of nodes at each level of the banyan is $s^{\ell-1}$. An addressing scheme has also been discussed in [9]. Each node has two address fields, one for the level to which it belongs and the second field is the address within the level. The level address contains ℓ digits in base system of the spread (i.e., s). When the spread is 2, the addressing is in binary. The levels are labelled from 0 to $\ell - 1$. A node at any intermediate level i is connected to every node at level $i + 1$, in which all digits other than the $(i + 1)$th digit are identical. When $s = 2$, than a node with level address 100 at level 1 will be connected to nodes 110 and 100 at level 2 (i.e., node 01100 is connected to nodes 10110 and 10100). The banyan networks have self routing algorithms of $O(\ell)$ complexity and they have a diameter of ℓ. In [12] the banyan network is used as the interconnection network between processors and memory, where the base and apex nodes serve as the processors and memory respectively.

A (h, k, s) Banyan-Hypercube (BH) Network is constructed by initially taking the first h levels ($h \leq k+1$) of a $k + 1$ level rectangular banyan of spread and fan-out equal to s, where s is a power of 2. Each level has s^k nodes representing processing elements, labelled from 0 to $s^k - 1$ in binary. We interconnect the nodes at each level as a hypercube (i.e., connect nodes whose level addresses differ in exactly one bit). Note that the hypercube itself can be viewed as a BH$(1, k, s)$ network. Figure 4.5 shows a $(4, 3, 2)$ network. Each node has two address fields, the level address and the address within the level. Thus the node 10000 is at level 2 and has address 000 at level 2. We have shown, in [17], that a BH(h, k, s) network has a diameter of $\min\{2k, k \log s\}$ which is less than that of a hypercube of the closest size. In addition it is also shown that the average distance and degree of a BH(h, k, s) network is comparable to that of a hypercube.

The routing on a BH network is based on a self routing scheme and combines the routing in the banyan and the hypercube. The complexity of the routing algorithm is $O(k)$ and the algorithm is shown below. Note that when routing between the same level we can use the hypercube routing algorithm. If the data is to be sent to a different level then the banyan connections are used. For this case, when we go up (or down) a level we change the level address and by selecting the ***oblique*** link we can change one bit of the intra-level address of the node. Therefore, we can go from node 00000 to node 01001 in one step (i.e., we have changed two bits of the node address). The algorithm is shown below. The node address is denoted as the ordered pair $(L, x_{k-1}, \ldots, x_0)$, where L is the level address and $x_{k-1}, \ldots, x_0$ is the address within the level (intra-level address).

Routing Algorithm for the Banyan-Hypercube

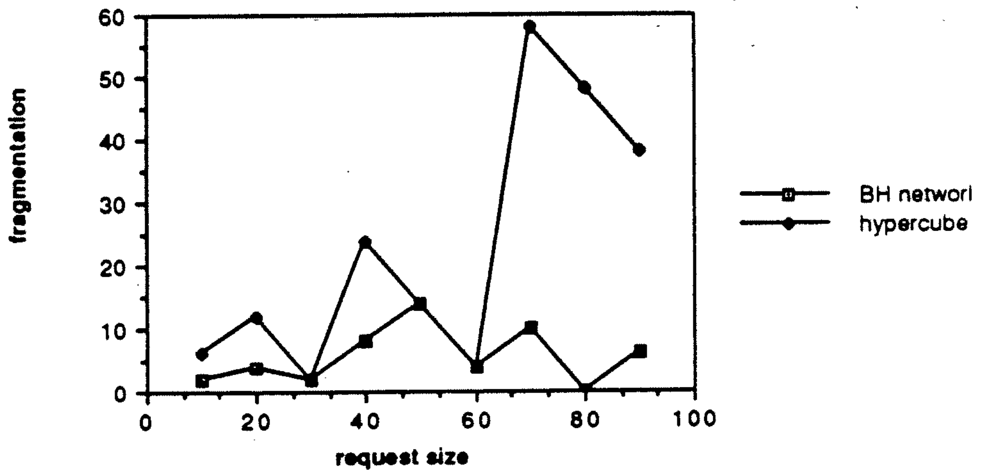


The dashed lines show the hypercube connections
The solid lines show the Banyan Connections
A (4,3,2) Banyan-Hypercube Network
Figure 4.5

Route from Current node X with address $(L, x_{k-1} \ldots x_1 x_0)$ to
Destination Node D with address $(L', d_{k-1} \ldots d_1 d_0)$

```
Algorithm Route(X,D);
begin
        Case:
         $L' > L$: /* go up */
          if $x_l = d_l$ then
                Send through up-straight link
          else
                Send through up-oblique link
         $L' < L$: /* go down /*
          if $x_{l-1} = d_{l-1}$ then
                Send through down-straight link
          else
                Send through down-oblique link
         $L' = L$: /* horizontally /*
          Route in Hypercube
        End Case
end
```

The banyan-hypercube network is seen to efficiently embed rings, meshes, trees and pyramids [17]. This makes it an efficient underlying network for our system since it allows us to efficiently embed the communication structures requested by the primitives. A BH(h, k, s) network can be partitioned horizontally, vertically or a combination thereof. Horizontal partitioning results in one BH$(1, k, s)$ (a hypercube) and one BH$(h-1, k, s)$ network. Vertical partitioning results in s BH$(h, k-1, s)$ networks and is allowed only when $h < k+1$. The details of the partitioning methods are discussed in [17]. The banyan-hypercube offers more flexibility in partitioning in the sense that the size of a partition is not restricted to a power of 2 which is the case in the hypercube. This leads to a lower degree of internal fragmentation, i.e., the number of additional processors that must be assigned to the partition. For example a request of 20 processors can be satisfied by a 24 node BH(3, 3, 2) network with a resulting internal fragmentation of 4 extra idle nodes, whereas a hypercube of 32 is the smallest cube that can fit 24 nodes and thus has 8 idle extra nodes. Figure 4.6 compares the fragmentation incurred on the banyan-hypercube and the hypercube for request sizes that are multiples of 10 (from 10 to 90) (note that when a request size is a power of 2 there is no internal fragmentation). The graph demonstrates the greater flexibility of partition sizes provided by the banyan-hypercube. This would lead to a better matching of system and task resources and a greater utilization.



Internal Fragmentation on BH(h, k, 2) and Hypercubes
Figure 4.6

We now outline the interconnections between the control processors at the lowest level and the set of PEs. Since we wanted to keep the number of I/O ports in the PEs to a minimum, we have one bus that connects all PEs lying on the same level of the Banyan-hypercube. Thus, we need only have one extra I/O port per PE (in addition to the I/O ports required for the Banyan-hypercube connections). There are a total of h buses, where h is the number of levels in the Banyan-hypercube, and h control processors at the lowest level of the CP hierarchy that serve as the various controllers of the partitions of the Banyan-hypercube system (Figure 4.4shows the connections when we have four levels in the Banyan-hypercube). Therefore, an $h \times h$ crossbar would suffice in order to connect a controller to any of the h buses and have it broadcast SIMD instructions to the PEs at the corresponding level of the Banyan-hypercube. To allow flexibity in partitioning, the controllers are allowed to select more than one bus if the partition consists of PEs at various levels or a controller can control more than one partition if they consist of PEs at the same level. However, note that two distinct controllers cannot be connected to the same bus (i.e., they cannot control the same set of PEs). The bandwidth of the bus is proportional to the number of controllers (h). This interconnection scheme allows for a modular interconnection, and has no contentions for the bus. The common bus (linking PEs at the same level) is pipelined to take advantage of the bus bandwidth. Several microinstructions (the SIMD instructions to a partition of PEs) can be pipelined. Since the partitions at any one level are sets of disjoint PEs, microinstruction broadcasts are interleaved across several partitions in the level (i.e., the bus is used to broadcast instructions while the partitions are executing the received instruction). This scheme allows for a greater number of partitions to be formed (i.e., the number of partitions are not restricted to

h). For example, if the bus cycle time is half of a PE intruction cycle then we can allow each controller to control two partitions. If T is the cycle time of the PEs, the controller first broadcasts instructions to PEs in partition 1, and then it broadcasts intruction to partition 2 at time $T/2$. Thus it is able to control two partitions while keeping the operations of the two partitions independent of each other.

The proposed architecture provides us with a model on which we can implement the knowledge based system (i.e., Layers 1, 2 and 3). It has greater flexibility than the PASM in terms of partitioning, and due to the capabilities of the Banyan-hypercube network it is able to provide a better embedding of networks than PASM or REPLICA. The architecture proposed here (and the Banyan-hypercube network) is more cost-effective than the NETRA system (which uses a crossbar network). The efforts in the future shall be geared towards partitioning algorithms (i.e., layer 3 routines), a thorough evaluation of the interconnection networks and a comparison with other systems, and introducing non-homogeneuity into the PEs in the Banyan-hypercube.

§5. Conclusions

We have presented a system that uses the knowledge of task characteristics to match the task requirements to the system resources. We have shown that the concept of primitives reduces the burden of parallel programming on the user. The paper has provided a sample of primitives for the domain of image understanding and vector processing. We demonstrated how the system uses the knowledge base for its scheduling attempts, thus removing this responsibility from the user. In addition, we discussed a model architecture to efficiently support our knowledge based system. The Banyan-hypercube network provides distinct advantages in terms of partitioning flexibility while keeping the routing time comparable to the hypercube.

There are many research issues to be investigated. In the domain of primitives, we need to define more primitives and their specifications. One also needs to investigate the inclusion of more parameters, such as the type of processors etc., into the primitives table. In the area of scheduling we are looking into more complex scheduling startegies and the possiblility of having Layer 1 generate the precedence graph from the input/output dependencies of the primitives. This precedence graph can be used by the scheduler at Layer2 [5]. The architecture has to be further investigated and evaluated, and we plan to employ non-homogenous processors at the various levels of the banyan-hypercube.

§6. References

[1] V. Cantoni and S. Levialdi, "PAPIA: A Case History," *Parallel Computer Vision*, L. Uhr, editor, Academic Press, 1987, pp.3–14.

[2] M. Chen, and K.G. Shin, "Processor Allocation in an N-Cube Multiprocessor Using Gray Codes," *IEEE Transactions on Computers*, Vol.C-36,No.12, pp. 1396–1407, December 1987.

[3] A.N. Choudhary and J.H. Patel, "A Parallel Processing Architecture for an Integrated Vision System," *1988 International Conference on Parallel Processing*, August 1988, pp. 383–387.

[4] W. Crother, et. al., "Performance Measures on a 128-Node Butterfly Parallel Processor," *1985 International Conference on Parallel Processing*, August 1985, pp.531–540.

[5] E.J. Delp, H.J.Siegel, A.Whinston and L.H. Jamieson, "AN Intelligent Operating System for Executing Image Understanding Tsks on Reconfigurable Parallel Architectures," *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, November 1985, pp.217–224.

[6] Dennis B. Gannon, and J. Van Rosendale, " On the Impact of Communication Complexity on the Design of Parallel Numerical Algorithms," *IEEE Transactions on Computers*, Vol.C-33, December 1984.

[7] R.L. Goke, "Banyan Networks for Partitioning Multiprocessor Systems," Doctoral Dissertation, University of Florida, 1976.

[8] Leah H. Jamieson, "Characterizing Parallel Algorithms," in *The Characteristics of Parallel Algorithms*, edited by Leah H. Jamieson, Dennis B. Gannon, and Robert J. Douglass, MIT Press, 1987.

[9] R.M. Jenevein and T. Mookken, "Traffic Analysis of Rectangular SW-Banyan Networks," *The 15th Annual International Symposium on Computer Architecture*, Computer Architecture News, Vol.16, No.2, May 30– June 2, 1988, pp. 333–342.

[10] Y.W. Ma and R. Krishnamurti, "The Architecture of REPLICA – A Special Purpose Computer System for Active Multi-sensory Perception of 3-Dimensional Objects," *Proceedings of the 1984 International Conference on Parallel Processing*, August 1984, pp.30–37.

[11] C. Seitz, "The Cosmic Cube," *Communciations of the ACM*, Vol.28, January 1985, pp.22-33.

[12] M. Sejnowski, et. al. "An Overview of the Texas Reconfigurable Array Computer," *Proceedings of the AFIPS Conference*, Vol.49, NCC, 1980, pp.631–641.

[13] Howard J. Siegel, Leah J. Siegel, F.C. Kemmerer, P.T. Mueller,Jr., H.E. Smalley, and S.D. Smith, "PASM : A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition, " *IEEE Transactions on Computers*, Vol.C-30, No.12, December 1981.

[14] M. Sharma, N. Ahuja and J.H. Patel, "NETRA: An Architecture for a Large Scale Multiprocessor Vision System," *Parallel Computer Vision*, L. Uhr, editor, Academic Press, 1987, pp87–106.

[15] S.L. Tanimoto, T.J. Logocki, and R. King, "A Prototype Pyramid Machine for Hierarchical Cellular Logic," *Parallel Computer Vision*, L. Uhr, editor, Academic Press, 1987, pp.43–84.

[16] Charles Weems, Allen Hanson, Edward Riseman, and Azriel Rosenfeld, "An Integrated Image Understanding Benchmark: Recognition of a 2 1/2 D Mobile," Technical Report, University of Massachussets, 1988.

[17] A. Youssef and B. Narahari, "The Banyan-Hypercube Network: A Synthesis of Banyans and Hypercubes," *Proceedings of the Third Annual Parallel Processing Symposium*, Fullerton, CA, March 1989.