SCHEDULING ON PARALLEL PROCESSING SYSTEMS USING PARALLEL PRIMITIVES

H-A. Choi B. Narahari S. Rotenstreich A. Youssef

Dept. of Electrical Engineering & Computer Science
The George Washington University, Washington DC 20052

Abstract

An approach for integrating task and parallel architecture characteristics is presented, with the objective of easing the burden of parallel programming on the user and to increase the system efficiency through more informed embedding, partitioning and scheduling. The approach discussed in the paper uses the concepts of *parallel primitives* and a *primitives table* that serves as the knowledge base for the system. Parallel primitives are parallel SIMD computations and can be used to write parallel programs. The primitives table stores the specifications of each primitive such as the number of processors required, the communication structure, the execution time, and the code. In addition, for each primitive several alternative implementations corresponding to alternative network structures are provided and stored in the table. This paper will discuss these concepts and give a layered implementation of a parallel processing system with parallel primitives. The paper will also examine the positive implications of the primitives table on scheduling as well as determining the optimal machine configurations for tasks using precedence graphs. The latter problem will be shown to be NP-complete. However, a linear algorithm is found for the case when the precedence graph is a tree.

§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, it introduces many new problems that do not arise in sequential programming. These include parallelization of algorithms, matching algorithms to the underlying parallel architecture, proper reconfiguration of the architecture for each task, and partitioning the system for multiple tasks. Parallelization of algorithms is a heavy programming burden, and if left to the system, a great potential for parallelism and speedup may be lost. Proper matching of algorithms with the architecture is also a very difficult problem that affects the performance of the system, and is 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 [8]. 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. As noted by Jamieson [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 [13]. 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. Although past research has discussed the use of task characteristics, at the present time very little research has addressed the problem of obtaining task characteristics. One of the primary objectives of this research is to find techniques to identify task characteristics and use them to improve system performance.

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 PASM [11], NETRA [12], the Cosmic Cube [10] and the Butterfly [5]. 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.

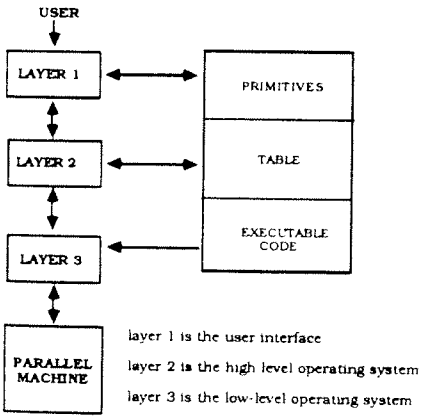
Knowledge of the task characteristics, such as the number of processors required, the communication structure, and the execution time, simplifies the problems of selecting an efficient architecture configuration (i.e., network topology and number of processors), deciding the partition sizes, mapping tasks onto the partitions, and scheduling these partitions. Having the user specify the parallel subtasks, the interconnection network, the partition sizes, and the mapping onto the partitions, would place a tremendous burden on the user and therefore one needs a methodology for acquiring this information.

CH2728-4/90/0000/0056$01.00 © 1990 IEEE

1.1 Outline

To acquire the knowledge pertaining to task characteristics, we are using the concept of parallel *primitives* and a *primitives table* [9]. The set of primitives and their characteristics are stored in the primitives table, which serves as a database of task characteristics. The primitives are frequently used basic parallel computations that are implemented as parallel algorithms. Examples of primitives are vector addition, vector inner product, computing the sum, and basic image processing operations. The user would define his task as a composition of primitives. 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 algorithm for each alternative. Thus we have a set of algorithms corresponding to each primitive and we store the executable code for each algorithm. The primitives table (of the system) 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, is organized as three layers. The first layer (Layer 1) serves as the user interface and receives programs written using the primitives. Layer 1 accesses the first component of the primitives table 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



System Overview
Figure 1

primitives and the precedence graph (dependency graph) for the task and sends this information to Layer 2. The second layer is a high level operating system, which could be viewed as an *intelligent operating system* [6] [4]. The high level operating system serves as the interface between the user and the architecture, and it uses the algorithm characteristics to select an efficient algorithm and network for the primitive, and schedules a partition of processors to execute this algorithm. Using the precedence graph, supplied by layer 1, the system first determines an efficient configuration (i.e., algorithm and network) for each primitive so as to minimize the total execution time for the task. Once the configurations, for each primitive, are determined the high level operating system allocates and schedules partitions of processors to execute these primitives. In our system the selection of the network (and algorithm), partitioning, mapping 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 level operating system places the processors in the required configuration and downloads the executable code stored in the primitives table 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. The components at layer 1 and layer 2, can be implemented on *any* partitionable parallel architecture and thus form a *portable parallel processing system*. We assume that the parallel processor is reconfigurable, partitionable and can efficiently embed common interconnection topologies such as rings, meshes, trees, and pyramids. A hypercube based architecture would be a suitable candidate for our system.

This paper develops this new approach and its concepts, and discusses some aspects of the three-layer implementation of the approach. The positive implications of the approach on scheduling and partitioning will be examined, and the effect on easing parallel programming is illustrated by examples from the domains of image processing and vector computations.

The next section presents the concept of parallel primitives and the organization of the primitives table. Section 3 describes the process of determining the efficient configurations for the tasks using the precedence graph for the task. We show that the problem (of determining efficient configurations) is NP-hard, and we provide a linear time algorithm to solve the problem when the precedence graph is a tree. Section 4 describes some of the functions of the Layer 2 operating system and demonstrates how it interacts with the primitives table to perform the scheduling functions.

§2. Parallel Primitives

A computational task may be decomposed into many subtasks each of which computes a specific function. Formally, one can view a task as computing a set of outputs $\{Y_1, \ldots Y_m\}$ from the set of inputs $\{X_1, \ldots X_n\}$, where $\{Y_1, \ldots Y_m\} = f(X_1, \ldots X_n)$. The function f may be a composition of many simpler functions. For example, multiplication of two matrices A and B can be 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 ($\bullet$ 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 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.

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 [13] [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 [13]. 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 existence of frequently used parallel operations and the need for easing the user's burden of parallel programming provide the underlying motivation behind the concept of *parallel 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.

2.1 Specification of Primitives

In what follows we discuss the specification of the low-level SIMD operations, which we refer to as low-level primitives, and in the future we plan to include high-level primitives for MIMD operations. In Table 1 we give a sample of primitives for our 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 primitive definition is identical to procedure definitions. The user programs are input to Layer 1, the user interface, which then performs a "syntax check" by accessesing the knowledge base to determine if the user program is well defined (i.e., the primitives are defined and the formal and actual parameters match correctly). Upon successful completion of the syntax checking phase, Layer 1 then generates the precedence graph for the task (where each node in the precedence graph corresponds to a primitive) and send this graph to Layer 2. In addition to the precedence graph, layer 1 also sends the list of primitives to be executed, for the task at hand, to the high level operating system at Layer 2. The justification behind this is discussed in the following sections. The user has no knowledge of the particular algorithms that will be selected for execution, their schedule or the configuration of the architecture. All these functions will be performed by high-level operating system at Layer 2. This approach clearly decreases the burden of parallel programming on the user. To extend the set of primitives, 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 system requires the communication primitives to move data around or align the data from the output of one primitive to the input of the next primitive. This is illustrated in the following example. Primitives themselves may be expressed as a composition of other primitives thus leading to a hierarchy of 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 co 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:

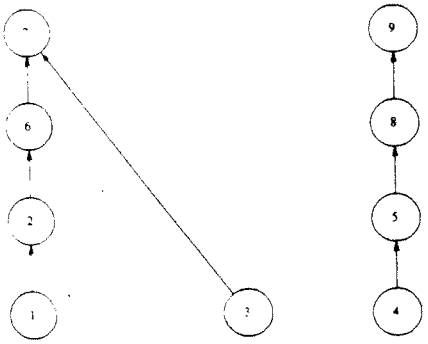
FFT-2$D(I, B(r))$
  I: $N \times N$ input image;
begin
  for $i = 1$ to N do in parallel
    FFT-1D(N);
  endfor
  Matrix-Transpose(I, I);
  for $i = 1$ to N do in parallel
    FFT-1D(N);
  endfor
end

2.2. Example. As an example of a complete task provided by a user, we consider an example of a benchmark image understanding task defined by Weems et.al. in [13]. In this task, there are two inputs, the intensity image X_I and the depth image X_D, 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, before the top-down probe, 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(X_D, 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;

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. The user's task, composed of primitives, is input to Layer 1 which proceeds to check the 'syntax' of the 'program' (i.e., verifies that the primitives are defined and there is a match of formal and actual parameters) and then generates the precedence graph for the task and send this information to Layer 2. The precedence graph for Example 2.2 above is shown in Figure 2, the numbers i at each node correspond to the primitive at step i in the program. (Layer 1 also generates the list of primitives to be executed and sends this list to Layer 2.) The concept of parallel primitives provides a paradigm for parallel programming, and would serve as an object oriented programming language.



Precedence Graph for Example 2.2
Figure 2

2.2 The Primitives Table

The concept of using a database of task characteristics in a parallel processing system was studied by Chu, Delp, Jamieson, et. al. at Purdue in [4] and [6]. They investigated an Image Understanding System, which stores the code and heuristics of appropriate algorithms, that has an intelligent operating system for scheduling and selecting algorithms. However, their system differs from ours in three main areas. First theirs is a special purpose system that can only call and execute image understanding tasks stored in the image database. Second, their objects are entire tasks which are not meant to be primitives used to define new programs. Third, they do not address the communication structure required as a task characteristics while our table includes this important parameter. Their objective does not include reducing the burden of parallel programming.

The set of primitives (with just their definition) 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. Towards this goal, we store the run-time information in the primitives table. 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 one of the parameters of the table. 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 high-level operating system at Layer 2 would attempt to allocate the optimal number of processors required connected in the optimal configuration (i.e., the best network topology and the algorithm for this topology), 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 details of the process of contructing the table and selecting the primitives are discussed in [9], and in this paper we attempt to give a comprehensive outline of only the table organization.

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 3. 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 primitives table 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), and alternative algorithms on the same network for each choice. We are investigating the process of generation of a list of primitives for the user's task. This will involve the use of code parallelization techniques employed in parallel compilers.

The function of the high level operating system (layer 2) is to select a set of algorithms (and the network configurations) 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. Once the algorithms (i.e., configurations) are determined the scheduler is called to allocate and schedule partitions of processors. The system accesses the primitives table to determine efficient algorithms for each primitive. The following sections discuss some features of the layer 2 operating system. We first discuss the process of selecting a set of algorithms (for the set of

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>	<code 1>
	<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	$kN^2/P + c4N/P$	N^2	2-D Array	2-D Array	<code for mesh>
	Hypercube	$kN^2/P + c8N/P$	N^2	2-D Array	2-D Array	<code for cube>
	Ring	$kN^2/P + c4N/\sqrt{P}$	N	2-D Array	2-D Array	<code for ring>
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	$k(\log P + N/P) + c\sqrt{P}$	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 table

Organization of the Primitives Table
Figure 3

primitives to be executed for the task), using the precedence graph, with the objective of minimizing the total execution time. Since selection of an algorithm for a primitive implies selection of an efficient network topology for the algorithm, we would in effect be determining efficient configurations for the task where a configuration is the network topology and the algorithm to run on the topology.

§3. Determining Optimal Configurations

The precedence graph describes the primitives to be executed and the order in which to execute the primitives. The goal now is to determine what alternatives to choose (from the primitives table) so as to minimize the completion time for the task. There are two components of time that we must consider; (1) the execution time for each primitive and (2) the time to reconfigure from one network configuration to another. The execution time includes the computation and communication time for the algorithm that computes the primitive and this time is stored in the primitives table. Since we allow for the selection of different network configurations for executing each primitive, the time to permute (or move) data from one network topology to another (when using the output of one primitive as input to another) must be taken into account. For example, let the first primitive require a Mesh network and let the second primitive (whose input is the output generated by the first primitive) require a tree. For the second primitive to execute, we must move the data in the mesh so that the processors are now configured as a tree. The time to perform this movement of data contributes to the overall execution time. The time for such reconfigurations depends on the complexity of the embedding function that embeds meshes into trees (if we use the same partition of processors for both primitives) or the I/O time (if we assign a different partition for each primitive). The algorithms we discuss will allow for both cases, and one of the contributions in this work is the inclusion of such overheads during selection of optimal configurations for the user's tasks. We also refer to this reconfiguration time as the time for *data movements*. We assume that the data movement times are also stored in the system.

The problem of determining optimal configurations can now be stated as follows:

> Given a precedence graph as input, where each node represents a primitive, *find* for each primitive the algorithm and network (from the primitives table) to be executed such that the completion time, which is the sum of the execution time and the data movement time, is minimized.

Once the optimal selections for each primitive are made, this list of algorithms is sent to the scheduler which then proceeds to allocate and schedule processors to compute the task. In what follows we show that the problem of determining optimal configurations is NP-hard and then present a linear time solution for the special case when the precedence graph is a tree. When the precedence graph is not a tree then we use a general scheduling strategy, for lists of primitives, which is discussed in the next section.

3.1 Complexity of Determining Optimal Configurations

The problem, of determining optimal configurations, will be formulated in graph theoretic terms in order to derive the complexity and efficient algorithms. Let $G(V, E)$ be a graph with vertex set V and edge set E. A graph is called a *directed* graph if there is a direction associated with every edge. A graph $H(V, E)$ is called a *subgraph* of $G(V, E)$ if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. A subgraph $H(V, E)$ of $G(V, E)$ is called

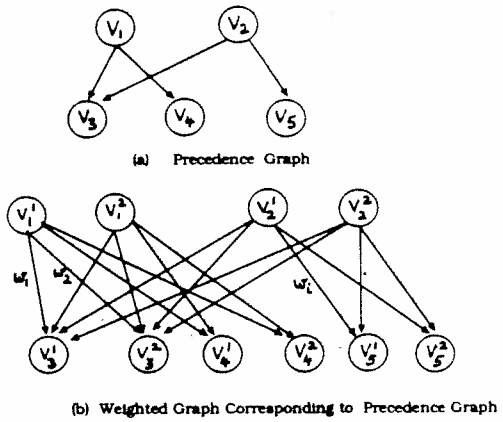
an *induced subgraph* of G if $V(H) \subseteq V(G)$ and $E(H)$ is the set of those edges that have both ends in $V(H)$. For any two sets of vertices V_i and V_j, let $E(V_i, V_j)$ be the set of edges which have one end in V_i and the other end in V_j. For any subgraph H of G, let $w(E(H)) = \sum_{e \in E(H)} w(e)$, $w(e)$ is a pre-assigned value to $e \in E(G)$.

The precedence graph, which is the input to layer 2, is a directed graph where each node represents a primitive to be executed. The edges denote the dependencies between primitives. Since each primitive has k choices (in our table $k = 3$) of algorithms that can be selected, each node V_i in the precedence graph can be denoted by k nodes $V_i^1, V_i^2, \ldots, V_i^k$ where the node V_i^l represents the l-th choice from the table for primitive V_i, $1 \leq l \leq k$. An edge edge between nodes V_i and V_j in the precedence graph indicates that primitive V_i must be computed before primitive V_j, and therefore the time for primitive V_j to complete is the sum of the time for V_i and V_j. If we select the l-th choice from the table for primitive V_i, i.e., V_i^l, and the p-th choice for V_j, then the earliest time for start of V_j is the sum of the execution time for V_i^l, which can be determined from the primitives table, and the time to reconfigure from network required by V_i^l to the network required by V_j^p. For example, if V_i^l the l-th choice required a Mesh and V_j^p required a Tree then the shortest time before we can start V_j is sum of the times for the l-th choice for primitive V_i and the time to reconfigure a Mesh to a Tree. This information can be incorporated into the graph by defining a weight to the edges between primitives, where $w(e)$ the weight of an edge between V_i^l and V_j^p is the sum of the computation time for V_i^l and the reconfiguration time (to reconfigure from network required by V_i^l to the network required by V_j^p). We note that a single edge between V_i and V_j in the precedence graph will be denoted by k^2 edges between the k choices for the two primitives. In other words, an edge (V_i, V_j) in the precedence graph will be replaced by a complete bipartite graph between $\{V_i^1, \ldots, V_i^k\}$ and $\{V_j^1, \ldots, V_j^k\}$. Transforming the precedence graph into the graph representing the various choices for each primitive, and their associated times, as described above results in a weighted graph. We refer to this graph as the weighted graph G corresponding to the precedence graph G_0. Figure 4 gives an example of a precedence graph and its corresponding weighted graph (when we have only two choices for each primitive). The problem of finding an optimal choice of algorithms for each primtive, such that the total time is minimized, is now equivalent to finding a subgraph H of G (the weighted graph) that is isomorphic to the precedence graph G_0 such that the total weight of the edges is minimized, i.e., minimize $w(E(H))$. We refer to this problem as the Minimum Weighted Subgraph problem and in what follows we formally define it and determine its complexity.

Minimum Weighted Subgraph Problem

Instance : A directed acyclic graph $G(V, E)$ and a positive number p such that (i) the vertices of G are partitioned into l disjoint sets $V_1, V_2, \ldots, V_l$; (ii) the edge set of G is defined in such a way that $E(V_i, V_i) = \emptyset$ for all i, $1 \leq i \leq l$ and for any pair of sets V_i and V_j, $1 \leq i, j \leq l$ either $E(V_i, V_j) = \emptyset$ or $E(V_i, V_j) = \{(v_i, v_j)\}$ for all $v_i \in V_i$ and $v_j \in V_j$}, and (iii) to each edge $e = (v_i, v_j) \in E(G)$, there is associated a positive number $w(e)$.

Question : Is there an induced subgraph $H(V, E)$ of $G(V, E)$ with $|V(H)| = l$ such that for all i, $1 \leq i \leq l$, $|V_i \cap V(H)| = 1$ and $w(E(H)) \leq p$?



Weighted Graphs of Precedence Graphs
Figure 4

Note that the induced subgraph with minimum weight corresponds to a set of choices (of algorithms for each primitive) that will minimize the total time.

In this section, we will show that finding a minimum weighted subgraph, as defined earlier, is NP-hard by showing that a restricted case is NP-complete. As usual, our NP-completeness proof will be based on a transformation from a known NP-complete problem, namely the Maximum 2-Satisfiability problem, which is described below.

Maximum 2-Satisfiability Problem

Instance : Set U of variables, collection C of clauses over U such that each clause $c \in C$ has has two literals, and positive integer $k \leq |C|$.
Question : Is there a truth assignment for U that simultaneously satisfies at least k of the clauses in C?

3.1 Theorem. *Let $G(V, E)$ and p be an instance of the Minimum Weighted Subgraph problem. Then it is NP-complete to decide whether there exists an induced subgraph H of G such that $w(E(H)) \leq p$, even if G is bipartite, $|V_i| \geq 2$ for all i, $1 \leq i \leq l$, and the length of any directed path in G is equal to one.*

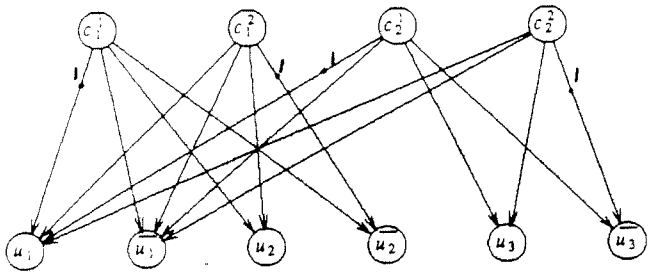
Proof. It is not difficult to see that the problem belongs to the class NP. We next present a transformation from an instance of Maximum 2-Satisfiability problem to an instance of the problem.

Let $U = \{u_1, u_2, \ldots, u_n\}$ be the set of variables, $C = \{c_1, c_2, \ldots, c_m\}$ be the set of clauses and k be the given constant as defined in the Maximum 2-Satisfiability problem. We first construct a bipartite directed graph $G(V, E)$ as follows. The vertex set of G is

$$V(G) = \{c_i^1, c_i^2 | 1 \leq i \leq m\} \cup \{u_j, \overline{u_j} | 1 \leq j \leq n\}.$$

The edge set of G is $E(G) = \{(c_i^1, u_j), (c_i^1, \overline{u_j}), (c_i^2, u_j), (c_i^2, \overline{u_j}) |$ literal u_j or $\overline{u_j}$ appears in clause $c_i\}$. Note that c_i^1 and c_i^2 are new formal vertices that correspond to the first and the second literal of clause c_i, respectively, assuming that the literals in each clause are ordered. Figure 5 shows $G(V, E)$ when $U = \{u_1, u_2, u_3\}$ and $C = \{c_1, c_2\}$ where $c_1 = (u_1 + \overline{u_2})$ and $c_2 = (u_1 + u_3)$. Let $e = (c_i^a, b) \in E(G)$, where $a \in \{1, 2\}$ and $b \in \{u_j, \overline{u_j} | 1 \leq j \leq n\}$. We set $w(e)$ to be equal to one if the

a^{th} literal of clause c_i is b and m^2 otherwise. In Figure 5, $w(e)$ is equal to one, if $e \in \{(c_1^1, u_i), (c_1^2, \overline{u_2}), (c_2^1, u_1), (c_2^2, \overline{u_3})\}$ and m^2, otherwise. It is not difficult to see that the above construction produces a graph G as an instance to the Minimum Weighted Subgraph problem, where the V_i's are $\{c_i^1, c_i^2\}$ for $1 \leq i \leq m$ and $\{u_j, \overline{u_j}\}$ for $1 \leq j \leq n$. Furthermore, G is bipartite, the length of any path in G is one, and the sizes of V_i's are all equal to two. It will be shown that there exists a truth assignment for U that simultaneously satisfies at least k of the clauses in C if and only if there exists an induced subgraph $H(V, E)$ of $G(V, E)$ with $|V(H)| = m + n$ such that either c_i^1 or c_i^2, but not both, is in $V(H)$, either u_j or $\overline{u_j}$, but not both, is in $V(H)$ and $w(E(H)) \leq k - km^2 + 2m^3$.



$w(e) = m^2$ for all edges not marked with weight 1
Figure 5

Suppose there exists a truth assignment for U that simultaneously satisfies k_0 ($\geq k$) of the clauses in C. One can construct an induced subgraph $H(V, E)$ of $G(V, E)$ as follows. The vertex set of H is

$V(H) = \{u_i \in V(G) | u_i \text{ is true }\} \cup \{\overline{u_i} \in V(G) | \overline{u_i} \text{ is true }\} \cup \{c_i^a | 1 \leq i \leq m \text{ such that if the first literal of clause } c_i \text{ is true then } a = 1 \text{ else } a = 2.\}$

The edge set of H is the set of those edges in $E(G)$ that have both ends in $V(H)$. In order to compute the value of $w(E(H))$, we note that the number of edges in $E(H)$ incident at $c_i^a \in V(H)$, for $a = 1$ or 2, is two. Suppose c_i is true and c_i^1 (respectively, c_i^2) is in $V(H)$. Then, the first (respectively, the second) literal of c_i, say z^1 (respectively, z^2), is also in $V(H)$ and the weight of edge (c_i^1, z^1) (respectively, edge (c_i^2, z^2)) is equal to one. Since the weight of the other edge incident at c_i^1 or c_i^2 is equal to m^2, the sum of the weights of these two edges incident at c_i^1 or c_i^2 is equal to $(1 + m^2)$. On the other hand, assume that c_i is not true. Then c_i^2 is in $V(H)$ and the weights of both edges incident at c_i^2 are m^2. This implies that $w(E(H)) = k_0(1 + m^2) + (m - k_0)2m^2$, which is $k_0(1 - m^2) + 2m^3$. Then, $w(E(H)) \leq k(1 - m^2) + m^3$, because $k_0 \geq k$ and $1 - m^2 \leq 0$.

Let $H = (V, E)$ be an induced subgraph of G with $w(E(H)) \leq k - km^2 + 2m^3$ such that $|V(H) \cap \{c_i^1, c_i^2\}| = 1$ and $|V(H) \cap \{u_i, \overline{u_i}\}| = 1$. Then, there are at least k vertices in $V(H) \cap \{c_i^1, c_i^2 | 1 \leq i \leq m\}$ such that the sum of the weights of the edges incident at those vertices are $(1 + m^2)$. To verify this, suppose that there are only k' $(< k)$ such vertices. It follows that since $|V(H) \cap \{c_i^1, c_i^2 | 1 \leq i \leq m\}| = m$, $w(E(H)) = k'(1 + m^2) + (m - k')2m^2$, which is larger than $(k - km^2 + 2m^3)$ assuming that $m > 1$ without loss of any generality. These observations establish the existence of a truth assignment for U that simultaneously satisfies at least k of the clauses in C. In particular, assigning true value to literal u_i (or $\overline{u_i}$) if $u_i \in V(H)$ (respectively, if $\overline{u_i} \in V(H)$) gives a desired assignment. This completes the proof of the theorem. ∎

3.2 A Linear Time Algorithm

In this subsection we shall present and discuss a linear time (linear in the number of edges) algorithm that determines the optimal configurations when the precedence graph is a tree. The precedence graph of many tasks can be represented by a tree. The precedence graph of Example 2.2 is seen to be a composition of two disjoint trees as can be seen from Figure 2.

Let $G(V, E)$ be a directed graph such that $V = V_1 \cup V_2 \cup ... \cup V_l$ as defined in the minimum weighted subgraph problem. We construct a directed graph $G_0(V, E)$ with $|V(G_0)| = l$ as follows. The vertex set of G_0 is

$$V(G_0) = \{v_i | 1 \leq i \leq l\}$$

and the edge set of G_0 is

$$E(G_0) = \{(v_i, v_j) | E(V_i, V_j) \neq \emptyset, \text{ for all } V_i, V_j \subseteq V\}.$$

We call such a graph G_0 the *basis* graph of G. Note that the NP-completeness result in the previous theorem holds even when G_0 is a bipartite graph. However, if G_0 is a tree, there exists a polynomial time algorithm which yields a minimum weighted subgraph of G. Note that in practice, the basis graph G_0 represents the precedence graph. When G_0 is a tree, the children of a node are its predecessors and the root is the terminal node.

Before we describe our algorithm, we note that an input to the algorithm is a weighted graph $G(V, E)$ as defined in the minimum weighted subgraph problem, such that $V = \cup_{1 \leq i \leq l} V_i$ and the basis graph G_0 of G is a tree. Let $V_i = \{v_i^1, v_i^2, ..., v_i^{k_i}\}$ for all i, $1 \leq i \leq l$, and $V(G_0) = \{v_1, v_2, ..., v_l\}$, where v_i corresponds to subset V_i of V. We are now ready to describe our algorithm.

Initially, we have that $w(v_i^j) = 0$ for all $v_i^j \in V$ such that the corresponding vertex v_i in G_0 is a leaf. If $v_i \in V(G_0)$ is not a leaf, then $w(v_i^j)$ is not defined yet. Let $v_i \in V(G_0)$ and $\{v_{i_1}, v_{i_2}, ..., v_{i_p}\}$ be the set of predecessors of v_i in G_0. Note that vertex $v_i \in V(G_0)$ corresponds to subset $V_i \subseteq V$ and vertex $v_{i_q} \in V(G_0)$ to subset $V_{i_q} \subseteq V$ for all q, $1 \leq q \leq p$. For each $v_i^j \in V_i$, compute

$$w(v_i^j) = \sum_{1 \leq q \leq p} \{min\{w(v_{i_q}^{j'}, v_i^j) + w(v_{i_q}^{j'}) | 1 \leq j' \leq k_{i_q}\}\}.$$

This can be done by first computing $w(v_i^j)$ for all $v_i^j \in V$ such that the predecessors of $v_i \in V(G_0)$ are all leaves. We then recursively compute the values of all the remaining vertices until the values of the vertices in V corresponding to the terminal vertex in $V(G_0)$ are computed. After we have completed the computation of $w(v_i^j)$ for all $v_i^j \in V$, the weight of a minimum weighted subgraph of G becomes $min\{w(v_r^j) | 1 \leq j \leq k_r\}$, where $v_r \in V(G_0)$ is the root of G_0. In the above computation, if there exists a tie, we can arbitrarily select one among them. Figure 6 shows the weights computed for the precedence graph shown in the figure. The time complexity of our algorithm is $O(|E'|)$, since each edge (u, v) is considered exactly once to compute $w(v)$.

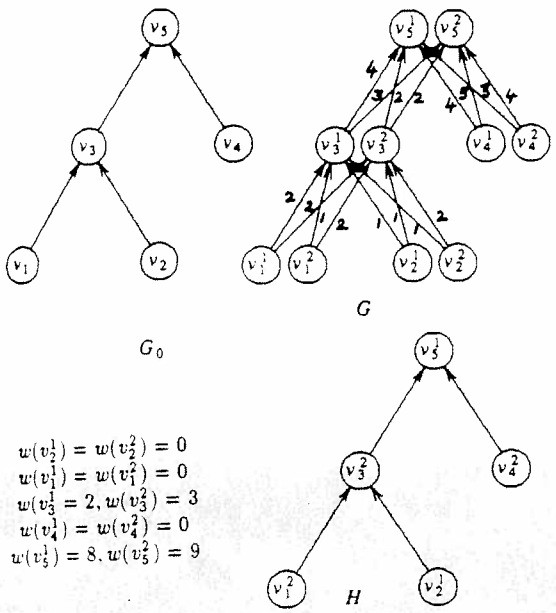


Figure 6

Note that in practice, the basis graph G_0, which represents the precedence graph, is taken as the initial input, while G can but does not have to be explicitly generated. The implementation of the above algorithm in the practical setting of scheduling precedence graphs is given in the next section.

4. Scheduling

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 primitives table to determine a good selection. Layer 2 operates in two modes, when the precedence graph generated by Layer 1 is a tree, Layer 2 employs a linear time algorithm that minimizes the run time of the precedence graph that will be called a *precedence tree*. The details of this part of Layer 2 are discussed in section 4.1. When the predence graph generated by Layer 1 is not a tree, Layer 2 performs scheduling based on the list of primitives supplied by Layer 1, the information in the primitives table, and the current configuration of the system. The details of this part of Layer 2 are discussed in section 4.2. Both sections describe the functionality of Layer 2 which interfaces with both Layer 1 and Layer 3. Therefore, we next describe the the interface between Layer 2 and the these two layers.

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

4.1 Scheduling of a Precedence Tree

Layer 2 starts the scheduling of a precedence tree by invoking *MinRunTime* that computes the minimal run time of a precedence tree. When a precedence graph is a tree, we call it a precedence tree. The parameter of the procedure is the root of the precedence tree. For simplicity of exposition assume that the precedence tree is an abstract data type which exports the following functions and procedures:

Predecessor(V,i) - provides the i-th predecessor of node V.
NumPredecessors(V) - returns the number of predecessors node V has.
Root - provides the root of the precedence tree.
Leaf(V) - returns *true* if V is a leaf in the precedence tree.
AddWeight(V,i,weight) - adds weight to the i-th alternative of node V.
NextPrimitive - returns the primitive and the configuration specified by the next node in the precedence tree to be executed.
Primitive(V) - the primitive specified by node V in the precedence tree.

Assume also that a function called *MinAlternativeWeight* is provided. It takes as parameters a node of a tree and the index of the alternative and computes the minimum weight according to the discussion in section 3.2. After choosing the minimal alternative, the chosen alternative is recorded in the node representing that alternative.

A high level code of *MinRunTime* follows.

```
MinRunTime(Node)
   i,j:integer;
begin
   for i:=1 to Alternatives(Primitive(Node)) do
      for j:=1 to NumPredecessors(Node) do
         if Leaf(Predecessor(Node,j)) then
         AddWeight(Node,i,MinAlternativeWeight(Node,j));
         else
            MinRunTime(Predecessor(Node,j));
         endif;
      endfor;
   endfor;
end RunMinTime;
```

When presented with a precedence tree, Layer 2 issues the call *MinRunTime(Root)* to compute the minimal run time of the tree. After this computation, Layer 2 is ready to schedule the primitives specified by the tree in the chosen configurations. It calls *NextPrimitive* to acquire the primitive and configure that have to be scheduled next. Using this data, Layer 2 attempts to schedule the primitive in the right configuration by calling on Layer 3 to ascertain that the resources needed are available. There are two possibilities in this case, the resources may be available from a primitive in the precedence tree that has just terminated or no previous resources are available. In the first case an attempt to schedule the primitive on the al-

ready allocated resources is made. In the second case, Layer 3 is requested to allocate new resources. If the resources, i.e., number of processors and interconnection network, are available, Layer 3 is requested to allocate the resources, download the algorithm specified in the primitives table, and execute the primitive. If the resources required to execute the primitive are not available, the request is queued using the queueing policy employed by Layer 2. The queued request will be scheduled once it becomes the highest priority request in the queue and the needed resources becomes available.

Layer 3 notifies Layer 2 when a primitive completes execution and retains the data produced by the terminating primitive and the resources allocated to it allocated to the precedence tree. Layer 3 now attempts to scedule the primitive succeeding the terminating primitive. If there is no such primitive, and there are no other nodes remaining in the precedence tree, the task in terminated. If there is a succeeding primitive, Layer 2 attempts to schedule the primitive on the resources available to the terminating one. If this is not possible, Layer 2 in cooperation with Layer 3, will take care of data movements, reconfiguration, and allocations as required.

4.2 Scheduling of Lists of Primitives

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 Neti. 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 primitives table (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.

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 our simulation runs it was observed that using the folding steps leads to a better system

performance (both in utilization and the total execution time). The decision to fold is based on the expected run-time of the algorithm and the current state of the system. Using the approximate run-time information in the table and the current status of the tasks executing on the system, the scheduler can 'predict' when the tasks (currently running on the system) will complete. This information is used by the scheduler to decide whether to allocate a currently available partition (which is not of the optimal size) by *folding* the algorithm or to wait till the first task finishes and assign a different configuration. The details of this process are omitted in this paper.

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 depending 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.

We simulated 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 The hypercube partitioning, i.e., *Layer3_Allocate* was based on a buddy system [2].

§5. Conclusions

This paper presented a system that uses the knowledge of task characteristics to match the task requirements to the system resources and the concept of parallel 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 primitives table to determine efficient configurations for the users tasks and for scheduling attempts, thus removing this responsibility from the user. We presented a linear time algorithm for determining optimal configurations of primitives when the precedence graph is a tree.

The proposed system provides an approach for parallel processing using parallel primitives and their characteristics. In the future, one needs to define more primitives and their specifications, also the inclusion of more parameters, such as the type of processors etc., into the primitives table. In the area of scheduling, an efficient heiuristic to determine optimal configurations for general precedence graphs is needed. To determine efficient partition allocation and scheduling policies we are evaluating various scheduling and partitioning algorithms.

§6. References

[1] J.A. Bondy and U.S.R. Murthy, *Graph Theory with Applications*, American Elsevier Publishing Co.,Inc., New York, 1976

[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] C.H.Chu, E.J.Delp, L.H.Jamieson, H.J.Siegel, F.J.Weil, A.B. Whinston, "A Model for an Intelligent Operating System for Executing Image Understanding Tasks on a Reconfigurable Parallel Architecture", *Journal of Parallel and Distributed Computing*, No.6, pp. 598–622, 1989.

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

[6] 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.

[7] M.R. Garey, D.S. Johnson, and L. Stockmeyer, "Some Simplified NP-Complete graph Problems," *Theoretical Computer Science*, Vol.1, pp. 237–267, 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] B. Narahari, S. Rotenstreich, A. Youssef "A Parallel Primitives Approach to Parallel Programming", *12th Minnowbrook Workshop: Software Engineering for Parallel Processing*, New York, July 1989.

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

[11] 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.

[12] 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.

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