NASA Contractor Report 189047
ICASE Report No. 91-79

ICASE

OPTIMAL PROCESSOR ASSIGNMENT FOR PIPELINE COMPUTATIONS

David M. Nicol
Rahul Simha
Alok N. Choudhury
Bhagirath Narahari

Contract No. NAS1-18605
October 1991

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

OPTIMAL PROCESSOR ASSIGNMENT FOR PIPELINE COMPUTATIONS

David M. Nicol* and Rahul Simha
College of William and Mary
Williamsburg, VA 23185

Alok N. Choudhury
Syracuse University
Syracuse, NY 13244

Bhagirath Narahari
George Washington University
Washington, DC 20052

ABSTRACT

The availability of large scale multitasked parallel architectures introduces the following processor assignment problem for pipelined computations. Given a set of tasks and their precedence constraints, along with their experimentally determined individual response times for different processor sizes, find an assignment of processors to tasks. Two objectives interest us: minimal response given a throughput requirement, and maximal throughput given a response time requirement. These assignment problems differ considerably from the classical mapping problem in which several tasks share a processor; instead, we assume that a large number of processors are to be assigned to a relatively small number of tasks. In this paper we develop efficient assignment algorithms for different classes of task structures. For a p processor system and a series-parallel precedence graph with n constituent tasks, we provide an $O(np^2)$ algorithm that finds the optimal assignment for the response time optimization problem; we find the assignment optimizing the constrained throughput in $O(np^2 \log p)$ time. Special cases of linear, independent, and tree graphs are also considered. In addition, we also examine more efficient algorithms when certain restrictions are placed on the problem parameters. Our techniques are applied to a task system in computer vision.

*Research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the author was in residence at the Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23665-5225.

1 Introduction

In recent years much research has been devoted to the problem of mapping large computations onto a system of parallel processors. Various aspects of the general problem have been studied, including different parallel architectures, task structures, communication issues and load balancing [11, 16]. Typically, experimentally observed performance (e.g., speedup or response time) is tabulated as a function of the number of processors employed. We are particularly interested in tabulations of response time, which we will refer to as *response-time functions*. Our work is also motivated by the growing availability of *multitasked* parallel architectures, such as PASM [37], the NCube system [18], and Intel's iPSC system [7], in which it is possible to map tasks to processors and allow parallel execution of multiple tasks in different logical partitions.

In this paper, we consider the problem of optimizing performance of a *task structure* on a parallel architecture, given a large supply of processors, and the experimentally determined response time functions for its constituent tasks. The task structure describes the sequencing of various computational activities (tasks) that are to be applied to each of many data sets; the data sets themselves are pipelined through the task structure. We refer to this class of computations as *pipeline computations*. This problem arises in data parallel applications such as the computer vision example we consider in this paper, when individual tasks, e.g. a fast Fourier transform, are highly parallelizable. Unlike prior treatments of the mapping problem we are interested in the case where there are many more processors than tasks. Rather than ask which tasks must share a processor, we ask how many processors each task should be allocated. We are interested in both the response time of the task structure on one data set, and in the throughput (data sets processed per unit time). We consider the dual problems of minimizing response time subject to a throughput constraint, and maximizing throughput subject to a response time constraint. These problems are complimentary, in the sense that allocation to increase throughput may have the side effect of increasing response time, and vice versa.

Under the assumption that the constituent task response time functions completely characterize performance, we show that p processors can be optimally allocated to an n-node *series-parallel* task structure in $O(np^2)$ time. We study separately the special cases of linear, and tree structures and show a $O(np^2)$ procedure; we also consider response time function characteristics such as convexity which are exploited to achieve even more efficient algorithms. Our methods are applied to the task of motion estimation in a computer vision system; we present several experimental results for both the response time as well as the throughput problem.

The problem of mapping workload to processors has attracted a great deal of attention in the literature, leading to a number of problem formulations. One often views the computation in terms of a graph, where nodes represent computations and edges represent communication; for an example, see [2]. In this case, mapping means assigning each node (task) to a processor. One view of the mapping problem is that the computation graph represents a distributed program, with a serial thread of control. Tasks have different affinities for different heterogeneous processors; the problem is to assign tasks to processors so that the total sum of execution times (of all tasks) and communication costs is minimized. Fundamental contributions to this problem are made in

[4, 39, 41]. However, the objective function for this problem does not capture any parallelism among the tasks. Another mapping problem formulation views the architecture as a graph whose nodes are processors and whose edges identify processors able to communicate directly. The *dilation* of a computation graph edge (v, v) is the minimum distance (in the processor graph) between the processors to which u and v are respectively assigned. The dilation of the graph itself is the maximum dilation among all computation graph edges. Dilation is a measure of how well the mapping preserves locality between nodes in the mapped computation graph. Results concerning the minimization of dilation can be found in [8, 19, 32, 36], and their references. Yet another formulation directly models execution time of a data parallel computation as a function of the chosen mapping, and attempts to find a mapping that minimizes the execution time. Workload may again be represented as a graph, with edges representing data communication. Nodes are mapped to processors in such a way that each processor's workload is approximately the same, for example, see [1, 5, 24, 33, 35]. Formulations using simulated annealing or neural networks attempt to minimize an "energy" function that heuristically quantifies the cost of the partition [6, 17]. Other interesting formulations consider mapping highly structured computations onto pipelined multiprocessors [25], and mapping systolic algorithms onto hypercubes [22]. The problem we study is distinctly different than these, in that it seeks the assignment of multiple processors to a task, rather than multiple tasks to a processor.

Recently, some studies consider the scheduling of tasks on multitasked parallel architectures where each task can be assigned a set of processors. The objective in such work, for example in [3, 13, 27], is to find a schedule that minimizes completion time. A fundamental difference, between the processor assignment problem studied in this paper and the above scheduling problems, is that scheduling formulations allow tasks to be *queued* or sequenced. In contrast, the nature of pipeline computations recommends assigning at least one processor to each task: executable images which would be swapped into main memory for each data set under scheduling, would remain in main memory under our assignment formulation. The problem of assigning processors to a set of independent tasks where each task is a chain of modules is considered in [10]. This differs from our problem, as neither response-time functions nor task precedence is treated. In other formulations, each task requires a specific number of processors; in this case, the problem of scheduling tasks on a partitionable hypercube or mesh connected architectures has been studied [9, 14, 23, 29]. Pipeline computations are studied in [25, 38]. In [38], heuristics are given for scheduling planar acyclic task structures and in [25], a methodology is presented for analyzing pipeline computations using Petri nets together with techniques for partitioning computations. We have not discovered treatments that address optimal processor assignment to pipeline computations, although our solution approach (dynamic programming) is related to those in [4] and [41].

This paper is organized as follows. Section §2 introduces notation, and formalizes the response-time problem and the throughput problem. Section §3 develops some preliminary results about response time functions that will be used throughout the paper. Section §4 closely examines two response-time problems associated with linear arrays of tasks, and Section §5 applies these results to tasks structured as trees or more general series-parallel graphs. Section §6 shows how the problem

	Number of processors							
tasks	1	2	3	4	5	6	7	8
t_1	29	16	11	9	7	6	4.5	4
t_2	40	21	14	11	8.5	8	7	5
t_3	10	5.5	3.4	3	2.5	2	1.5	2
t_4	20	12	10	9	8	7	6	5
t_5	15	10	8	5	4	3.5	3	2.5

Table 1: Example of Response time functions

of maximizing throughput subject to a response-time constraint can be solved using solutions to the response-time problem. Section §7 discusses application of our techniques to actual problems, and Section §8 summarizes this work.

2 Problem Definition

A *pipeline computation* is a quadruple $\mathcal{P} = < K, T, F, G >$ where

- $K = \{1, ..., p\}$ is a set of identical processors.
- $T = \{t_1, ..., t_{n+1}\}$ is a set of tasks labeled such that t_1 is always the first task and t_{n+1} the last task executed on each data set. We will assume that the last task t_{n+1} is a "dummy" task that requires no processing—it is used for convenience of notation in the graph G, described below.
- $F = \{f_1, ..., f_{n+1}\}$ is a collection of response-time functions $f_i : K \rightarrow \mathbb{R}^+$ for each task. For notational convenience we assume that $f_i(0) = \infty$ for all $i = 1, \ldots, n$. We also assume that $f_{n+1}(x) = 0$ for all x, so that no processors need ever be assigned to the dummy task. It is often convenient to think of the discrete function f_i as a *table*, a format we shall use in this paper. Later, we will also use F to denote the response time functions for a whole task structure.
- $G = (T, E)$ is a directed acyclic graph (DAG) describing the precedence relation for the tasks in T. Thus, $(t_i, t_j) \in E$ if t_i immediately precedes t_j.

An example of response time table for $n = 5$ and $p = 8$ is shown in Table 1. Each row of the entire table is a response time function for a particular task. In the course of the paper we will be constructing examples to demonstrate the use of our algorithms for various graph structures; these examples will use the response time functions in this table.

Our definition of a pipeline computation extends earlier definitions [25, 38] to include the empirically determined response-time functions. Observe that $f_i(k)$ may include the communication costs inherent in executing t_i on k processors, as well as the communication costs t_i may suffer communicating with predecessor and/or successor tasks in T. This paper assumes that all performance dependencies on communication are captured in the response time functions. Our problem formulation does not therefore attempt to deal with any issues related to "matching" the task structure topology to the architecture topology. It implicitly assumes that performance is independent of *which* processors are assigned to a task. These assumptions are reasonable when the cost of communication is largely independent of the distance between communicating processors (as is the case with the Intel iPSC/2 [7]), and the communication bandwidth is sufficiently high for us to ignore effects due to contention between pairs of communicating tasks. They are also reasonable for compute-bound applications, for which load-balancing of the type we study is a major concern. The computer vision application we later consider is compute-bound.

Let $A : T \rightarrow \mathbb{Z}$ denote a feasible assignment of processors to tasks such that $\sum_{i=1}^{n} A(t_i) \leq p$ and $A(t_i) \geq 1$ for all t_i where $1 \leq i \leq n$. Observe that we do not require all p processors to be assigned, as it is possible that increasing the number of processors used actually hampers performance. In addition, observe that each task must be assigned at least one processor; this condition clearly differentiates between an assignment and a schedule.

For a pipeline computation $\mathcal{P}$ and assignment (mapping) A, define the following:

- $S(\mathcal{P}, A) = \max_{1 \leq i \leq n} f_i(A(t_i))$, the largest response time, under A, among all tasks.

- $\Lambda(\mathcal{P}, A) = S(\mathcal{P}, A)^{-1}$. We will later argue that this quantity is the maximal throughput under assignment A, i.e., the maximum rate at which successive data sets can be processed by the task system.

- $L = \{l | l \text{ is a path in } G \text{ starting from } t_1 \text{ ending in } t_{n+1}\}$. L is thus the set of all complete paths through G. We will write each $l \in L$ as a set $\{i_1, ..., i_k\}$, $i_1 = 1, i_k = n+1$, $1 \leq k \leq n+1$, with l consisting of the edges $(t_{i_1}, t_{i_2}), ..., (t_{i_{k-1}}, t_k)$.

- $R(\mathcal{P}, A) = \max_{l \in L} \sum_{i \in l} f_i(A(t_i))$, the "length" of the longest path through G. $R(\mathcal{P}, A)$ is thus the total time required to execute one data set, i.e., the response time.

With these definitions we formulate two problems.

Response time problem:

> Given a pipeline computation $\mathcal{P}$ and throughput requirement λ, find an assignment A^* such that $\Lambda(\mathcal{P}, A^*) \geq \lambda$, and $R(\mathcal{P}, A^*) \leq R(\mathcal{P}, A)$ for every feasible assignment A which satisfies $\Lambda(\mathcal{P}, A) \geq \lambda$.

We are also interested in determining how the optimal response time $R(\mathcal{P}, A^*)$ behaves as a function of p, the maximum number of available processors. In other words, we are interested in obtaining the response time function for the entire computation $\mathcal{P}$: the values of $R(\mathcal{P}, A^*)$ for different values of p. We will call this $\mathcal{P}$'s optimal response time function, or sometimes simply the response time function (the optimality being understood).

Throughput problem:

> Given a pipeline computation $\mathcal{P}$ and response time requirement ρ, find an assignment A^* such that $R(\mathcal{P}, A^*) \leq \rho$, and $\Lambda(\mathcal{P}, A^*) \geq \Lambda(\mathcal{P}, A)$ for every feasible assignment A which satisfies $R(\mathcal{P}, A) \leq \rho$.

The response time problem arises when we have a steady stream of input data arriving at a fixed rate and the system must complete processing each data set as soon as possible. The throughput problem arises when there is flexibility in the amount of time it takes to process one data set but the throughput must be maximized to handle high input data rates. Both conditions appear in real-time applications. Our approach will be to focus first on the response-time problem, for different task structures; in Section §6 we then show how solutions to the response time problem can be used to solve the throughput problem.

3 Preliminaries

Much of this paper is devoted to the issue of decomposing a large task structure into a set of smaller task structures and constructing a response time function for the large structure from response time functions for the smaller structures. This is accomplished by first separately studying algorithms for handling simple task structures such as tasks in series and tasks in parallel. Then more complex task structures such as trees and series-parallel graphs are treated by decomposing the optimization procedure to handle series and parallel components of the overall task structure.

Given x ($x \leq p$) processors and a task structure consisting only of two tasks t_1, t_2, with response time functions f_1, f_2, we wish to determine y such that assigning y processors to t_1 and $x - y$ to t_2 satisfies the throughput requirement and minimizes the overall response time. If we tabulate this minimal response time for each value of x, then we obtain a response time function for the aggregate of t_1 and t_2. Note that this function captures optimality and is thus an *optimal* response time function. In general, given a set of task structures $\{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$, where for $j = 1, \ldots, m, \mathcal{P}_j =< K, T_j, F_j, G_j >$, we extend the notion of response time function for a single task to a response time function for an entire pipeline computation; let $F_j : \mathbb{Z} \rightarrow I\!R$ be the response time function for $\mathcal{P}_j$, i.e., $F_j(x)$ is the optimal response time achieved for $\mathcal{P}_j$ using x processors. Suppose also that we have an m-node graph $\mathcal{G}$ that describes a precedence relation on $\{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$. We may view each $\mathcal{P}_j$ as an arbitrary task, even though $\mathcal{P}_j$ may itself have a complex subtask structure. We wish to construct the optimal response time function for the structure $\mathcal{Q} = \left\langle K, \{\mathcal{P}_1, \ldots, \mathcal{P}_m\}, \cup_{j=1}^{m}\{F_j\}, \mathcal{G}\right\rangle$, given a throughput constraint λ. We accomplish this by solving a number of response-time problems: for every $x \in [1, p]$ processors, we determine the minimal response time $h(x)$ achievable by allocating no more than x processors among the task structures $\mathcal{P}_j$ in such a way that the throughput requirement is satisfied. $h(x)$ becomes the optimal response time function for $\mathcal{Q}$, which now can be treated as a task itself with a known response-time function.

We are interested in properties of optimal response time functions that are conserved through such an aggregation procedure. Two questions are particularly important: (i) what is the minimum number of processors needed for $\mathcal{Q}$ to meet the throughput constraint, and (ii) what is the maximum number of processors that $\mathcal{Q}$ should be allocated? The answer to the first question is straightforward whereas the answer to the second requires additional analysis.

First consider the throughput constraint question. Let $u_\lambda(\mathcal{P}_j)$ denote the minimum number of processors $\mathcal{P}_j$ must be allocated in order to meet throughput constraint λ. For a single task t_i, $u_\lambda(t_i)$ denotes the minimum that must be assigned to task t_i, i.e, $u_\lambda(t_i) = \min_{k \in \mathbb{Z}}\{k : f_i(k) \leq \lambda^{-1}\}$. Observe that any distribution of tasks to $\mathcal{Q}$ must assign at least $u_\lambda(\mathcal{P}_j)$ processors to $\mathcal{P}_j$ if $\mathcal{Q}$ is to meet the throughput requirement. As this is true for each $\mathcal{P}_j$, it is clear that

$$u_\lambda(\mathcal{Q}) \geq \sum_{j=1}^{m} u_\lambda(\mathcal{P}_j). \tag{1}$$

This is true regardless of the structure of $\mathcal{Q}$. It is also true that if every $\mathcal{P}_j$ is allocated $u_\lambda(\mathcal{P}_j)$ processors, then $\mathcal{Q}$'s throughput is at least λ. One need only perform an easy induction on the number of nodes in the precedence graph to establish that $\mathcal{Q}$'s throughput is the inverse of the maximal response-time among all tasks in $\mathcal{Q}$. This shows that the inequality in equation (1) can be reversed, thereby implying equality. Thus, the rule for computing minimal processor requirements for $\mathcal{Q}$ is simple, and general: add the minimal requirements of $\mathcal{Q}$'s constituent tasks.

To answer the second question, especially when $\mathcal{Q}$ is complex, we need to manipulate the functions so that certain conditions are satisfied. For a response time function $f(x)$, define the *reduced response time function* $\bar{f}(x)$ as:

$$\bar{f}(x) = \min_{0<y\leq x}\{f(y)\}$$

Note that $\bar{f}$ is monotonically decreasing (non-increasing), whereas f need not be, and can be defined both for single tasks as well as for whole computations by using the appropriate response time function. In several applications, increasing communication costs when a large number of processors is used can force response times to *increase* with increasing x. In general, we would like to treat response time functions that behave arbitrarily (exhibit several local minima) with increasing x. The adjustment above will prevent assigning "too many" processors. A processor assignment x is called *reducible* if $\exists y < x : \bar{f}(y) < \bar{f}(x)$. It is otherwise *irreducible*. For obvious reasons, we seek irreducible assignments. In the example in Table 1 the response time for task t_3, *i.e.*, $f_3(x)$, can be reduced while all other functions cannot. After the adjustment, we have the reduced response time function with $\bar{f}_3(8) = 1.5$ which assigns only 7 processors to task t_3.

We next derive some properties of reduced response time functions that we will later use in our algorithms. Consider first a simple case of two elemental tasks t_1 and t_2 and their aggregate, s. Suppose $f_1(x)$ and $f_2(x)$ are the response time functions for t_1 and t_2 and $H(x_1, x_2)$ is a real-valued function increasing in both arguments. Define

$$f_s(x) = \min_{0<y\leq x}\{H(f_1(y), f_2((x-y))\}. \tag{2}$$

Here f_s is the optimal response time function of the aggregate task s, written as some function of the response time functions of t_1 and t_2. In this paper, H is usually a sum (for series tasks) or a maximum (for parallel tasks). Define

$$\underline{f}_s(x) = \min_{0<y\leq x} \{H(\bar{f}_1(y), \bar{f}_2(x-y))\}. \tag{3}$$

We next show that:

Lemma 3.1 *For all $x = 1, \ldots, p$, $\bar{f}_s(x) = \underline{f}_s(x)$.*

Proof: We first show that $\underline{f}_s(x)$ is monotone decreasing in x, and therefore $\underline{f}_s(x)$ is already irreducible. Since $\bar{f}_1$ and $\bar{f}_2$ are monotone decreasing and H is increasing, for any y

$$H(\bar{f}_1(y), \bar{f}_2(x-y)) \geq H(\bar{f}_1(y), \bar{f}_2(x+1-y)).$$

Therefore,

$$\min_{0<y\leq x} \{H(\bar{f}_1(y), \bar{f}_2(x-y))\} \geq \min_{0<y\leq x} \{H(\bar{f}_1(y), \bar{f}_2(x+1-y))\},$$

that is, $\underline{f}_s(x)$ is decreasing.

Next, for any $x \geq y > 0$, $\bar{f}_1(y) \leq f_1(y)$ and $\bar{f}_2(x-y) \leq f_2(x-y)$. Thus

$$H(\bar{f}_1(y), \bar{f}_2(x-y)) \leq H(f_1(y), f_2(x-y))$$

and hence

$$\underline{f}_s(x) = \min_{0<y\leq x} \{H(\bar{f}_1(y), \bar{f}_2(x-y))\} \leq \min_{0<y\leq x} \{H(f_1(y), f_2(x-y))\} = f_s(x).$$

As this is true for all $x = 1, \ldots, p$, it follows that

$$\min_{0<y\leq x} \{\underline{f}_s(y)\} \leq \min_{0<y\leq x} \{f_s(y)\} \quad \text{for all } x.$$

But, the left-hand-side of the above is simply $\underline{f}_s(x)$ (by definition); the right-hand-side is $\bar{f}_s(x)$ (also by definition), showing that $\underline{f}_s(x) \leq \bar{f}_s(x)$ for all $x = 1, \ldots, p$.

Finally, we show $\underline{f}_s(x) \geq \bar{f}_s(x)$. For the sake of contradiction suppose $\exists x_0 : \bar{f}_s(x_0) > \underline{f}_s(x_0)$. Then

$$\min_{0<y\leq x_0} \min_{0<w\leq y} \{H(f_1(w), f_2(y-w))\} > \min_{0<z\leq x_0} \{H(\bar{f}_1(z), \bar{f}_2(x_0-z))\}$$

and thus,

$$\forall y \leq x_0 : \min_{0<w\leq y} \{H(f_1(w), f_2(y-w))\} > \min_{0<z\leq x_0} \{H(\bar{f}_1(z), \bar{f}_2(x_0-z))\}. \tag{4}$$

Next let the minimum of the right side of inequality (4) be achieved at $z = z_0$ with value

$$H(\bar{f}_1(z_0), \bar{f}_2(x_0 - z_0)) \;=\; H(f_1(a), f_2(b))$$

with $\bar{f}_1(z_0) = f_1(a)$ and $\bar{f}_2(x_0 - z_0) = f_2(b)$ for some $a \leq z_0, b \leq x_0 - z_0$ and $a + b \leq x_0$. Note that a and b are obtained through the reduction of f_1 and f_2. We may also rewrite inequality (4) as

$$\forall y \leq x_0 : \min_{0<w\leq y} \{H(f_1(w), f_2(y - w))\} \;>\; H(\bar{f}_1(z_0), \bar{f}_2(x_0 - z_0)). \qquad (5)$$

But, with $y = a + b \leq x_0$ above, we get

$$\min_{0<w\leq y} \{H(f_1(w), f_2(y - w))\} \;\leq\; H(f_1(a), f_2(b)) \;=\; H(\bar{f}_1(z_0), \bar{f}_2(x_0 - z_0))$$

which contradicts (5) and therefore, $\bar{f}_s(x) = \underline{f}_s(x)$ ∎

Thus, we have shown that no information is lost in reduction, since the desired optimal response time function of the aggregate $\bar{f}_s$ is obtained using the reduced response time functions of the constituent tasks. This is an important point: we will build up response-time functions for complex tasks using increasing functions H. and minimization equations of the form shown in equation (2). We have just shown that if we start with reduced response time functions, then we will construct reduced response time functions, and the assignments associated with them will be irreducible.

The lemma can be generalized through an easy induction argument for multiple, complex tasks.

Lemma 3.2 *Let $s_1, \ldots, s_k$ be k complex tasks with optimal response time functions $g_1, \ldots, g_k$ and $H(x_1, \ldots, x_k)$ be an increasing function in each argument. If s is the task that represents the aggregate of tasks $s_1, \ldots, s_k$ with reduced optimal response time function $h(x)$ and defining*

$$\underline{h}(x) = \min_{\substack{y_1,\ldots,y_k \in [1,x] \\ y_1 + \ldots + y_k = x}} \{H(\bar{g}_1(y_1), \ldots, \bar{g}_k(y_k))\}.$$

then $\bar{h}(x) = \underline{h}(x)$.

Remark 3.1 *If the irreducible minimums of the functions $\bar{g}_1, \ldots, \bar{g}_k$ occur at $x_1, \ldots, x_k$, then the irreducible minimum of $\underline{h}$, x_0, satisfies $x_0 \leq \sum_{i=1}^{k} x_i$.*

The last remark implies that when constructing $\underline{h}$ we may restrict our attention to only those assignment vectors $(y_1, \ldots, y_k)$ for which $\sum_{i=1}^{k} y_i \leq \sum_{i=1}^{k} x_i$. This will result in improved execution time for our optimization algorithms when $\sum_{i=1}^{k} x_i < O(p)$. Next, we begin our presentation of the algorithms by first treating the two simpler task structures, linear series tasks and linear parallel tasks.

4 Linear Task Structures

Linear task structures are interesting both because many pipelines are simple linear chains [25] and because chains appear as tasks in more complex task structures. We examine two different ways of assessing the cost of a linear chain. The first is when the chain is a linear pipeline, and the response time function is the sum of the response times of each of the 'stages' [25]. This is called a *series* task structure. The second is when the constituent tasks execute in parallel on different aspects of the same data set, a *parallel* task structure. For both problems we show how to construct the optimal response time function for the aggregate task, and, for every $q = 1, \ldots, p$, how to recover the optimal assignment of q processors from information computed as the response time function was constructed.

In the treatments of both problems we consider $s_1, \ldots, s_m$ to be the set of m constituent tasks, and $g_1, \ldots, g_m$ to be their respective response-time functions. Let s be the aggregate task whose optimal response time function $h(x), 0 < x \leq p$, we are interested in computing. Note that each constituent task s_j may already be an aggregation of the elemental tasks t_i. Our immediate goal is to construct the overall reduced response time function for processors in the range $[1, p]$ and also, to recover the optimal assignment when required.

4.1 Series Tasks

First we describe an algorithm that constructs the optimal response time function $h(x)$ for linear task structures when each function $g_i(x)$ is convex (see [30], pp. 445-454) in x, i.e., when the efficiency of parallelism is decreasing (see pp. 217 in [16] for an example). We later treat the general case.

Let the assignment be recorded in $I(s, x) = (x_1, \ldots, x_k)$ where x_j denotes the number of processors assigned to task s_j; also let h_G denote the response time function created by our algorithm. As a first step, we must ensure that every task s_i is allocated enough processors $u_\lambda(s_i)$ to meet the throughput constraint. For each $i = 1, \ldots, m$, let $x_i = u_\lambda(s_i)$ be this initial assignment. Of course, the algorithm terminates at this point if $\sum_{i=1}^m x_i > p$, because no feasible assignment exists. Note that this first step does not require the presumed convexity of each g_i. Let $t = \sum_{i=1}^m x_i$; we set $h_G(x) = \infty$ for all $x < t$ to reflect an inability to meet the throughput requirement, set $h_G(t) = \sum_{i=1}^m g_i(x_i)$, and let $x = t$. Next, for each s_i, compute $d(i, x_i) = g_i(x_i + 1) - g_i(x_i)$, the change in response time achieved by allocating one more processor to s_i. Build a max priority heap [20] where the priority of s_i is $|d(i, x_i)|$. Finally, enter a loop where, on each iteration,

- The task (say s_j) with highest priority is allocated another processor.
- Let a denote the number of processors previously assigned to s_j. Compute $h_G(x) = h_G(x - 1) + d(j, a)$, and set $I(s, x) = (x_1, \ldots, x_j + 1, \ldots, x_k)$.
- Increment x.
- Compute s_j's new priority, and adjust the priority heap accordingly.

We iterate until all available processors have been assigned, or the top element of the heap is non-negative, i.e., $d(j, x_j)$ is non-negative. If the top element becomes non-negative when $x = y$, then we assign $h_G(z) = h_G(y-1)$ and $I(s,z) = I(s,y-1)$ for all $z = y, \ldots, p$.

Each iteration of the loop allocates the next processor to the task which stands to benefit most from the allocation. When the individual task response functions are convex, then the greedy response time function h_G it produces is optimal, and is irreducible.

Prop. 4.1 *Suppose that $g_i(k)$ is convex over $x \in [1,p]$, for all $i = 1, \ldots, n$. Then for all $x \in [1,p]$, $h_G(x) = h(x)$, the optimal response time function. Furthermore, $h_G(x)$ is irreducible.*

Proof: Clearly, each task s_i must receive at least $u_\lambda(s_i)$ tasks in order for the throughput condition to be satisfied. Recalling that $t = \sum_{i=1}^{m} u_\lambda(s_i)$, it is clear that $h_G(x) = h(x) = \infty$ for all $x \in [1, t-1]$. Now consider $x = t$. For all $j = 1, \ldots, p-t$ the remainder of the algorithm should assign "the next" j processors in such a way to obtain the maximal possible decrease in response time given j additional processors. The proposed algorithm does exactly that. $D = \{d(i, x_i + j) | 1 \leq i \leq n, 1 \leq j \leq p-x\}$ is the set of all possible changes for the remainder of the assignment. For every $j = 1, \ldots, p-t$, the maximal decrease is obtained by choosing the j largest (in magnitude) elements of D. Since each g_i is convex, $|d(i, x_i + j_1)| \leq |d(i, x_i + j_2)|$ for $j_1 > j_2$ (see [30], pp. 453-454) and so the j elements with largest magnitude in D are selected as given in the algorithm.

The irreducibility of h_G follows from its construction.

∎

The complexity of this algorithm is low. The throughput condition is checked in m steps. The initial priority heap is constructed in $O(m \log m)$ time; the highest priority heap element is found in $O(1)$ time and each heap adjustment requires only $O(\log m)$ time using standard heap algorithms. Thus the overall complexity is $O(m \log m) + O(p \log m) = O(p \log m)$. This is an example of how the structure of the response time function (convexity) can be used to obtain higher algorithmic efficiency than might otherwise be achievable, as we will see below for general response time functions.

A different approach, based on dynamic programming, is needed when the task response time functions are not convex. In fact, we anticipate that this condition will be the norm when considering chains whose tasks are themselves aggregates of other tasks. Since convexity need not be preserved in aggregation, we must turn to a slightly more complicated algorithm. The new approach has a higher complexity—$O(mp^2)$—but it permits completely general response time functions. We will show that certain algorithmic efficiencies are possible when bounds on the least minimums are known ahead of time.

For any $j = 1, \ldots, m$, we can view the subchain $s_1, \ldots, s_j$ as a (larger) task itself. We will call this task S_j, and compute its optimal response time function: for $x = 1, \ldots, p$ let $G_\lambda(j, x)$ be the minimal response time of S_j, subject to throughput constraint λ, achievable when no more than

x processors are allocated to it. The function $G_\lambda(j,\cdot)$ is thus S_j's optimal response time function; in computing this function we will simultaneously check the throughput constraint—hence the subscript λ. Using the principle of optimality[12], we may write a recursive definition for $G_\lambda(j,x)$ as follows.

$$G_\lambda(j,x) = \begin{cases} \infty & \text{if } u_\lambda(s_j) + u_\lambda(S_{j-1}) > x \\ \bar{g}_1(x) & \text{if } j = 1 \text{ and } u_\lambda(s_1) \leq x \\ \min\limits_{u_\lambda(s_j) \leq i \leq x - u_\lambda(S_{j-1})} \{\bar{g}_j(i) + G_\lambda(j-1, x-i)\} & \text{otherwise.} \end{cases} \quad (6)$$

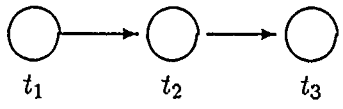
These equations define response time to be ∞ whenever insufficiently many processors are allocated to s_j or S_{j-1} to meet the throughput constraint; we define $u_\lambda(S_0) = 0$ as a boundary condition. Observe that $\bar{h}(x) = G_\lambda(m,x)$. Note that the H function (Lemma 3.2) is the 'sum' operator here, in the third part of the equation.

The dynamic programming equation is more intuitively explained by reading it 'top down'. Suppose we had somehow computed the response time table for the first $j-1$ tasks (the 'large' task S_{j-1}), i.e., $G_\lambda(j,x)$. Then, given x processors to distribute between tasks s_j and S_{j-1}, we try every combination subject to the throughput constraints: i processors for s_j and $x-i$ processors for S_{j-1}. Since the equation is written as a recursion, the computation will actually build response time tables for larger tasks 'bottom up', starting with task s_1 in the second part of the equation. Note that similar explanations may be given for the dynamic programming equations that appear later in the paper. The optimal assignment of q $(1 \leq q \leq p)$ processors to tasks is found by setting the appropriate value of I as we solve for the value $G_\lambda(j,x)$. Suppose that i solves $G_\lambda(j,x) = \bar{g}_{j-1}(i) + G_\lambda(j-1, x-i)$. Then we set $I(S_j,x) = (x_1,\ldots,x_{j-1},i)$, where $I(S_{j-1},x-i) = (x_1,\ldots,x_{j-1})$.

An important consequence of Lemma 3.2 is that each function $G_\lambda(j,\cdot)$ (and hence each assignment $I(S_j,x)$) is irreducible. This follows directly from the fact that equation (6) has the form specified by equation (3). The more complex bounds on the minimum's index variable in equation (6) serve simply to keep the index i away from regions where either $\bar{g}_j(\cdot)$ or $G_\lambda(j-1, \ )$ are known to take value ∞.

If we have already solved for the minimal response time function $G_\lambda(j-1,\cdot)$, we may use equation (6) to determine $G_\lambda(j,\cdot)$. The cost of determining one individual $G_\lambda(j,x)$ value is seen to be $O(x) = O(p)$; the cost of determining the whole function $G_\lambda(j,\cdot)$ is thus $O(p^2)$, and the cost of determining all such functions (and hence the desired response time function $G_\lambda(m,\cdot)$) is $O(mp^2)$.

The application of the above dynamic programming procedure, in equation (6), is illustrated in Figure 1 (which shows the computation of $G_\lambda(j,\cdot)$) for a task structure with three tasks. The response time functions, $g_i(x)$, for the three tasks t_1, t_2 and t_3 are taken from Table 1 and the throughput constraint $\lambda = 1/40$. Since we use tasks from Table 1, we revert to using t_i for the constituent tasks. The first column of the table identifies the aggregated task S_j, for $1 \leq j \leq 3$; here $S_1 = t_1$, $S_2 = (t_1,t_2)$ and $S_3 = (t_1,t_2,t_3)$. A row j corresponds to the response time function $G_\lambda(j,x)$, for aggregated task S_j; entry $[k,l]$ in the table (row k, column l) gives the value, and the corresponding assignment, for $G_\lambda(k,l)$. The last row shows the assignment produced by the



S_j	x					
	3	4	5	6	7	8
S_1 (t_1)	11 (3)	9 (4)	7 (5)	6 (6)	4.5 (7)	4 (8)
S_2 (S_1, t_2)	50 (1,2)	37 (2,2)	30 (2,3)	25 (3,3)	22 (3,4)	19.5 (3,5)
S_3 (S_2, t_3)	79 (1,1,1)	60 (1,2,1)	47 (2,2,1)	40 (2,3,1)	35 (3,3,1)	30.5 (3,3,2)

Figure 1: Application of Algorithm for series tasks: $G_\lambda(j, x)$ for $1 \le j \le 3$, $1 \le x \le 8$

algorithm; this assigns 3 processors to tasks t_1 and t_2 and 2 processors to t_3 with minimum response time of 30.5 and an achieved throughput of 1/14. Note that in our example above, and in all other examples to follow, we have omitted the dummy task that is the last task executed on the data set, since it plays no role in the computation.

The dynamic programming equations can sometimes be solved more efficiently, when each $\bar{g}_i$ has an irreducible minimum at z_i, and each z_i is small relative to p. Suppose $z_i \le L$ for all $i = 1, \ldots, m$. We next show how the optimality equations can be solved in $O(m^2L^2)$ time. This is advantageous when $L < O(p/\sqrt{m})$.

As we solve for each $G_\lambda(j, k)$, Remark 3.1 also tells us that we need not consider assigning any more than $z_j \le L$ processors to s_j. This means we can rewrite the optimality equations as

$$G_\lambda(j, x) = \begin{cases} \infty & \text{if } u_\lambda(s_j) + u_\lambda(S_{j-1}) > x \\ \bar{g}_1(x) & \text{if } j = 1 \text{ and } u_\lambda(s_1) \le x \\ \displaystyle\min_{\max\{u_\lambda(s_j),\, x - \sum_{a=1}^{j-1} z_a\} \le i \le z_j} \{\bar{g}_j(i) + G_\lambda(j-1, x-i)\} & \text{otherwise.} \end{cases} \tag{7}$$

The complex lower bound on i prohibits indexing values of i such that S_{j-1} cannot meet the throughput constraint, and values indexing beyond S_j's known minimum. Thus, the cost of computing $G_\lambda(j, x)$ is only $O(L)$. Since we need only compute $G_\lambda(j, k)$ for $x \le \sum_{i=1}^{j} z_j$, the cost of computing $G_\lambda(j, \cdot)$ is $O(jL^2)$, so that the cost of solving the overall problem is $O(\sum_{j=2}^{m} jL^2) = O(m^2L^2)$.

4.2 Parallel Tasks

In this subproblem, we have a sequence S of tasks $s_1, \ldots, s_m$ with irreducible response-time functions $\bar{g}_1, \ldots, \bar{g}_m$ for which we need to determine the irreducible optimal response-time function $\bar{h}(x)$

for the *maximum* where

$$h(x) = \min_{\substack{x_1,\ldots,x_m \\ x_1+\cdots+x_m = x}} \max\{\bar{g}_1(x_1), \bar{g}_2(x_2), \ldots, \bar{g}_m(x_m)\}.$$

In this case, the function H (in Lemma 3.2) is the maximum operator. The basic idea behind the algorithm is that after processors are allocated to meet the throughput requirement, we can only drive the maximum response time down by allocating a processor to the task whose response time under the present allocation is maximal. This process is repeated until the maximum number of needed processors is allocated. This idea is now made more precise.

Suppose that the irreducible minimum of each $\bar{g}_i$ occurs at z_i, and let $z_h = \sum_{i=1}^{m} z_i$. First, observe that the response time function value at all processor counts smaller than $t = \sum_{i=1}^{m} u_\lambda(s_i)$ is ∞. Thus, for $i = 1, \ldots, m$, we begin by assigning $u_\lambda(s_i)$ processors to task s_i. This is also reflected in the initialization of the data structure recording assignments, as $I(S,t) = (u_\lambda(s_1), \ldots, u_\lambda(s_m))$. Set $h(x) = \infty$ for $x = 1, \ldots, t-1$, and $h(t) = \max_{1 \le i \le m}\{\bar{g}_i(u_\lambda(s_i))\}$. Next build a max-priority heap on the tasks, where $\bar{g}_i(u_\lambda(s_i))$ is the priority for task s_i. Let $x = t+1$, and enter a loop where the following is performed for at most $z_h - t$ iterations.

- Give an additional processor to the task whose priority is greatest. Let y_x be that maximal priority.
- If that task (say s_i) was previously assigned x_i processors, and if $x_i = z_i$, then terminate the algorithm.
- If that task (say s_i) was previously assigned $x_i < z_i$ processors, reset its new priority to $\bar{g}_i(x_i + 1)$. Set $I(S,x) = (x_1, \ldots, x_i + 1, \ldots, x_m)$, where $I(S, x-1) = (x_1, \ldots, x_i, \ldots, x_m)$.
- Adjust the max-priority heap to reflect the task's new priority, and set $h(x)$ to the maximum value in the heap.
- Increment x.

If the loop terminates with $x = y$, then set $h(z) = h(y-1)$ and $I(S,z) = I(S,y-1)$ for all $z = y, \ldots, p$.

The termination condition follows from the observation that if s_i has the maximum response time but already has z_i processors assigned, no further assignment of processors to s_i can reduce its response time. Since the objective function is the maximum response time among tasks, that objective function cannot be further reduced. It is clear then that the procedure we describe constructs an irreducible function. The algorithm's correctness is established with the following lemma.

Lemma 4.1 *For every $x = t, \ldots, p$, $h(x) = \bar{h}(x) = y_x$.*

Proof: For every $i = 1, \ldots, m$, let $S_i = \{\bar{g}_i(x) \mid x = u_\lambda(s_i), \ldots, z_i\}$ be the set of feasible response times for s_i following its initial assignment, and let $S = \cup_{i=1}^{m} S_i$. Since the objective function value for an assignment is the maximum response time under that assignment and since we stop assigning processors once the objective function can no longer be minimized, S contains every value of y_x generated by our algorithm. Furthermore, the sequence $y_t, y_{t+1}, \ldots,$ describes the elements of S in descending order. Now if an assignment is to achieve cost y_x, the response time of every task must be no greater than y_x. We argue that our algorithm finds an assignment achieving cost y_x, using the minimum number of processors. For every y_i let $T(y_i)$ be the task from whose response-time function y_i is taken. Our algorithm allocates an additional processor to $T(y_1)$, then another to $T(y_2)$, and so on. For every $x = t, \ldots, z_h$ and $j = 1, \ldots, m$ let $P_j(x)$ be the number of elements y_a with $a < x$ for which $T(y_a) = s_j$. $P_j(x)$ is thus the number of additional processors our algorithm has allocated to s_j by the $(x - t)^{th}$ pass through the loop, and is also the minimum number of additional processors (after $u_\lambda(s_j)$) that s_j must be assigned if its response is to be no greater than y_x. As this is true for every task for every y_x, it follows that the assignment generated by our algorithm achieves each cost y_x with the minimum number of processors. The lemma's conclusion is a restatement of this fact. ∎

Since the algorithm's loop is executed at most $z_h - t$ times, the overall cost of the algorithm is $O(m \log m + z_h \log m)$. The optimal assignment is found in $I(S, p)$. An example of the application of this algorithm is shown in the next section; in Figure 2 the row for B_1 shows the response time function (and the corresponding assignment) of a parallel task composed of tasks t_1 and t_2.

While the problems studied in this paper are distinctly different from those addressed in the literature, a closer look reveals that the above algorithm (for parallel tasks) is a generalization of the algorithm independently conceived in [27]. While they address the problem of finding a nonpremptive schedule for a set of n independent tasks, *i.e.*, parallel tasks, their algorithm in fact finds an assignment which satisfies the feasibility conditions of our problem. Our algorithm is a generalization in the sense that they do not "construct" a reduced response time table for the entire parallel task that provides the response time as a function of the number of processors. This is essential for our solution technique which views complex task structures as composition of simpler task structures.

5 Complex Tasks

The algorithms we have developed to analyze series and parallel task structures can be used to analyze task-structures whose graphs form trees, or series-parallel graphs. We now show how the response time function for a tree task with n nodes and arbitrary branching is computed in $O(np^2)$ time, and how a series-parallel task with arbitrary branching is analyzed in $O(np^2)$ time. Note that the complex tasks we consider usually determine a whole pipeline computation and thus, we will henceforth use n (as in Section 2) to denote the number of nodes in the task graph. Series-parallel graphs arise frequently in applications where data in a set is split, processed separately, and then rejoined. The basic idea behind our algorithms is that these complex structures can be viewed as a composition of series and parallel tasks, thus facilitating the use of the algorithms designed thus far.

5.1 Tree Tasks

Suppose the precedence graph for $\mathcal{P}$ forms a tree with n nodes. Either out-trees (edges directed to child nodes) or in-trees (edges directed to parent node) are permissible. Without loss of generality (because path lengths are unaffected by arc direction) our discussion will concern out-trees.

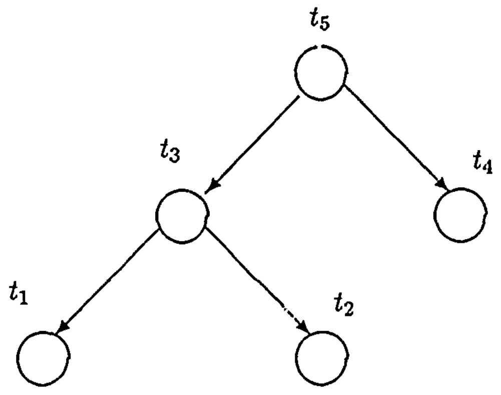
For notational convenience we assume that every non-leaf node has exactly b children; our approach extends immediately to the general case. For every task s_j, let $c_{j,1},\ldots,c_{j,b}$ be s_j's children. s_j is the root of a subtree which can be viewed as a subtask T_j with its own response time function. Dynamic programming again expresses the optimal response time function for each T_j. The optimal response time function for T_1 is the overall problem solution.

Let $G_\lambda(j,x)$ be the optimal response time achievable by T_j when subject to throughput constraint λ. Let $\mathcal{I}$ be the set of interior tree tasks, and $\mathcal{L}$ be the set of leaf tasks. The principle of optimality states that

$$G_\lambda(j,x) = \begin{cases} \infty & \text{if } s_j \in \mathcal{L} \text{ and } u_\lambda(s_j) > x \\ \min\limits_{\substack{x_0,\ldots,x_b \\ x_0+\cdots+x_b=k}} \{\bar{f}_j(x_0) + \max\limits_{1\le i\le b} \{G_\lambda(c_{j,i},x_i)\}\} & \text{otherwise.} \end{cases}$$

The formidable recursive expression simply takes the minimum cost over all possible partitionings of k processors among s_j and the b subtrees rooted in its children. Fortunately, the results developed in Section §4 may be employed to solve this equation efficiently. The subtasks $c_{j,1}$ through $c_{j,b}$ form a single parallel task, B. The algorithm developed in the previous section constructs B's irreducible response time function in $O(p\log b)$ time. Next we can view T_j as a series task, composed of s_j and B. Given B's response time function, T_j's irreducible response time function is computed in $O(p^2)$ additional time using the algorithm described in Section §4.1. Thus, the cost of computing the serial composition dominates. The complexity of computing response time functions for all T_j where $s_j \in \mathcal{I}$ is $O(\sum_{s_j\in\mathcal{I}} p^2)$. Note however that $b|\mathcal{I}| = n$, which implies that the total cost of processing interior tasks is $O(np^2/b)$. Since the cost of processing all leaf tasks is $O(n)$, the total cost in the general case is $O(np^2/b)$.

The procedure is illustrated by the example in Figure 2, a tree with 5 constituent tasks; here $\lambda = 1/40$. The tasks t_1, t_2 form a parallel task, denoted B_1; B_1 and t_3 form a series task, denoted T_3. Similarly, the aggregate task T_3 and t_4 form a parallel task B_2; B_2 and t_5 form a series task T_5 whose response time gives us the response time of the entire task. Note that the tasks $t_1,\ldots,t_5$ are taken from Table 1. Each row of the table shows the response time assignment for the corresponding aggregated task. The minimum response time achieved by the assignment is 41



task aggregates	x = 5	6	7	8
B_1 (t_1, t_2)	16 (2,3)	14 (3,3)	11 (3,4)	11 (4,4)
T_3 (t_3, B_1)	31 (2,2,1)	26 (2,3,1)	21.5 (2,3,2)	19.5 (2,3,3)
B_2 (t_4, T_3)	39 (1,2,1,1)	31 (2,2,1,1)	26 (2,3,1,1)	21.5 (2,3,2,1)
T_5 (t_5, B_2)	65 (1,1,1,1,1)	54 (1,2,1,1,1)	46 (2,2,1,1,1)	41 (2,3,1,1,1)

Figure 2: Application of Algorithm for Tree Structures

(by assigning 2 processors to t_1, 3 to t_2 and one processor to each of the other three tasks) and the achieved throughput is 1/20.

Better complexities are achievable when the irreducible minima z_i for each s_j satisfy $z_i \leq L$ where $L \ll p$. The computation of B's response time function is fast—$O(bL \log b)$ time. For $s_j + B$, let z_{T_j} be the sum of the z_i values for all nodes in the subtree rooted in s_j. Since we need not consider any assignment that gives more than z_j processors to s_j, the response time function for $s_j + B$ is computed in $O(z_{T_j} L)$ time. This cost dominates that of computing B's response time function, provided that $b \log b < L$, which we will assume here for simplicity.

The total cost of analyzing the tree is maximized when each X_{T_j} is as large as possible. This occurs when the tree is actually just a linear chain, in which case $X_{T_n} = L$, $X_{T_{n-1}} = 2L$, $X_{T_{n-2}} = 3L$, and so on. As we have seen, the total cost is then $O(n^2L^2)$. The best topology is a full tree; for example, consider a full binary tree. A subtree T_j consisting of exactly 3 tasks has $x_{T_j} \leq 3L$, and an analysis cost of $O(3L^2)$. $n/2$ such subtrees are analyzed. Then, $n/4$ subtrees are analyzed where $x_S \leq L + 3L + 3L = 7L$. Each of these requires $O(7L^2)$ time to analyze. Continuing in this fashion we determine a complexity bound of

$$O(\sum_{i=1}^{\log n} \frac{n}{2^i}(2^{i+1} - 1)L^2) = O(L^2 n \log n).$$

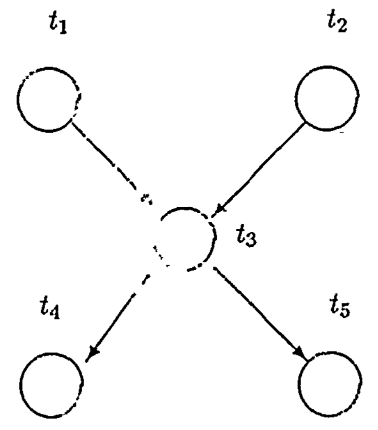
5.2 Series-Parallel Tasks

Finally, we consider series-parallel task graphs. We show that the response time function for such a graph (with n nodes) can be computed in $O(np^2)$ time. A number of different but equivalent definitions of series-parallel graphs exist. The one we will use is taken from [42], which studies *vertex series-parallel* DAGs. However, based on their results on the equivalence of edge series-parallel DAGs and vertex series-parallel DAGs, we use the term series-parallel to mean both cases and use their definition of vertex series-parallel DAGs. A series-parallel DAG (SP) is defined recursively as follows.

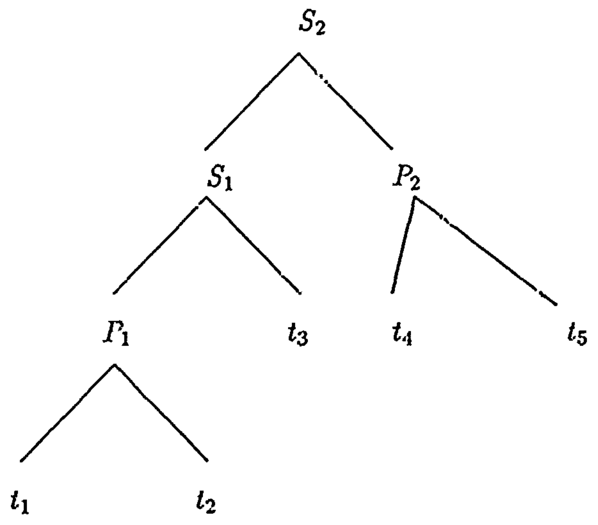
1. (i) The DAG having a single vertex and no edges is SP.
2. (ii) If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are two SP DAGs, so are the DAGs constructed by each of the following two operations:
   (a) *Parallel composition*: $G_p = (V_1 \cup V_2, E_1 \cup E_2)$.
   (b) *Series composition*: $G_s = (V_1 \cup V_2, E_1 \cup E_2 \cup (T_1 \times S_2))$, where T_1 is the set of sinks of G_1 and S_2 is the set of sources of G_2.

A node t_i in $G = (V, E)$ is a sink if there are no outgoing edges from t_i, i.e., there is no edge (t_i, t_j) in E. A node t_i is a source if there are no incoming edges to the node, i.e., there is no edge (t_j, t_i) in E. It is shown in [42] that any SP DAG can be parsed as a binary decomposition tree (BDT). Figure 3 illustrates a series-parallel graph, and the BDT that represents the graph. The internal nodes are labeled S_i or P_i to denote the series or parallel composition. There is a one-to-one correspondence between BDT leaves and DAG nodes. Each internal BDT node a represents either a series (labeled S) or parallel (labeled P) composition of two SP subgraphs represented by the subtrees rooted in a. For example, suppose a's subtrees are simply leaf nodes. The corresponding nodes in the DAG are SP graphs, composed by the operation specified in a's label. a can be thought to be representing that composition. Now if a's BDT parent is some node q and q has another child a', then we know that a' represents an SP subgraph of the original DAG, and q represents the series or parallel composition of the subgraphs represented by a and by a'. A BDT thus shows the selection and ordering of compositions necessary to establish that the original DAG is SP with respect to the definition above.

There is an obvious correspondence between SP compositions and the methods we have developed to compute response time functions for series and parallel task structures. If we think of an SP DAG's nodes as representing tasks, a series composition corresponds to the aggregation of two tasks into a series task structure: two tasks are replaced by one, and the serial edge between them



(a) A series-parallel graph



(b) Binary decomposition tree

Figure 3: A Series-Parallel Graph and corresponding BDT

Task aggregates	Number of processors			
	5	6	7	8
P_1 parallel:(t_1, t_2)	16 (2,3)	14 (3,3)	11 (3,4)	11 (4,4)
S_1 serial:(p_1, t_3)	31 (2,2,1)	26 (2,3,1)	21.5 (2,3,2)	19.4 (2,3,3)
P_2 parallel:(t_4, t_5)	10 (3,2)	10 (3,3)	9 (4,3)	8 (5,3)
$G = S_2$ serial: (s_1, p_2)	70 (1,1,1,1,1)	59 (1,2,1,1,1)	51 (2,2,1,1,1)	46 (2,3,1,1,1)

Table 2: Computation of Response times for series-parallel structures

disappears. Similarly, a parallel composition corresponds to the aggregation of a set of tasks into a parallel task structure. It is thus quite straightforward to construct the response time function for a series-parallel graph, once the associated BDT is known. Starting with the individual tasks' response time functions, we compose response-time functions in the order specified by the BDT. The response time functions created during intermediate steps represent aggregate subtasks in much the same way as task T_j represented an entire subtree in Section §5.1. Likewise, the optimal assignment is recovered by backtracking through intermediate optimal assignments in the same way as was described for trees.

An application of our procedure, for the series-parallel graph in Figure 3, is shown in Table 2 for throughput constraint $\lambda = 1/40$. Each row shows the response time function, and corresponding assignment, for the aggregate task formed by a series or parallel composition. For example, the row labeled S_1 corresponds to the aggregate task formed by the series composition of P_1 (which is a parallel composition of t_1 and t_2) and t_3. The minimum response time in the above assignment is 46 (assigning 2 processors to t_1, 3 to t_2 and one processor each to t_3, t_4 and t_5) and the achieved throughput is 1/20.

Once the BDT is known, the cost of determining the optimal assignment is $O(np^2)$, as every response-time function composition has cost $O(p^2)$; there are at most n such compositions performed. As we have seen before, the cost is reduced to $O(L^2 n \log n)$ when the irreducible minima z_i for each s_i satisfies $z_i \leq L$. It is shown in [42] that a BDT can be constructed time proportional to the number of edges which is $O(n^2)$ time. Since we assume $n < p$, the $O(np^2)$ analysis cost dominates the procedure.

6 The Throughput Problem

In computations where the input data rates must be maximized to handle real time constraints, the objective of the system is to achieve a high throughput. Typically, there is a limit on the amount of time the system can take to process a single data set, *i.e.,* the response time. Under these conditions the objective of an assignment becomes maximization of the throughput subject to a specified response time requirement. We have referred to this problem as the *throughput* problem. In this section we show how solutions to the response-time problem can be used to solve the throughput problem. If one can solve the response-time problem for a given pipeline computation in $O(C(n,p))$ time, then one can solve its throughput problem in $O(np\log(pn) + \log(np)C(n,p))$ time.

Our approach depends on the fact that minimal response times behave monotonically with respect to the throughput constraint.

Lemma 6.1 *For any pipeline computation $\mathcal{P} =< K,T,F,G >$, let $\rho(\lambda)$ be the minimal possible response time of $\mathcal{P}$, given throughput constraint λ. Then $\rho(\lambda)$ is a monotone non-decreasing function of λ.*

Proof: Recall that $u_\lambda(t_i)$ is the minimum number of processors required for task t_i to meet throughput constraint λ. For every t_i, $u_\lambda(t_i)$ is clearly a monotone non-decreasing function of λ. Call an assignment A λ-*feasible* if, for all $i = 1,\ldots,n$ it assigns at least $u_\lambda(t_i)$ processors to t_i. Finally, let $\mathcal{A}_\lambda$ be the set of all λ-feasible assignments. Whenever $\lambda_1 < \lambda_2$, we must have $\mathcal{A}_{\lambda_2} \subseteq \mathcal{A}_{\lambda_1}$, because of the monotonicity of each $u_\lambda(t_i)$. Since $\rho(\lambda)$ is the minimum cost among all assignments in $\mathcal{A}_\lambda$, we have $\rho(\lambda_2) \leq \rho(\lambda_1)$. ∎

This result can be viewed as a generalization of Bokhari's graph-based argument for monotonicity of the minimal "sum" cost, given a "bottleneck" cost [5].

Suppose for a given pipeline computation we are able to solve for $\rho(\lambda)$, given any λ. The set of all possible throughput values is $\{1/f_i(k) \mid i = 1,\ldots,n; k = 1,\ldots,p\}$; $O(pn\log(pn))$ time is needed to sort them. Now suppose a response time constraint $\hat{\rho}$ is given. For any given throughput λ we may compute $\rho(\lambda)$, and determine whether $\rho(\lambda) \leq \hat{\rho}$. $\rho(\lambda)$ is monotone in λ, which permits us to perform a binary search over the sorted space of throughputs and identify the greatest one, say λ^*, for which $\rho(\lambda^*) \leq \rho$. The assignment associated with $\rho(\lambda^*)$ is the one maximizing throughput using p processors, subject to response time constraint $\hat{\rho}$. If the cost of solving one response-time problem is $O(C(n,p))$, then the cost of solving the throughput problem is $O(pn\log(pn) + C(n,p)\log(pn))$.

Lemma 6.2 *Let $\mathcal{P}$ be a pipeline computation, and suppose that the complexity of solving the response-time problem for $\mathcal{P}$ is $O(C(n,p))$. Then the complexity of solving the throughput problem for $\mathcal{P}$ is $O(pn\log(pn) + C(n,p)\log(pn))$.*

When solving the response time problem, we typically compute an entire response time function, which essentially gives the "answer" (minimal response time) for a whole range of processors. When we solve the throughput problem in the manner just described, we compute a single answer, for a single processor count. If we desire a range of throughputs for a range of processors, we need to repeat the procedure above once for every processor count.

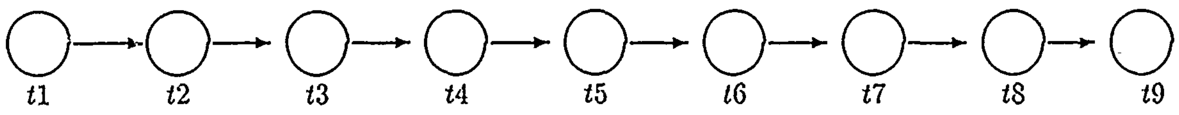


Figure 4: Computation Flow for Motion Estimation

The complexity of the algorithms for the throughput problem are seen to be higher, by a logarithmic factor, than those for the response time problem. For example, the complexity for serial task structures is seen to be $O(np^2 \log np^2) = O(np^2 \log p)$ which has increased by a logarithmic factor. Future endeavors include the pursuance of more efficient algorithms for the throughput problem.

7 An Application

In this section we illustrate our methods by considering an application requiring pipelined execution - a motion estimation system in computer vision. Motion estimation is an important problem in computer vision in which the goal is to characterize the motion of moving objects in a scene. ¿From a computational point of view, continually generated images from a camera must be processed by a number of tasks. In order to process the images (data sets), throughput and response time constraints are imposed on the tasks and therefore, the appropriate model of computation is a pipeline computation. The application itself is described in detail in [11, 28] It should be noted that there are many approaches to solving the motion estimation problem. We are only interested in an example, and therefore, the following algorithm is not presented as the only or the best way to perform motion estimation. A comprehensive digest of papers on the topic of motion understanding can be found in [31]. The following subsection briefly describes the underlying computations.

7.1 A Motion Estimation System

Figure 4 shows the task structure of our motion estimation system [11] - a linear task structure. The data sets input to the task system are a continuous stream of stereo image pairs of a scene containing the moving vehicles. The required output is a list of 3-dimensional points (or features) that describe the motion at each time step.

The system consists of nine major tasks:

1. Task t_1. The first task performs 2-D convolution on the input image pair. The convolution window size is an image-size independent input parameter.

2. Task t_2. The second task extracts the zero crossings of the convolved image using a thresholding algorithm. Zero crossings represent edge features in the image.

3. Task t_3. The third task fits patterns to the edge features by using a template matching algorithm. There are 24 possible patterns that can be fit to an edge [21].

4. Task t_4. The fourth task performs a stereo match algorithm to match features from the left and right images of the same time frame [28]. To find a match for a feature in the left image from the right image, weighted sum of the correlation coefficient and the directional difference weight between the feature in the left image and for all the features in the search space of the right image are calculated. The feature in the right image that has the maximum total weight is considered as the matched feature. Details are provided in [28, 11].

5. Tasks t_5, t_6 and t_7. These are similar to t_1, t_2 and t_3 respectively except that the algorithms are applied to stereo images separated in time by wider margins, depending on the desired accuracy for estimation.

6. Task t_8. This task performs a time match algorithm between matched features of the left image obtained from t_4 and features of the left image obtained from t_7. The time match process is similar to the stereo match process except for the fact that first stereo match guides the time match process and the search space for the time match algorithm is much larger.

7. Task t_9. Finally, the ninth task performs a second stereo match between the left and right images of the stereo images from later time frames. The output of t_9 is a set of 3-D feature points that describe the motion of an object between the two time frames.

All nine tasks are repeated for image inputs obtained continuously. In order to represent real-time motion estimation at video frame rates the entire process must be completed in 0.0333 seconds. The Image Understanding Benchmark [13] has a similar structure of computation flow several tasks must be performed in a sequence in order to recognize an object in the scene and find the model that best describes the object.

7.2 Shared and Distributed Multiprocessors

All nine tasks were implemented on a distributed memory machine, the Intel iPSC/2 [7] and a shared memory machine, the Encore Multimax [15]. The Intel iPSC/2 is a circuit-switched hypercube multiprocessor. We used a 32 node iPSC/2 machine. Each node consists of an Intel 80386 processor and a floating point co-processor together with 4 Mbytes of RAM and and 64 Kbyte cache. The Encore Multimax 520 is a bus based system installed with eight dual processor cards. Each dual incorporates two NS32532 processors each with its of own 256 Kbyte cache of fast static RAM. It has 128 Mbytes of shared memory.

7.3 Implementation Results for Individual Tasks

We implemented the task system described above using outdoor images [11]. Several methods for implementing each algorithm (e.g., block partitioning, dynamic partitioning [11]) were used, for each task, we have selected the best performance numbers from these alternatives. The completion times for each algorithm were tabulated and are shown in Tables 3 and 4. Note that for each multiprocessor size, the completion times include all the overheads, computation time and communication time. Therefore, when selecting a partition of processors for a task, the corresponding response time will include all the overheads, computation time and communication times (including transferring data from one task to the next). The times in the table are only shown for selected multiprocessor sizes, although individual tasks can be executed on an arbitrary number of processors. Since the sizes of the machines available to us were limited, for the purposes of illustration, we extrapolated the completion times for larger machines as shown in the tables. Extrapolation was done using the immediate speedup available from the largest multiprocessor. For example, we computed the speedup (percentage improvement in response times) going from 16 to 32 processors for Intel iPSC/2 and then reduced this number by five percent (the degradation in speedup in the range 8 to 32); the resulting number was taken as the speedup going from 32 to 64 processors. The portion of each response time table with times for 64, 128 and 256 processors was estimated in this manner. It should be noted that the absolute values of completion times have no impact of the execution of the assignment algorithms proposed. If individual completion times are different, the allocation may be different. The response time functions in both tables are found to be decreasing and convex.

A basic premise of our assignment algorithms is that we can measure response time functions of elemental tasks, then accurately compute the response time functions of aggregate tasks. The premise was validated on this application—the *measured* response time function for the entire system was found to deviate from the *predicted* response time function by no more than 5% at any processor count. This accuracy is largely due to the fact that the application is compute-bound; the computation-to-communication ratio is 100 to 1. Any errors introduced by our simplistic approach to communication costs are bound to be low. The accuracy is also due in part to the fact that all possible mappings of the pipeline were constructed to avoid shared communication channels one can always embed a chain in a hypercube. Thus, no effects due to channel contention exist in the measurements. It remains to see how well our approach predicts response time functions on less compute-intensive applications. Nevertheless, applications of the type we consider here are practical, and important.

7.4 Experimental Results

7.4.1 The Response Time Problem

The algorithm for serial tasks with convex response time functions (in Section 4) was run using Tables 3 and 4 for a range of desired throughput constraints. As an example of the output generated by the algorithm, Table 5 shows the processor assignment for individual tasks for various sizes of the Intel iPSC/2. The last row of the table also shows the minimum response time for the given throughout constraint (λ = 0.05 tasks/second). We observe that some throughput conditions cannot be met by all sizes of multiprocessors. For example, a throughput of 0.125 tasks/second cannot be achieved for a 32 or 64 processor machine but it can be achieved for a 128 or 256 processor machine for which the minimum response time was observed to be 22.18 and 12.98 seconds respectively. Furthermore, the achieved throughput for a 128 processor machine was 0.157

Table 3: Completion times for individual tasks on the Intel iPSC/2 of various sizes (* indicates extrapolated values)

No. of Proc.	Response Times for Individual Tasks (Sec.)								
	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7	Task 8	Task 9
1	109.0	6.15	0.32	24.67	109.0	6.15	0.32	129.02	18.20
2	54.76	3.07	0.16	12.52	54.76	3.07	0.16	67.70	9.15
4	27.51	1.58	0.081	6.32	27.51	1.58	0.081	34.22	4.58
8	13.88	0.81	0.042	3.22	13.88	0.81	0.042	17.50	2.39
16	7.07	0.40	0.022	1.76	7.07	0.40	0.042	10.30	1.52
32	3.78	0.20	0.012	1.01	3.78	0.20	0.012	6.36	1.01
64*	2.12	0.11	0.007	0.61	2.12	0.11	0.007	4.13	0.71
128*	1.25	0.06	0.004	0.38	1.25	0.06	0.004	2.81	0.52
256*	0.77	0.04	0.002	0.26	0.77	0.77	0.04	0.002	0.40

Table 4: Completion times for individual tasks on the Encore Multimax of various sizes (* indicates extrapolated values)

No. of Proc.	Response Times for Individual Tasks (Sec.)								
	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7	Task 8	Task 9
1	352.20	16.54	0.85	51.70	352.20	16.54	0.85	212.00	25.50
2	176.08	8.33	0.69	28.00	176.08	8.33	0.69	103.77	13.10
4	88.38	4.26	0.60	15.10	88.38	4.26	0.60	51.70	7.10
8	45.42	2.14	0.32	8.70	45.42	2.14	0.32	25.98	4.25
16	26.99	1.23	0.20	5.00	26.99	1.23	0.20	15.23	2.76
32*	16.84	0.74	0.13	3.01	16.84	0.74	0.13	9.37	1.88
64*	11.03	0.47	0.09	1.91	11.03	0.47	0.09	6.06	1.34
128*	7.59	0.31	0.06	1.27	7.59	0.31	0.06	4.11	1.01
256*	5.48	0.22	0.05	0.89	5.48	0.22	0.05	2.93	0.80

Table 5: An example processor allocation for minimizing response time for several sizes of iPSC/2 (MRT = Minimum Response Time, Specified Throughput = 0.05 tasks/sec., No. of processors allocated to individual tasks are shown)

Task No.	Multiprocessor Size (No. of Procs.)							
	32		64		128		256	
	Proc. Asgn.	Time (Sec.)	Proc. Asgn.	Time (Sec.)	Proc. Asgn.	Time (Sec.)	Proc. Asgn.	Time (Sec.)
1	8	13.88	16	7.07	32	3.78	64	2.12
2	1	6.15	2	3.07	8	0.81	16	0.40
3	1	0.32	1	0.32	1	0.32	2	0.16
4	2	12.52	6	4.77	8	3.22	16	1.76
5	8	13.88	16	7.07	32	3.78	64	2.12
6	1	6.15	2	3.07	6	1.19	12	0.60
7	1	0.32	1	0.32	1	0.32	2	0.16
8	8	17.50	16	10.30	32	6.36	64	4.13
9	2	9.15	4	4.58	8	2.39	16	1.52
MRT		79.87		40.57		22.18		12.98

tasks/seconds and for a 256 processor machine the achieved throughput was 0.242 tasks/seconds.

Figure 5 shows the optimal response time function for the entire pipeline computation together with the achieved throughput using the hypercube data. As we might expect, the response time function is decreasing and the achieved throughput is increasing. Figure 6 shows response times for specified throughput of $\lambda = 0.05$ tasks/second for different hypercube sizes. Along with the response time function from Figure 5, two curves are shown to provide a comparison with non-optimal, yet simple, heuristics for processor assignment. The first heuristic, called the equal allocation heuristic, allocates an equal number of processors to each task, thus ignoring the response time functions of the individual tasks (this takes $O(n)$ time). The second heuristic, called the ratio heuristic, attempts to take these functions into account through the use of ratios: initially each task is assigned a processor; the remaining processors are distributed in proportion to the quantities $f_i(1), 1 \leq i \leq n$ for each of the n tasks (requiring $O(n)$ time). Our optimal algorithm ($O(n \log p)$) always achieves a lower response time than the two simple $O(n)$ heuristics. Comparing the achieved throughputs in Figure 7, it can be observed that the ratio heuristic achi es higher throughput than the optimal algorithm because it does not tradeoff throughput for achieving the minimum response time, i.e., the heuristic is not guaranteed to satisfy the response-time constraint. The equal allocation strategy performs rather poorly as one might expect.

The tradeoff of response time versus throughput constraint (using optimal response time functions) is studied in Figures 8 and 9 for a 128- and 256-processor hypercube. Figure 8 shows the response time and Figure 9 shows the corresponding achieved throughput as a function of the specified throughput. As we can observe, the response time curve follows the throughput curve

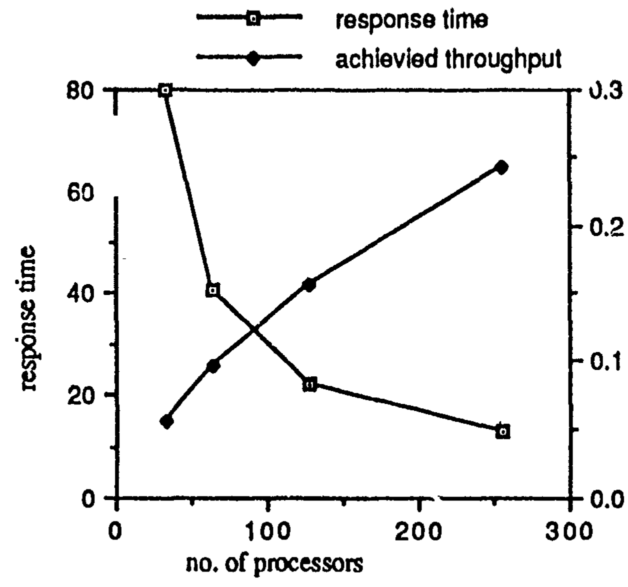


Figure 5: Response Time Problem: Response Time and Achieved Throughput

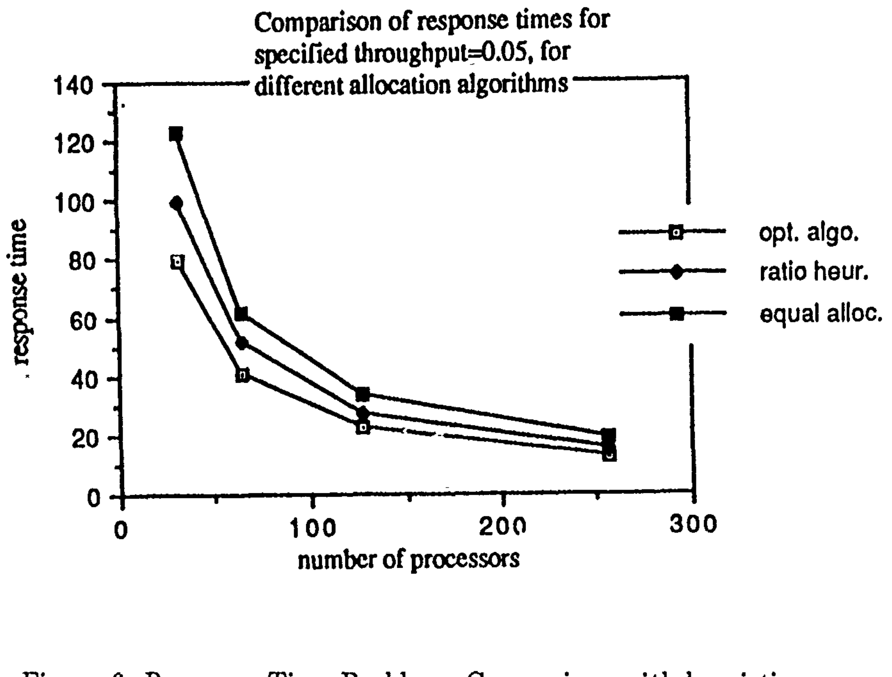


Figure 6: Response Time Problem: Comparison with heuristics

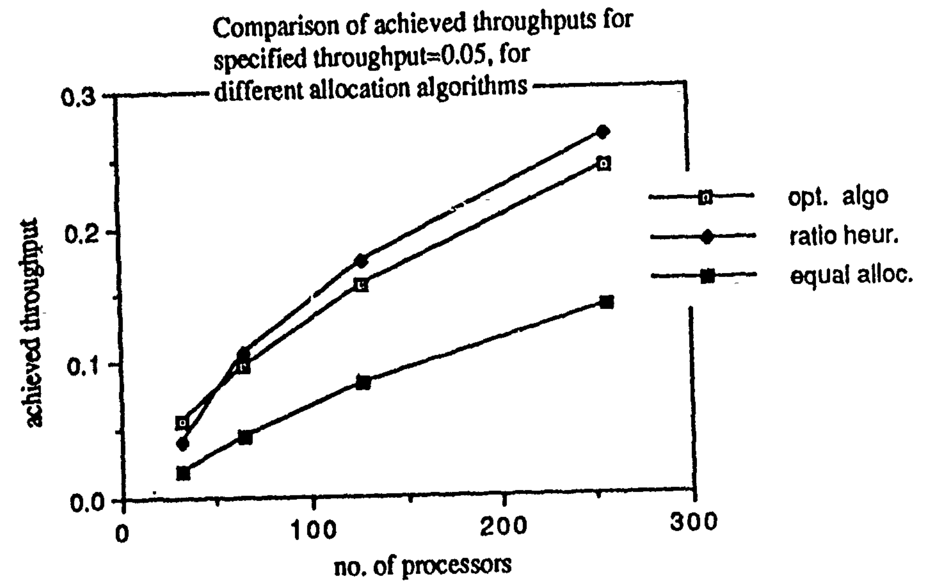


Figure 7: Response Time Problem: Achieved throughputs for heuristics

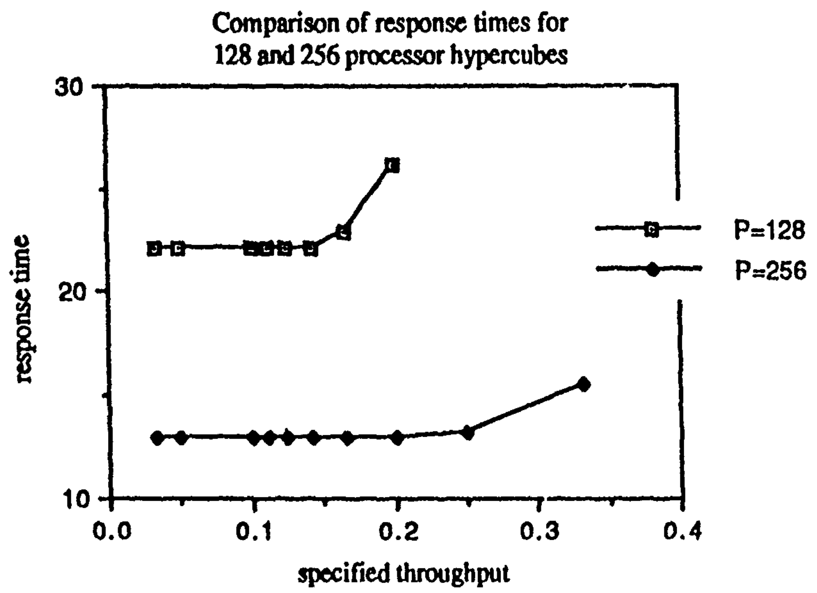


Figure 8: Response Time Problem: Response time with increasing throughput constraint

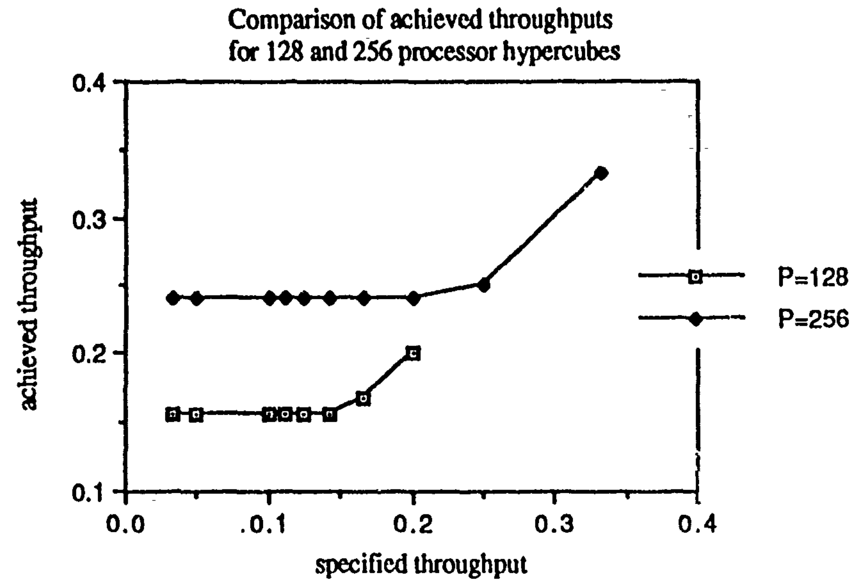


Figure 9: Response Time Problem: Achieved throughput with increasing throughput constraint

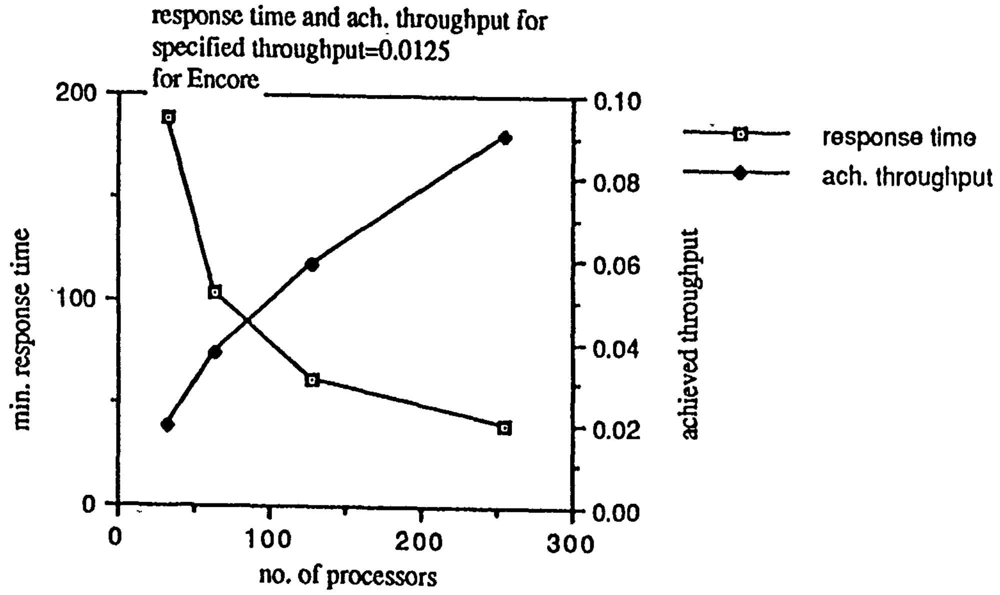


Figure 10: Response Time Problem: Results for Encore

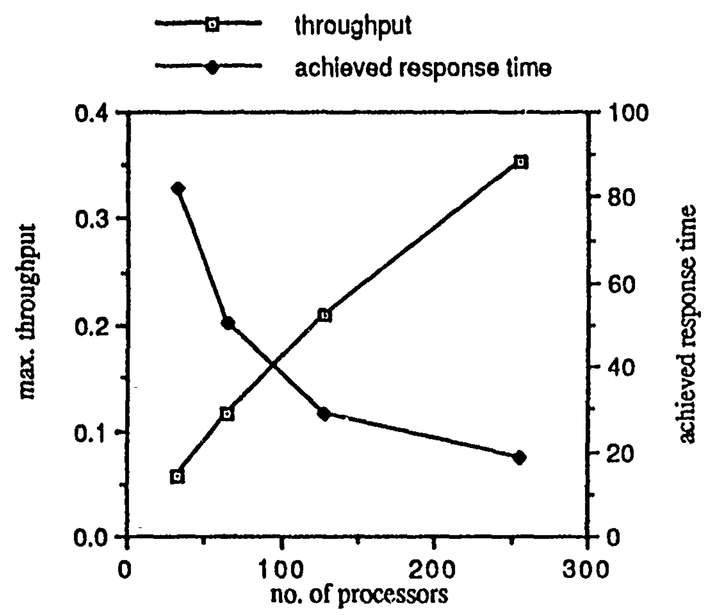


Figure 11: Throughput Problem: Throughputs and achieved response times

in shape; this clearly indicates that the algorithm trades off response time to achieve the specified throughput. This is exemplified at high throughput constraints where the minimum response time increases significantly in order to achieve the specified throughput. For low values of specified throughput, the change in minimum response time is insignificant because the throughput can be achieved easily with the given number of processors. For a larger system the knee of the curves shifts to the right as expected due to the additional resources (as shown for a 256-processor system). Finally, Figure 10 plots the response time as a function of the number of processors for the Encore data. The graph is seen to closely resemble Figure 5. To avoid repetition, we do not show further results for the Encore.

7.4.2 The Throughput Problem

Figure 11 illustrates the maximum throughput obtained and the corresponding achieved response time for our task system when the specified response time $\rho = 100$ seconds. The results generated by the two heuristics described earlier are presented in Figure 12. The optimal algorithm generates higher throughputs than achieved by the two heuristics. Figure 13 shows the achieved response times when using the heuristics. The ratio heuristic achieves a lower response time than that by the optimal algorithm because it does not necessarily satisfy the throughput constraint.

The tradeoff between response time and throughput is shown once again, this time in the context of the throughput problem, in Figures 14 and 15 for 128 and 256 processor hypercubes as a function of the specified response time. The solid line shows the maximum possible throughput when there is no response time constraint. Therefore, for any specified response time, the difference between the maximum throughput and unconstrained maximum throughput represents the amount of throughput tradeoff to achieve the specified response time. Furthermore, we can observe that as the specified response time increases, the difference between the unconstrained maximum

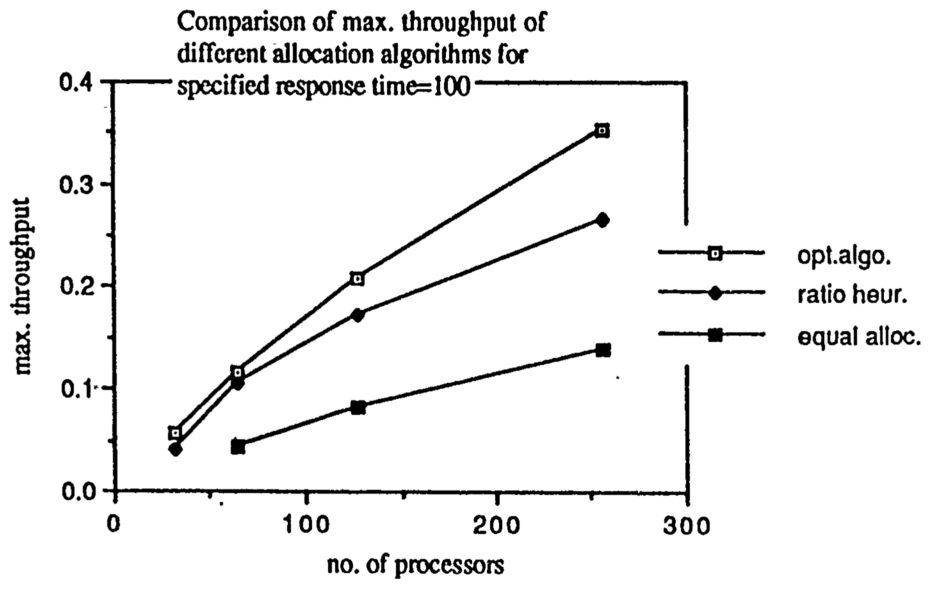


Figure 12: Throughput Problem: Throughputs obtained by heuristics

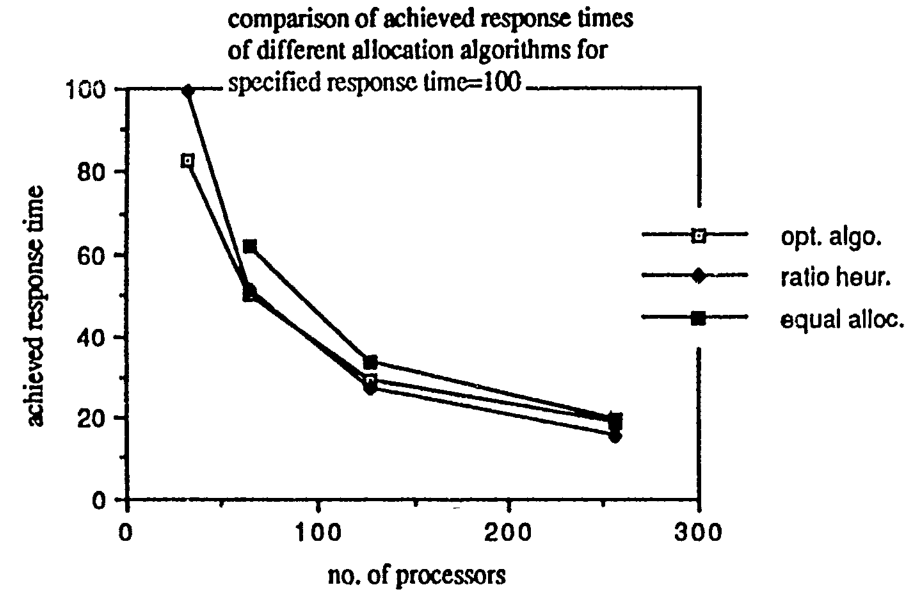


Figure 13: Throughput Problem: Achieved response times for heuristics

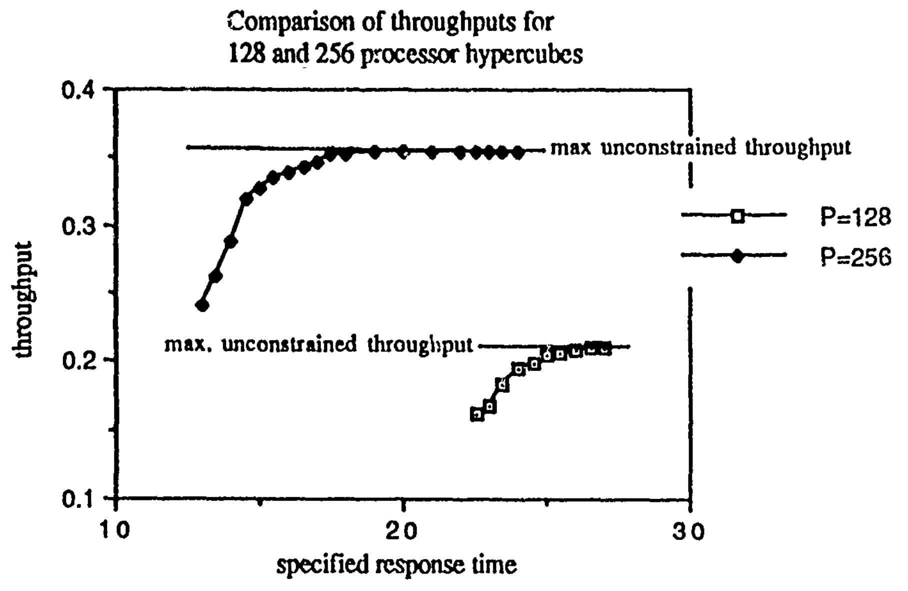


Figure 14: Throughput Problem: Maximum throughput with increasing response time constraint

throughput and throughput reduces because of the weakening of the response time constraint. Beyond a certain point, the response time constraint is so weakened that the maximum unconstrained throughput is achieved as shown by the plateau in the throughput curve. This phenomenon is also observed in functional pipelines in processor designs where inserting delays in the pipeline stages results in higher throughout at the cost of response time [26, 34, 40].

8 Summary

In this paper we have formulated the problem of optimizing the performance of a pipeline computation, represented by a task structure, on a parallel architecture, given a large supply of processors, and the experimentally determined response time functions for its constituent tasks. Unlike prior treatments of the mapping problem we considered the case where there are many more processors than tasks and where tasks are not queued or scheduled. We considered the dual problems of minimizing response time subject to a throughput constraint, and maximizing throughput subject to a response time constraint. As we observed in our sample application, these problems are complimentary, in the sense that allocation to increase throughput may have the side effect of increasing response time, and vice versa.

The problem posed in this paper was shown to be solvable in polynomial time for a useful class of task structures. Specifically we presented $O(np^2)$ algorithms (where n is the number of tasks and p is the number of processors), for the response time problem, for the cases where the task structures are linear, tree-structured and series-parallel graphs. The algorithms designed for the response time problem can be used to solve the throughput problem with an additional logarithmic factor in complexity. To place the work in a realistic setting we considered an application, stereo image matching on two parallel architectures, and evaluated the performance of our assignment

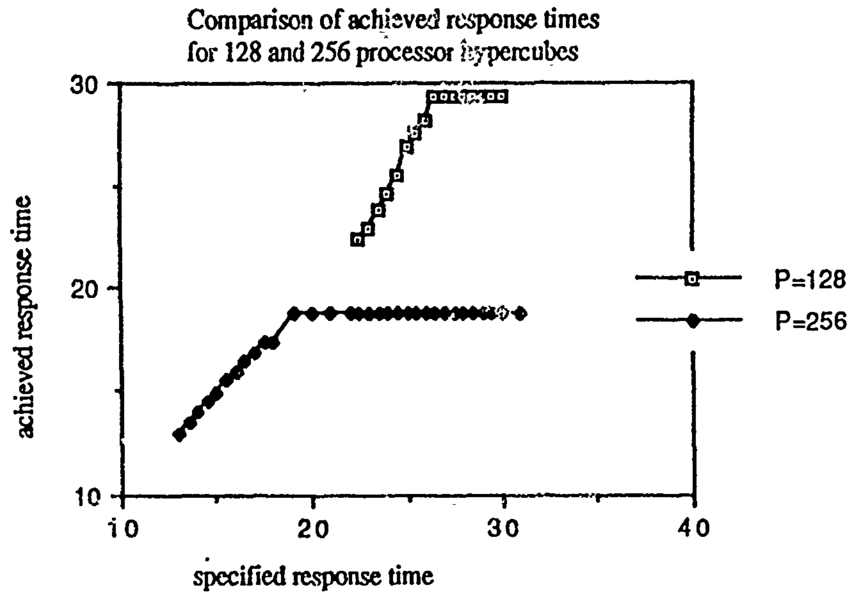


Figure 15: Throughput Problem: Achieved response times with increasing response time constraint

algorithms. Future endeavors include the provision of algorithms for general task structures and investigation of faster and parallelized assignment algorithms.

References

[1] M J Berger and S H Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. on Computers*, C-36(5):570–580, May 1987.

[2] F Berman and L Snyder. On mapping parallel algorithms into parallel architectures. *Journal of Parallel and Distributed Computing*, 4:439–458, 1987.

[3] J Blazewicz, M Drabowski, and J. Welgarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Trans. on Computers*, C-35(5):389–393, May 1986.

[4] S H Bokhari A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Trans. on Soft. Eng.*, SE-7(6):583–589, Nov. 1981.

[5] S H Bokhari Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. on Computers*, 37(1):48–57, January 1988.

[6] S Bollinger and S Midkiff. Heuristic Technique for Processor and Link Assignment in Multicomputers. *IEEE Trans. on Computers*, 40(3):325–336, March 1991.

[7] L Bomans and D. Roose. Benchmarking the iPSC/2 Hypercube Multiprocessor. *Concurrency. Practice and Experience*, 1(1):3–18, Sept. 1989.

[8] M.Y Chan and F Y.L. Chin. On embedding rectangular grids in hypercubes. *IEEE Trans. on Computers*, 37(10):1285–1288, October 1988.

[9] M Chen, and K.G. Shin, Processor Allocation in an N-Cube Multiprocessor Using Gray Codes. *IEEE Trans. on Computers*, C-36(12):1396–1407, December 1987.

[10] H-A Choi and B Narahari, Algorithms for Mapping and Partitioning Chain Structured Parallel Computations. Tp appear in *1991 International Conference on Parallel Processing*.

[11] A. N. Choudhary and J. H. Patel. Parallel Architectures and Parallel Algorithms for Integrated Vision Systems. *Kluwer Academic Publishers*, Boston, MA, 1990. Video images obtained from the Army Research Office.

[12] E. Denardo. *Dynamic Programming: Models and Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[13] J. Du and Y-T. Leung. Complexity of Scheduling Parallel Task Systems. *SIAM J. Discrete Math.* 2(4):473–487, November 1989.

[14] S. Dutt and J.P. Hayes Subcube Allocation in Hypercube Computers. *IEEE Trans. on Computers,* 40(3):341–352, March 1991.

[15] Encore Computer Corp. Promotional Literature. Marlborough, MA. 1986.

[16] G. Fox, M. Johnson, G. Lyzenga. S. Otto, J. Salmon and D. Walker. *Solving Problems on Concurrent Processors (Vol. I and II).* Prentice Hall, Englewood Cliffs, NJ, 1990.

[17] G. Fox, A. Kolawa, and R. Williams. The implementation of a dynamic load balancer. *Technical Report* C3P-287a, Caltech, February 1987.

[18] J. P. Hayes, T. N. Mudge, Q. F. Stout, and S. Colley. Architecture of a hypercube supercomputer. *Proc. of the 1986 International Conference on Parallel Processing.*

[19] C.-T. Ho and S.L. Johnsson. On the embedding of arbitrary meshes in boolean cubes with expansion two dilation two. In *Proceedings of the 1987 Int'l Conference on Parallel Processing*, pages 188–191, August 1987.

[20] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*, Chapter 2, Computer Science Press, Maryland, 1985.

[21] A. Huertas and G. Mediono. Detection of intensity changes with subpixel accuracy using Laplacian-Gaussian masks. *IEEE Trans. PAMI,* PAMI-8 pp. 651-664, Sep. 86.

[22] O.H. Ibarra and S.M. Sohn. On mapping systolic algorithms onto the hypercube. *IEEE Trans on Parallel and Distributed Systems*, 1(1):48–63, January 1990.

[23] M. Jeng and H.J. Siegel. A distributed management scheme for partitionable parallel computers *IEEE Trans. Parallel and Distributed Systems,* 1(1):120–126, January 1990.

[24] R. Kincaid, D.M. Nicol, D. Shier, and D. Richards. A multistage linear array assignment problem. *Operations Research*, 38(6):993–1005, November-December 1990.

[25] C.-T. King, W.-H. Chou, and L.M. Ni. Pipelined data-parallel algorithms. *IEEE Trans. on Parallel and Distributed Systems*, 1(4):470–499, October 1990.

[26] P. M. Kogge. *The Architecture of Pipelined Computers.* McGraw Hill, New York, 1981.

[27] R. Krishnamurti and Y.E. Ma. The processor partitioning problem in special-purpose partitionable systems. *Proc. 1988 International Conference on Parallel Processing,* Vol. 1, pp. 434–443.

[28] M. K. Leung and T. S. Huang. Point matching in a time sequence of stereo image pairs. *Technical Report*, CSL, University of Illinois, Urbana-Champaign, 1987.

[29] L. Li and K.H. Cheng. Job scheduling in partitionable mesh connected systems. *Proc. 1989 International Conference on Parallel Processing.*

[30] A.W.Marshall and I.Olkin, *Inequalities. Theory of Majorization and Its Applications*, Academic Press, 1979.

[31] W N. Martin and J. K. Aggarwal (editors). *Motion Understanding, Robot and Human Vision.* Kluwer Academic Publishers, Boston, MA 1988.

[32] R G. Melhem and G.-Y Hwang. Embedding rectangular grids into square grids with dilation two. *IEEE Trans. on Computers*, 39(12):1446–1455, December 1990.

[33] D M Nicol and D R. O'Hallaron. Improved algorithms for mapping parallel and pipelined computations. *IEEE Trans. on Computers*, 40(3):295-306, March 1991.

[34] J. H. Patel and E. S. Davidson. Imroving the Throughput of a Pipeline by Insertion of Delays. *Proceedings of the Third Annual Computer Architecture Symposium*, pp. 159-163, 1976.

[35] P Sadayappan and F Ercal. Nearest-neighbor mapping of finite element graphs onto processor meshes. *IEEE Trans. on Computers*, 36(12):1408–1424, December 1987.

[36] D S Scott and R. Brandenburg. Minimal mesh embeddings in binary hypercubes. *IEEE Trans. on Computers*, 37(10):1284–1285, October 1988.

[37] H. J. Siegel, L. 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 Trans. on Computers*, C-30(12), December 1981.

[38] C V Stewart and C R. Dyer. Scheduling Algorithms for PIPE (Pipelined Image-Processing Engine). *Journal of Parallel and Distributed Computing*, 5:131–153, 1988.

[39] H Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. on Soft. Eng.*, SE-3(1):85–93, January 1977.

[40] H. S Stone. *High-Performance Computer Architecture (2nd ed.).* Addison-Wesley, 1990.

[41] D Towsley Allocating programs containing branches and loops within a multiple processor system. *IEEE Trans. on Soft. Eng.*, SE-12(10):1018–1024, October 1986.

[42] J. Valdes, R E Tarjan, and E.L. L[illegible]ler. The Recognition of series parallel digraphs. *SIAM J. Comput.*, 11(2):298–313, May 1982.

[43] C Weems, A Hanson, E. Riseman, and A. Rosenfeld. An integrated image understanding benchmark for parallel computers. *Journal of Parallel and Distributed Computing.* January, 1991.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE October 1991	3. REPORT TYPE AND DATES COVERED Contractor Report

4. TITLE AND SUBTITLE

OPTIMAL PROCESSOR ASSIGNMENT FOR PIPELINE COMPUTATIONS

5. FUNDING NUMBERS

NAS1-18605
505-90-52-01

6. AUTHOR(S)

David M. Nicol, Rahul Simha, Alok N. Choudhury, and Bhagirath Narasami

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

Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

8. PERFORMING ORGANIZATION REPORT NUMBER

ICASE Report No. 91-79

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

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA CR-189047
ICASE Report No. 91-79

11. SUPPLEMENTARY NOTES

Langley Technical Monitor: Michael F. Card
Final Report

Submitted to IEEE Trans. on Parallel & Distributed Systems

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The availability of large scale multitasked parallel architectures introduces the following processor assignment problem for pipelined computations. Given a set of tasks and their precedence constraints, along with their experimentally determined individual response teims for different processor sizes, find an assignment of processor to tasks. Two objectives interest us: minimal response given a throughput requirement, and maximal throughput given a response teim requirement. These assignment problems differ considerably from the classical mapping problem in which several tasks share a processor; instead, we assume that a large number of processors are to be assigned to a relatively small number of tasks. In this paper we develop efficient assignment algorithms for different classes of task structures. For a p processor system and a series-parallel precedence graph with n constituent tasks, we previde an $O(np^2)$ algorithm that finds the optimal assignment for the response time optimization problem; we find the assignment optimizing the constrained throughput in $O(np^2 \log p)$ time. Special cases of linear, independent, and tree graphs are also considered. In addition, we also examine more efficient algorithms when certain restrictions are placed on the problem parameters. Our techniques are applied to a task system in computer vision.

14. SUBJECT TERMS

mapping; pipelining; assignment; parallel processing

15. NUMBER OF PAGES

36

16. PRICE CODE

A03

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

SUPPLEMENTARY

INFORMATION

ERRATA ADA 242793

NASA Contractor Report 189047

OPTIMAL PROCESSOR ASSIGNMENT FOR
PIPELINE COMPUTATIONS

David M. Nicol, Rahul Simha, Alok N. Choudhury,
and Bhagirath Narahari

October 1991

NASA Contractor Report 189047 has an incorrect report number. The correct report number is NASA Contractor Report 189550. Please mark your copies on the cover and Report Documentation Page to reflect this change.

Issued January 1992