# Optimal Processor Assignment for a Class of Pipelined Computations


Alok N. Choudhary[1]

*Department of Electrical and Computer Engineering*

*Syracuse University*

*Syracuse, NY 13244*

Bhagirath Narahari

*Department of Electrical Engineering and Computer Science*

*George Washington University*

*Washington, DC 20052*

David M. Nicol[2] Rahul Simha[3]

*Department of Computer Science*

*College of William and Mary*

*Williamsburg, VA 23185*



[1]Partially supported by NSF grant MIP-9110810

[2]Supported in part by NASA grant NAG-1-995, and by NSF grant ASC 8819393.

[3]Partially supported by NSF grant NCR-8907909

**Abstract**

The availability of large scale multitasked parallel architectures introduces the following processor assignment problem: we are given a long sequence of data sets, each of which is to undergo processing by a collection of tasks whose inter-task data dependencies form a series-parallel partial order. Each individual task is potentially parallelizable, with a known experimentally determined execution signature. Recognizing that data sets can be pipelined through the task structure, the problem is to find a "good" assignment of processors to tasks. Two objectives interest us: minimal response time per data set given a throughput requirement, and maximal throughput given a response time requirement. Our approach is to decompose a series-parallel task system into its essential "serial" and "parallel" components; our problem admits the independent solution and recomposition of each such component. We provide algorithms for the series analysis, and use an algorithm due to Krishnamurti and Ma for the parallel analysis. For a $p$ processor system and a series-parallel precedence graph with $n$ constituent tasks, we give a $O(np^2)$ algorithm that finds the optimal assignment (over a broad class of assignments) for the response time optimization problem; we find the assignment optimizing the constrained throughput in $O(np^2 \log p)$ time. Our techniques are applied to a task system in computer vision.

# 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 [8, 13]. Typically, experimentally observed performance (e.g., speedup or response time) is tabulated as a function of the number of processors employed, a function sometimes known as the *execution signature* [10], or *response time* function. In this paper we use such functions to determine the number of processors to be allocated to each of several tasks when the tasks are part of a pipelined computation. This problem is natural, given the growing availability of *multitasked* parallel architectures, such as PASM [29], the NCube system [14], and Intel's iPSC system [5], in which it is possible to map tasks to processors and allow parallel execution of multiple tasks in different logical partitions.

We consider the problem of optimizing the performance of a complex computation applied to each member of a sequence of data sets. This type of problem arises, for instance, in imaging systems, where each image frame is analyzed by a sequence of elemental tasks, e.g., fast Fourier transform or convolution. Other applications include network software, where packets are pipelined through well-defined functions such as checksum computations, address decoding and framing. Given the data dependencies between the computation's multiple tasks, we may exploit parallelism both by pipelining data sets through the task structure, and by applying multiple processors to individual tasks.

There is a fundamental tradeoff between assigning processors to maximize the overall throughput (measured as data sets per unit time), and assigning processors to minimize a single data set's response time. We manage the tradeoff by maximizing one aspect of performance subject to the constraint that a certain level of performance must be achieved in the other aspect. Under the assumptions that each of $n$ tasks is statically assigned a subset of dedicated processors and that an individual task's response time function completely characterizes performance (even when using shared resources such as the communication network) we show that $p$ processors can be assigned to a *series-parallel* task structure in $O(np^2)$ time so as to minimize response time while achieving a given throughput. We are also able to find the assignment that maximizes throughput while achieving a given minimal response time, in $O(np^2 \log p)$ time.

The assumption of a static assignment arises naturally in real-time applications, where the overhead of swapping executable task code in and out of a processor's memory threatens performance. Without this assumption, the optimization problem becomes much more difficult.

Our method involves decomposing a series-parallel graph into series and parallel components using standard methods; we present algorithms for analyzing series components and use Krishnamurthy and Ma's algorithm [20] to analyze the parallel components.

We assume that costs of communication between tasks are completely captured in the given response-time functions. Thus, our techniques can be expected to work well on compute-bound task systems; our example application is representative of this class, having a computation to communication ratio of 100. Our techniques may not be applicable when communication costs that depend on the particular sets of processors assigned to a task (e.g., contention) contribute significantly to overall performance.

A large literature exists on the topic of mapping workload to processors, see, for instance [1, 3, 4, 6, 15, 17, 18, 23, 24, 26, 27, 31, 33]. A new problem has recently emerged, that of scheduling of tasks on multitasked parallel architectures where each task can be assigned a set of processors. Some formulations consider scheduling policies with the goal of achieving good average response time and good throughput, given an arrival stream of different, independent parallel jobs, e.g., [28]. Another common objective, exemplified in [2, 11, 20, 25], is to find a schedule of processor assignments that minimizes completion time of a single job executed once. The problem we consider is different from these specifically because we have a parallel job which is to be *repeatedly* executed. We consider issues arising from our need to pipeline the repeated executions to get good throughput, as well as apply parallel processing to the constituent tasks to get good per-execution response time. Yet another distinguishing characteristic of our problem is an underlying assumption that a processor is statically assigned to one task, with the implication that every task is always assigned at least one processor.

Two previously studied problems are close to our formulation. The assignment of processors to a set of independent tasks is considered in [20]. The single objective is the minimization of the makespan, which minimizes response time if the tasks are considered to be part of a single parallel computation, or maximizes throughput if the tasks are considered to form a pipeline. The problem of assigning processors to independent chains of modules is considered in [7]; this assignment minimizes the response time if the component tasks are considered to be parallel, and maximizes the throughput if the component chains are considered to form pipelines. Pipeline computations are also studied in [19, 30]. In [30], heuristics are given for scheduling planar acyclic task structures and in [19], 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 for general pipeline computations, although our solution approach (dynamic programming) is related to those in [3] and [33].

This paper is organized as follows. Section §2 introduces notation, and formalizes the response-time problem and the throughput problem. Section §3 presents our algorithms for series systems, and §4 shows how to optimally assign processors to series-parallel systems. Section §5 shows how the problem of maximizing throughput subject to a response-time constraint can be solved using solutions to the response-time problem. Section §6 discusses the application of our techniques to an actual problem, and Section §7 summarizes this work.

## 2 Problem Definition

We consider a set of tasks, $t_0, t_1, \ldots, t_{n+1}$, that comprise a computation to be executed using up to $p$ identical processors, on each of a long stream of data sets. Every task is applied to every data set. We assume the tasks have a series-parallel precedence relation constraining the order in which we may apply tasks to a given data set; tasks unrelated in the partial order are assumed to process duplicated copies (or, different elements) of a given data set. Under these assumptions we may pipeline the computation, so that different tasks are concurrently applied to different data sets.

Each task is potentially parallelizable; for each $t_i$ we let $f_i(n)$ be the execution time of $t_i$ using $n$ identical processors. $f_i$ is called a *response-time function* (also known as an *execution signature* [10]). We assume that $f_0$ and $f_{n+1}$ are dummy tasks that serve respectively to identify the initiation and completion of the computation; correspondingly we take $f_0(n) = f_{n+1}(n) = 0$ for all $n$. However, $f_i(0) = \infty$ for all $i = 1, \ldots, n$; these conditions ensure that no processor is ever assigned to $t_0$ or $t_{n+1}$, and that at least one processor is assigned to every other task.

An example of the response time functions for a computation with 5 tasks on up to 8 processors is shown in Table 1. Each row of the table is a response time function for a particular task. Observe

| tasks | Number of processors 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $t_1$ | 29 | 16 | 11 | 9 | 7 | 6 | 4.5 | 4 |
| $t_2$ | 90 | 50 | 20 | 15 | 12 | 10 | 9 | 9.5 |
| $t_3$ | 80 | 43 | 18 | 14 | 11 | 9 | 8 | 8.5 |
| $t_4$ | 20 | 12 | 10 | 9 | 8 | 7 | 6 | 5 |
| $t_5$ | 15 | 10 | 7 | 5 | 4 | 3.5 | 3 | 2.5 |

Table 1: Example response time functions. Table gives tasks' execution time (in seconds) as a function of the number of processors used.

that individual functions need not be convex, nor monotonic.

We may describe an assignment of *numbers* of processors to each task by a function $A$: $A(i)$ gives the number of processors statically and exclusively allocated to $t_i$. A feasible assignment is one where $\sum_{i=1}^{n} A(i) \leq p$, and $A(i) > 0$ for $i = 1, \ldots, n$.

Given $A$, $t_i$'s execution time is $f_i(A(i))$, and the maximal data set throughput is $\Lambda(A) = \max_i\{f_i(A(i))^{-1}\}$. The response time for a data set is obtained by computing the length $R(A)$ of the longest path through the graph where each $t_i$ is a node weighted by $f_i(A(i))$, and the edges are defined by the series-parallel precedence relation.

Given some throughput constraint $\lambda$ and processor count $q$, we define $\mathcal{T}_\lambda(q)$ to be the set of all feasible assignments $A$ that use no more than $q$ processors, and achieve $\Lambda(A) \geq \lambda$. The response-time problem is to find $\mathcal{F}_\lambda(p)$ the minimum response time over all feasible assignments in $\mathcal{T}_\lambda(p)$, that is, the response time for which there is an assignment $A$ for which $R(A)$ is mimimal over all assignments with $p$ or fewer processors that achieve throughput $\lambda$ or greater. This problem arises when data sets must be processed at least as fast as a known rate $\lambda$ to avoid losing data; we wish to minimize the response time among all those assignments that achieve throughput $\lambda$. Similarly, given response time constraint $\gamma$ and processor count $q$ we define $\mathcal{R}_\gamma(q)$ to be the set of all feasible assignments $A$ using no more than $q$ processors, and achieving $R(A) \leq \gamma$. The *throughput problem* is to find $A \in \mathcal{R}_\gamma(p)$ for which $\Lambda(A)$ is maximized. This problem arises in real-time control applications, where each data set must be processed within a maximal time frame in order to meet

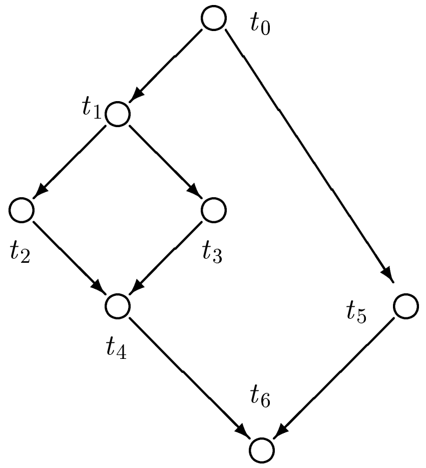


Figure 1: Example of series-parallel task system $T$

processing deadlines. We will focus on solutions to the response time problem first and later show how these may be used to solve the throughput problem.

Since a response-time function completely defines a task, elemental or composite, we will also use the term "task" to refer to compositions of the more elemental tasks $t_i$. Let $\tau_i$ denote such a composite task and let $F_i$ be its optimal response time function. Our general approach is illustrated through an example. Consider the series-parallel task $T$ in Figure 1 with response-time functions given Table 1 (here, $t_0$ and $t_6$ are dummy tasks). We may think of $t_2$ and $t_3$ as forming a parallel subtask—call it $\tau_1$. Given the response time functions for $t_2$ and $t_3$, we will construct an optimal response time function called $F_1$ for $\tau_1$, after which we need never explicitly consider $t_1$ or $t_2$ separately from each other—$F_1$ completely captures what we need to know about both of them. Next, we view $\tau_1$ and $t_1$ as a series task, call it $\tau_2$, and compute the optimal response time function for $\tau_2$. The process of identifying series and parallel subtasks and constructing response-time functions for them continues until we are left with a single response time function that describes the optimal behavior of $T$. By tracking the processor assignments necessary to achieve the optimal response times at each step, we are able to determine the optimal processor allocations for $T$. A solution method for parallel tasks has already been given in [20]; we present algorithms for series tasks.

We will assume that every response-time function is monotone nonincreasing, since, as argued in [20], any other response-time function can be made decreasing by disregarding those assignments of processors that cause higher response times. Also, observe that response time functions may include inherent communication costs due to parallelism, as well as the communication costs that are suffered by communicating with predecessor and successor tasks. These assumptions are reasonable when the communication bandwidth is sufficiently high for us to ignore effects due to contention between pairs of communicating tasks. Our methods may not produce good results when this assumption does not hold.

## 3 Individual Parallel Tasks and Series Tasks

The problem of determining an optimal response-time function for parallel tasks has already essentially been solved in the literature [20]. We describe this solution briefly. Let $t_1, \ldots, t_k$ be the tasks used to compose a parallel task $\tau$. For each $t_i$ we know $u_\lambda(t_i)$—the minimum number of processors needed so that every elemental task involved in $t_i$ has response-time no greater than $1/\lambda$. We initialize by allocating $u_\lambda(t_i)$ processors to each $t_i$. If we run out of processors first then no processor allocation can meet the throughput requirement. Otherwise, the initial allocation uses the fewest possible number of processors that do meet this requirement. We then incrementally add the remaining processors to tasks in such a way that at each step the response time (the maximum of task response times) is reduced maximally. This algorithm has an $O(p \log p)$ time complexity.

Series task structures are interesting in themselves because many pipelines are simple linear chains [19]. We first describe an algorithm that constructs the optimal response time function $\mathcal{F}_\lambda$ for a linear task structure $T$ when each function $f_i(x)$ is convex in $x$. While convexity in elemental functions is intuitive, nonconvex response-time functions arise from parallel task compositions. Consequently, a different algorithm for series compositions of nonconvex response-time functions will be developed later.

Like the parallel composition algorithm, we first assign the minimal number of processors needed to meet the throughput requirement. The mechanism for this is identical. Supposing that this step does not exhaust the processor supply, define $x_i$ to be the number of processors currently assigned to $t_i$, initialize $x_i = u_\lambda(t_i)$, and define $y = \sum_{i=1}^{n} x_i$ to be the total number of processors already allocated. We then set $\mathcal{F}_\lambda(x) = \infty$ for all $x < y$ to reflect an inability to meet the throughput

| function | Number of processors 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $f_2$ (for task $t_2$) | 90 | 50 | 20 | 15 | 12 | 10 | 9 | 9.5 |
| $f_3$ (for task $t_3$) | 80 | 43 | 18 | 14 | 11 | 9 | 8 | 8.5 |
| $F_1$ (for task $\tau_1$) | $\infty$ | 90 | 80 | 50 | 43 | 20 | 18 | 15 |

Table 2: Response time function $F_1$ for parallel task $\tau_1$

requirement, and set $\mathcal{F}_\lambda(y) = \sum_{i=1}^{n} f_i(x_i)$. Next, for each $t_i$, compute $d(i, x_i) = f_i(x_i + 1) - f_i(x_i)$, the change in response time achieved by allocating one more processor to $t_i$. Build a max-priority heap [16] where the priority of $t_i$ is $|d(i, x_i)|$. Finally, enter a loop where, on each iteration the task with highest priority is allocated another processor, its new priority is computed, and the priority heap is adjusted. We iterate until all available processors have been assigned. 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 response time function $\mathcal{F}_\lambda$ it greedily produces is optimal, since the algorithm above is essentially one due to Fox [12], as reported in [32]. Simple inspection reveals that the algorithm has an $O(p \log n)$ time complexity. Unlike the similar algorithm for parallel tasks, correctness here depends on convexity of component task response times.

The need to treat nonconvex response-time functions arises from the behavior of composed parallel tasks. Return to our example in Figure 1 and consider the parallel composition $\tau_1$ of elemental tasks $t_2$ and $t_3$, with throughput requirement $\lambda = 0.01$. The response-time function $F_1$ is shown in Table 2. Note that $F_1$ is not convex, even though $f_2$ and $f_3$ are. This nonconvexity is due to the peculiar nature of the maximum of two functions and cannot be avoided when dealing with parallel task compositions. We show below that nonconvexity can be handled, with an additional cost in complexity.

We begin as before, allocating just enough processors so that the throughput constraint is met. Assuming so, for any $j = 1, \ldots, n$, we will denote the subchain comprised of $t_1, \ldots, t_j$ as task $T_j$, and compute its optimal response time function, $C_j$, subject to throughput constraint $\lambda$. Using the principle of optimality[9], we write a recursive definition for $u_\lambda(T_j)$ and $C_j(x)$ :

$$u_\lambda(T_j) = \begin{cases} u_\lambda(t_1) & \text{if } j = 1 \\ u_\lambda(t_j) + u_\lambda(T_{j-1}) & \text{otherwise} \end{cases}$$

$$C_j(x) = \begin{cases} f_1(x) & \text{if } j = 1 \\ \min\limits_{u_\lambda(t_j) \le i \le x - u_\lambda(T_{j-1})} \{f_j(i) + C_{j-1}(x-i)\} & \text{otherwise.} \end{cases}$$

The dynamic programming equation is understood as follows. Suppose we have already computed the function $C_{j-1}$. This implicitly asserts that we know how to optimally allocate any number $y \le p$ processors to $T_{j-1}$. Next, given $x$ processors to distribute between tasks $t_j$ and $T_{j-1}$, we try every combination subject to the throughput constraints: $i$ processors for $t_j$ and $x - i$ processors for $T_{j-1}$. The principle of optimality tells us that the least-cost combination gives us the optimal assignment of $x$ processors to $T_j$. Since the equation is written as a recursion, the computation will actually build response time tables from 'bottom up', starting with task $t_1$ in the first part of the equation.

This procedure requires $O(np^2)$ time. We have been unable to find a solution that gives a better worst-case behavior in all cases. Some of the difficulties one encounters may be appreciated by study of our previous example. Consider the construction of $\tau_2$, comprised of the series composition of $t_1$ and $\tau_1$. As before, let $F_1$ denote the response time function for $\tau_1$. Table 3 gives the values of $f_1(u)+F_1(v)$ for all $1 \le u, v < 8$ with $u+v \le 8$. The set of possible sums associated with allocating a fixed number of processors $x$ lie on an *assignment diagonal* moving from the lower left (assign $x-1$ processors to $\tau_1$, one to $t_1$) to the upper right (assign one processor to $\tau_1$, $x-1$ to $t_1$) of the table, illustrated by use of a common typeface on a diagonal. Brute force computation of $\tau_2(x)$ consists of generating all sums on the associated diagonal, and choosing the allocation associated with the least sum. In the general case this is equivalent to looking for the minimum of a function known to be the sum of a function that decreases in $i$ (e.g. $f_1(i)$) and one that increases (e.g. $F_1(x-i)$). Unlike the case when these functions are known to be convex as well, in general their sum does not have any special structure we can exploit—the minimum can be achieved anywhere, implying that we have to look for it everywhere. It would seem then that dynamic programming may offer the least-cost solution to the problem.

We note in passing that a straightforward optimization may reduce the running time, but does

| | $f_1(1)$ 29 | $f_1(2)$ 16 | $f_1(3)$ 11 | $f_1(4)$ 9 | $f_1(5)$ 7 | $f_1(6)$ 6 | $f_1(7)$ 4.5 |
|---|---|---|---|---|---|---|---|
| $F_1(1) = \infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| $F_1(2) = 90$ | 119* | *106** | 101 | *99* | 97 | *96* | |
| $F_1(3) = 80$ | *109* | 96 | *91* | 89 | *87* | | |
| $F_1(4) = 50$ | 79* | *66** | 61 | *59* | | | |
| $F_1(5) = 43$ | *72* | 59 | *54* | | | | |
| $F_1(6) = 20$ | 49* | *36** | | | | | |
| $F_1(7) = 18$ | *47* | | | | | | |

Table 3: Sum of response time functions $f_1$ and $F_1$. The minimum value on each assignment diagonal is marked by *.

not have a better asymptotic complexity. If both functions being summed are convex, then the minimum values on adjacent assignment diagonals must be adjacent in a row or column. This fact can considerably accelerate the solution time, since given the minimum on the $x$-processor assignment diagonal we can find the minimum on the $(x + 1)$-processor diagonal by generating and comparing only two additional entries (this is a consequence of the greedy algorithm described earlier). Although we cannot in general assume that both functions are convex, we *can* view them as being piece-wise convex. Thus, if $t_1$ is convex over $[a, b]$, and $\tau_1$ is convex over $[c, d]$, then $t_1 + \tau_1$ is convex over $[a, b] \times [c, d]$ and we can efficiently find minima on assignment diagonals restricted to this subdomain. Working through the details (which are straightforward), one finds that the complexity of this approach is $O(rnp)$, where $r$ is the maximum number of convex subregions spanned by any given assignment diagonal. Of course, in the worst case $r = O(p)$, leaving us still with an $O(np^2)$ algorithm.

# 4 Series-Parallel Tasks

Algorithms for the analysis of series and parallel task structures can be used to analyze task-structures whose graphs form series-parallel directed acyclic graphs. We show that the response

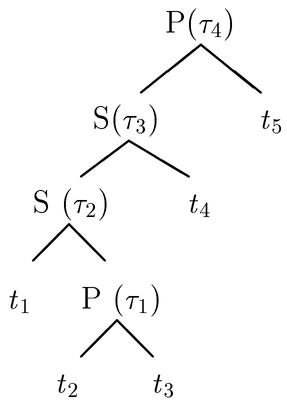


Figure 2: Binary decomposition tree

time function for any such 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 [34], in which a series-parallel DAG can be parsed as a binary decomposition tree (BDT) in time proportional to the number of edges. The leaves of such a tree correspond to the DAG nodes themselves and internal tree nodes describe either parallel (P) or series (S) compositions. Figure 2 illustrates the BDT (labeling S and P nodes by task names used in discussion) corresponding to the task in Figure 1.

The structure of a BDT specifies the precise order in which we should apply our analyses. The idea is to build up the overall optimal response-time function from the bottom up. Conceptually we mark every BDT node as being computed or not, with leaf nodes being the only ones marked initially. We then enter a loop where each iteration we identify an unmarked BDT node whose children are both marked. We apply a series composition or parallel composition to those childrens' response-time functions depending on whether the node is of type S or P, and mark the node. The algorithm ends when the root node is marked.

In the example, $\tau_1$'s response time function is generated using the parallel algorithm on $t_2$ and $t_3$, the series composition is applied to $t_1$ and $\tau_1$, (for composite task $\tau_2$), which is then composed via another series composition with $t_4$, creating $\tau_3$; finally, $t_5$ is combined via a parallel composition with $\tau_3$ to create the response time function for the overall task structure. At each step one must record the actual number of processors assigned to each task in order to compute the optimal assignment; this is straightforward and needs no discussion.

From the above, we see that the cost of determining the optimal assignment from a BDT is $O(np^2)$, as every response-time function composition has worst case cost $O(p^2)$ and there are $n-1$ such compositions performed.

# 5 The Throughput Problem

Real time applications often require that the processing of every data set meet a response-time deadline. At system design time it becomes necessary to assess the maximal throughput possible under the constraint. This is our throughput problem. In this section we show how solutions to the response-time problem can be used to solve this new problem in $O(np^2 \log p)$ time.

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

**Lemma 5.1** *For any pipeline computation let $\mathcal{F}_\lambda(p)$ be the minimal possible response time using $p$ processors, given throughput constraint $\lambda$ and the assumption of static processor-to-task mapping. Then for every fixed $p$, $\mathcal{F}_\lambda(p)$ is a monotone nondecreasing function of $\lambda$.*

**Proof:** Let $p$ be fixed. As before, let $u_\lambda(t_i)$ be the minimum number of processors required for all elemental tasks comprising $t_i$ to meet throughput constraint $\lambda$. For every $t_i$, $u_\lambda(t_i)$ is clearly a monotone nondecreasing function of $\lambda$. Recall that $\mathcal{T}_\lambda(p)$ is the set of all assignments that meet the throughput constraint $\lambda$ using no more than $p$ processors. Whenever $\lambda_1 < \lambda_2$, we must have $\mathcal{T}_{\lambda_2}(p) \subseteq \mathcal{T}_{\lambda_1}(p)$, because of the monotonicity of each $u_\lambda(t_i)$. Since $\mathcal{F}_\lambda(p)$ is the minimum cost among all assignments in $\mathcal{T}_\lambda(p)$, we have $\mathcal{F}_{\lambda_2}(p) \leq \mathcal{F}_{\lambda_1}(p)$. ∎

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 [4].

Suppose for a given pipeline computation we are able to solve for $\mathcal{F}_\lambda(p)$, given any $\lambda$. The set of all possible throughput values is $\{1/f_i(x) \mid i = 1, \ldots, n; x = 1, \ldots, p\}$; $O(pn\log(pn))$ time is needed to generate and sort them. Given response time constraint $\hat{\gamma}$, and tentative throughput $\lambda$, we may determine whether $\mathcal{F}_\lambda(p) \leq \hat{\gamma}$. Since $\mathcal{F}_\lambda(p)$ is monotone in $\lambda$, we use a binary search to identify the greatest $\lambda = \lambda^*$ for which $\mathcal{F}_{\lambda^*}(p) \leq \hat{\gamma}$. The associated processor assignment maximizes throughput (using $p$ processors), subject to response time constraint $\hat{\gamma}$. There being $O(\log p)$ solutions of the response-time problem, the complexity for the throughput problem is $O(np^2 \log p)$.

# 6 An Application

In this section we report the results of applying our methods to a motion estimation system in computer vision. Motion estimation is an important problem 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. A primary goal is to ensure that the computational throughput meets the input data rate. Subject to this constraint, we desire that the response time be as small as possible. The application itself is described in detail in [8, 21]. 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 [22]. The following subsection briefly describes the underlying computations.

## 6.1 A Motion Estimation System

Our example problem is a linear pipeline with nine stages, each stage is a task. The data sets input to the task system are a continuous stream of stereo image pairs of a scene containing the moving vehicles. The tasks perform well-known vision computations such as 2-D convolution, extracting zero crossings and feature matching, similar to computations in the Image Understanding Benchmark [35]. All nine tasks were implemented on a distributed memory machine, the Intel iPSC/2 hypercube [5]. We applied the system above to a problem using outdoor images [8]. The relevant response-time functions are shown in Table 4 for selected processor sizes. Measurements include all overheads, computation time and communication times.

| 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 Intel iPSC/2 of various sizes, in seconds (* indicates extrapolated values)

Figure 3: Minimal response time as a function of the throughput constraint

### 6.2 Experimental Results

We applied the series task algorithm using Table 4, for a range of possible 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, given constraint $\lambda = 0.05$ frames/second. The response times shown are those *predicted* by our algorithms. Nevertheless, *observed* response times using the computed allocations were observed to be in excellent agreement with these figures—the relative error was less than 5% in all measurable cases.

The processor allocation behavior is intuitive. Tasks $t_1$, $t_5$, and $t_8$ have much larger response times than the others. As increasingly more processors are allocated to the problem, these three tasks receive the lion's share of the additional processors.

Figure 3 illustrates the tension between response time and throughput by plotting the minimal response time function for the entire pipeline computation, as a function of the throughput constraint. For any problem there will be a throughput $\lambda_{\min}$ achieved when processors are allocated entirely to minimize response time. The flat region of the curve lies over throughput constraints $\lambda \leq \lambda_{\min}$. The response time curve turns up, sometimes dramatically, as the throughput constraint moves into a region where response time must be traded off for increased throughput.

| | Multiprocessor Size (No. of Procs.) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 32 | | 64 | | 128 | | 256 | |
| Task No. | 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 |

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

# 7 Summary

In this paper we consider performance optimization of series-parallel pipelined computations. The problem arises when a system of individually parallelizable tasks is to be applied repeatedly to a long sequence of data sets. Given a large supply of processors, parallelism can be exploited both by pipelining the data sets through the task structure, and by allocating multiple processors to individual tasks. We treat the dual problems of minimizing response time subject to a throughput constraint, and maximizing throughput subject to a response time constraint.

We showed that problems with $p$ processors and $n$ tasks satisfying series-parallel precedence constraints can be solved in low-order polynomial time: response time (subject to a throughput constraint) is minimized in $O(np^2)$ time, and throughput (subject to a response time constraint) is maximized in $O(np^2 \log p)$ time. To place the work in a realistic setting we evaluated the performance of our assignment algorithms on the problem of stereo image matching. The results predicted by our analysis were observed to be very close to measured on actual systems.

Future endeavors include the provision of algorithms for general task structures and investigation of dynamic assignment algorithms. Also, we believe that our results can be extended to task models that include "branching", such as are encountered with CASE statements. This feature essentially forces us to treat response times and throughputs as being stochastic. We also believe that our approach can be extended to consider the effects of certain types of communication contention.

# 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] 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.

[3] 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.

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

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

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

[7] H-A. Choi and B. Narahari, Algorithms for mapping and partitioning chain structured parallel computations. In *Proceedings of the 1991 International Conference on Parallel Processing.*, St. Charles, IL., pp. 625-628.

[8] 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.

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

[10] K. Dussa and B. Carlson and L. Dowdy and K.-H. Park. Dynamic partitioning in a transputer environment. *Proceedings of the 1990 ACM SIGMETRICS Conference*, 203–213, May 1990.

[11] J. Du and Y-T. Leung. Complexity of scheduling parallel task systems. *SIAM J. Discrete Math.* 2(4):473–487, November 1989.

[12] B. Fox. Discrete optimization via marginal analysis. *Management Science*, vol. 13, 909-918, May 1974.

[13] 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.

[14] 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.*

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] C.D. Polychronopoulos, D.J. Kuck and D.A. Padua. Utilizing multidimensional loop parallelism on large scale parallel processor systems. *IEEE Trans. on Computers*, 38(9):1285–1296, September 1989.

[26] 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.

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

[28] K. Sevcik. Characterizations of parallelism in applications and their use in scheduling . *Proceedings of the 1989 ACM SIGMETRICS Conference*, 171-180.

[29] 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.

[30] 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.

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

[32] H. S. Stone, J. Turek, and J.L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. on Computers*, 41(9):1054–1068, September 1992.

[33] 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.

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

[35] 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.