Optimal Latency–Throughput Tradeoffs for Data Parallel Pipelines

Jaspal Subhlok
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213
jass@cs.cmu.edu

Gary Vondran
Hewlett Packard Laboratories
Hewlett Packard Company
Cambridge, MA 02142
vondran@hpl.hp.com

Abstract

This paper addresses optimal mapping of parallel programs composed of a chain of data parallel tasks onto the processors of a parallel system. The input to this class of programs is a stream of data sets, each of which is processed in order by the chain of tasks. This computation structure, also referred to as a data parallel pipeline, is common in several application domains including digital signal processing, image processing, and computer vision. The parameters of the performance of stream processing are latency (the time to process an individual data set) and throughput (the aggregate rate at which the data sets are processed). These two criterion are distinct since multiple data sets can be pipelined or processed in parallel. We present a new algorithm to determine a processor mapping of a chain of tasks that optimizes the latency in the presence of throughput constraints, and discuss optimization of the throughput with latency constraints. The problem formulation uses a general and realistic model of inter-task communication, and addresses the entire problem of mapping, which includes clustering tasks into modules, assignment of processors to modules, and possible replication of modules. The main algorithms are based on dynamic programming and their execution time complexity is polynomial in the number of processors and tasks. The entire framework is implemented as an automatic mapping tool in the Fx parallelizing compiler for a dialect of High Performance Fortran.

1 Introduction

A fundamental problem in parallel computing is to find the most efficient mapping of a parallel program onto the processors of a parallel system. In simple data parallel computing, all available processors combine to execute every computation step in a program. Complex applications are often composed of multiple coarse grain tasks, where each task is data parallel and can execute on multiple processors, while different tasks may execute concurrently on different groups of processors. Most modern parallel machines support MIMD execution, and therefore combined data and task (or function) parallel computing. Compiler and runtime support for task and data parallel computing is an active area of research, and several solutions have been proposed [3, 4, 8, 9, 14]. Recent research has also examined the benefits of mixed task and data parallel programming [2, 6, 13].

This paper specifically addresses the mapping of applications composed of a linear chain of data parallel tasks that act on a stream of input data sets. Each task repeatedly receives input from its predecessor task, performs its computation, and sends the output to its successor task. The first task reads external input and the last task generates the final output. Figure 1 shows a chain of three tasks.

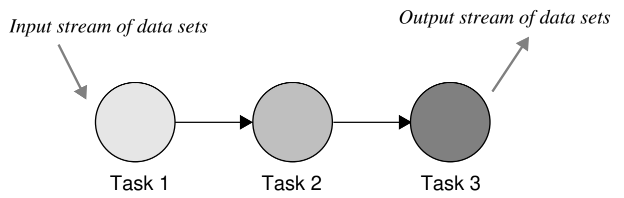

Figure 1: A three task pipeline

A chain of tasks is a relatively simple model of task and data parallel computing, yet it is fairly common in several application areas, including computer vision, image processing, and signal processing [7]. These applications are generally driven by physical sensors like cameras and antennas. They act on streams of relatively small data sets but often have strict throughput and latency requirements. An example application is multibaseline stereo [17], where the first task captures three (or more) images from the cameras, the second task computes a *difference image* for each of 16 disparity levels, the third task computes an *error image* for each difference image, and the final task performs a minimum reduction across error images and computes the final *depth image*.

An application composed of a chain of data parallel tasks can be mapped onto a parallel machine in a variety of ways, as shown in Figure 2. Figure 2(a) shows a pure data parallel mapping where all tasks execute on all processors. Figure 2(b) shows a task parallel mapping where a subset of the processors is dedicated to each task, and multiple data sets are pipelined. It may be possible to have replicated copies of the program to process multiple data sets in parallel on

Sponsored in part by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152. Gary Vondran was at the Department of Electrical and Computer Engineering at Carnegie Mellon University for the duration of this research.

Appears in the Eighth Annual ACM Symposium on Parallel Algorithms and Architecture (SPAA), June 1996, Padua, Italy

different sets of processors, as shown in Figure 2(c). Finally, a mix of task and data parallelism with replication is shown in Figure 2(d).

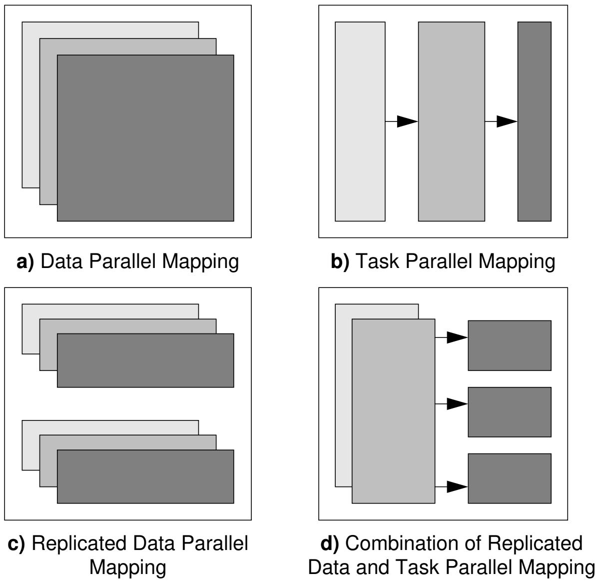

Figure 2: Combinations of data and task parallel mappings

The goal of this research is to identify the best mapping for a chain of data parallel tasks. However, there are two distinct criterion for judging the quality of a mapping: *latency* and *throughput*. The latency is the time taken to process an individual data set, while the throughput is the aggregate rate at which the data sets are processed. Since multiple data sets may be processed in parallel or pipelined, latency and throughput are not directly related. If the objective is to minimize the latency, a simple data parallel mapping is often the optimal solution, since it makes all processors available to each task, and the tasks can execute at the fastest possible speed. For optimizing the throughput, however, it can be more beneficial to process multiple data sets simultaneously, in parallel or as a pipeline. This causes higher latency for individual data sets, but is likely to improve the throughput, since a single data set can be processed more efficiently by a smaller number of processors. The mapping decisions are particularly important for programs that do not scale well, often because of a relatively small data set size, and are therefore not able to use all processors efficiently with pure data parallelism.

The central problem addressed in this paper is as follows: *Given a chain of data parallel tasks and a minimum throughput constraint, identify a mapping corresponding to minimum latency.* The problem of finding a mapping corresponding to maximum throughput with a given maximum latency constraint is also discussed.

Our basic approach is to execute the user program with different mappings to automatically infer the computation and communication characteristics of the tasks in the program, and use these characteristics to derive an optimal mapping. We have used this methodology to make mapping decisions at compile time, but it can also be the basis of a runtime mapping tool. This approach assumes that the execution behavior of a program is *not* strongly dependent on the properties of the input data sets.

There is a large large volume of literature on mapping and scheduling parallel programs, and some excellent references are [1, 12, 19]. We focus only on mapping coarse grain task parallelism, and assume that a data parallel compiler is available to exploit fine grain parallelism. We compare our approach to two other research efforts that address assignment of processors to coarse grain tasks.

Choudhary et. al. [5] address the problem of optimal processor assignment to a pipeline of coarse grain tasks, assuming that the communication cost can be folded into the computation cost. In our experience, a realistic model of communication is very important for a practical automatic mapping system. Ramaswamy et. al. [11] use convex programming to address the problem of minimizing the completion time for one data set (latency), but do not address the throughput constraints. Also, our algorithms are not tied to any particular way of modeling execution , unlike mathematical programming techniques like convex programming. Finally, both these approaches address the processor assignment problem, but do not address clustering of tasks into modules, or replication of tasks. Our earlier work [15] addressed unconstrained throughput optimization, while this paper addresses tradeoffs between latency and throughput optimization.

The methods developed in this paper are implemented as an automatic mapping tool for the Fx compiler, and have been used to map several applications, including multibaseline stereo and narrowband tracking radar [7]. Fx is a subset High Performance Fortran [10] compiler with support for task parallelism [14, 18]. The targets for Fx are the Intel Paragon, the Intel iWarp,the IBM SP2, the Cray T3D, and networks of workstations running PVM.

This paper is organized as follows. Section 2 introduces the algorithmic problem to be solved. Section 3 presents an optimal dynamic programming solution to the general latency optimization problem, and section 4 outlines how it can be used to solve the constrained throughput optimization problem. Section 5 discusses the estimation of execution behavior. Section 6 illustrates the importance of mapping algorithms with examples, and discusses some practical issues in automatic mapping. Section 7 contains conclusions.

2 Problem statement

2.1 Execution model

A program in our model consists of a sequence of tasks $t_1, t_2, t_3, ...t_k$, where each task receives a data set from its predecessor, processes it, and sends the output to its successor; the first task reads external input data sets and the last task generates the final output data sets. Each task can execute on one or more processors, and the execution time of each task is a function of the number of processors. The execution time functions for the chain of tasks are represented as follows:

$$f_1^{exec}, f_2^{exec}, f_3^{exec}, ..., f_k^{exec}$$

A pair of adjacent tasks may be assigned to the same set of processors or to different sets of processors. If a pair of tasks is assigned to the same set of processors, the communication between them is a potential internal redistribution of data. The communication time is a function of the number of processors assigned to the tasks. The internal communication time functions for the chain of tasks are represented as follows:

$$f^{icom}_{1\to2}, f^{icom}_{2\to3}, f^{icom}_{3\to4}, \ldots, f^{icom}_{(k-1)\to k}$$

If a pair of tasks is assigned to different sets of processors, the communication between them involves moving data between different groups of processors. The communication time is a function of the number of processors assigned to the sending task and the receiving task. The external communication time functions for the chain of tasks are represented as follows:

$$f^{ecom}_{1\to2}, f^{ecom}_{2\to3}, f^{ecom}_{3\to4}, \ldots, f^{ecom}_{(k-1)\to k}$$

The response time of a task, which is the total time that a task spends processing one data set, including the time to receive an input data set and the time to send the output data set, is a function of the number of processors assigned to the task and the tasks adjacent to it. The response time functions for the chain of tasks are represented as follows:

$$f_1, f_2, f_3, \ldots, f_k \quad \text{where} \quad f_i = f^{com}_{(i-1)\to i} + f^{exec}_i + f^{com}_{i\to(i+1)}$$

and f^{com} represents the applicable internal or external communication function.

The sender and the receiver tasks are involved in the entire duration of a communication step. The execution of one data set by a part of a chain of tasks is modeled in Figure 3 using $p_1, p_2, p_3, \ldots p_k$ to denote the number of processors assigned to the corresponding tasks in the chain $t_1, t_2, t_3, \ldots t_k$.

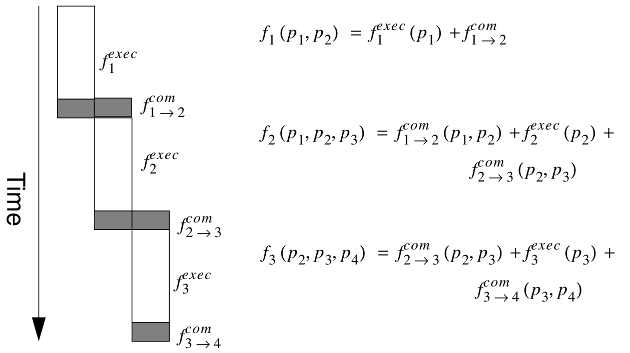

Figure 3: Execution model of a chain of tasks. The total time spent on a data set by (say) task 2 includes the time to receive a data set from task 1, which is a function of p_1 and p_2 denoted by $f^{com}_{1\to2}(p_1, p_2)$, the time to process the data set, which is a function of p_2 denoted by $f^{exec}_1(p_2)$, and the time to output the data set to task 3, which is a function of p_2 and p_3 denoted by $f^{com}_{2\to3}(p_2, p_3)$. Thus the total processing time of task 2 is a function of p_1, p_2 and p_3, denoted by $f_2(p_1, p_2, p_3)$.

We point out some important aspects of this execution model and their implications. First, the communication cost of sending a data set between a pair of tasks executing on different sets of processors is assumed to be an arbitrary function of the number of processors assigned to the sender as well as the receiver. This coupling makes it harder to develop efficient processor assignment algorithms, but our experience suggests that it is important to model communication costs accurately. Second, the task chain acts on a long input stream of independent input data sets, and therefore, multiple data sets can be processed concurrently. Finally, the execution and communication times are static functions of the relevant numbers of processors. This is an important restriction on the applicability of this approach.

2.2 The mapping problem

We discuss the components of a mapping and show how a mapping is represented.

Processor assignment problem

If we assume that each task is assigned a disjoint subset of the processors, the mapping problem is that of finding an assignment A from tasks to processors, i.e. $A(i)$ equals the number of processors assigned to the task t_i. For a valid assignment of P processors to k tasks:

$$\sum_{i=1}^{k} A(i) <= P$$

For the three task example shown in Figure 1, the problem is simply to divide the available processors among the three tasks according to the optimality criterion. Note that the computation is pipelined and the tasks are typically working on different data sets.

Clustering into modules

A pair of adjacent tasks in a chain may execute on the same set of processors. We assume that any two tasks execute on identical or disjoint sets of processors. For the purpose of mapping, subchains of tasks are clustered into modules, and each module is assigned a disjoint subset of the available processors. A *clustering* of a chain of tasks is a list of modules $m_1, m_2, m_3, \ldots m_l$, where each module contains a subchain of the task chain $t_i, t_{i+1}, t_{i+2}, \ldots t_j$, and every task belongs to exactly one module.*

As an example, one way to map the three task pipeline of Figure 1 is to cluster tasks 1 and 2 in one module, the third task in a different module, and divide the available processors among the two modules, as shown in Figure 4. Each data set will be processed by the first two tasks on the processors executing the first module, and then transferred to the processors executing the second module for processing by the third task.

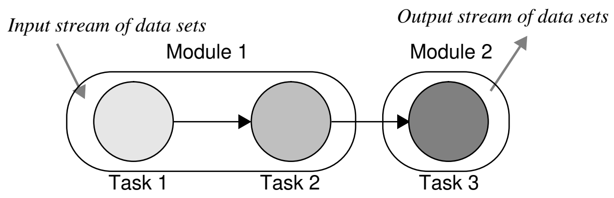

Figure 4: Clustering a three task pipeline into modules

*We assume that only adjacent tasks in a chain can be clustered together. Other clusterings (e.g. tasks 1 and 3 in one cluster and 2 in a different cluster) are rarely profitable because of the higher communication cost and are not considered.

Replication of modules

Suppose a set of p processors were available to a module. It may be profitable to divide the p processors in two or more disjoint groups and process alternate data sets on these disjoint groups of processors, as shown in Figure 5. While the response time for each data set, and therefore, the overall latency, will increase, the throughput may improve since multiple data sets are processed in parallel. This is called replication, and its legality depends on data dependences. For this paper, we assume that the tasks are known to be replicable or non-replicable. Replication is primarily used to improve throughput.

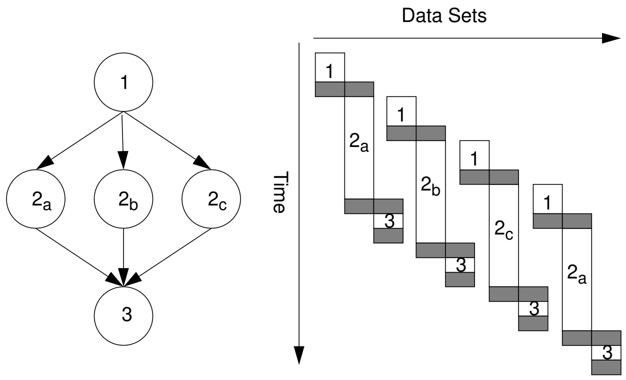

Figure 5: Replication. Module 2 is replicated to 3 instances. At any given time, up to 3 data sets can be processed by module 2, one each by the group of processors executing 2_a, 2_b and 2_c. Module 1 sends each output data set to one of the instances of module 2 in round robin sequence.

Representation of a mapping

A mapping is a result of clustering, replication and processor assignment decisions. The mapping of a chain of tasks $t_1, t_2, t_3, ..., t_k$ can be expressed as a list of modules $M_1, M_2, M_3, ..., M_l$, where M_i is a triplet $(\mathcal{T}, r, p)$, where $\mathcal{T}$ is the subchain of tasks clustered into that module, r is the number of times the module is replicated, and p is the number of processors assigned to each instance of the module.

The execution and communication time functions of a module can be composed from the corresponding functions of the tasks that constitute the module. For a mapping containing l modules, the latency , which is the time for processing an individual data set, can be expressed as:

$$\sum_{i=1}^{l}(f_i^{exec} + f_{i \to i+1}^{ecom})$$

and the throughput, which is the rate of processing data sets, as:

$$1/(\max_{i=1}^{l}(f_i/r_i))$$

where f , f^{exec}, and f^{ecom} correspond to the modules in the mapping and apply to the number of processors in each instance of a module. Also, since the last module does not have a successor, $f_{l \to l+1}^{ecom}$ is undefined and a value of 0 is used.

3 Latency optimization

In this section, we present an algorithm to find the mapping of a chain of tasks that minimizes the latency, while ensuring that a given minimum throughput constraint is satisfied. We first present an algorithm for unconstrained latency minimization without considering clustering or replication. We then extend this algorithm, first to include a throughput constraint, and subsequently to include clustering and replication. The final algorithm solves the general constrained latency optimization problem.

3.1 Unconstrained latency minimization

We begin by presenting a dynamic programming algorithm to find an optimal assignment of processors for a chain of tasks, without a throughput constraint. We assume that each task must execute on a disjoint set of processors, and that there is no replication. Thus, we are dividing the available processors among the given set of tasks such that the total execution time of the task chain (for one data set) is minimized. Every task corresponds to a unique module, so we will refer only to tasks and not to modules. For simplicity of presentation, we also assume that a task can use all processors available to it, and that the optimal assignment uses all available processors.

Approximate latency minimization can be done by adding processors one at a time to the task chain, by always selecting the task whose execution time decreases the most by adding a processor. Tasks that take longer to execute and tasks that scale well are favored, since their execution time is likely to decrease more with an additional processor. But this simple method does not always lead to an optimal mapping since it does not consider all possible assignments.

We first state our algorithm intuitively. The optimal assignment of p processors for a subchain of tasks, say from task t_1 to t_j, can be computed without consideration of the remaining parts of the chain, except that the number of processors assigned to the task t_{j+1} must be known, since the communication cost from t_j to t_{j+1} depends on it. This is illustrated in Figure 6. If we build a table of optimal assignments for this subchain for all possible values of the number of processors assigned to this subchain, and all possible values of the number processors assigned to the next task, then we can use this information to build a table for the subchain from task t_1 to t_{j+1}. Thus we can build optimal assignments for incrementally longer subchains, and finally the optimal assignment for the full chain.

We now state the problem formally. Let T represent a chain of k tasks, labeled $t_1, t_2, ..., t_k$, and T_j represent the subchain $t_1, t_2, ..., t_j$ of the chain T. Let A_j represent an assignment of processors for a subchain of tasks T_j, i.e. $A_j(i)$ equals the number of processors assigned to the task t_i. An assignment A_j for a subchain of length j can be appended with a new value to obtain an assignment for a subchain of length $j+1$, e.g. $A_{j+1} = A_j \oplus \{(A(j+1) = p_{j+1})\}$. We will omit the subscript j of A_j for conciseness, whenever the length of the assignment is clear from the context.

Let L be a function that returns the latency of an assignment A for a task subchain T_j. The latency function L for T_j includes the computation time of the tasks in the subchain, the communication time between them, as well as the communication time between the last task t_j and the next task t_{j+1}, if it exists. Hence L is a function of the assignment A, and p_{next}, the number of processors assigned to the next task t_{j+1}. The latency function for an assignment to the subchain T_j can be evaluated as follows:

$$\begin{aligned} L(A, p_{next}) &= \sum_{i=1}^{j} f_i^{exec} A(i) \\ &+ \sum_{i=1}^{j-1} f_{i \to i+1}^{ecom}(A(i), A(i+1)) \\ &+ f_{j \to j+1}^{ecom}(A(j), p_{next}) \end{aligned}$$

We show how an optimal assignment for a subchain of tasks is related to the optimal assignment for the full chain. Let A_j be a unique optimal assignment of p processors to a task subchain T_j, assuming that p_{j+1} processors are assigned to the next task t_{j+1}. We argue that A_j is the assignment for T_j in the final optimal assignment for the full chain T, if p and p_{j+1} are the numbers of processors assigned to T_j and t_{j+1}, respectively, in the optimal assignment for the full chain. This is the case since no other factors influence the final assignment for the subchain T_j. Note that p_{j+1}, the number of processors assigned to the task following the subchain T_j, is needed to compute the optimal assignment for T_j, since the communication time between the last task in T_j and the next task t_{j+1} is part of the latency of the subchain T_j. We have the following result:

Lemma 1 Suppose the function $\mathcal{A}_j(p_{total}, p_{next})$ returns the optimal assignment of p_{total} processors to a subchain T_j of a task chain T, given that the task immediately following the subchain is assigned p_{next} processors. (If no *next* task exists, p_{next} is redundant and assigned ϕ.) If an optimal assignment for the full chain T has p and p_{j+1} processors assigned to the subchain T_j and the task t_{j+1}, respectively, then the assignment of processors to T_j in this optimal mapping is that returned by $\mathcal{A}_j(p, p_{j+1})$.

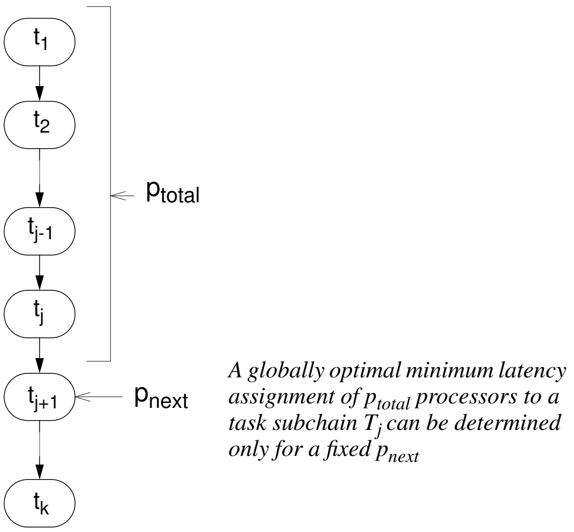

Figure 6: Processor assignment with dynamic programming

The lemma is illustrated in Figure 6. The optimal assignment of P processors to a chain of k tasks, and the corresponding minimum latency, is obtained directly from the function $\mathcal{A}_k$.

$$\begin{aligned} A^{opt} &= \mathcal{A}_k(P, \phi) \\ L^{opt} &= L(A^{opt}, \phi) \end{aligned}$$

We now present a constructive recursive definition of $\mathcal{A}$.

$$\begin{aligned} \mathcal{A}_j(p_{total}, p_{next}) &= A_j \equiv \min_{q=1}^{p_{total}} L\,[\,(\mathcal{A}_{j-1}(p_{total} - q, q) \\ &\quad \oplus\{(A(j) = q)\}), p_{next}\,] \qquad \text{if } j > 1 \\ &= \{(A(1) = p_{total})\} \qquad \text{if } j = 1 \end{aligned}$$

We explain the meaning of the above equation and describe how it is used to compute the processor assignment for minimum latency. Suppose we have a table defining the optimal assignment $\mathcal{A}_{j-1}$, and the corresponding minimum latency, for every possible number of processors assigned to the task subchain T_{j-1} and the next task t_j. To determine an entry in the table for $\mathcal{A}_j$, say $\mathcal{A}_j(p_{total}, p_{next})$, we need to find the optimal latency assignment for the subchain T_j, given that the number of processors assigned to T_j, and the next task t_{j+1}, are p_{total} and p_{next}, respectively. If we assume that q processors are assigned to t_j, the corresponding optimal assignment and latency for T_{j-1} can be looked up from the table for $\mathcal{A}_{j-1}$, and the corresponding latency for the subchain T_j can be directly computed, given that p_{next} processors are assigned to t_{j+1}. By comparing the latency of the assignments for all possible values of q, we obtain the optimal assignment $\mathcal{A}_j(p_{total}, p_{next})$, and the corresponding minimum latency. The optimal assignment and the minimum latency for the complete chain of k tasks is obtained by repeating the procedure for incrementally longer subchains. **Complexity:** The size of a definition table for $\mathcal{A}$ is at most $O(P^2)$. Each entry of a new table $\mathcal{A}_j$ is determined by comparing at most P values, each of which is calculated from $\mathcal{A}_{j-1}$ and the response time functions in $O(1)$ operations. Hence $\mathcal{A}_j$ is obtained from $\mathcal{A}_{j-1}$ in $O(P^3)$ operations. Since there are k such steps, the total complexity of the algorithm is $O(P^3 k)$ operations.

3.2 Latency minimization with a throughput constraint

The throughput of a chain of tasks is the rate at which data sets are processed. Intuitively, to increase the throughput of a chain of tasks, processors are added to the slowest or the *bottleneck* task, even though adding them to another task may increase the response time of that task, as well as the total latency, by a larger amount. This is the case since the throughput is determined by the processing rate of the slowest task in a pipeline, and therefore, to improve the throughput, the slowest task has to be speeded up. When a throughput constraint is given, we have to ensure a certain minimum processing speed for every task, and then minimize the total latency of the chain.

The throughput of an assignment A to a subchain T_j, given that the next task t_{j+1} is assigned p_{next} processors, can be computed by the following function:

$$\begin{aligned} H(A, p_{next}) &= min[\{\min_{i=1}^{j-1} 1/f_i(A(i-1), A(i), A(i+1))\}, \\ &\quad \{1/f_j(A(j-1), A(j), p_{next})\}] \end{aligned}$$

We restate that the response time function f_i represents the sum of the computation and the communication times of the task t_i, and is therefore a function of the number of processors assigned to the task t_i and to the tasks before and after it in the task chain.

Note that the number of processors assigned to the task immediately following a subchain T_j is also needed to compute the throughput of a subchain. The algorithm for unconstrained latency minimization can be modified in a simple way to compute the minimum latency with a minimum throughput constraint. During the building of the table of optimal assignments, only assignments that satisfy the throughput constraint are considered.

We introduce the constrained latency function $L^{\mathbf{h}}$, which returns the latency of an assignment if the corresponding throughput is greater than $\mathbf{h}$, and *infinity* otherwise.

$$\begin{aligned} L^{\mathbf{h}}(A, p_{next}) &= L(A, p_{next}) && \text{if } H(A, p_{next}) \geq \mathbf{h} \\ &= \infty && \text{if } H(A, p_{next}) < \mathbf{h} \end{aligned}$$

Using the constrained latency function in place of the unconstrained latency function, the equation system from the previous section can be used for latency minimization with a throughput constraint. We have:

$$\begin{aligned} \mathcal{A}_j(p_{total}, p_{next}) &= A_j \equiv \min_{q=1}^{p_{total}} L^{\mathbf{h}}\,[\,(\mathcal{A}_{j-1}(p_{total} - q, q) \\ &\quad \oplus\{(A(j) = q)\}), p_{next}\,] \quad \text{if } j > 1 \\ &= A_1 \equiv L^{\mathbf{h}}\{(A(1) = p_{total})\} \quad \text{if } j = 1 \end{aligned}$$

When the constrained latency function $L^{\mathbf{h}}$ returns infinity, the corresponding assignment is undefined. If all entries for the optimal mapping of a subchain are undefined, then no assignment satisfies the throughput constraint, and therefore, no solution exists.

The algorithm for finding the constrained optimal latency is essentially the same as that for unconstrained latency optimization presented earlier, with L replaced by L^h. The computation complexity is also the same, that is $O(P^3k)$.

3.3 Latency minimization with clustering

In the previous subsections, we have assumed that every task is assigned a disjoint set of processors. However, it is often more beneficial to group a set of tasks together in a module, and assign a set of processors to be shared by the tasks in the module. The tasks inside a module are sequentialized, that is, all the processors are used by one task until it finishes processing a data set, then by the other tasks in order. Coalescing a pair of tasks in a single module can potentially reduce the communication time between the tasks and improve the latency, but it can also reduce the throughput if the tasks do not scale well, since they may not execute as efficiently on a larger number of processors.

We outline the changes to the dynamic programming algorithm to take clustering into account. Because of lack of space, we will not present the modified equation system and, and instead refer to [16].

Given a chain of tasks T, the central problem, as before, is to find an optimal mapping of a subchain T_j using p processors, which is guaranteed to be the mapping for T_j in a global optimal mapping of T, if T_j is allocated p processors.

Let M_{next} be the module to which the task t_{j+1} immediately following T_j is assigned. To determine the globally optimal mapping of the subchain T_j in the presence of clustering, we need to know the number of processors assigned to the next module M_{next}, just as we needed to know the number of processors assigned to the next task t_{j+1} in latency optimization without clustering. This information is sufficient if we assume that the last task t_j is *not* clustered in M_{next}. However, to consider the case where t_j is clustered with M_{next}, we must be able to check the response time of M_{next} (with t_j and possibly other preceding tasks assigned to it), to verify if it satisfies the throughput constraint, since it becomes a part of the mapping of subchain T_j. To determine the response time of M_{next}, we need to know the number of tasks following t_{j+1} that M_j is clustered with, and the number of processors assigned to the module following M_{next}.

The dynamic programming solution is similar to that presented for optimal mapping without clustering. The difference is that we work with the mapping function $\mathcal{M}_j(p_{total}, p_{next}, p_{next+}, nextmodlength)$ instead of $\mathcal{A}_j(p_{total}, p_{next})$, where $nextmodlength$ is the length of the subchain after the task t_{j+1} assumed to be clustered with t_{j+1}, and p_{next+} is the number of processors assigned to the module following the first module after T_j. Clearly $nextmodlength < k$, the number of tasks, and $p_{next+} < P$, the total number of processors. The computation complexity is increased by a multiplicative factor Pk. Hence, the computation complexity of latency optimization with a throughput constraint and in the presence of clustering is $O(P^4k^2)$.

3.4 Latency minimization with replication

With replication, processors available to a module are divided into multiple groups that process input data sets in a round robin sequence, as shown in Figure 5 and discussed in Section 2.2. Replication implies that each data set uses fewer processors, and therefore, it generally increases the latency, but it can improve the throughput. Hence, the purpose of replication is to help satisfy the throughput constraints.

We briefly discuss how the mapping algorithm is modified to incorporate replication. We use the algorithm in the previous subsection, with the exception that if a particular assignment of tasks and processors to M_{next} does not satisfy the throughput constraint, we determine the number of copies of M_{next} required to satisfy the throughput constraint, and assume their existence if that does not exceed the available number of processors. The complexity of the mapping algorithm is unchanged. Hence, for the general problem of minimizing latency with a throughput constraint, we have the following:

Lemma 2 *Given a chain of k tasks, P processors, and a minimum throughput constraint, an optimal latency mapping for the task chain can be be determined in $O(P^4k^2)$ time.*

3.5 Execution efficiency of latency optimization

The execution time complexity of the latency minimization algorithm is $O(P^4k^2)$. The practical cost of executing this algorithm is an issue for large P, particularly when the algorithm is used at runtime to map and remap programs.

The reason for the quadratic complexity in P (rather than square or cubic) is that we model communication between two tasks as an arbitrary function of the number of processors assigned to the sender and the receiver tasks. If communication cost is modeled as two independent sending and receiving costs, the complexity of the algorithm is lower, but our experience is that this is not a realistic model of communication. We also make no assumptions about the behavior of the execution and communication time functions. Several optimizations based on the properties of the execution time functions can be used to improve the performance, but a discussion is beyond the scope of this paper.

However, we will briefly discuss the use of small processor blocks, instead of individual processors, as the assignment units for the mapping algorithms. This leads to a simple tradeoff between precision and complexity. Using a block of b processors instead of 1 would reduce the computation complexity of the general algorithm by a factor of b^4, while ensuring that the mapping obtained is within b processors of the optimal mapping.[†] In general, if we are willing to be off by one processor for every P_0 processors in the system, i.e. by a fixed fraction of the total number of processors, the total complexity is $O(P_0^4 k^2)$. This is independent of the actual number of processors, and is a function of the desired precision. We believe that this is a practically useful way to use these algorithms for large numbers of processors.

4 Throughput optimization

The general throughput optimization problem is to find a mapping that corresponds to the maximum throughput with a maximum latency constraint. In previous work [15], we developed an algorithm for optimizing the throughput, but without a latency constraint. It appears that there is no polynomial solution to the problem of optimizing throughput with a latency constraint.

However, repeated application of the algorithm for latency optimization can be used to find a near optimal solution to the throughput problem. We first present the following result, which essentially states that increasing the minimum throughput constraint when solving the latency optimization problem cannot decrease the optimal latency.

Lemma 3 *Given a task chain and a fixed number of processors, let H be the maximum possible throughput for a maximum allowed latency $\mathbf{l}$. Let L_1 be the minimum latency for the chain with a minimum throughput constraint $\mathbf{h_1}$, and L_2 the minimum latency with a minimum throughput constraint $\mathbf{h_2}$. Then:*

$$\mathbf{h_2} \geq H \geq \mathbf{h_1} \Leftrightarrow L_2 \geq \mathbf{l} \geq L_1$$

This result is fairly obvious and has the following implication. If we find the optimal latency for two different throughput constraints, and the given latency constraint is between the two optimal latency values, then the optimal throughput is between the two throughput constraints. Using this result, we can obtain successively improved approximations to the optimal throughput for a given latency constraint by using binary search and repeatedly solving the optimal latency problem. An initial range for the throughput can be determined by solving the unconstrained throughput problem, which has the same complexity as the general latency problem [15].

Therefore, the general throughput problem can be solved in $O(log(H_0/\delta)P^4k^2)$ time by this procedure, where H_0 is the range of possible throughput values, and δ is the maximum allowed difference between the approximate and the exact optimal throughput. We will not discuss the optimization of throughput any further.

[†]This means that by changing the number of processors assigned to a task at most by b, an optimal or better than optimal mapping can be obtained.

5 Estimation of execution behavior

The mapping algorithms presented in this paper use the execution time and communication time functions for the tasks. One advantage our algorithms have over those based on mathematical programming (e.g. linear or convex programming) is that they are not tied to any particular execution model. The execution time and communication time functions may be mathematical functions computed at compile time or runtime, or they may be defined pointwise, possibly using interpolation.

In our mapping tool, we model computation and communication times as polynomial functions. The parameters of the model are derived automatically by analyzing the profile information from a set of executions.

The execution time of a task-subroutine on a set of p processors is modeled as:

$$f^{exec}(p) \quad = \quad C_1^{exec} + C_2^{exec}/p + C_3^{exec} * p$$

The constant term C_1^{exec} reflects fixed cost sequential and replicated computation, the second term represents parallel computation whose cost decreases linearly with the number of processors, and the last term represents overheads that increase with the number of processors.

The external communication function denotes the time for transferring data between a pair of tasks mapped on different sets of processors, and is modeled as a function of the number of processors assigned to the sending and the receiving tasks (p_s and p_r) as follows:

$$\begin{aligned} f^{ecom}(p_s, p_r) \quad &= \quad C_1^{ecom} + C_2^{ecom}/p_s + C_3^{ecom}/p_r \\ &\quad +C_4^{ecom} * p_s + C_5^{ecom} * p_r \end{aligned}$$

The internal communication function denotes the time for a potential redistribution of data when a pair of communicating tasks are mapped to the same set of processors, and is modeled in the same manner. In this case the sending processors are also the receiving processors, and hence there are only 3 terms.

$$f^{icom}(p) \quad = \quad C_1^{icom} + C_2^{icom}/p + C_3^{icom} * p$$

6 Results

The execution model and the mapping algorithms presented in this paper are implemented as an automatic mapping tool in the Fx compiler [9]. We present a set of results using a 64 processor Intel iWarp, and 64 processors of an Intel Paragon. The purpose is to illustrate the mapping procedure, demonstrate that the mapping tool automatically finds optimal or nearly optimal mappings in practice, and to examine the practical reasons for inaccuracies in finding optimal mappings.

6.1 Example Program: FFT-Hist

The FFT-Hist program takes a sequence of m $n \times n$ complex arrays from a sensor (e.g., a camera). For each of the m input arrays, a two dimensional fast Fourier transform (FFT) is performed, followed by statistical analysis. The main loop of the program and the task graph are shown in Figure 7. For each iteration of the loop, the `colffts` function inputs the array `A` and performs 1D FFTs on the columns. The `rowffts` function performs 1D FFTs on the rows, and the `hist` function analyzes and outputs the results. The `rowffts` and `colffts` functions are parallel with no communication, while `hist` has a significant amount of internal communication. This is an interesting program because it represents the structure of many applications in image and signal processing.

```
do i = 1,m
   call colffts(A)
   call rowffts(A)
   call hist(A)
enddo
```

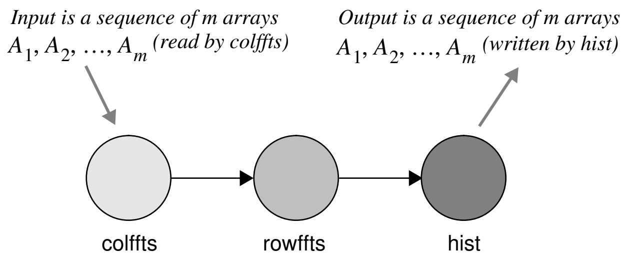

Figure 7: FFT-Hist example program and task graph

6.2 Modeling and mapping FFT-Hist

We used our mapping tool on the FFT-Hist program for two data set sizes (256×256 and 512×512) and two parallel machines, the Intel iWarp and the Intel Paragon. In each case, the program was run through a training set of sample mappings to build a computation and communication model for the tasks, as discussed in section 5.

The mapping tool used the algorithms presented in Section 3 to determine the mapping corresponding to the optimal latency, with and without a minimum throughput constraint. The optimal latency mappings obtained for the 512×512 FFT-Hist on an Intel Paragon are shown in Figure 8, and the corresponding latency and throughput are shown in Table 1. We briefly discuss the intuitive reasoning that leads the tool to infer these optimal mappings.

The mapping obtained for minimizing latency with no throughput constraint is the simple data parallel mapping. This is expected since clustering into multiple modules and replication typically improve the throughput, but also increase the latency, since the number of processors that execute a task is reduced.

The second mapping corresponds to a minimum throughput constraint of 2 data sets per second. In this case, all tasks are again clustered together, but the module is replicated. Replication improves the throughput since the module instances can use 32 processors more efficiently than 64. The optimal latency is significantly higher than the optimal latency without a throughput constraint.

The third mapping was obtained for a more restrictive throughput constraint of 4 data sets per second. This mapping has two modules, *module 1* contains `colffts`, and *module 2* contains `rowffts` and `hist`. *Module 1* is replicated to 2 instances and each instance is assigned 8 processors, while *Module 2* has 3 instances and 14 processors per instance. The primary reason for having two modules is that the communication step between *colffts* and *rowffts* is somewhat more efficient if they are on different groups of processors. The modules are replicated to satisfy the throughput constraint. We were somewhat surprised that a few processors were not used, but experimentation with nearby mappings that used all processors confirmed that the selected mapping was indeed the most efficient.

We have used this example to illustrate some of the factors that influence the mapping, but the optimal mapping is really determined by a quantitative comparison of the impact of several factors and can be hard to predict. For the FFT-Hist example, the nature of the mappings obtained for different throughput constraints was very different for the 256×256 and 512×512 problems, and was also different for the same problem executing on the iWarp and the Paragon.

6.3 Experience with the mapping tool

We have used the mapping tool and the Fx compiler to generate mappings for several task and data parallel programs. In Table 1 we present results from FFT-Hist and two other applications; narrowband tracking radar and multibaseline stereo [7]. The relevant properties of the latter two programs are discussed in [13].

Table 1 shows the optimal latency obtained with and without a throughput constraint. We observe that adding a throughput constraint can lead to a mapping with a significantly higher latency. The sole exception is the radar program, for which the latency remains unchanged. The reason is that the radar program, when optimized for latency, is able to use only a few processors effectively, and additional throughput is obtained by replicating the computation on unused processors.

6.4 Accuracy of performance estimates

Table 1 also shows that the predicted and measured latency generally differ. The maximum estimation error for these examples was 17.9 percent. The mapping algorithms used are optimal, hence the inaccuracy in prediction is entirely because of the inaccuracy in the modeling of the execution behavior. Modeling of execution is not a contribution of this paper, and we have selected a relatively simple model. A small to moderate difference between predicted and measured values is expected, and in fact the reason that some cases have no error is that the optimal mapping happened to coincide with one of the data points used for execution modeling. Some of the factors that cause inaccuracy in performance prediction are as follows:

1. The execution performance sometimes changes abruptly as the number of processors is changed. For example, some tasks execute significantly more efficiently with power of 2 number of processors.

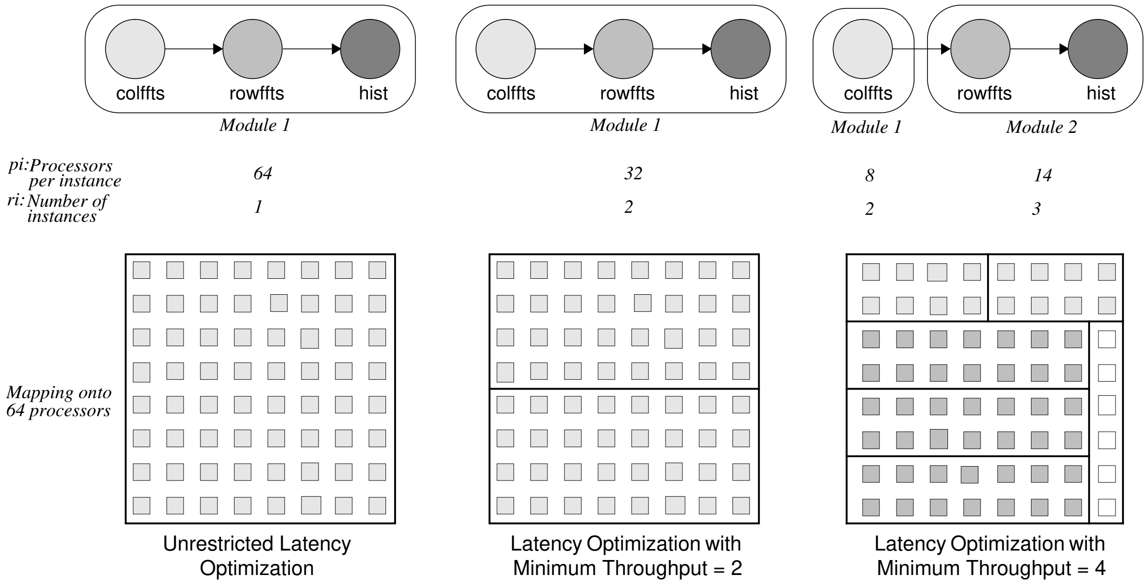

Figure 8: Mappings of a 512x512 FFT-Hist program on an Intel Paragon

Problem Description		Name	Throughput	Optimal Latency		Percent	Measured
Name of Program	Array Size	of Machine	Constraint datasets/sec	Predicted sec	Measured sec	Difference	Throughput datasets/sec
FFT-Hist	256x256	Paragon	None	.232	.256	10.3	3.90
			8	.357	.293	-17.9	13.3
FFT-Hist	256x256	iWarp	None	.463	.444	-4.1	2.25
			8	.484	.491	1.5	8.14
FFT-Hist	512x512	Paragon	None	.578	.502	-13.1	1.99
			2	.853	.807	-5.4	2.48
FFT-Hist	512x512	iWarp	None	.739	.739	0.0	1.35
			2	.845	.845	0.0	2.37
Radar	512x10x4	Paragon	None	.041	.043	4.9	23.4
			50	.041	.043	4.9	70.2
Radar	512x10x4	iWarp	None	.049	.049	0.0	20.3
			50	.049	.049	0.0	60.9
Stereo	256x240	Paragon	None	.275	.275	0.0	3.64
			12	.486	.514	5.7	11.67
Stereo	256x240	iWarp	None	.115	.115	0.0	8.7
			12	.152	.149	-2.3	13.4

Table 1: Performance results

2. A small change in the number of processors can result in a dramatic change in the cache behavior. This is one of the reasons why the predictions on the iWarp, which has a flat memory, were generally more accurate than the predictions on the Paragon.

3. The communication cost between two modules is often significantly lower when the number of processors in the sender and the receiver are the same, or one is a multiple of the other.

More sophisticated modeling of execution can certainly improve the accuracy of the predictions, but further discussion is beyond the scope of this paper. The algorithms presented in this paper are independent of how the execution behavior is modeled, and therefore can be used with other modeling approaches.

7 Conclusions

This paper offers a solution to the problem of finding processor mappings to satisfy the latency and throughput requirements of data parallel pipelines. We have presented an algorithm for minimizing the latency of a chain of data parallel tasks in the presence of a minimum throughput constraint, and discussed how it can be used to solve the converse problem of maximizing the throughput with a maximum latency constraint. The algorithm addresses the general mapping problem, which includes processor assignment, clustering, and replication, and is independent of the details of the computation and communication modeling.

This research is directed toward applications that process a stream of data sets from a physical sensor like a camera, where each data set is relatively small but parallel computing is needed to achieve the desired latency and throughput. Data parallel computing is generally not an adequate solution since small data sets may not execute efficiently on a large number of processors. We have implemented the ideas in this paper as a mapping tool that finds the optimal task and data parallel mapping for such applications. The mapping tool is integrated with the Fx parallelizing compiler for a subset of High Performance Fortran. We have demonstrated with examples that this tool chain can be used effectively to parallelize these applications and to tune them to their performance goals.

8 Acknowledgements

We would like to thank the Fx compiler team at Carnegie Mellon, in particular Peter Dinda, James Stichnoth, Tom Stricker and Bwolen Yang, for building and supporting the compiler infrastructure used in our implementation. Thomas Gross and David O'Hallaron participated in several fruitful discussions about this research.

References

[1] Bokhari, S. *Assignment Problems in Parallel and Distributed Computing.* Kluwer Academic Publishers, 1987.

[2] Chakrabarti, S., Demmel, J., and Yelick, K. Modeling the benefits of mixed data and task parallelism. In *Seventh Annual ACM Symposium on Parallel Algorithms and Architectures* (Santa Barbara, CA, July 1995).

[3] Chandy, M., Foster, I., Kennedy, K., Koelbel, C., and Tseng, C. Integrated support for task and data parallelism. *International Journal of Supercomputer Applications 8*, 2 (1994), 80–98.

[4] Chapman, B., Mehrotra, P., Van Rosendale, J., and Zima, H. A software architecture for multidisciplinary applications: Integrating task and data parallelism. Tech. Rep. 94-18, ICASE, NASA Langley Research Center, Hampton, VA, Mar. 1994.

[5] Choudhary, A., Narahari, B., Nicol, D., and Simha, R. Optimal processor assignment for a class of pipelined computations. *IEEE Transactions on Parallel and Distributed Systems 5*, 4 (April 94), 439–445.

[6] Crowl, L., Crovella, M., LeBlanc, T., and Scott, M. The advantages of multiple parallelizations in combinatorial search. *Journal of Parallel and Distributed Computing 21* (1994), 110–123.

[7] Dinda, P., Gross, T., O'Hallaron, D., Segall, E., Stichnoth, J., Subhlok, J., Webb, J., and Yang, B. The CMU task parallel program suite. Tech. Rep. CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, Mar. 1994.

[8] Foster, I., Avalani, B., Choudhary, A., and Xu, M. A compilation system that integrates High Performance Fortran and Fortran M. In *Proceeding of 1994 Scalable High Performance Computing Conference* (Knoxville, TN, October 1994), pp. 293–300.

[9] Gross, T., O'Hallaron, D., and Subhlok, J. Task parallelism in a High Performance Fortran framework. *IEEE Parallel & Distributed Technology*, 3 (1994), 16–26.

[10] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0*, May 1993.

[11] Ramaswamy, S., Sapatnekar, S., and Banerjee, P. A convex programming approach for exploiting data and functional parallelism. In *Proceedings of the 1994 International Conference on Parallel Processing* (St Charles, IL, August 1994), vol. 2, pp. 116–125.

[12] Sarkar, V. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* The MIT Press, Cambridge, MA, 1989.

[13] Subhlok, J., O'Hallaron, D., Gross, T., Dinda, P., and Webb, J. Communication and memory requirements as the basis for mapping task and data parallel programs. In *Supercomputing '94* (Washington, DC, November 1994), pp. 330–339.

[14] Subhlok, J., Stichnoth, J., O'Hallaron, D., and Gross, T. Exploiting task and data parallelism on a multicomputer. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, CA, May 1993), pp. 13–22.

[15] Subhlok, J., and Vondran, G. Optimal mapping of sequences of data parallel tasks. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Santa Barbara, CA, July 1995), pp. 134–143.

[16] Vondran, G. Optimization of latency, throughput and processors for pipelines of data parallel tasks. Master's thesis, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, May 1995.

[17] Webb, J. Latency and bandwidth consideration in parallel robotics image processing. In *Supercomputing '93* (Portland, OR, Nov. 1993), pp. 230–239.

[18] Yang, B., Webb, J., Stichnoth, J., O'Hallaron, D., and Gross, T. Do&merge: Integrating parallel loops and reductions. In *Sixth Annual Workshop on Languages and Compilers for Parallel Computing* (Portland, Oregon, Aug 1993).

[19] Yang, T. *Scheduling and Code Generation for Parallel Architectures.* PhD thesis, Rutgers University, May 1993.