Appears in the Proceedings of The Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, July 1995, Santa Barbara, CA

Optimal Mapping of Sequences of Data Parallel Tasks

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

Gary Vondran
Advanced LaserJet Operation
Hewlett Packard Company
Boise, ID 83714
vondran@boi.hp.com

Abstract

Many applications in a variety of domains including digital signal processing, image processing, and computer vision are composed of a sequence of tasks that act on a stream of input data sets in a pipelined manner. Recent research has established that these applications are best mapped to a massively parallel machine by dividing the tasks into modules and assigning a subset of the available processors to each module. This paper addresses the problem of optimally mapping such applications onto a massively parallel machine. We formulate the problem of optimizing throughput in task pipelines and present two new solution algorithms. The formulation uses a general and realistic model for inter-task communication, takes memory constraints into account, and addresses the entire problem of mapping which includes clustering tasks into modules, assignment of processors to modules, and possible replication of modules. The first algorithm is based on dynamic programming and finds the optimal mapping of k tasks onto P processors in $O(P^4k^2)$ time. We also present a heuristic algorithm that is linear in the number of processors and establish with theoretical and practical results that the solutions obtained are optimal in practical situations. The entire framework is implemented as an automatic mapping tool for the Fx parallelizing compiler for High Performance Fortran. We present experimental results that demonstrate the importance of choosing a good mapping and show that the methods presented yield efficient mappings and predict optimal performance accurately.

1 Introduction

A fundamental problem in parallel computing is how to map a parallel program onto the processors of a parallel machine. In data parallel computing, all available processors combine to execute a computation step. Complex applications are often composed of multiple tasks, where each task is data parallel and can execute on multiple processors, while different tasks may execute concurrently on different groups of processors on a parallel machine. 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 [2, 3, 7, 8, 13]. Recent research has also examined the tradeoffs involved in mapping task and data parallel programs [5, 12]. In this paper we address the problem of optimizing the throughput of task and data parallel programs.

We address 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, performs its computation, and sends the output to its successor. The first task reads external input and the last task generates the final output. This is a relatively simple model of task and data parallel computing. However, a large class of real applications in computer vision, image processing, and signal processing conform to this model [6]. For example, in multibaseline stereo program [15] 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 1. Figure 1(a) shows a pure data parallel mapping where all tasks execute on all processors. Figure 1(b) shows a task parallel mapping where a subset of processors is dedicated to each task. It may be possible to have multiple replicated copies of the data parallel mapping executing on different sets of processors, as shown in Figure 1(c). Finally, a mix of task and data parallelism with replication is shown in Figure 1(d).

The fundamental question addressed in this paper is: "Which of the many feasible mappings with data and task parallelism is optimal, and how can this be determined automatically?" Our criterion for optimality is maximum throughput

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.

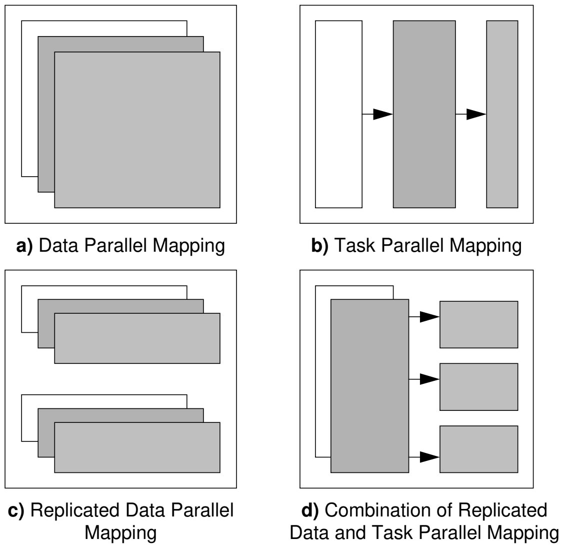

Figure 1: Combinations of data and task parallel mappings

measured in data sets per second. If all the tasks in an application as well as communication steps between them were perfectly scalable, a simple data parallel mapping would be optimal and demonstrate a perfect speedup. In practice, it is rarely the case. Typically processor efficiency decreases as more processors are used for executing a task. The challenge of finding an optimal mapping is to partition the processors such that the tasks use the processors efficiently while minimizing the cost associated with data movement between tasks.

Our basic approach is to execute the user program with different mappings to automatically infer how the time spent in execution of tasks and communication between tasks varies with the number of processors, and use these characteristics to predict an optimal mapping for the program. Our methodology can be the basis for a feedback driven compile time, or a runtime tool. It is applicable to programs whose execution behavior is not strongly data dependent.

A large volume of literature exists on mapping and scheduling parallel programs, and some excellent references are [1, 11, 17]. We focus only on mapping coarse grain function level task parallelism and assume that a data parallel compiler is available to exploit fine grain parallelism. We compare our approach to some other research efforts that address assignment of processors to coarse grain tasks.

Choudhary et. al. [4] address the problem of optimal processor assignment to a pipeline of coarse grain tasks but assume no communication cost (or that the communication cost can be folded into computation cost). In our experience, a realistic model for communication is very important for a practical automatic mapping system. Ramaswamy et. al. [10] use convex programming to address the problem of minimizing completion time for one data set (latency), but do not address the optimization of throughput for a sequence of data sets. Also, our algorithms are not tied to a particular execution model as is necessary when using a mathematical programming technique like convex programming.

The main contributions of this paper are, first, a solution to the processor assignment problem where the communication cost can be any function of the number of processors in the sending and receiving tasks, and second, a combined solution to the processor assignment, clustering and replication problems. We present an optimal algorithm and also show that the problem can be solved effectively with a fast heuristic algorithm.

The results presented in this paper are part of an automatic mapping tool that is integrated with the Fx compiler, and have been used to map several applications including multibaseline stereo and narrowband tracking radar [6]. Fx is a subset High Performance Fortran [9] compiler with support for task parallelism [13, 16]. The targets for Fx are the Intel Paragon, Intel iWarp, IBM SP2, 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 problem and section 4 presents a fast greedy solution. Section 5 shows how we estimate execution behavior. Section 6 illustrates the algorithms and their importance with examples and discusses some practical issues in automatic mapping. Section 7 contains conclusions.

2 Problem statement

2.1 Execution model

Consider a program consisting of a sequence of tasks $t_1, t_2, t_3, \ldots t_k$ 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. All tasks can execute on multiple 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:

$$f_1^{exec}, f_2^{exec}, f_3^{exec}, \ldots, f_k^{exec}$$

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:

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

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:

$$f^{ecom}_{1\rightarrow 2}, f^{ecom}_{2\rightarrow 3}, f^{ecom}_{3\rightarrow 4}, \ldots, f^{ecom}_{(k-1)\rightarrow k}$$

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

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

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. Thus, the execution of the chain is modeled as shown in Figure 2. Note that the task chain acts on a long input stream and different tasks can execute in parallel by processing different data sets in the stream.

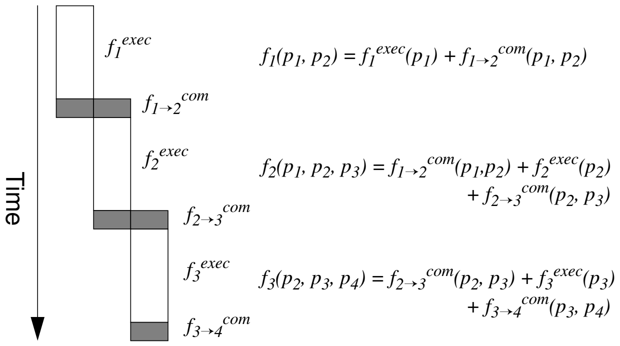

Figure 2: Execution model of a chain of tasks

An important underlying assumption in this model is that the execution and communication times are static functions of the relevant numbers of processors. Clearly the model is not applicable to programs whose execution behavior strongly data dependent. We discovered that other factors like processor locations and interference with external communication are a second order effect even for communication intensive programs.

2.2 The mapping problem

The throughput of a task pipeline is the rate at which the data sets are processed (data sets per second). The problem we are addressing is to find a *mapping* of the tasks onto a given fixed number processors that maximizes the throughput. For the purpose of finding a good algorithm, we assume that the number of processors P is much larger than the number of tasks k, which is generally the case when tasks are data parallel.

Processor allocation problem

If we assume that each task must execute exclusively on a subset of 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, such that the throughput is maximized. For a valid assignment of P processors to k tasks:

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

Clustering into modules

In general, tasks need not execute on their own exclusive subset of processors. A pair of adjacent tasks in a chain may execute on the same set of processors. We assume that any two tasks execute on either the same or disjoint sets of processors*. For the purpose of mapping, subsequences of tasks are clustered into modules and each module is mapped to an exclusive subset of processors. A *clustering* of a sequence of tasks is a list of modules $m_1, m_2, m_3, \ldots m_l$, where each module contains a subsequence of tasks $t_i, t_{i+1}, t_{i+2}, \ldots t_j$ and every task belongs to exactly one module.

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 groups and process alternate data sets on distinct sets of processors, as shown in Figure 3. While the response time for each data set will increase, the total throughput is also likely to increase since multiple data sets are processed in parallel. The legality of replication depends on data dependence constraints whose discussion is beyond the scope of this paper. We will assume that the tasks are known to be replicable or non-replicable. Only modules composed exclusively of replicable tasks are replicable. Further, each instance of a module requires a minimum number of processors to execute which limits the degree of replication. The number of instances of each replicable module is an important aspect of mapping.

A mapping is a result of clustering, replication and processor allocation decisions. The mapping of a chain of tasks $t_1, t_2, t_3, \ldots, t_k$ can be expressed as a list of modules M. $M(i)$ is a triplet (T, r, p), where T is the subsequence of tasks that are clustered into the 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 functions of the modules can be composed from the corresponding functions of the tasks that constitute the module. The throughput of such a mapping is determined by the slowest, or the *bottleneck*

*It can be shown that permitting allocation of tasks to partially intersecting sets of processor does not improve the best possible performance.

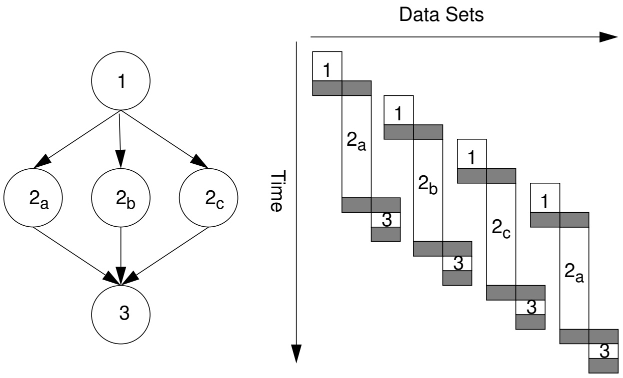

Figure 3: Replication

module, and is:

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

where $f_1, f_2, f_3, \ldots, f_l$ represent the response time functions for the chain of modules, and f_i is replicated r_i times.

3 Dynamic programming solution

We present an algorithm to find the mapping for a chain of tasks that corresponds to maximum throughput. We first present an algorithm to solve the processor assignment problem and then outline how the algorithm is extended to include replication and clustering.

3.1 Optimal processor assignment

If the communication time between tasks is negligible as compared to the computation time then the response time of each task is purely a function of the number of processors allocated to that task. In this case, the optimal throughput problem is solved in $O(Pk)$ time by assigning the processors one at a time to the slowest task until all processors are used. However, this approach is not optimal when the communication time is a function of the number of processors in the sending and the receiving tasks, since adding a processor to a task also affects other tasks.

We begin by presenting a dynamic programming algorithm to find an optimal assignment of processors to tasks without regard to clustering or replication. We assume that each task must execute on an exclusive set of processors, and that there is no replication. This implies that every task corresponds to a unique module, so we will refer to tasks and not modules. For simplicity of presentation, we also assume that a task can execute on any number of processors and that the optimal assignment uses all available processors. In practice, a task may require a certain minimum number of processors to execute and the optimal assignment may not use all available processors. These scenarios can be taken into account without a significant change to the algorithm.

Let T represent a chain of k tasks labeled $t_1, t_2, \ldots, t_k$ and T_j represents the subchain $t_1, t_2, \ldots, t_j$ of the chain T. Let P be the number of available processors. Let A represent an assignment of processors to a subchain of tasks, i.e. A(i) equals number of processors assigned to task t_i. An assignment A can be appended with a new value to obtain an assignment for the new task subchain, e.g. $A_{j+1} = A_j \oplus \{(A(j+1) = p_{j+1})\}$

Let F be a function that returns the throughput for an assignment A to a task subchain T_j, *not* including the last task t_j if a task t_{j+1} exists (since the response time for t_j depends on the number of processors assigned to the next task t_{j+1} which is not a part of A). If t_i is the task with the longest response time f_i, F is simply $1/f_i$.

It is easily seen that the optimal processor assignment to T_j is defined only for a fixed number of processors p_{j+1} for t_{j+1}. Suppose A represents the optimal assignment to the subchain T_j, given that t_{j+1} is assigned p_{j+1} processors. However A may not be the assignment to T_j in the global optimal mapping for the full chain T even if t_{j+1} is assigned p_{j+1} processors, unless it is also known that t_j, the last task in the subchain T_j, is assigned the same number of processors p_j in the global optimal mapping as in the mapping $A^{\ddagger}$. Hence A represents a part of the global optimal mapping only under the assumption that the last task t_j and the next task t_{j+1} are assigned p_j and p_{j+1} processors respectively, in the global optimal mapping. We have the following result:

Lemma 1 Suppose the function $\mathcal{A}_j(p_{total}, p_{last}, p_{next})$ returns the optimal assignment of p_{total} processors to a subchain of j tasks, given that the last task in the subchain is assigned p_{last} processors, and 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, p_j, and p_{j+1} processors assigned to T_j, t_j, and t_{j+1} respectively, then the assignment of processors to T_j in this optimal mapping is that returned by $\mathcal{A}_j(p, p_j, p_{j+1})$. The lemma is illustrated in Figure 4.

The optimal assignment and the corresponding maximum throughput is obtained directly from $\mathcal{A}_k$ for the chain of k tasks:

$$\begin{aligned} A^{opt} &= A \equiv \max_{q=1}^{P} F(\mathcal{A}_k(P, q, \phi)) \\ F^{opt} &= F(A^{opt}) \end{aligned}$$

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

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

‡Another mapping may assign a different number of processors to the last task t_j, say p'_j that leads to a lower response time for t_{j+1}, resulting in an optimal global throughput for the full chain T which is less than or equal to the throughput represented by A.

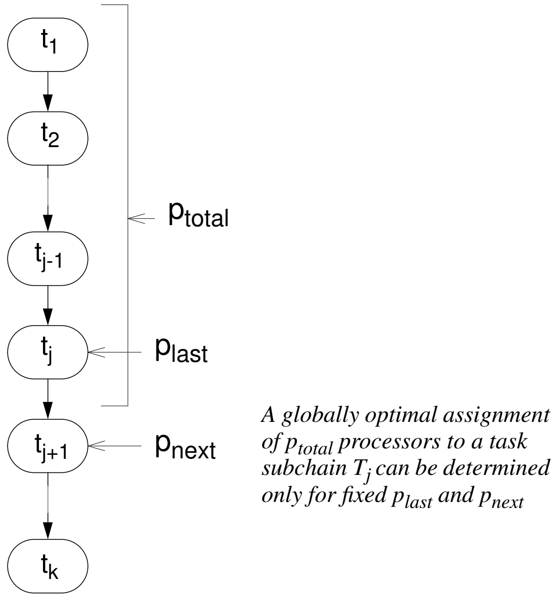

Figure 4: Processor assignment with dynamic programming

We explain the meaning of the above equation and describe how it is used to compute the processor assignment for maximum throughput. Suppose we have a table defining $\mathcal{A}_{j-1}$ and the corresponding maximum throughput $F(\mathcal{A}_{j-1})$, i.e. for every argument set, we have the optimal assignment and the corresponding maximum throughput. To determine an entry in the table for $\mathcal{A}_j$, say $\mathcal{A}_j(p_{total}, p_{last}, p_{next})$ we need to find the optimal throughput assignment for T_j, given that the processors assigned to t_j, t_{j+1}, and available processors for assignment to subchain T_j, are p_{last}, p_{next} and p_{total}, respectively. If we assume that q processors are assigned to t_{j-1}, the optimal assignment to T_j can be looked up from the table for $\mathcal{A}_{j-1}$. By comparing the throughput of the optimal assignments for all possible values of q, we obtain the optimal assignment and throughput entry for $\mathcal{A}_j(p_{total}, p_{last}, p_{next})$. We compute $\mathcal{A}_k$ applying the above equation k times and obtain the optimal assignment and throughput for the chain of tasks.
Complexity: The size of a definition table for $\mathcal{A}$ is at most $O(P^3)$. 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 $O(P^4)$ operations. Since there are k such steps, the total complexity of the algorithm is $O(P^4 k)$ operations.

3.2 Optimal assignment with replication

Each task t_j requires a known minimum number of processors p_j^{min} to execute due to memory constraints. We assume that the execution and communication functions do not exhibit superlinear behavior, i.e. if a processor is added to a task executing on k processors, the decrease in communication and execution time is by a factor of at most $1/(k+1)$. Under this assumption, it is always profitable to replicate maximally subject to memory constraints, i.e. if p processors are available for a task, then the task is replicated into $\lfloor p/p^{min} \rfloor$ instances and the processors divided equally among the instances.

In this scenario, the *effective* number of processors used for computation and communication is equal to the number of processors in each instance, which is a known function of the total number of processors. We can still use the dynamic programming algorithm from the last subsection using effective number of processors instead of total number, and using *effective response time* for throughput computation, where effective response time is $f_i(p)/r$ for a task with r instances and p processors per instance.

3.3 Optimal assignment with clustering

We outline the changes in the dynamic programming algorithm to take clustering into account. The mapping of a chain of tasks now includes clustering task sequences into modules. It is assumed that the communication and computation characteristics of a module obtained by combining two tasks or modules can be computed in $O(1)$ time from the characteristics of the constituent tasks or modules. Also the memory requirement of each module and the minimum number of processors needed to execute it are assumed to be known. These hold for our execution and memory models which are presented in section 5.

Given a chain of tasks T, consider the optimal mapping for subchain T_j including clustering for the subchain. As previously discussed, an optimal mapping is defined only for a given fixed number of processors p_{j+1} assigned to t_{j+1}. To determine an optimal mapping for T_j, we have to decide if t_j and t_{j+1} are clustered in the same module. However, this decision can be made only if it is assumed which other tasks after t_{j+1} are clustered with it, since the minimum number of processors required to meet the memory requirements of a module is known only if all the tasks belonging to the module are known.

The dynamic programming solution is similar to that discussed earlier, except that we work with the mapping function $\mathcal{M}_j(p_{total}, p_{last}, p_{next}, nextmodlength)$ instead of $\mathcal{A}_j(p_{total}, p_{last}, p_{next})$, where *nextmodlength* is the length of the subchain after t_{j+1} which is assumed to be clustered with t_{j+1}. Clearly $nextmodlength < k$, the number of tasks. The computation complexity is increased by a multiplicative factor k. Hence we have the following result:

Lemma 2 *The optimal mapping of a chain of k tasks onto P processors can be determined in $O(P^4 k^2)$ time.*

4 A fast heuristic algorithm

The dynamic programming algorithm discussed in the last section determines the optimal mapping for a chain of k tasks on P processors in $O(P^4k^2)$ steps. This computation cost can be unacceptably high when the number of processors is large, particularly when mapping tasks dynamically. In this section we present a simple algorithm that is effective in computing good mappings in practice. We argue that the algorithm generally produces optimal or near optimal mappings, and prove that it produces optimal mappings under a set of realistic assumptions.

We present some observations about the three main interdependent decisions that have to be made, i.e. processor assignment, replication, and clustering. Processor assignment and degree of replication are detailed quantitative decisions, while clustering is a coarse decision. Typically mappings corresponding to optimal or near optimal throughput have the same clustering. Hence it is generally possible to predict optimal clustering with an approximate notion of replication and processor assignment.

We first do an approximate mapping with clustering, replication and processor assignment. We assume that the clustering obtained is optimal and decide the processor assignment and replication in a second refinement phase.

4.1 Greedy processor assignment algorithm

We first present an algorithm for assignment of processors to tasks without regard to replication and clustering, i.e. it is assumed that clustering has been determined and replication is not permissible. Since the tasks and modules are the same in this case, we will directly assign processors to tasks.

Procedure: Greedy(T, P)
Input: T, a chain of k tasks labeled $t_1, t_2, \ldots, t_k$, P available processors, and the computation and communication time functions (and thus the maximum throughput function F for all subchains) of the tasks.
Output: An assignment A of processors to tasks that maximizes throughput.

1. For each task, assign the minimum number of processors that are necessary for its execution. Let A represent this assignment and let $A^{opt} = A$.
2. While *available processors > 0* do step 3.
3. Determine the task with the longest response time, say t_i. Let F_{i-1}, F_i, and F_{i+1} be the throughput values for the new assignment if a processor was added to t_{i-1}, t_i, and t_{i+1}, respectively (The throughput value is assigned zero if the corresponding predecessor or successor task does not exist). Assign a processor corresponding to $max(F_{i-1}, F_i, F_{i+1})$, giving the new assignment A. If the throughput of A is greater than A^{opt}, then assign $A^{opt} = A$.
4. A^{opt} is the best assignment.

End procedure
Complexity: Step 3 takes $O(k)$ time since it has to compare the response time of all k tasks, and can be invoked at most P times. Step 1 also takes $O(k)$ time and is invoked just once. Hence the computation complexity is $O(Pk)$.

We present the reasoning behind this algorithm and discuss why the results may not be optimal. Let A_p be the assignment generated by the algorithm for p processors. In the next step, the slowest task t_i is identified. Starting from the assignment A_p, addition of another processor can improve throughput if and only if the response time of task t_i is decreased. This can be potentially achieved by adding a processor to task t_i, its predecessor t_{i-1} or successor t_{i+1}. We select the task that leads to maximal improvement in throughput and assign another processor to it. Note that the predecessor and successor tasks have to be considered since the response time includes communication time which depends on the number of processors assigned to them.

The assignment A_{p+1} is optimal only if we assume that processors already assigned in A_p are part of the optimal assignment. However, in general this may not be true even if A_p is the optimal assignment for p processors. For example, optimal assignment A_p may have p_i processors assigned to t_i, but optimal assignment A_{p+1} may have only $p_i - 1$ processors assigned to t_i with t_i having a lower response time due to a different number of processors assigned to tasks t_{i-1} and/or t_{i+1}. Thus the greedy algorithm is not guaranteed to produce the optimal assignment.

Intuitively the reason for potentially reaching a non-optimal distribution is the *bad* behavior of communication and/or computation time functions. As an extreme example, assigning 2 to 9 processors to a task may have no impact on the communication and computation time, but adding a 10th processor may dramatically improve both. Hence, an optimal throughput configuration will assign either 1, or at least 10 processors. Since the greedy algorithm adds one processor at a time, it cannot go from a configuration with 1 processor to one with 10 processors. However, in our experience the computation and communication functions are generally well behaved, and the assignments reached are typically either optimal or close to optimal. In many practical situations, the computation time is much larger than the communication time which also makes it more likely that the algorithm will yield an optimal result (if the communication cost is ignored, the results obtained are provably optimal). An option is to add backtracking. A full backtracking algorithm is exponential in the number of processors and can be prohibitively expensive, but limited backtracking is of practical interest.

We present two sets of additional assumptions that lead to provably optimal assignments.

Theorem 1 A modified greedy algorithm in which the processors are always added to the slowest task (i.e. not to neighbors) is optimal if the communication time increases monotonically with the number of processors involved in a communication step, i.e. $f^{com}(i, j) < f^{com}(i + x, j + y)$ for $x, y >= 0$.

The proof is based on the following observation. Let A_p be the optimal assignment with p processors, and let t_i be the slowest task in A_p. The only place to add a processor to potentially decrease the response time of t_i is to t_i itself, since adding to the neighbors can only increase the response time. This implies that the algorithm builds the optimal assignment incrementally.

When processors are added to a task, the overhead of sending and receiving messages increases but parallelism in communication also increases. Theorem 1 holds when the former dominates, which is often the case in practical systems where the software overhead and not the communication bandwidth dominates communication behavior.

Theorem 2 The greedy algorithm can overallocate at most two processors to a task compared to the optimal assignment, if the following conditions are true:

1. All computation and communication functions are convex, i.e. the decrease (positive or negative) in the computation and communication time on adding a processor to an assignment A is less than the decrease corresponding to any processor addition used to reach the assignment A.

2. For any assignment A of p processors, if δ is the decrease in computation time of a task t_i on adding a processor and δ_c is the decrease in its communication time on adding one processor to itself or to its neighbor, then $\delta > 4 * \delta_c$.

The proof (by induction) is omitted here for brevity, and is included in [14]. Using this theorem, a post processing backtracking algorithm will yield the optimal assignment. The overall complexity of such an algorithm is ($O(Pk + 2^k)$). The algorithm is still linear in the number of processors and reasonable if k is very small.

The first condition of the theorem is generally (but certainly not always) true, since the effect of adding one processor on response time decreases as more processors are assigned due to the nature of parallelism. The second condition implies that the computation inside tasks dominates inter-task communication.

While the conditions for the two theorems may be difficult to verify, and indeed not be true, the general conclusion is that the greedy algorithm yields optimal or near optimal assignments in practice, and the accuracy can be improved by relatively inexpensive limited backtracking. This is verified by our experience with programs discussed in Section 6.

4.2 Replication and clustering

We use a modified response time function to capture the effect of replication as discussed in Section 4 and will not discuss replication any further.

Clustering is a coarse decision which is dependent on how many processors are assigned to different tasks. We run the greedy processor assignment algorithm one time to determine the clustering. We start with a clustering in which every task is assigned to a single module, compare pairs of tasks (or modules) at every step to check if they should be merged, and then check if the merged tasks should be separated. Given a pair of tasks and the number of processors assigned to the pair, it can be determined whether merging them will improve performance. At the end of this phase, we get a clustering which is usually (but not provably) optimal. We re-execute the algorithm with this clustering to obtain the final processor assignment and replication [§].

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 computations, 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 receiving tasks (p_s and p_r) as follows:

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

The 1st constant term reflects fixed overheads of communication, the 2nd and 3rd terms reflect the part of communication cost that decreases with increasing number processors due to parallelism, and the last two terms reflect overhead that increase as more processors are added.

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

[§] If k is very small, it is also feasible to exhaustively test all clusterings terms.

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

The memory requirements are modeled by separately measuring the memory used for global and system variables, local variables, and compiler buffers. All the parameters of this model can be computed using 8 executions.

6 Results

The execution model and the mapping algorithms presented in this paper are implemented as an automatic mapping tool in the Fx compiler. We will illustrate the mapping procedure with an example program that was mapped onto a 64 processor iWarp computer using the Fx compiler. Results from this program and other applications are presented to evaluate the mapping algorithms. We first discuss some practical considerations in using the mapping algorithms.

6.1 Compiler and machine constraints

The algorithms presented in this paper assume that any number of processors can be allocated to any module. However, the Fx compiler allows only a rectangular subarray of processors to be mapped to a module making some assignments of processors to modules infeasible. Even when all processor assignments can be individually mapped to rectangular processor subarrays, it may not be possible to map all the modules due to geometrical constraints. The architecture of a parallel machine can add additional constraints on mapping. In some of our examples, we used *systolic* mode communication of iWarp where communicating modules are connected by logical pathways. This caused some mappings to be infeasible because of a limit on the number of pathways that can pass through a physical communication link.

In our discussion of results, we will point out the situations where machine constraints were a problem, and how they were addressed. In some cases we modified the mapping algorithms to include machine constraints but a discussion is beyond the scope of this paper.

6.2 Example Program: FFT-Hist

We use FFT-Hist as an example program to illustrate and validate our algorithms and mapping tool. This 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 5. 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
```

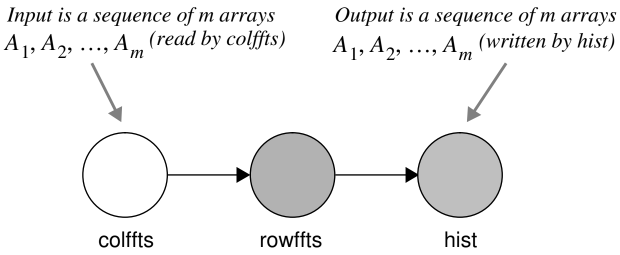

Figure 5: FFT-Hist example program and task graph

6.3 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 for two communication models that were available to us (message passing and systolic). 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. We checked the accuracy of the model by comparing the predicted and actual communication and computation times for a set of mappings and the difference averaged less than 10%. We expect an error of this order since we model a wide range of computation and communication behavior using a small number (eight) of executions; it is certainly possible to develop a more accurate model that uses a larger number of executions.

The mapping tool then used the the algorithms presented in Sections 3 and 4 to predict the optimal mapping. A key result is that for all cases the dynamic programming and the greedy algorithms reached the same optimal mapping. Since the results from dynamic programming are provably optimal, this supports our claim that the greedy algorithm is a good heuristic algorithm.

The optimal mapping obtained for the 256×256 FFT-Hist using message passing communication is shown in Figure 6. The optimal mapping has two modules, *module 1* contains `colffts` and *module 2* contains `rowffts` and `hist`. *Modules 1* is replicated to 8 instances and each instance is assigned 3 processors. *Module 2* has 10 instances and 4 processors per instance.

We outline the reasoning behind this mapping. `rowffts` and `hist` use the same distributions for the data that is transferred between them, hence merging them together eliminates the data transfer cost. Combining `colffts` with this module would not reduce communication cost significantly since there is a transpose step between `colffts` and `rowffts`, whose cost is comparable whether they are mapped together or separately. Merging `colffts` with the module containing `rowffts` and `hist` also has the disadvantage that total memory requirement for the combined module is higher, which implies that a larger number of processors are required to execute each instance of the module. Since `hist` has significant communication, using a larger number of processors causes it to run inefficiently, leading to a reduction in total throughput. To satisfy the memory requirements, each instance of *Module 1* and *Module 2* must be assigned at least 3 and 4 processors respectively. Both modules are replicated maximally subject to this constraint.

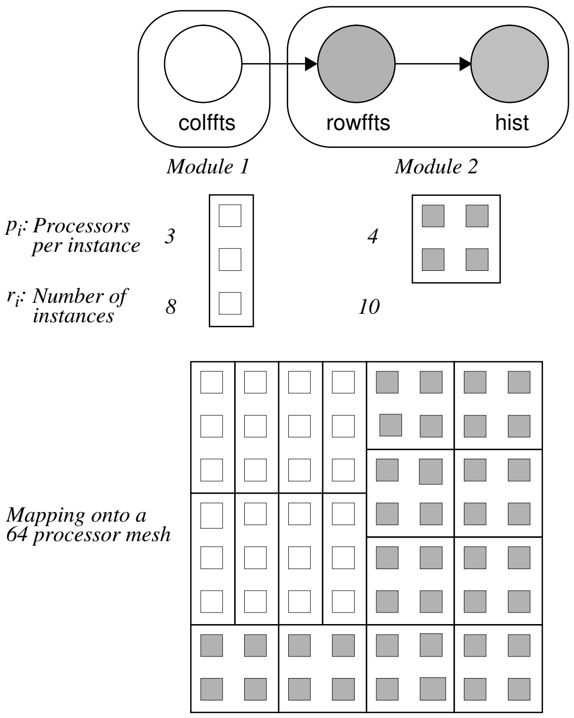

Figure 6: FFT-Hist program mapping(256, Message)

Table 1 shows the optimal mappings for the four versions of FFT-Hist predicted by the mapping algorithm. In all cases, *module 1* contains `colffts` and *module 2* contains `rowffts` and `hist`, but the number of processors assigned to each instance of the module (p_i) and the number of replicated module instances (r_i) vary. The table also shows the feasible optimal mappings computed with the constraint that every module is rectangular as required by the Fx compiler. The mappings differ in one case with a small decrease in predicted throughput. In rest of this section we will refer to the optimal mapping subject to this constraint as the optimal mapping.

6.4 Evaluating the mapping algorithms

We have used the mapping tool and the Fx compiler to generate mappings for several task and data parallel programs. In Table 2 we present results from FFT-Hist and two other applications; narrowband tracking radar and multibaseline stereo [6]. The properties of the latter two programs that influence their mapping are discussed in [12].

Table 2 compares the predicted optimal throughput, corresponding measured throughput, and the measured throughput for a simple data parallel mapping. Even when all modules are rectangular, it may not be possible to map all of them together due to machine or compiler constraints discussed earlier. In these cases (marked in Table 2) we used a smaller number of instances of one or more modules for execution, and extrapolated the results for the purpose of meaningfully validating predicted optimal results, but did not change the mapping in any other way.

We experimentally verified that the optimal mapping predicted by the algorithms was in fact superior to all the others that we tried, and Table 2 shows that the predicted optimal throughput and measured optimal throughput were between 0 to 12 percent of each other. This validates the claim that the algorithms are effective in finding the optimal mapping and predicting the optimal throughput. The differences between predicted and measured throughput are due to inaccuracies in our modeling of performance parameters, and second order effects like interference between communication inside tasks and communication between tasks, which are not considered. The table also shows that the optimal mapping outperforms the pure data parallel mapping by a factor of 2 to 9 for our examples. This underlines the importance of finding a good mapping and shows that the inaccuracies in predicting an optimal mapping for a practical system are small as compared to the benefits that are obtained by choosing a good mapping.

7 Conclusions

We have addressed the problem of mapping a sequence of data parallel tasks to maximize throughput. The main contributions are an optimal algorithm for this problem and a fast heuristic algorithm which is optimal for most practical purposes. The algorithms developed are independent of how the computation and communication are modeled. They are implemented in an automatic tool which is a part of the Fx parallelizing compiler for task and data parallel programs. The compiler and the tool have been used to develop a variety of parallel programs and we have demonstrated that the tool is effective in finding good mappings and predicting the best possible performance. We believe that this research makes a significant contribution towards efficiently compiling and mapping a large class of applications onto parallel computers.

Problem Description		Optimal Mapping					Optimal Feasible Mapping				
Size of Data set	Commun-ication	Module 1		Module 2		Throughput	Module 1		Module 2		Throughput
		p_1	r_1	p_2	r_2	data sets / sec	p_1	r_1	p_2	r_2	data sets / sec
256x256	Message	3	8	4	10	14.60	3	8	4	10	14.60
256x256	Systolic	3	6	4	11	14.74	3	6	4	11	14.74
512x512	Message	20	1	14	3	3.14	20	1	14	3	3.14
512x512	Systolic	12	2	13	3	2.99	12	2	12	3	2.83

Table 1: Optimal and Feasible Optimal Mappings for FFT-Hist

Problem Description			Optimal Throughput		Percent	Data Parallel	Ratio
Name of Program	Array Size	Commun-ication	Predicted data sets / sec	Measured data sets / sec	Difference %	Throughput data sets / sec	Optimal / Data Parallel
FFT-Hist	256x256	Message	14.60	16.28	+11.51	1.86	8.75
FFT-Hist†	256x256	Systolic	14.74	14.35	-2.65	1.86	7.72
FFT-Hist†	512x512	Message	3.14	2.93	-6.69	1.35	2.17
FFT-Hist†	512x512	Systolic	2.83	2.65	-6.36	1.35	1.96
Radar	512x10x4	Systolic	81.21	81.18	-0.03	18.95	4.28
Stereo	256x100	Systolic	43.12	43.15	+0.07	15.67	2.75

† Measured results extrapolated from execution with at least one less module instance.

Table 2: Performance Results

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] GROSS, T., O'HALLARON, D., AND SUBHLOK, J. Task parallelism in a High Performance Fortran framework. *IEEE Parallel & Distributed Technology*, 3 (1994), 16–26.

[9] HIGH PERFORMANCE FORTRAN FORUM. *High Performance Fortran Language Specification, Version 1.0*, May 1993.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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, 1995. In preparation.

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

[16] 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).

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