# Compile-time Partitioning and Scheduling of Parallel Programs

**Vivek Sarkar and John Hennessy**
**Computer Systems Laboratory**
**Stanford University**

### Abstract

Partitioning and scheduling techniques are necessary to implement parallel languages on multiprocessors. Multiprocessor performance is maximized when parallelism between tasks is optimally traded off with communication and synchronization overhead. We present compile-time partitioning and scheduling techniques to achieve this trade-off.

## 1. Introduction

One of the biggest challenges facing compiler writers is to efficiently implement programming languages on multiprocessors. We need to find compilation techniques for general-purpose parallel languages; these techniques should be adaptable to a wide range of multiprocessor architectures.

There are three fundamental problems to be solved when compiling a program for parallel execution on a multiprocessor:

1. Identifying potential parallelism.
2. Partitioning the program into sequential tasks.
3. Scheduling the concurrent execution of these tasks.

We address the latter two problems and suggest that they be solved at compile-time instead of run-time for applications with fairly predictable execution times. In these cases the benefits are tremendous. A global, compile-time analysis reduces communication overhead for the entire program; such an analysis cannot be done on the fly at run-time. Compile-time partitioning and scheduling also eliminates task scheduling overhead and load balancing overhead at run-time.

A compile-time partitioner-cum-scheduler has been implemented to process program graphs in the intermediate language, IF1 [21]. IF1 represents computation as dataflow graphs, as described later in Section 6. A list of target parameters (e.g. number of processors, communication overhead, inter-processor distances) drives the partitioning for a given multiprocessor architecture. Using a front-end from SISAL [15] to IF1, we apply this system to different benchmark programs written in the single-assignment language SISAL. Like other functional languages, SISAL is implicitly parallel; this eliminates the need to extract parallelism from the program. However, our approach is applicable to any environment where a program graph representation can be obtained.

## 2. Overview of our approach

Our approach is to expose enough parallelism in the "main program" function graph and then assign computations to processors so as to minimize the parallel execution time, while considering communication overhead. As shown in Figure 2-1, there are four basic steps in this process:

1. Cost Assignment: Traverse the program graph and assign execution time costs to nodes and communication size costs to edges.
2. Graph Expansion: Expand the graph so that the main function contains sufficient parallelism to keep all processors busy. Nodes in the expanded program graph are mapped to tasks.
3. Internalization: Decide which tasks must go together on the same processor, even if other processors are available. This internalizes their communication and eliminates its overhead, at the expense of sequentializing the tasks.
4. Processor Assignment: Assign tasks to processors so as to minimize parallel execution time. Tasks in the

This work has been supported by the National Science Foundation under grant # DCR8351269 and by the Defense Research Projects Agency under contract # MDA 903-83-C-0335.

same block of the internalized partition must be assigned to the same processor.

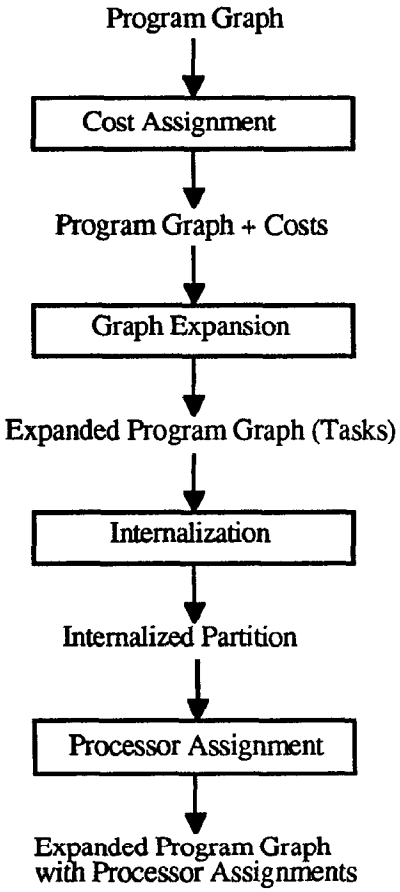


**Figure 2-1:** Overview

Note that we rely heavily on compile-time estimates of communication and computation costs. These costs drive the expansion, internalization and scheduling of parallel tasks. We have found that our cost estimates yield good partitions for a range of programs and input data.

These four phases for compile-time partitioning and scheduling are described in later sections. We begin with a discussion of multiprocessor models and real multiprocessors.

## 3. Multiprocessor Scheduling Theory

Let's start by examining the traditional model for multiprocessor scheduling, which consists of:

- P identical, independent processors.
- A set of N tasks, $T = \{T_1, ..., T_N\}$ with execution times $t_1, ..., t_N$.
- A partial order $\rightarrow$ on T.

The partial order describes precedence constraints on the tasks, so that $T_i \rightarrow T_j$ forces $T_j$ to start only after $T_i$ has completed execution.

The problem is to find a valid schedule with the smallest completion time. This problem (along with several restricted versions) has been shown to be NP-complete [14]. However, there exist efficient scheduling algorithms that generate schedules with a worst-case performance bound of 2, relative to an optimal schedule [11]. Thus the NP-completeness of the scheduling problem is not an impediment to achieving linear speed-up in multiprocessors.

This model is inadequate for our purpose because it ignores the overhead of inter-processor synchronization and communication. Most schemes for considering communication overhead do so by separately maximizing parallelism and minimizing communication. We believe that these parameters should be combined into a single objective function. The next section shows how we extend this model to incorporate communication costs.

## 4. Modelling Communication Costs

Communication costs are represented by a communication matrix. C[i,j] is the size of communication (in bytes, say) from task $T_i$ to $T_j$. For simplicity, we assume that communication only occurs along precedence edges, though all precedence edges need not have an associated communication. The data along a communication edge is only available to the consumer after the producer has completed execution. We also define an inter-processor distance matrix; D[k,l] gives the communication distance (number of hops) between processors k and l. This is a property of the multiprocessor rather than the tasks.

Communication overhead in a multiprocessor has two components:

1. *Processor component* - the duration for which a processor participates in its communication.
2. *Delay component* - the fraction of communication time during which the producer and consumer processors are free to execute other tasks.

Let $\pi(i)$ be the processor to which task $T_i$ is assigned. The communication overhead for each non-local edge from $T_i$ to $T_j$ (i.e. $\pi(i) \neq \pi(j)$) is modeled as:

1. *Processor component* - add $\omega(C[i,j], D[\pi(i),\pi(j)])$ to the execution time of task $T_i$, and $\rho(C[i,j], D[\pi(i),\pi(j)])$ to the execution time of $T_j$.
2. *Delay component* - require that $T_j$ not be able to start till $\delta(C[i,j], D[\pi(i),\pi(j)])$ time has elapsed after $T_i$ completed execution.

$\rho$, $\omega$ and $\sigma$ are simple functions for the costs of reading inputs, writing outputs and communication delay respectively.

They convert communication size to execution time units for the target multiprocessor. So far, we have used functions of the form $K_1 + K_2 \times C[i,j] \times D[\pi(i),\pi(j)]$, where $K_1$ and $K_2$ are constants.

We ignore the effect of communication demand on communication overhead in this model. This is valid when the demand is less than the available bandwidth. If contention for communication resources is to be considered, it can be approximated by using average overhead values.

Synchronization between tasks is modeled by a synchronization relation (boolean matrix) S. The pair <i,j> is in S if task $T_j$ must synchronize and wait for task $T_i$'s completion. It would be safe to make S the same as the precedence relation $\rightarrow$, but would be inefficient because $\rightarrow$ is a partial order and contains transitive edges which lead to unnecessary synchronizations. The communication matrix does not entirely determine the synchronization relation, because some synchronizations are due to control dependencies. Besides, the communication matrix can also have transitive edges. So, for efficiency and generality, we represent synchronization by a separate relation.

Each pair <i,j> in S adds to the execution times of tasks $T_i$ and $T_j$ due to *signal* and *wait* operations respectively. The execution cost used for a *wait* is actually the cost of a successful *wait*. Time spent spinning in a busy-wait loop is considered to be idle time and is not added to the task's execution time.

## 5. Real Multiprocessors

Multiprocessors are general-purpose, asynchronous, MIMD parallel machines. They can be classified as being "tightly coupled" or "loosely coupled". Tightly coupled multiprocessors (e.g. Sequent [19], Encore [8], Ultracomputer [10], Cm* [12]) communicate through a shared memory. Loosely coupled multiprocessors (e.g. the Cosmic Cube [18]) communicate by exchanging messages. Our model is designed to be tractable on this wide class of architectures and we discuss how some of these machines are modeled in our system.

Machines like the Sequent and Encore communicate through a single bus connected to a shared memory and individual processor caches. The communication overhead consists entirely of its processor component, since the processors are directly involved in accessing the shared memory. Also, the distance matrix is uniform, so that D[k,l] = 1 for all pairs of processors. Functions $\rho$ and $\omega$ for the processor component represent the time taken to respectively read from or write to the shared memory. Both machines use write-through caches and invalidating snoopy caches. Because they use invalidation, the value of $\rho$ is the cost of a cache miss (i.e. a main memory access - on the order of 5-10 cycles). Because writes are buffered and a write-through cache is used, writes will cost one cycle when there is no contention. Thus, the value of $\omega$ depends primarily on bus and main memory traffic. Bus contention increases the cost to read shared memory because cache misses take longer to satisfy; it increases the cost to write shared memory, because delays may cause write stalls.

Synchronization costs differ on these machines because they use different hardware mechanisms for synchronization. The Sequent has 64 hardware "gates" that can be used to ensure mutual exclusion, making synchronization cheap. The Encore has a test-and-set instruction. The difficulty in accurately modeling such machines is that the cost of various operations (reading and writing shared data and synchronization) depends on the amount of contention for shared resources.

A single bus architecture is feasible for only a small number of processors. Interconnection networks are used to support more processors in shared-memory machines like the Ultracomputer. Communication and synchronization overhead is modeled in basically the same way as single bus architectures. There are two important differences: the time to access shared memory increases with the processor count (because of the interconnection network delay), and the communication costs are less affected by contention.

Another important class of machines are those using point-to-point communication. These include the Cosmic Cube, the Intel Advanced Scientific Processor and the NCube machine. All of these machines use a boolean N-cube interconnect, which defines the distance matrix. Since these machines use a "message passing" approach we can easily model their communication properties. Assuming that communication contention is negligible, the following table summarizes the properties of the Cosmic Cube and Intel cube. The processor component is for the initiating processor and is given as X + Y, where X is the start-up and Y is the cost per 100 bytes, both in milliseconds. The delay component is in milliseconds per hop for each 100 bytes. Packetization introduces a nonlinearity in communication costs, but we ignore this effect.

| Machine | Processor Component | Delay Component |
|---|---|---|
| Cosmic Cube | 1.5 + 0.4 | 0.4 |
| Intel Cube | 6.0 + 0.08 | 0.08 |

As we have indicated, the primary limitation of this model is its inability to deal with contention for communication or

synchronization resources. As we measure more problems running on real machines, we believe that we can refine the model to realistically accommodate such issues.

Having discussed the target architectures, let's now examine the intermediate language used in our system.

## 6. IF1 Program Graphs

Our compilation system operates on a graphical representation of programs, namely IF1 [21]. IF1 is an intermediate form for applicative languages. It is strongly based on the features of single-assignment languages SISAL [15] and VAL [1].

An IF1 program is a hierarchy of acyclic dataflow graphs [7]; the nodes denote operations and the edges carry data. Nodes are either *simple* or *compound*. A simple node's outputs are direct functions of its inputs. IF1 has about 50 simple nodes, e.g. Plus, ArrayCatenate, FunctionCall. A compound node contains subgraphs and its outputs depend on the interaction between these subgraphs. The following table lists the five compound nodes available in IF1. These compound nodes obviate the need for labels, goto's and cycles in the program graph.

| Compound Node | Subgraphs |
|---|---|
| Select | Selector, Alternatives |
| TagCase | Alternatives (for Union) |
| Forall | Generator, Body, Results |
| While, Until | Init, Test, Body, Returns |

Nodes have numbered ports connected by edges. An edge contains the node and port numbers of its producer and consumer. It also contains an optional type number, which is used for strongly typed languages like SISAL. Literals are special edges used for constant values. A literal has no producer - its value is given by a string. All data is carried by edges. No variables or memory locations are used.

Basic types include boolean, character, integer, real and double. Arrays, streams, records and unions are used to construct more complex types. Arrays are dynamically extendible. Nodes and edges in IF1 can use pragmas to carry additional information. We use pragmas to store profile-based frequency counts, communication and computation costs, graph partitions and processor assignments.

This program graph representation is well suited for compile-time partitioning and scheduling. However, generation of sequential machine code is more complicated than from traditional, sequential intermediate languages. It is imperative to avoid unnecessary copying when an update-in-place is possible. This effectively coalesces data on input and output edges to be the same "variable". A few research projects are under way to address this problem. The SISAL [15] project includes code generation from IF1 for the VAX 780 and Cray-2 architectures. A project is under way at Stanford to translate SAL [6] graphs (similar to IF1) to U-code [20]. Our partitioner will benefit from all advances in this field, as sequential code generation and optimization techniques can be applied to intra-task computations.

## 7. Cost Assignment

The first step in compile-time partitioning and scheduling is to estimate computation and communication costs in the program. Communication costs are determined by examining the data type of an edge and assessing its size in an appropriate unit (e.g. bytes). Estimation of node execution times is more difficult and is undecidable in general. The unknown parameters are:

- The frequency distribution of subgraphs in a compound node (e.g. number of iterations for a While Body, probability distribution of Alternatives in a Select)
- Array size for nodes that operate on entire arrays.
- Recursion depth for recursive function calls.

Average node execution times are determined by using average values for these frequency parameters. These frequency values can be estimated using simple rules of thumb, can be provided by the programmer through pragmas, or can be derived from profile information. Our current implementation uses profile data.

Given these parameters, it is a straightforward task to compute the cost of a node from the cost of its components via a depth-first traversal of the program graph. The cost of a function call is determined by the cost assigned to the callee. The strongly connected components (SCC's) in the call graph reveal groups of mutually recursive functions. The recursion depth estimate is used to evaluate the costs of functions in the same SCC. The reduced inter-SCC graph is acyclic and is traversed in topological order so that the callee's costs are assigned before processing the caller.

## 8. Graph Expansion

Given execution time costs, the next step is to create a set of parallel tasks. This phase begins by considering the body of the "main program" function to be a single task and proceeds by recursively expanding the current tasks to reveal more parallelism. A task containing an entire acyclic dataflow graph can be replaced by a set of new tasks - one for each node in the graph. A task corresponding to a function call node can be replaced by tasks for nodes in the callee's function body, as in conventional procedure integration.

A Forall node is special because we know that all its iterations can be executed in parallel. Thus a task for a Forall node can be replaced by S+2 new tasks - a Scatter task, S sub-Forall tasks and a Gather task. The value of S is determined in part by F, the number of iterations in the original Forall. Assuming that all iterations take the same time, the smallest S that yields an optimal completion time on P processors is $\lceil F / \lceil F/P \rceil \rceil$. This makes S = P for large F.

A task for a non-Forall compound node is replaced by a task for each of its subgraphs. These subgraphs are totally ordered by control dependencies, according to the semantics of the compound node. This ordering avoids the possibility of wasted work through eager evaluation of (say) Alternative subgraphs in a Select node. Instead, the Alternative will only start after the Selector has been evaluated, at which time it is known which Alternative should be evaluated. It's possible to perform a Parafrase-style [13] dependency analysis on While expressions and try to convert them to Forall's. This would be a compatible pre-pass to our partitioning system. We have not pursued that approach because we assume that the programs were written with a view to parallelism, and leave it to the programmer to use Forall's where appropriate.

We'd like the final task system to:

- Contain sufficient parallelism for the given number of processors,
- Not have an impractically large number of tasks (e.g. one task per instruction is too many!).

Both objectives can be conveniently quantified using costs. The task system will have sufficient parallelism if no task is a *bottleneck*. Task $T_i$ is a bottleneck when all tasks that can be executed in parallel with it together contain insufficient work to keep P-1 processors busy during $T_i$'s execution, i.e.

$$\Sigma_{T_j \text{ parallel to } T_i} \text{cost}(T_j) < (P-1) \times \text{cost}(T_i)$$

Only bottleneck nodes are considered for further expansion.

The problem of determining parallel tasks is equivalent to finding the transitive closure of the graph's adjacency matrix. Transitive closure algorithms have a worst-case execution time between $O(N^{2.5})$ and $O(N^3)$, making them impractical for large programs [5]. We use a divide-and-conquer approach on the hierarchical structure of IF1 program graphs to compute the path relation more efficiently. It is only necessary to use the $O(N^3)$ algorithm on dataflow graphs at each level. Their path relations are then efficiently combined in a depth-first traversal of the entire hierarchy. This corresponds to the notion of path-preserving homomorphic graph structures [17], and makes it practical to determine the path relation for a large program graph.

To enforce a reasonable limit on the number of tasks, we employ a Granularity Threshold Factor, ε. Any computation with execution time less than ε × (Total Program Cost) / P is considered not worthwhile for further expansion. This threshold value controls task granularity - 0.001 is a typical value for ε. A smaller value of ε usually increases the number of tasks and hence the execution time for compile-time partitioning. Programs with sufficient coarse-grain parallelism are unaffected by ε; a few expansions remove all bottleneck nodes causing the expansion process to terminate before tasks reach the granularity threshold size.

After task expansion, there is scope for further economy on the number of tasks by merging small tasks when their total cost is less than the threshold value. These small tasks are usually simple nodes (e.g. Plus, ArrayBuild) that were exposed along with larger computations during task expansion. By merging tasks, we map a set of IF1 nodes to a single task. This set must:

- Have a total execution time that's smaller than the threshold value.
- Form a convex subgraph of the original precedence graph so that the reduced precedence graph with the single merged task will still be acyclic. A simple way to form convex subgraphs is by picking intervals on any linear completion of the precedence graph.
- Form a connected subgraph so that it does not destroy any parallelism outside the merged task.

Task expansion and task merging partition the program graph nodes into tasks. Each task should either be a non-bottleneck or have a smaller execution time than the threshold value. A task that satisfies neither property is expanded, if possible. Tasks are merged if their total cost is still less than the threshold value. Both expansion and merging can be incorporated in a single depth-first traversal of the program graph. At each level, all subcomputations are first recursively processed, which determines the expanded nodes. A second pass at the same level performs the merging, and then returns to the parent level.

## 9. Internalization Pre-pass

Once the task boundaries have been established, the problem is represented in terms of our model for multiprocessor scheduling with communication. We have tasks with execution times and communication edges. A single pass scheduling algorithm is unsuitable for handling communication costs. For example, in Figure 9-1, if tasks A and B are assigned to different processors, a single pass algorithm is later on forced to make one of $C_1$ or $C_2$ non-local, and incur its overhead. This is inevitable no matter how large $C_1$ or $C_2$ may be. It could be avoided by backtracking on previous assignments, but that would be too inefficient. Instead, we first perform an Internalization pass that partitions

tasks into blocks, so that all tasks in the same block must be assigned to the same processor. After Internalization, a single pass Processor Assignment algorithm can be used to assign internalized task blocks to processors.

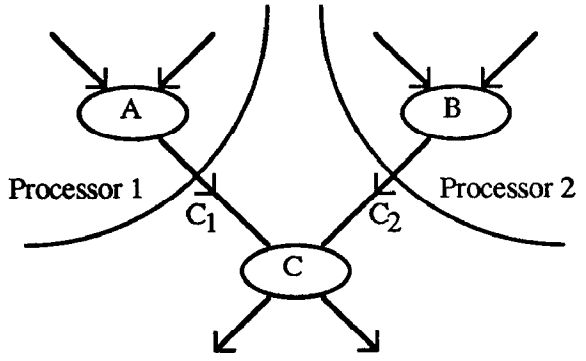


**Figure 9-1:** Counter-example for one-pass scheduling

The Internalization problem is to find a partition that minimizes the critical path length of the task system, i.e. minimizes the completion time of the task system on an unbounded number of processors. If we ignore communication overhead, this optimal completion time can be simply achieved by assigning each task to a different processor. This is not so with communication costs, since the optimal critical path may only occur when some parallel tasks are assigned to the same processor. It is a harder problem and is in fact NP-complete.

So, we designed a greedy approximation algorithm to solve this problem. It begins with the trivial partition that places each task in a separate block. It also maintains a table DeltaCPL[i,j], which represents the decrease in the critical path length obtained by merging blocks i and j. The algorithm then iteratively merges the best pair of blocks - the pair that yields the largest decrease in the critical path length - and terminates when no remaining merger could possibly reduce the critical path length (i.e. all entries in DeltaCPL are negative).

In computing the critical path length, we force all tasks in the same block to execute sequentially since they will be assigned to the same processor. There could be several possible task sequences consistent with the precedence constraints, and it's for that very reason that the problem is NP-complete. An algorithm that tries all possible sequences will have a worst-case exponential time. Instead we just use an arbitrary topological order (priority list) to provide a sequence for tasks in the same block. Figure 9-2 outlines the body of procedure DetermineCompletionTime, used to compute the completion time for a given partition. Because of step 3, it has an O(N+E) worst-case execution time, for N tasks and E synchronization and communication edges.

Inputs: Tasks, costs, partition, priority list.

Output: Completion time.

Algorithm:

1. **for each** task $T_i$ **do**
   Add non-local synchronization and communication costs to $T_i$'s execution time. (Use processor component for communication)
   **end for**
2. **for each** block B **do**
   BlockTime[B] ← 0
   **end for**
3. **for each** task $T_j$ in priority list order **do**
   StartTime ←
   max( BlockTime[Block($T_j$)],
   CompletionTime[i] ∀ <i,j> ∈ S,
   CompletionTime[i] + delay component of C[i,j]
   ∀ non-local input communication C[i,j])
   CompletionTime[j] ← StartTime + ExecutionTime[j]
   BlockTime[Block($T_j$)] ← CompletionTime[j]
   **end for**
4. **return** max(CompletionTime[j] ∀ tasks $T_j$)

**Figure 9-2:** Procedure DetermineCompletionTime

The internalization algorithm has an $O(N^2 \times (N+E))$ execution time because there are $O(N^2)$ entries in DeltaCPL. In the worst case, $E = O(N^2)$, making this an $O(N^4)$ algorithm. Just like the path relation in the previous section, the critical path can be obtained by combining critical path values of subgraphs. The algorithm incurs an $O(N^4)$ worst-case execution time at each level, but is practical for large programs. Most programs have a small number (< 100) of tasks at each level. We have seen programs with over 5000 nodes containing fewer than 20 tasks per level. This is because there's not much computation that can be expressed without using compound nodes. Even if there are several simple nodes at the same level, they often get merged into a small number of tasks, due to the granularity threshold value.

Though this is an approximation algorithm, we have shown that it finds the optimal partition for a restricted class of communication graphs, namely series-parallel graphs. Further, we have shown that this algorithm has a worst-case performance bound of 2, relative to the optimal critical path. The proofs of these results are beyond the scope of this paper.

## 10. Processor Assignment

With the internalization pre-pass completed, the ground is finally set for the actual assignment of tasks to processors. Tasks in the same internalized block must be assigned to the

same processor. We use a modified Priority List scheduling algorithm [11] to perform the processor assignment. An outline of this algorithm is given in Figure 10-1.

Inputs: Tasks, costs, internalized partition, priority list.

Output: Processor assignment for each task.

Algorithm:

1. **for each** task $T_i$ **do**
   Processor[i] ← 0
   **end for**
2. **for** proc ← 1 **to** P **do**
   ProcessorBlock[proc] ← 0
   **end for**
3. **for each** task $T_j$ in priority list order **do**
   **if** Processor[j] = 0 **then**
   a. **for** proc ← 1 **to** P **do**
      Call procedure DetermineCompletionTime for partition that merges blocks Block($T_j$) and ProcessorBlock[proc].
      Set BestProc to the value of proc with the smallest completion time.
      **end for**
   b. Merge blocks Block($T_j$) and ProcessorBlock[BestProc].
   c. ProcessorBlock[BestProc] ← Block($T_j$)
   d. **for each** task $T_i$ with Block($T_i$) = Block($T_j$) **do**
      Processor[i] ← BestProc
      **end for**
   **end if**
   **end for**
4. **return** Processor

**Figure 10-1:** Procedure Schedule

The algorithm visits tasks in priority list order, so that a task is only scheduled after all its predecessors have been scheduled. Processor[i] stores the processor number for task $T_i$. It is initialized to zero and is set when visiting the first task in $T_i$'s block. Like the Internalization algorithm, Processor Assignment proceeds by merging blocks in the partition. It terminates when there are at most as many blocks as processors. At that time, all tasks $T_i$ in ProcessorBlock[*proc*] will have Processor[i] set to *proc*. Once again, we use Procedure DetermineCompletionTime to compute the completion time for a given partition.

This algorithm has a worst-case O(B×(N+E)) execution time, where B is the number of internalized blocks. The scheduling problem does not lend itself to an efficient divide-and-conquer algorithm, as the path relation and critical path problems, because all sub-computations share the same set of processors. Instead, its efficiency lies in being able to assign an entire block of tasks to a processor at a time.

## 11. Code Generation Issues

The output of the processor assignment phase consists of P task sequences for P processors. Each task's computation is translated to sequential code, as in uniprocessor compilation. However, synchronization primitives and communication code for any non-local synchronizations and communications must be appropriately placed in a prologue and epilogue for each task. These non-local synchronizations and communications are barriers that must not be crossed when optimizing and reordering instructions. Their rearrangement could form a cycle in inter-processor synchronization and lead to deadlock during execution. Even if it avoided deadlock, the schedule would be different from the one chosen by our algorithm, and could have a larger parallel execution time. However, the code generator is free to reorder and optimize instructions that do not cross an external synchronization or communication. There should be a large scope for such conventional code optimizations, since we exploit outer-level parallelism and each task can have a lot of computation buried inside it.

## 12. Preliminary Results

As mentioned earlier, a partitioner-cum-scheduler based on these techniques has been implemented to process IF1 program graphs. We have instrumented the Livermore IF1 interpreter to provide statistics for a multiprocessor simulation. The simulation uses processor assignments generated by the partitioner. Execution time values are based on actual run-time frequencies and data sizes.

Figure 12-1 shows the speed-up obtained for the following SISAL programs:

1. Towers of Hanoi. A program to solve the puzzle for a tower of height 10. Graph expansion unwound the recursive function calls to get a binary tree with $\lceil \lg P \rceil$ levels. Hence the non-linearity when the number of processors is a power of 2.
2. Batcher's iterative merge-exchange sorting algorithm [4] on 100 integers. This is an excellent algorithm for parallel sorting. It consists of two nested While loops, each with log N iterations, and an inner Forall with N iterations. Graph expansion successively expanded the While bodies and finally the Forall, which contains the parallelism.
3. Eight Queens - a recursive program to generate all solutions to the 8 queens problem. A recursion depth value of 8 directed the graph expansion algorithm to expand the recursive call to 8 levels. The Forall at each level was then expanded.
4. Multi-precision multiplication. A divide-and-conquer solution to the problem of multiplying two N-bit numbers [2]. The algorithm breaks each number into halves, recursively finds 3 sub-products and combines them to get the full product. Using 3 (instead of 4)

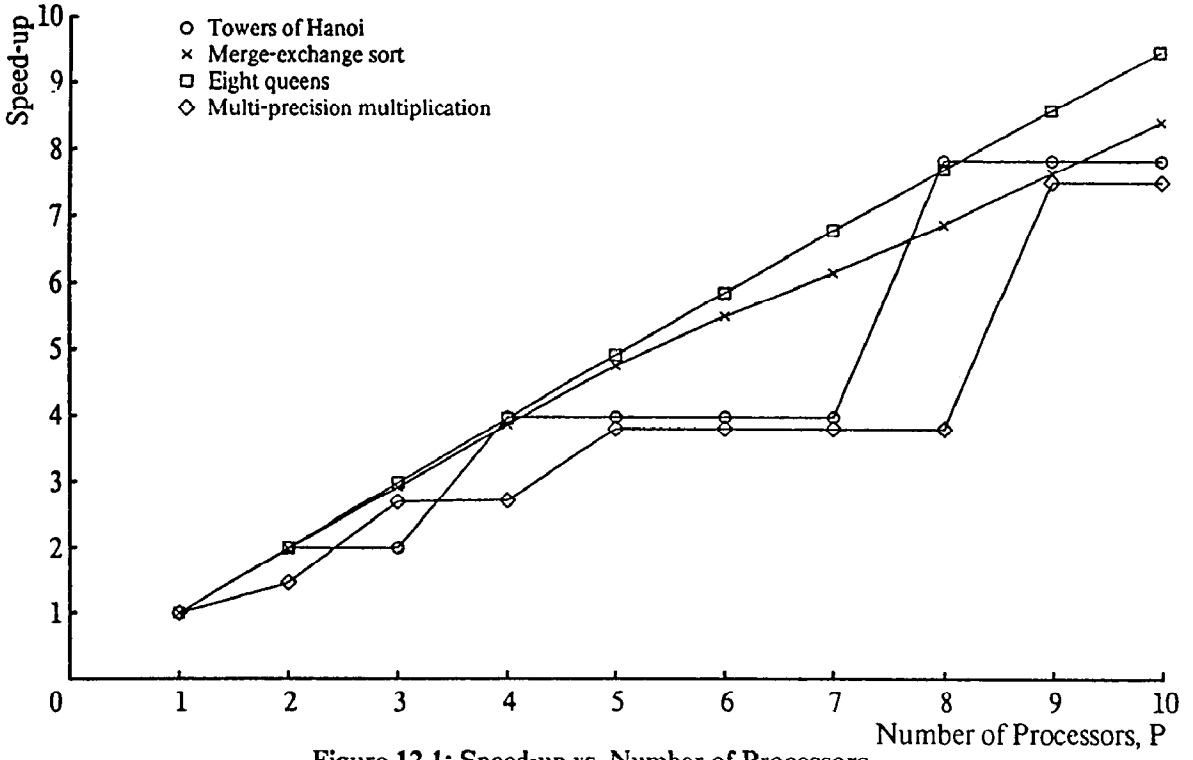


Figure 12.1: Speed-up vs. Number of Processors

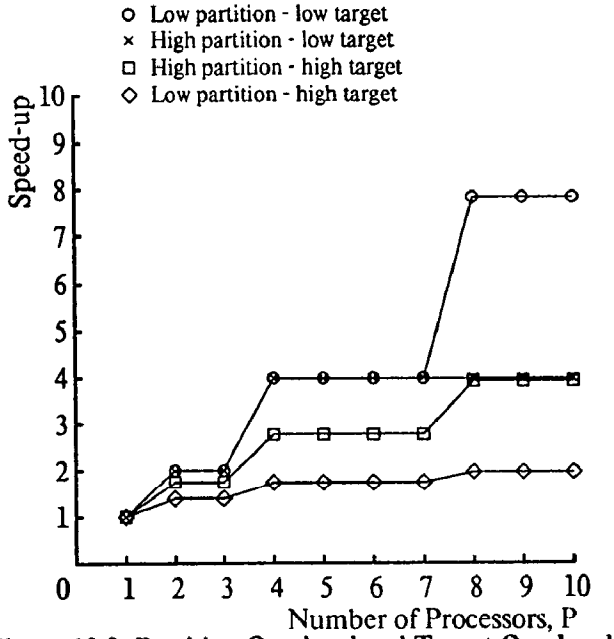


Figure 12.2: Partition Overhead and Target Overhead

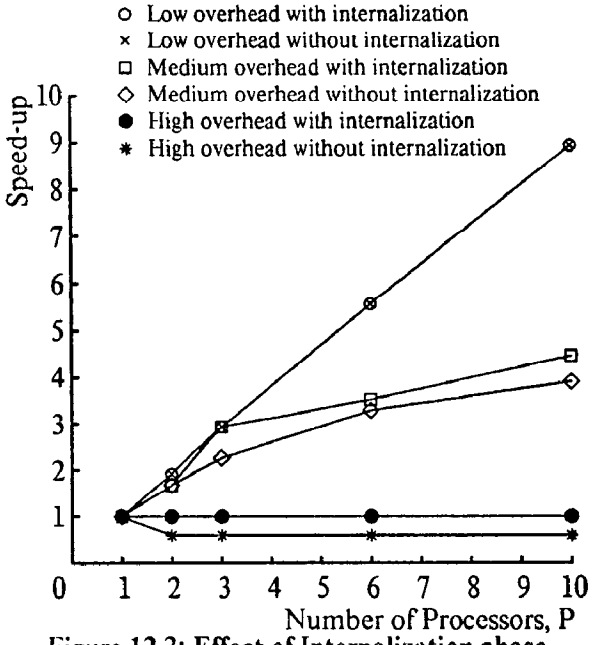


Figure 12.3: Effect of Internalization phase

recursive multiplications makes this an $O(N^{lg\ 3}) = O(N^{1.59})$ algorithm (instead of $N^2$).

These speed-up curves show that compile-time processor assignment can be successfully used to exploit parallelism. We'd like to make similar speed-up measurements on real machines, e.g. Sequent, Encore, iPSC. That will be possible when the translator from IF1 to U-code is ready. Another approach is to translate IF1 to C. We have already hand-coded the partitioned merge-exchange sort program in C and observed linear speed-up on the Sequent (12 processors) and Encore (20 processors).

Figure 12-2 illustrates the match between a partition and its target multiprocessor parameters. These measurements were taken for the Towers of Hanoi program, using two sets of target parameters that represent low and high communication overhead. The four curves show all four combinations of the two partitions with the two targets. Naturally, the low overhead target curves show a better speed-up than the high overhead target. But, for a given target, the partition that was generated for it performed better than the other partition.

Figure 12-3 shows the effect of the Internalization phase on partitioning. The SISAL program used was a simple matrix multiplication of two 20x20 integer matrices. The program was partitioned and simulated for three sets of target parameters representing low, medium and high communication overhead. Two partitions were generated in each case - one with and one without using the Internalization phase. As seen in the figure, the partition with Internalization performed better, especially for high communication overhead.

## 13. Related Work

As mentioned in the introduction, a compiler system for parallel machines must deal with the problems of extracting parallelism, partitioning the program into tasks and scheduling tasks. The major effort so far has been in solving the first problem. Kuck's Parafrase [13], [16] and the Rice vectorizer [3] have been successful in extracting parallelism from Fortran programs. They have been used for vector machines, where partitioning and scheduling is not an issue. This parallelism is typically local, since global parallelism (say between subroutines) is difficult to automatically extract from a sequential language. Both these systems could be used to produce an IF1-like graph representation that serves as input to our partitioner. Whether such an approach would be effective depends on the ability of these systems to recognize larger-grain parallelism.

The Bulldog compiler [9] extracts local parallelism and schedules operations for VLIW (Very Long Instruction Word) architectures. Loop unrolling and trace scheduling are among the techniques used to get more parallelism than that available within basic blocks. This is similar in spirit to our task expansion, except that we start at the outermost level and move inwards looking for parallelism at the macro level. The Bulldog approach attempts to generate a set of synchronous, fine-grained parallel operations that can be "packed" into a single wide instruction word. The primary technique used to find parallelism is local expansion of basic blocks. Our partitioner is targeted to asynchronous multiprocessors, which perform most efficiently with coarse-grain parallelism.

In the Hughes Data Flow Machine compiler [5], dataflow nodes (actors) are statically allocated to processing elements. The allocation is based on heuristics to minimize communication and maximize parallelism. The heuristic functions use inter-processor distances and a count of parallel actors; they do not consider the frequency count of individual actors or the communication size of data. Static allocation of an actor causes all its invocations to be sequential, since they are executed on the same processing element. This can reduce parallelism for an actor in the body of a Forall or in a function called more than once in parallel. We address the problem by task expansion, so that different sub-Forall's or different calls to the same function can be executed on different processors. It is necessary to do a transitive closure of the dataflow graph to determine parallel actors. Their "local allocator" uses an $O(N^3)$ transitive closure algorithm and took 3 VAX CPU hours to schedule 415 actors. Because of this high cost, they use a "global allocator" to approximate the heuristics by partitioning the graph and set of processing elements into separate pieces that can be individually handled by the local allocator. Transitive closure is done more efficiently in our partitioner, because we exploit the program graph hierarchy to determine the path relation, e.g. it took only 10 VAX CPU seconds for a program graph with over 1500 nodes.

## 14. Conclusions

We have demonstrated that the problem of partitioning and scheduling parallel programs can be solved at compile-time. Our techniques are practical and have been implemented to process IF1 program graphs. They rely on estimates of frequency parameters, which we obtain from execution profile data.

These techniques do not assume any particular multiprocessor architecture. Instead, they are driven by a table of parameters that describe the target multiprocessor.

The central issue in partitioning and scheduling is the trade-off between parallelism and the overhead of

synchronization and communication. We use costs to incorporate these overheads in our model for multiprocessor scheduling.

The implementation has already been used to partition many benchmark programs, and the simulation results are very encouraging. As more multiprocessors become available, we will use this implementation as a basis to compare alternative architectures and their interaction with different application programs.

## References


**1.** Ackerman, W. B. & Dennis, J. B. VAL -- a value-oriented algorithmic language. Preliminary reference manual. MIT/LCS/TR-218, Laboratory for Computer Science, MIT, June, 1979.

**2.** Aho, A. V., Hopcroft, J. E., Ullman, J. D.. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

**3.** Allen, J. R. & Kennedy, K. PFC: A Program to Convert Fortran to Parallel Form. The Proceedings of the IBM Conference on Parallel Computers and Scientific Computations, March, 1982.

**4.** Batcher, K. E. "Sorting networks and their applications". *1968 Spring Joint Computer Conf., AFIPS Proc. 32* (1968), 307-314.

**5.** Campbell, M. L. Static Allocation for a Dataflow Multiprocessor. Proc. 1985 Int. Conf. Parallel Processing, 1985, pp. 511-517.

**6.** Celoni, J. R. & Hennessy, J. L. SAL: A Single-Assignment Language for Parallel Algorithms. ClaSSiC-83-01, Center for Large Scale Scientific Computation, Stanford University, Sept., 1983.

**7.** Davis, A. L. & Keller, R. M. "Data Flow Program Graphs". *IEEE Computer 15*, 2 (Feb. 1982).

**8.** *Using the Encore Multimax*. Argonne National Laboratory, Mathematics and Computer Science Division, ANL/MCS-TM-65, 1986.

**9.** Fisher, J. A. *et al.* "Parallel Processing: A Smart Compiler and a Dumb Machine". *SIGPLAN Notices 19*, 6 (June 1984).

**10.** Gottlieb, A. *et al.* "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer". *IEEE Trans. Computers C-32*, 2 (Feb. 1983).

**11.** Graham, R. L. "Bounds on Multiprocessing Timing Anomalies". *SIAM J. Appl. Math. 17*, 2 (March 1969).

**12.** Jones, A. K., Gehringer, E. F. The Cm* Multiprocessor Project: A Research Review. CMU-CS-80-131, Computer Science Department, Carnegie-Mellon University, 1980.

**13.** Kuck, D. J. *et al.* Dependence Graphs and Compiler Optimizations. Proc. 8th ACM Symp Principles Programming Languages, Jan., 1981, pp. 207-218.

**14.** Lenstra, J. K. & Rinnooy Kan, A. H. G. "Complexity of Scheduling under Precedence Constraints". *Operations Research 26*, 1 (Jan-Feb 1978).

**15.** McGraw, J. *et al.* SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual, Version 1.2. M-146, LLNL, March, 1985.

**16.** Padua, D. A., Kuck, D. J. & Lawrie, D. H. "High-Speed Multiprocessors and Compilation Techniques". *IEEE Trans. Computers C-29*, 9 (1980).

**17.** Pfaltz, J. L.. *Computer Data Structures*. McGraw-Hill, Inc., 1977.

**18.** Seitz, C. L. "The Cosmic Cube". *CACM 28*, 1 (Jan. 1985).

**19.** *Using the Sequent Balance 8000*. Argonne National Laboratory, Mathematics and Computer Science Division, ANL/MCS-TM-66, 1986.

**20.** Sites, R. *et al.* Machine-independent Pascal Optimizer Project: Final Report. UCSD/CS-79/038, University of California at San Diego, Nov., 1979.

**21.** Skedzielewski, S. & Glauert, J. IF1 -- An Intermediate Form for Applicative Languages, Version 1.0. M-170, LLNL, July, 1985.