Automatic Generation of Self-Scheduling Programs

Ian Foster

Abstract— **We describe techniques for the automatic generation of self-scheduling parallel programs. Both scheduling algorithms and the concurrent components of applications are expressed in a high-level concurrent language. Partitioning and data dependency information are expressed by simple control statements, which may be generated either automatically or manually. A self-scheduling compiler, implemented as a source-to-source transformation, takes application code, control statements, and scheduling routines and generates a new program that can schedule its own execution on a parallel computer. The approach has several advantages compared to previous proposals. It generates programs that are portable over a wide range of parallel computers. There is no need to embed special control structures in application programs. Finally, the use of a high-level language to express applications and scheduling algorithms facilitates the development, modification, and reuse of parallel programs.**

Index Terms—**Algorithmic abstraction, concurrent programming, high-level language, load balancing, portability, program transformation, self-scheduling program.**

I. Introduction

MANY interesting parallel algorithms require *load-balancing* mechanisms that dynamically *partition* a problem into suitably sized tasks and *schedule* these tasks on processors. Load balancing is often viewed as the responsibility of the hardware, operating system, or run-time system—that is, low-level system components. For example, in the Rediflow system, processors exchange load information and migrate processes to neighbors with less load [13]. Sequent multiprocessors use a special bus to communicate load information [17]. Parallel Prologs incorporate specialized schedulers [14].

Unfortunately, low-level support for load balancing is inflexible. Automatic mechanisms cannot easily exploit particular properties of an application, such as locality. Reliance on low-level facilities also compromises portability. Hence, we prefer an alternative approach that incorporates scheduling code in applications to yield *self-scheduling* programs that require little low-level support. This approach permits scheduling strategies to be tailored to particular applications. Self-scheduling programs can be generated manually. However, bookkeeping and synchronization issues provide many opportunities for error. It is preferable if the programmer can invoke a compiler to *automatically* generate a self-scheduling version of a source program.

The automatic generation of self-scheduling programs has been studied previously in the context of Fortran by Polychronopoulos [15], among others. However, Polychronopoulos focuses on the scheduling of fine-grained tasks (DO-loop iterations). Babb [1], Boyle *et al.* [2], and Dongarra and Sorenson [6] describe techniques for larger-grained tasks. However, their techniques require that the programmer restructure application code and embed calls to control macros. Furthermore, the techniques are based on a shared-memory model and hence are not directly portable to local-memory machines.

We advocate an alternative approach. We assume that applications are expressed using a *bilingual* programming style in which a high-level concurrent language such as Strand [9] or PCN [4] is used to organize the concurrent execution of sequential components implemented in low-level languages (Fortran and C) [7]. We show how a self-scheduling version of a bilingual program can be generated by an automatic source-to-source transformation of the program's concurrent component. The transformation is directed by simple control information that specifies partitioning and data dependency information. The self-scheduling version of the program is linked with scheduling routines contained in libraries, to provide a program that can schedule its own execution on a parallel computer. We describe transformations appropriate for an important class of programs, and report on our experiences employing the techniques in applications.

This approach has a number of advantages. The use of a high-level notation makes it easy to express and modify both application-specific algorithms and scheduling algorithms. Furthermore, the separation of concerns achieved by separate specification of application, partitioning, and scheduler reduces overall complexity. As the self-scheduling compiler is implemented as a source-to-source transformation, the portable compilers and run-time systems that have been developed for Strand [10] ensure portability of self-scheduling programs across both shared-memory and local-memory computers. Finally, as computationally intensive components may be written in languages such as Fortran or C, performance and existing software can be preserved.

II. Preliminaries

A. Strand

The high-level programming language Strand [9] is used throughout this paper to express concurrent algorithms. Here we summarize key features of the notation.

Strand is a member of the family of languages commonly referred to as *concurrent logic languages.* Research in concurrent logic programming originated with the Relational Language of Clark and Gregory [5]. Subsequent proposals have included Concurrent Prolog, Parlog, FCP, and Guarded Horn Clauses. Strand captures the essential concepts of previous proposals in a simple and practical parallel programming tool.

A Strand program is a collection of *guarded rules* with the form

$$H :- G_1, G_2, \cdots, G_m | B_1, B_2, \cdots, B_n \qquad m, n \geq 0$$

where H is the rule *head,* ":−" is the implication operator, the G's are the rule *guard,* "|" is the commit operator, and the B's are

Manuscript received February 1, 1990; revised May 17, 1990. This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy, under Contract W-31-109-Eng-38.

The author is with the Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439.

IEEE Log Number 9040817.

the rule *body*. Rules in which the heads have the same name and number of arguments are grouped into a *process definition.* The head and guard of a rule define conditions that must be satisfied before a process can execute (reduce) using the rule; the body specifies new processes to replace the process if these conditions are satisfied. The program in Fig. 1 illustrates the notation.

The notation **[Head|Tail]** denotes a list structure with a **Head** and a **Tail**. An alternative tuple notation can also be used to denote structured data: A term $\{T1, \cdots, Tn\}$ denotes a tuple with subterms $T1, \cdots, Tn$. Strings beginning with uppercase letters denote variables, while those with lowercase letters denote constants. The assignment primitive ":=" is used to assign values to variables, which have the single assignment property: the value of a variable is initially undefined and, once provided, cannot be modified. An attempt to assign to a variable that already has a value is signaled as a run-time error.

The state of a Strand computation is represented by a pool of extremely lightweight processes. Execution proceeds by repeatedly selecting and attempting to reduce processes in this pool. For example, if an initial process pool contains the single process **go(4)** (defined in Fig. 1), this can be immediately reduced with rule 1 to create two new processes, **producer** and **consumer**.

Processes communicate by reading and writing shared variables. The example program illustrates a common communication structure, in which a **producer** incrementally instantiates a shared variable to a list structure (R2), hence communicating a stream of values (the list elements) to a **consumer** (R4). In the example, the values communicated are themselves variables $(X, X', \cdots)$. The **consumer** process acknowledges these "communications" by assigning each variable that it receives the value **sync** (R4).

The availability of data serves as the synchronization mechanism. Conditions expressed by nonvariable terms in a rule head define dataflow constraints: a rule can be used to reduce a process only when the process's arguments match its own. Synchronous communication protocols can be constructed in terms of this inherently asynchronous model. In the example, **producer** and **consumer** use acknowledgment messages to communicate synchronously. The **consumer** process waits for a communication from **producer**; the recursive call to **producer** waits for the variable it has communicated to be assigned the value **sync**. After sending four messages, the two processes terminate.

Note that read and write operations on variables are clearly distinguished. *Read* operations are performed when attempting to reduce a process. They suspend if they encounter an unbound variable. *Write* operations are performed by the assign primitive.

Strand is supported on a wide variety of MIMD parallel computers including hypercubes, meshes, transputer surfaces, and shared-memory machines. On a parallel computer, the processes comprising a computation may be distributed over many processors. A run-time system located at each processor ensures that read and write operations complete successfully, wherever shared variables are located.

B. Program Development Strategies

Strand encourages the following approach to the development of parallel programs. A set of process definitions is constructed that defines the operations that must be performed to solve a problem and the data consumed and produced by each operation. Individual operations may themselves be specified in Strand or may invoke sequential code written in other languages. This initial program specifies a parallel algorithm but not how operations are to be clustered or allocated to processors. Partitioning and mapping issues are addressed in a separate development stage, typically by annotating the source program to indicate processes that are to be executed on remote processors [9]. A preprocessor translates the annotated program into a form suitable for execution on a parallel computer.

An important advantage of this approach is that it is frequently possible to experiment with alternative partitioning and scheduling strategies simply by providing alternative annotations. This form of experimentation is typically not supported by conventional approaches to parallel execution, where an alternative partitioning requires a complete restructuring of the application.

The separation of concerns between application and scheduling issues has previously been supported by tools based on the concept of *virtual machines* [16]. A virtual machine is an abstract connection topology and an associated set of mapping annotations. For example, ring connectivity with annotations **@bak** and **@fwd**, and total connectivity with annotations **@random** and **@N**. Although useful, these tools do not support load balancing and hence are often ill suited for applications with irregular or dynamic load. In this paper, we develop alternative and complementary techniques that enable separate specification of partitioning and mapping for applications that require load balancing.

C. Directed Partitions

The techniques developed in this paper are applicable to an important subset of Strand programs, namely, those programs for which it is possible to define a *directed partition.* We introduce this class of programs here.

By default, a process and its offspring execute on the same processor. However, a *partition* of a Strand program identifies certain processes in rule bodies as *tasks.* These processes (and hence their offspring) are candidates for execution on other processors. For example, one possible partition of the program in Fig. 1 would identify the **producer** and **consumer** processes in rule 1 as distinct tasks. Execution of a process **go(4)** with this program would then generate two tasks. In general, the recursive nature of Strand programs means that the number of tasks generated by a program's execution can be arbitrary and dynamic.

The execution of a program can conveniently be represented by a *process tree,* in which a node represents a process and a node's offspring represent processes in the body of the rule used to reduce that process. For example, Fig. 2 represents the upper levels of the tree corresponding to a computation initiated by the process **go(4)** using the program in Fig. 1. A task in a program corresponds to a subtree in a process tree. The two tasks specified by the partition described previously are highlighted in Fig. 2.

A task (subtree) A is said to be *data dependent* on another task (subtree) B if B is required to execute in order that data required by A become available. In the example partition, the **consumer** and **producer** tasks are data dependent on each other: **consumer** processes in the **consumer** task require messages produced by **producer** processes; **producer** processes in the **producer** task require acknowledgments produced by **consumer** processes.

A partition is said to be *directed* if there are no cyclic data dependencies between the tasks that it creates. Note that this definition does not preclude data dependencies, but requires that it be possible to define a sequential ordering of task executions that will execute to completion without deadlock. Many, but by no means all, parallel algorithms can be expressed with directed partitions, which have the advantage of never requiring

```
go(N) :- producer(N,Xs,sync), consumer(Xs).                          % R1

producer(N,Xs,sync) :-                                                % R2
    N > 0 | Xs := [X | Xs1], N1 is N - 1, producer(N1,Xs1,X).
producer(0,Xs,_) :- Xs := [].                                         % R3

consumer([X | Xs]) :- X := sync, consumer(Xs).                        % R4
consumer([])                                                          % R5
```

Fig. 1. Example Strand program.

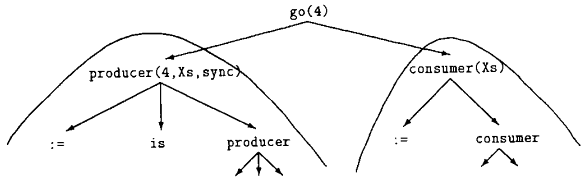


Fig. 2. Producer/consumer process tree.

coscheduling. Note that the example partition is not directed; the **producer** and **consumer** tasks are mutually data dependent and must be scheduled concurrently to avoid deadlock.

Fig. 3 presents a program for which a useful directed partition can be defined. This is the (somewhat simplified) top level of a solution to the state-space search problem. This problem, which occurs in areas as diverse as VLSI design and molecular dynamics, may be stated as follows: Start from an initial state and apply transition rules to obtain further states, until one or more final states are reached. Alternatively, the problem may be viewed as the exploration of a tree (rooted in the initial state) looking for leaves that satisfy some criterion (i.e., are final states).

The three rules in this program can be read as follows. A problem is reduced by invoking a process to explore a portion of the corresponding tree, splitting any remaining problem into subproblems, and proceeding to reduce these subproblems (R1). Each subproblem is reduced independently (R2-3). Solutions are collected using a programming technique called a *difference list* [9]. This technique permits many processes to cooperate in building a list. Each process is given the head and tail of a part of the list and can either insert values into this sublist or assign the list tail to the head, hence inserting nothing. Note that the program does not provide an explicit specification of a parallel execution strategy. Nevertheless, the potential for parallel execution is clear and can be expressed in terms of a directed partition. Each **reduce** process is made a task.

This is, of course, a very simple example. Nevertheless, it allows us to point to some advantages of our approach. The high-level specification is more concise than equivalent programs written in Fortran or C. Opportunities for concurrent execution are explicit and can be exploited (by mapping processes to processors in a parallel computer) without changing the result computed.

D. Formalization

We now provide a more formal description of directed partitions. This material is not essential to an understanding of the paper; the reader who is happy with the preceding informal presentation can safely skip the rest of this section.

We consider a view of program execution where a *computation* consists of a sequence of states. Each state can be represented by a process tree, and transitions between states are strictly defined by *transition rules* [11], [9]. We use a function **CMatch** to represent head matching and guard evaluation. This function takes as arguments a process p and a rule of the form $H\ :-\ G\,|\,B$. It executes the match algorithm (used to match a rule head with a process) followed by guard execution and returns a substitution Θ for variables in H if $\text{match}(p, H) = \Theta$ and $G\Theta = \text{true}$; or *suspend* otherwise.

States: The state of a computation is a process tree T whose nodes are labeled by processes. The distinction between a node (n) and the process that labels it (p) is strictly necessary because processes need not be unique. However, for brevity in exposition we will speak simply of a node p in subsequent discussion. Every process in T either is or is not an assignment process of the form $X := t$. We define a function **open_leaves**(T) which returns the set containing the open leaves of a process tree T. A node is *open* if it has not taken part in a transition (see below).

The initial state of a computation with processes $p_1, \cdots, p_k$ is the tree with nodes $p_1, \cdots, p_k$ as offspring of the root.

Transitions: A transition rule specifies a mapping between states. Execution of a program P proceeds according to the following transitions:

Reduction: $T \rightarrow T'$

If $p_i \in \textbf{open\_leaves}(T) \wedge p_i \neq (X := t) \wedge \textbf{CMatch}(p_i, H :- G\,|\,B) = \Theta$ where $H :- G\,|\,B$ is a *fresh copy* (new variables) of some $R \in P$, and T' is derived from T by adding the processes $B\Theta$ as offspring to p_i.

Assignment: $T \rightarrow T'$

If $p_i \in \textbf{open\_leaves}(T) \wedge p_i = (X := t)$ where X is a variable that does not occur in t, and T' is derived from T by applying the substitution $[X \backslash t]$.

Suspension: $T \rightarrow$ <suspend>

If $\textbf{open\_leaves}(T) = \phi \wedge \forall\, p_i \in \textbf{open\_leaves}(T)(p_i \neq (X := t) \wedge (\forall\, R \in P, \textbf{CMatch}(p_i, R) = \textit{suspend}))$.

Note that in the reduction rule no nodes are added to the new state if B is the empty body.

Computations: A computation is a sequence of states, each derived from the prior state by application of a transition rule, and ending in a terminal state (i.e., one in which no further rules can be applied). A computation that ends in a state T is called a *successful* computation, and T is its computation tree. Note that a computation contains information not represented by its tree: namely, the order in which independent reductions and assignments are performed.

In the following, we will sometimes refer to the computation tree for a computation C simply as "the tree C." We restrict our attention to successful computations. We write $\textbf{desc}_C(n)$ to denote the set of *descendants* of a node n in the tree C. We employ the function $\textbf{p\_map}_{CP}$, which maps nodes in C to processes in the

```
reduce(Prob,Solns,Solns2) :-                                  % R1
    process_prob(Prob,NewProb,Solns,Solns1),
    split_prob(NewProb,Probs),
    reduce_all(Probs,Solns1,Solns2).

reduce_all([Prob | Probs],Solns,Solns2) :-                    % R2
    reduce(Prob,Solns,Solns1),
    reduce_all(Probs,Solns1,Solns2).
reduce_all([],Solns,Solns1) :- Solns := Solns1.               % R3
```

Fig. 3. State-space search example.

program P executed by C. This function is defined for all nodes except the root node and its immediate offspring. It maps each offspring of a node n to the corresponding process in the rule in P used to reduce n.

Data Structures: A computation constructs a set of data structures by instantiating variables in the computation tree. Positions in these data structures can be referred to by data positions with the form $<n, u>$, where n is a tree node and u is a path to a position in n.

Each data position has one or more *providers* and zero or more *consumers*. A process *provides* a data position if it produces or propagates the value at the position. A process *consumes* a data position if it reads the value at the position. We formalize these notions by extending the transition rules to construct the following functions:

Provides: Data-Position $\rightarrow \mathcal{P}$(Tree-Nodes)

Consumes: Data-Position $\rightarrow \mathcal{P}$(Tree-Nodes).

Reduction Rule (which reduces a node n_i to B with substitution Θ, using rule $H : - G \mid B$). Each data position d in n_i that is read by the application of **CMatch** is tagged as consumed by $n_i : n_i \in$ **Consumes**(d). The data positions that are read by **CMatch** are those that are matched against nonvariable terms in the rule head H or to which guard tests in G are applied.

Application of the reduction rule also tags all data positions d' in B as provided by $n_i : n_i \in$ **Provides**(d'). Furthermore, the substitution Θ produced by **CMatch** and applied to body processes is assumed to propagate any tags on data positions in n_i to the corresponding data positions in B.

Assignment Rule (which executes a process $n_i = (X := t)$ and applies the substitution $[X \backslash t]$ to tree T to derive T'). T' is derived from T by placing an occurrence of t at each data position d containing X. All data positions d' at and below each such d are tagged as provided by $n_i : n_i \in$ **Provides**(d'). In addition, any tags on data positions in t at n_i are propagated to the corresponding positions at or below each d, and any tags on d are propagated to nodes below d. The propagation steps ensure that nodes provide entire subtrees, even if these subtrees are incomplete at the time the node is executed.

Node Ordering: Let $<_C$ be the least partial order on the nodes in a tree C that satisfies the following conditions. These state that 1) the provider of a data position executes before its consumers and 2) a parent executes before its offspring.

1) $\forall$ data position d, $\forall\, p, q \in \mathrm{Nodes}(C)$,
   $p \in \mathbf{Provides}(d), q \in \mathbf{Consumes}(d) \rightarrow p <_C q$.
2) $\forall\, p, q \in \mathrm{Nodes}(C), q \in \mathbf{desc}_C(p) \rightarrow p <_C q$.

A subtree rooted at node p in a tree C is *dependent* on the subtree rooted at node q if there exist nodes $a \in \mathbf{desc}_C(p), b \in \mathbf{desc}_C(q)$ such that $b <_C a$.

Definition 1: A program P is *directed* if no computation contains a pair of disjoint, mutually dependent subtrees. That is, for any successful computation C of P:

$$\forall\, p, q \in \mathrm{Nodes}(C), p \notin \mathbf{desc}_C(q), q \notin \mathbf{desc}_C(p) \rightarrow$$
$$\nexists\, (u, v \in \mathbf{desc}_C(p), x, y \in \mathbf{desc}_C(q)) : u <_C x \wedge y <_C v.$$

□

Partition: A *partition* of a program tags certain processes in rule bodies with task names. For simplicity, we consider only partitions in which each task name tags a single process. Let the function $\mathbf{task}_{PL}$ define a partition L of a program P: $\mathbf{task}_{PL}(p)$ returns a task name if process p is tagged by L and ϕ otherwise. Hence, the subtree rooted at a node n in a tree C of P is a task iff $\mathbf{task}_{PL}(\mathbf{p\_map}_{CP}(n)) \neq \phi$.

Definition 2: A partition L of a program P is *directed* if no computation contains a pair of mutually dependent tasks. That is, for any successful computation C of P:

$$\forall\, p, q \in \mathrm{Nodes}(C), \mathbf{task}_{PL}(\mathbf{p\_map}_{CP}(p)) \neq \phi,$$
$$\mathbf{task}_{PL}(\mathbf{p\_map}_{CP}(q)) \neq \phi, p \notin \mathbf{desc}_C(q),$$
$$q \notin \mathbf{desc}_C(p) \rightarrow$$
$$\nexists\, (u, v \in \mathbf{desc}_C(p), x, y \in \mathbf{desc}_C(q)) : u <_C x \wedge y <_C v.$$

□

Observation 1: All partitions of a directed program are directed. (Note, however, that this is not the case if partitions are permitted to define tasks containing two or more processes.)

Observation 2: Directed partitions can be defined for some programs that are not themselves directed.

III. The Approach

We now introduce the techniques used to generate self-scheduling programs. A *compiler* transforms an application into a form suitable for linking with scheduling routines provided in a *library*. Both the compiler and the scheduling library are application independent. Furthermore, the application/scheduler interface is designed to facilitate the substitution of alternative schedulers.

We introduce these ideas by presenting a simple scheduler, defining its interface, and showing how a compiler transforms an application program to fit this interface. Initially, we consider the problem of scheduling tasks in the absence of data dependencies.

A. Scheduler

We use a simple manager/worker scheduler to illustrate issues in scheduler implementation and scheduler/application interfaces. This scheduling algorithm employs a central *manager* which maintains a pool of pending tasks and allocates tasks to idle *workers.* Clearly, the use of a central manager renders this particular algorithm inappropriate for large parallel machines. In practice, we frequently use alternative hierarchical or random scheduling algorithms. However, we choose to present the manager/worker algorithm here as its simplicity permits a complete exposition within the confines of this paper. The essential techniques apply directly to the more sophisticated, scalable schedulers.

The process structure employed by the manager/worker scheduler is illustrated in Fig. 4. Six worker processes are shown ($W1, \cdots, W6$); these are responsible for both processing tasks and generating new tasks. A single manager (M) allocates tasks to idle workers. Each worker has two streams to the manager, which it uses to communicate requests for work and task pool contributions, respectively. The manager returns tasks to workers by assigning values to variables included in request messages.

The complete scheduler program is presented in Fig. 5. It consists of three distinct parts. Rules 1–3 create the process network illustrated in Fig. 4. A call to **scheduler** (R1) has the general form **scheduler(*N*, FirstTask)** where *N* is the number of workers to create and **FirstTask** is a term representing an initial process to be placed in the task pool. The **scheduler** process creates a **manager** and two **mergers** (R1), plus a **workers** process which recursively creates *N* **worker** processes (R2–3). The low-level process mapping annotation **@N** is used to place the worker processes on processors 1–*N*; other processes are created on processor 0 by default. The purpose of the two primitive **merger** processes associated with the manager (R1) is to combine the multiple streams from the workers into single request and task streams (*Rs*, *Ts*). The **merge** messages generated for each worker register request and task streams with the mergers, establishing connectivity (R2) [9].

The second part of the program, rule 4, implements the manager logic. This repeatedly matches requests for work (**req**(*R*)) with tasks (*T*). Tasks are returned to requesting workers by assigning to the variable included in a request message (*R*).

The third part of the program, rules 5–6, implements the worker logic. This repeatedly requests a new task from the manager and calls **execute** to execute this task to completion. The **execute** process provides the interface to the application code. It assigns the constant **[]** to the variable **Proceed** when execution of its task is complete. This permits a worker to sequence the execution of tasks. Sequencing is achieved with the head match [] (R6). A new task is requested each time a task is started; this permits overlapping of communication and computation.

The more subtle details of this program may not be immediately intelligible to the reader unfamiliar with Strand. However, it should be clear that the use of a high-level notation has permitted a succinct and elegant specification of a relatively complex parallel algorithm. It is emphasized that Fig. 5 constitutes a *complete* implementation of the manager/worker scheduler. Clearly, 16 lines of code cannot be too difficult to understand or modify.

B. Interface

A worker invokes an **execute** process to execute application-specific code. This process constitutes the scheduler/application interface. We consider the nature of this interface here.

An application process invoked by **execute** must return two data objects: a list of offspring tasks to be passed to the manager, and a variable that will be bound (assigned a value) when all offspring processes *not* dispatched as tasks complete execution. The latter information permits a worker to sequence task execution.

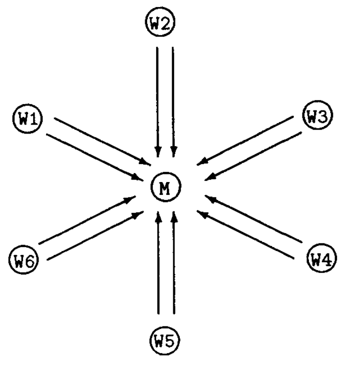


Fig. 4. Scheduler process structure.

The compiler extends the application program with code to construct these data objects. This code utilizes two related programming techniques, both of which involve threading variables through application code: the *difference list,* presented in Section II-C, and the *short circuit.* The latter technique permits termination detection in a set of concurrent processes. A process requiring termination detection is augmented with two additional variables representing the left and right sides of a circuit. This circuit is split among offspring processes and closed if a process terminates. A value placed on the left side will become available on the right only when the computation initiated by the original process has terminated [9].

The form of the scheduler/application interface is determined by the techniques used in the compiler to construct the task list and termination variable. A call to **execute** has the general form

$$\mathbf{execute(Task, Ts, Ts1, [\,], Proceed)}$$

where **Task** identifies the application process to be executed by the worker, **Ts** and **Ts1** are the head and tail of the difference list used to collect additional tasks, and **[ ]**, **Proceed** are a short circuit used to detect termination of the task. The variable **Proceed** is assigned the value **[ ]** when all nondispatched processes generated by execution of **Task** have completed.

C. Compiler

The scheduler/application interface requires that the application program be partitioned into tasks to be invoked by an **execute** process. Each task must 1) generate a (possibly empty) set of new tasks and 2) set a variable to signal completion. These requirements can be satisfied by transformation of the program, using three basic techniques.

1) *Dispatch tasks:* Each process in the original program that is to be dispatched as a task is translated into a message to the scheduler. The message comprises the process name and process arguments.
2) *Detect termination:* Process definitions invoked by tasks are augmented with termination detection code.
3) *Construct interface:* An **execute** rule is generated for each type of task dispatched by the program.

Consider the program in Fig. 3. Assume that each **reduce** process is to be a task. Application of the three techniques yields the program in Fig. 6. Note the additional arguments used to implement the difference list (**Ts**, etc.) and short circuit (**D**, etc.). Note also the replacement of the process **reduce(Prob, Solns, Solns1)** by a message representing this process

```
scheduler(N,FirstTask) :-                                          % R1
        manager([FirstTask | Ts],Rs),
        merger(Rs1,Rs), merger(Ts1,Ts),
        workers(N,Ts1,Rs1).

workers(N,Ts,Rs) :-                                                % R2
    N > 0 |
        Ts := [merge(W) | Ts1], Rs := [merge(R) | Rs1],
        worker(W,R)@N, N1 is N - 1,
        workers(N1,Ts1,Rs1).
workers(0,Ts,Rs) :- Ts := [], Rs := [].                            % R3

manager([T | Ts],[req(R) | Rs]) :- R := T, manager(Ts,Rs).         % R4

worker(Ts,Rs) :- Rs := [req(Task) | Rs1], worker1(Ts,Rs1,[],Task). % R5

worker1(Ts,Rs,[],Task) :-                                          % R6
        Rs := [req(Next) | Rs1],
        execute(Task,Ts,Ts1,[],Proceed),
        worker1(Ts1,Rs1,Proceed,Next).
```

Fig. 5. Simple manager/worker scheduler.

and its arguments (R2). The head and tail of the new task stream (**Ts, Ts1**) are not passed to **process_prob** or **split_prob**, as these processes cannot generate new tasks (R1). The single **execute** rule implements the scheduler/application interface: receipt of a **reduce** message causes the worker to start execution of the application's **reduce** process definition.

The reader is encouraged to study Figs. 5 and 6 carefully. Together, these constitute a self-scheduling state-space search program. A call to **scheduler** (Fig. 5) with an initial **reduce** message leads to the invocation of the **reduce** process (Fig. 6) with appropriate arguments. This in turn may lead to the generation of further **reduce** messages and hence additional invocations of the **reduce** process. The scheduler ensures that each call to **reduce** occurs on an idle processor.

D. Discussion

The program developed in this section (Figs. 5 and 6) need be augmented only with definitions for **process_prob** and **split_prob** to obtain a self-scheduling program that can be executed on a parallel computer. This program employs a load-balancing strategy to allocate tasks to idle processors and executes only a single task at a time on each processor. These techniques enhance processor utilization and reduce memory consumption.

We have used the self-scheduling program to good effect to parallelize two state-space search problems: a molecular dynamics code (5000 lines of Fortran) and a bin-packing problem (1500 lines of C). Both codes have been ported to shared-memory and local-memory parallel computers without modification. Good speedups were achieved on both classes of machine. On hypercubes, a hierarchical scheduler was substituted to reduce the bottleneck inherent in a central manager. This substitution was achieved without modification to the application code or to the compiler used to transform the application code.

IV. Defining Tasks

We now present a notation for specifying the partitioning and data dependency information required by self-scheduling compilers. Statements in the notation can be generated either automatically or manually; each approach has its advantages. In the rest of this paper, we assume that appropriate statements have been provided and do not concern ourselves with how they are generated.

The notation comprises two types of control statement. The first, **-initial,** specifies the name of a process to be provided as an initial task. The second, **-task,** has the general form

$$-\mathbf{task(Rule, TaskProcess, DependentProcesses)}$$

and specifies that, in a named **Rule**, the specified **TaskProcess** forms a task that can be safely scheduled after processes in the task's dependency set (those named in the list **DependentProcesses**) have completed execution. Both rules and processes are specified by "name/index" terms. The index can be omitted if the omission does not introduce ambiguity. For example, the partition developed in Section III-C for the state-space search example can be expressed using the following two control statements.

$$-\mathbf{initial(reduce)}.$$
$$-\mathbf{task(reduce\_all/1, reduce, [\,])}.$$

V. Data Dependencies

We now show how the scheduler and compiler techniques introduced in Section III can be extended to deal with data dependencies. We once again illustrate the discussion by presenting a simple scheduler library, describing the associated scheduler/application interface, and showing the results of applying the transformation required to support this interface to an example program.

A. Example Program

The program in Fig. 7 will be used to illustrate the discussion of scheduling in the presence of data dependencies. This is the (greatly simplified) top level of a program developed by Overbeek *et al.* to generate *alignments* (which identify similarities and differences) of related sequences of genetic material (RNA) [3]. Briefly, the rules state that one aligns a set of sequences by partitioning the set into *chunks* using structural information, aligning each chunk, and gluing the aligned chunks together (R1). Each chunk is aligned separately (R2). To align a single chunk, an attempt is made to find a *pin,* a unique subsequence

```
reduce(Prob,Solns,Solns2,Ts,Ts1,D,D3) :-                         % R1
    process_prob(Prob,NewProb,Solns,Solns1,D,D1),
    split_prob(NewProb,Probs,D1,D2),
    reduce_all(Probs,Solns1,Solns2,Ts,Ts1,D2,D3).

reduce_all([Prob | Probs],Solns,Solns2,Ts,Ts2,D,D1) :-           % R2
    Ts := [reduce(Prob,Solns,Solns1) | Ts1],
    reduce_all(Probs,Solns1,Solns2,Ts1,Ts2,D,D1).
reduce_all([],Solns,Solns1,Ts,Ts1,D,D1) :-                       % R3
    Solns := Solns1, Ts := Ts1, D1 := D.

execute(reduce(P,S,S1),Ts,Ts1,D,D1) :- reduce(P,S,S1,Ts,Ts1,D,D1).   % R4
```

Fig. 6. Transformed state-space search example.

```
alignment(Sequences,Alignment) :-                                % R1
    partition(Sequences,Partitions),
    align_chunks(Partitions,SubAligns),
    glue(SubAligns,Alignment).

align_chunks([Chunk | Tseqs],AlignedChunks) :-                   % R2
    AlignedChunks := [AlignedChunk | Tchunks],
    align_chunk(Chunk,AlignedChunk),
    align_chunks(Tseqs,Tchunks).
align_chunks([],AlignedChunks) :- AlignedChunks := [].           % R3

align_chunk(Chunk,AlignedChunk) :-                               % R4
    pins(Chunk,BestPin),
    divide(Chunk,BestPin,AlignedChunk).

divide(Seqs,BestPin,Alignment) :-                                % R5
  BestPin ≠ [] |
    split(Seqs,BestPin,Left,Right,UnPinned),
    align_chunk(Left,LAligned), align_chunk(Right,RAligned),
    align_chunk(UnPinned,UnPinnedAligned),
    combine(LAligned,BestPin,RAligned,UnPinnedAligned,Alignment).
divide(Seqs,[],Alignment) :- dynamic_prog(Seqs,Alignment).       % R6
```

Fig. 7. Top level of alignment program.

that occurs only once in each of a subset of all sequences (R4). A pin can be used to partition a chunk into left, right, and unpinned subchunks. If a pin is found, each subchunk is aligned separately and the resulting aligned chunks are combined (R5). Otherwise the entire chunk is aligned with a dynamic programming algorithm (R6).

The alignment program has considerable potential for parallel evaluation. For example, each **align_chunk** process created by **align_chunks** can be executed in parallel (R2), as can the **align_chunk** processes created by **divide** (R5). However, data dependencies constrain the order in which processes can be executed. For example, in R5 the **align_chunk** processes cannot be executed until the **split** process has completed execution, and the **combine** process cannot execute until the **align_chunk** processes have completed. This partition and associated data dependencies can be represented by the control statements presented in Fig. 8. These data dependencies must be taken into account when scheduling tasks. As each worker executes tasks serially and to completion, the scheduling of a task for which data dependencies have not been resolved can cause busy waiting and/or deadlock.

Note that the core of this program is a prototypical divide-and-conquer algorithm: rules 5 and 6 recursively partition a task into subtasks, solve the subtasks independently, and combine results to construct a final solution. We cannot solve this problem efficiently using the static mappings sometimes proposed for divide-and-conquer problems (e.g., [12]), as the number and size of tasks are data dependent and variable.

```
-initial(alignment).
-task(alignment,glue,align_chunks).
-task(align_chunks/1,align_chunk,[]).
-task(divide/1,align_chunk/2,split).
-task(divide/1,align_chunk/3,split).
-task(divide/1,combine,[align_chunk/1,align_chunk/2,align_chunk/3]).
```

Fig. 8. Control statements for alignment program.

B. Enhanced Scheduler

We illustrate the treatment of data dependencies in schedulers by presenting a variant of the manager/worker scheduler of Fig. 5 that deals with dependencies. As before, the choice of a simple example is motivated by a desire to present a complete algorithm. The new scheduler extends the program of Fig. 5 in two ways. First, a task is represented by a more complex data structure, a tuple of the form

$$\{\mathbf{Process}, \mathbf{Done}, \mathbf{DependentVars}\}$$

where **Process** is a representation of a task and its arguments; **Done** is the task's *termination variable,* a unique variable to be assigned a value after the task has completed; and **DependentVars** is a (possibly empty) list of termination variables of the processes on which this task is dependent.

The second extension to Fig. 5 is the introduction of a *filter* process, placed between the workers and the manager. This process delays each new task generated by workers until the termination variables of any tasks on which the task is dependent have been assigned a value. This ensures that the manager receives only those tasks that are ready for immediate execution.

The enhanced scheduler is presented in Fig. 9. Rules 2–5 are omitted as they are the same as in Fig. 5. Note the creation of the additional *filter* process (R1), and observe how this process creates an **await** process each time it receives a new task from a worker (R7). An **await** process delays until each termination variable in a task's dependency list has been assigned a value, and then passes the task to the manager (R8–9).

It is interesting to compare Figs. 5 and 9. Note that only a small extension to Fig. 5 was required to obtain the enhanced functionality of Fig. 9. This emphasizes an important advantage of a high-level notation: scheduling algorithms can be modified easily.

C. Interface

The new scheduler, like the old, is completely *application independent.* It can be used to execute *any* program that obeys its scheduler/application interface. This interface is expressed in terms of an **execute** process with seven arguments. The first five arguments fulfill the same functions as in the first scheduler. The last two arguments implement an additional short circuit, used to bind a *task termination* variable when execution of the whole task, including subtasks, has completed. The task termination variable is used to signal when dependent tasks can be passed to the scheduler. The **execute** process has the general form

$$\mathbf{execute}(\mathbf{Task}, \mathbf{Ts}, \mathbf{Ts1}, [\,], \mathbf{Proceed}, [\,], \mathbf{Done}).$$

As before, **Task** and **Ts**, **Ts1** represent the application process to be executed and the *task difference list,* respectively. The remaining terms represent a *local short circuit* and a *task short circuit,* used to bind the local and task termination variable (**Proceed** and **Done**, respectively).

A process invoked by **execute** either

1) executes **Task** to completion and then closes the difference list and the two short circuits; or
2) executes part of the task, closes the local short circuit, generates new tasks corresponding to the rest of the task, and links the termination variables of these tasks into the task short circuit.

A new task is represented by a term of the form {**Task, Done, DependentVars**}, as explained previously.

D. Enhanced Compiler

We present in Fig. 10 rules that can be applied to transform a program, under the direction of a set of control statements defining a directed partition, to yield a program that satisfies the enhanced scheduler/application interface requirements. The

```
scheduler(N,FirstTask) :-                                   % R1
    manager([{FirstTask,_} | Ts],Rs),
    filter(DTs,Ts1), merger(Ts1,Ts),
    merger(Rs1,Rs), merger(DTs1,DTs),
    workers(N,DTs1,Rs1)@fwd.

...

worker1(Ts,Rs,[],{Task,Done}) :-                            % R6
    Rs := [req(Next) | Rs1],
    execute(Task,Ts,Ts1,[],Proceed,[],Done),
    worker1(Ts1,Rs1,Proceed,Next).

filter([{Process,Done,DepVars} | In],Ts) :-                 % R7
    Ts := [merge(W) | Ts1],
    await(DepVars,{Process,Done},W),
    filter(In,Ts1).

await([[] | Vs],Task,W) :- await(Vs,Task,W).                % R8
await([],Task,W) :- W := [Task].                            % R9
```

Fig. 9. Enhanced manager/worker scheduler.

1. *Generate interface.* Generate an **execute** rule for each unique process *p* named in an **–initial** statement or in the second argument of a **–task** statement:

   execute(*p*(...),Ts,Ts1,L,L1,D,D1) :– *p*(...,Ts,Ts1,L,L1,D,D1).

2. For each control statement **–task(R,P,Ps)**:
   (a) *Insert P in task list.* Replace *P* in rule *R* by an assignment Tsi := [{P, Done, Vs} | Tsj], where Done is *P*'s termination variable, Vs lists termination variables for the processes named in *Ps*, and Tsi and Tsj are new variables.
   (b) *Link task into task short circuit.* Add a process link(Done,Di,Dj) to *R*, where Di and Dj are new variables. This process is defined as follows. The guard test **data** suspends until its argument is available, and then succeeds.

       link(Done,Di,Dj) :– data(Done) | Dj := Di.

3. Thread additional arguments through the head and non-assignment body processes of every program rule to implement:
   (a) A *task difference list.* This includes variables Tsi, Tsj associated with any assignment Tsi := [P | Tsj] introduced by rule 2(a).
   (b) A *local short circuit.*
   (c) A *task short circuit.* This includes variables Di, Dj associated with any process link(Done,Di,Dj) introduced by rule 2(b).

4. *Link non-task processes.* Replace the task short circuit variables (say Di, Dj) of any non-dispatched process named in a task's dependency set with the constant [] and the process's termination variable (say Dk), respectively. Add a process link(Dk,Di,Dj) to the rule containing the process.

5. *Close difference lists, short circuits.* Add assignment processes Ts := Ts1, D1 := D, L1 := L to any program rule with no non-assignment processes in the body.

Fig. 10. Enhanced compiler rules.

result of applying this transformation to Fig. 7 using the control statements in Fig. 8 is illustrated in Fig. 11.

The transformed application program and the definitions for **execute** and **link** constitute the output of the transformation. Note that variables introduced by the transformation (*D*, *L*, etc.) must not occur in the original program; the names **"link"** and **"execute"** must also be unique.

The implementation of a preprocessor that applies this trans-

```
alignment(Sequences,Alignment,Ts,Ts3,L,L2,D,D3) :-                        % R1
    partition(Sequences,Partitions,Ts,Ts1,L,L1,D,D1),
    align_chunks(Partitions,SubAligns,Ts1,Ts2,L1,L2,[],DV),
    Ts2 := [{glue(SubAligns,Alignment),DV1,[DV]} | Ts3],
    link(DV,D1,D2), link(DV1,D2,D3).

align_chunks([Chunk | Tseqs],AlignedChunks,Ts,Ts2,L,L1,D,D2) :-           % R2
    AlignedChunks := [AlignedChunk | Tchunks],
    Ts := [{align_chunk(Chunk,AlignedChunk),DV,[ ]} | Ts1],
    link(DV,D,D1),
    align_chunks(Tseqs,Tchunks,Ts1,Ts2,L,L1,D1,D2).
align_chunks([ ],AlignedChunks,Ts,Ts1,L,L1,D,D1) :-                       % R3
    AlignedChunks := [], Ts := Ts1, L1 := L, D1 := D.

align_chunk(Chunk,AlignedChunk,Ts,Ts2,L,L2,D,D2) :-                       % R4
    pins(Chunk,BestPin,Ts,Ts1,L,L1,D,D1),
    divide(Chunk,BestPin,AlignedChunk,Ts1,Ts2,L1,L2,D1,D2).

divide(Seqs,BestPin,Alignment,Ts,Ts5,L,L2,D,D5) :-                        % R5
  BestPin ≠ [] |
    split(Seqs,BestPin,Left,Right,UnPinned,Ts,Ts1,L,L1,[],DV),
    align_chunk(Left,LAligned,Ts1,Ts2,L1,L2,[],DV1),
    Ts2 := [{align_chunk(Right,RAligned),DV2,[DV]} | Ts3],
    Ts3 := [{align_chunk(UnPinned,UPAligned),DV3,[DV]} | Ts4],
    Ts4 := [{combine(LAligned,BestPin,RAligned,UPAligned,Alignment),
                      DV4,[DV1,DV2,DV3]} | Ts5],
    link(DV,D,D1), link(DV1,D1,D2), link(DV2,D2,D3),
    link(DV3,D3,D4), link(DV4,D4,D5).
divide(Seqs,[],Alignment,Ts,Ts1,L,L1,D,D1) :-                             % R6
    dynamic_prog(Seqs,Alignment,Ts,Ts1,L,L1,D,D1).
```

Fig. 11. Transformed alignment program.

formation is straightforward. A number of obvious optimizations can be employed to reduce the number of additional arguments and assignment processes, without significantly complicating the transformation. Our implementation totals about 50 lines of scheduler-specific code; these lines invoke library routines to perform common functions such as adding short circuits to sets of process definitions.

VI. Comparison to Other Approaches

A comprehensive comparison to other approaches to self-scheduling parallel programs is beyond the scope of this paper. However, we provide a comparison to one particularly well-known system, Schedule [6]. This system permits a self-scheduling version of a Fortran program to be developed by first identifying subroutines that can be executed as independent tasks (processes) and data dependencies between these tasks. Then, calls to the Schedule run-time system are embedded in the existing code to define tasks and data dependencies.

Dongarra and Sorenson present a Schedule solution to a problem that is in some respects similar to the divide-and-conquer algorithms used as examples in this paper [6]. This is a parallel vector multiplication program that decomposes matrix A and B into n subvectors and computes and sums the n inner products. The essential aspects of the original program are summarized in Fig. 12, using a Fortran-like pseudocode in which indentation is used to represent the extent of DO-loops and blank lines to separate procedures. The unlisted **inprod** procedure computes the inner product of two vectors. For simplicity, we assume that the vector length (**nvec**) is an integer multiple of the number of subvectors (**nslices**): **nvec = nslices*nelems.**

Dongarra and Sorenson present two self-scheduling versions of this code. The first assumes a static allocation of processes and

```
program main
real a(nvec), b(nvec), temp(nslices), sigma
call vecprod(nelems,nslices,a,b,temp,sigma)

subroutine vecprod(nelems,nslices,a,b,temp,sigma)
indx = 1
do j = 1, nslices
     call inprod(nelems,a(indx),b(indx),temp(j))
     indx = indx + nelems
call addup(nslices,temp,sigma)

subroutine addup(nslices,temp,sigma)
sigma = 0.0
do j = 1, nslices
     sigma = sigma + temp(j)
```

Fig. 12. Original vector multiplication program.

is constructed by replacing calls to **vecprod**, **inprod**, and **addup** with calls that define data dependencies and invoke the Schedule run-time system. The second assumes a dynamic allocation of processes and uses a Schedule call **SPAWN** to create processes. We present key components of the second version in Fig. 13.

Vector multiplication is of course an extremely simple problem that would normally not be parallelized in this way. Nevertheless, we see that implementation of even this simple process structure requires a significant restructuring of the original program. Code is moved between procedures, and additional subroutine calls and variables are introduced. These changes introduce many opportunities for error and produce a program that is difficult to understand and modify.

In contrast, a Strand solution to the vector multiplication problem comprises just eight lines of code. The recursive process

```
      subroutine vecprod(nelems,nslices,a,b,temp,sigma)
      indx = 1
      JOBTAG = 1
      ICANGO = 0
      NCHEKS = 1
      call DEP(jobtag,icango,ncheks,mychkn)
      call PUTQ(jobtag,ADDUP,jobtag,a,b,temp)

      subroutine addup(myid,nslices,a,b,temp,sigma)
      go to (11,22) IENTRY(myid)
11    m = nvec/nslices
      do j = 1, nslices
         call SPAWN(myid,jdummy,INPROD,nelems,a(indx),b(indx),temp(j))
         indx = indx + nelems
      if (WAIT(myid,nslices,2)) return
22    continue
      sigma = 0.0
      do j = 1, nslices
         sigma = sigma + temp(j)
```

Fig. 13. Transformed vector multiplication program.

definition spawns one **inprod** process for each subvector. The **inprod** process executes a Fortran procedure similar to that used in the Schedule program.

```
vecprod(Slices, Nelems, From, A, B, SoFar, Sigma) :-
    Slices > 0 |
        inprod(From, Nelems, A, B, Tmp),
        SoFar1 is SoFar + Tmp,
        From1 is From + Nelems,
        Slices1 is Slices - 1,
        vecprod(Slices1, Nelems, From1, A, B, SoFar1, Sigma).
vecprod(0, _, _, _, _, SoFar, Sigma) :- Sigma := SoFar.
```

A self-scheduling version of this code is generated by providing two control statements, **-initial(vm)** and **-task(vm, 1, inprod, [ ])**, and invoking our self-scheduling compiler. No manual rewrite of the Strand/Fortran program is required. The resulting program can be executed on a variety of MIMD computers. In contrast, the Schedule program is portable only between shared-memory computers.

We would like to emphasize that the intention of this comparison is not to single out Schedule for particular criticism. On the contrary, Schedule seems a well-conceived and useful solution to issues of portability and reuse of existing software. However, as its authors themselves admit [6], systems of this sort are in the long term no substitute for high-level notational support for the expression of concurrent algorithms.

VII. Experiences

The techniques described in this paper have been used to develop parallel implementations of a variety of codes. One of these, a state-space search code, has already been described. Although simple, this serves to illustrate the portability of the programs developed using our techniques. The same self-scheduling program has been executed on hypercube computers, shared-memory computers, and nonuniform memory access computers without modification.

We also have experience with more complex applications. For example, a genetic sequence alignment program provided to us by Overbeek and his colleagues at Argonne National Laboratory comprises 1200 lines of Strand code and 3000 lines of C [3]. A simplification of the top-level structure of this program has been presented in Fig. 7. A partitioning similar to that specified in Fig. 8 gave a speedup of 8.5 times on an 11-processor Encore Multimax, with a moderate-sized problem. Another application, a secondary structure prediction code which invokes the alignment code with a Monte Carlo algorithm, was parallelized with four control statements and achieved speedups of over 18 on a 20-processor Sequent Symmetry. In both cases, one processor was dedicated to the scheduler. Experimentation with these applications has been confined to shared-memory machines because C procedures called by the concurrent component create temporary data structures that are too large for the distributed-memory machines to which we have access. This problem is being addressed in a rewrite currently in progress.

Our experience with these and other applications motivates the following observations. First, self-scheduling programs can often be generated with little effort. In the computational biology codes, effective parallel programs were obtained by the addition of 4–6 control statements and the application of our preprocessor. As the original sources comprised over 4000 lines, this represents an insignificant incremental effort.

Second, automatic development of self-scheduling versions of programs represents a considerable saving in time. A comparison of Figs. 7 and 11 shows that the self-scheduling version of even a simple program can be complex. Manual development of a self-scheduling version of the complete alignment code (1200 lines) takes several hours and is a tedious and error-prone activity. This would be serious enough if our concern was simply to develop a single self-scheduling version. Yet our experience suggests that it is frequently useful to be able to explore several alternative partitioning strategies. Furthermore, programs are typically not static entities but evolve over time. It is much more convenient to modify a program in its nonself-scheduling form.

VIII. Conclusions

We have described techniques to support automatic generation of self-scheduling parallel programs. A high-level programming notation (e.g., Strand) is used to organize the concurrent execution of sequential components expressed in languages such as Fortran and C. Simple compilation techniques translate an application program into a form suitable for linking with a scheduler library. The resulting program is capable of scheduling its own execution on a parallel computer. The compiler is directed by control statements that specify a partition and data dependencies between tasks in the partition. Control statements may be generated automatically or manually. Programs generated by the preprocessor do not require access to any low-level mechanisms, beyond simple mechanisms used by Strand implementations, and hence are portable in principle to any MIMD computer.

Experience shows that the approach can greatly simplify the task of developing parallel programs. The incremental effort required to specify partitioning and data dependency information required for parallel execution is generally small. In two moderate-sized programs discussed in the paper, control statements totaled less than 0.2% of total code lines. The separation of concerns that is achieved between concurrent algorithm and load-balancing strategy reduces program complexity and encourages exploration of alternative scheduling strategies.

Program analysis tools can usefully complement our techniques by automatically generating the control statements required by our system. For example, we have used granularity analysis techniques to generate partitions. However, we are not convinced that automatic generation of control statements is either necessary or advantageous. Manual generation of control statements is rarely difficult. Furthermore, we believe that a programmer will frequently be able to exploit domain-specific knowledge and specify a better partition than could be achieved by purely automatic means.

The compilation techniques described in this paper constitute what is termed in [8] an *algorithmic motif:* a library program and associated compiler that together define a useful parallel programming abstraction. The scheduler motif applies only to the class of programs for which it is possible and useful to define directed partitions. Other motifs are required for programs that do not fit in this class: for example, those based on software pipelines. The design and evaluation of alternative motifs are topics of current research.

Acknowledgment

W. Winsborough's comments greatly improved the presentation in Section II-D.

References

[1] R. Babb, "Parallel processing with large grain data flow techniques," *IEEE Comput. Mag.,* vol. 17, no. 7, pp. 55–61, 1984.

[2] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens, *Portable Programs for Parallel Processors.* New York: Holt, Rinehart, and Winston, 1987.

[3] R. Butler, T. Butler, I. Foster, N. Karonis, R. Olson, R. Overbeek, N. Pfluger, M. Price, and S. Tuecke, "Aligning genetic sequences," in *Strand: New Concepts in Parallel Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1989, pp. 253–271.

[4] K. M. Chandy and S. Taylor, "Composing parallel programs," in *Beauty is our Business,* New York: Springer-Verlag, 1990.

[5] K. Clark and S. Gregory, "A relational language for parallel programming," in *Proc. 1981 ACM Conf. Functional Programming Languages Comput. Architectures,* ACM, 1981, pp. 171–178.

[6] J. Dongarra and D. Sorenson, "A portable environment for developing parallel Fortran programs," *Parallel Comput.,* vol. 5, pp. 175–186, 1987.

[7] I. Foster and R. Overbeek, "Experiences with bilingual parallel programming," in *Proc. 5th Distributed Memory Comput. Conf.,* 1990.

[8] I. Foster and R. Stevens, "Parallel programming with algorithmic motifs," in *Proc. 1990 Int. Conf. Parallel Processing,* Penn. State Univ. Press, 1990.

[9] I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

[10] ——, "Strand: A practical parallel programming tool," in *Proc. North Amer. Conf. Logic Programming.* MIT Press, 1989, pp. 497–512.

[11] R. Gerth, M. Codish, Y. Lichtenstein, and E. Shapiro, "Fully abstract denotational semantics for Flat Concurrent Prolog," in *Proc. Symp. Logic Comput. Sci.,* 1988.

[12] E. Horowitz and A. Zorat, "Divide-and-conquer for parallel processing," *IEEE Trans. Comput.,* vol. C-32, no. 6, pp. 582–585, 1983.

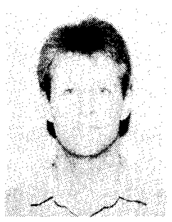
[13] R. Keller and F. Lin, "Simulated performance of a reduction based multiprocessor," *IEEE Comput. Mag.,* vol. 17, no. 7, pp. 70–82, 1984.

[14] E. Lusk, *et al.,* "The Aurora Or-parallel Prolog system," in *Proc. Fifth Generation Comput. Syst. Conf.,* Tokyo, 1988, pp. 819–830.

[15] C. Polychronopoulos, *Parallel Programming and Compilers.* Boston, MA: Kluwer Academic, 1988.

[16] S. Taylor, *Parallel Logic Programming Techniques.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

[17] S. Thakkar, P. Gifford, and G. Fieland, "The Balance multiprocessor system," *IEEE Micro,* Feb. 1988.



Ian Foster received the B.Sc. (Hons I) degree from the University of Canterbury, Christchurch, New Zealand, and the Ph.D. degree from Imperial College, London, both in computer science.

Since 1989 he has been a member of the Research Staff in the Mathematics and Computer Science Division at Argonne National Laboratory, Chicago, IL. His research interests include parallel languages, algorithms, and programming methodologies.