Compositional Parallel Programming Languages

IAN FOSTER

Argonne National Laboratory

Author's address: I. Foster, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439.

In task-parallel programs, diverse activities can take place concurrently, and communication and synchronization patterns are complex and not easily predictable. Previous work has identified *compositionality* as an important design principle for task-parallel programs. In this paper, we discuss alternative approaches to the realization of this principle, which holds that properties of program components should be preserved when those components are composed in parallel with other program components. We review two programming languages, Strand and Program Composition Notation, that support compositionality via a small number of simple concepts, namely monotone operations on shared objects, a uniform addressing mechanism, and parallel composition. Both languages have been used extensively for large-scale application development, allowing us to provide an informed assessment of their strengths and weaknesses. We observe that while compositionality simplifies development of complex applications, the use of specialized languages hinders reuse of existing code and tools, and the specification of domain decomposition strategies. This suggests an alternative approach based on small extensions to existing sequential languages. We conclude the paper with a discussion of two languages that realize this strategy.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms: Languages

Additional Key Words and Phrases: Compositionality, Parallel Languages, Parallel Programming

---

1. INTRODUCTION

Parallel programming is widely regarded as difficult: more difficult than sequential programming, and perhaps (at least this is our view) more difficult than it needs to be. In addition to the normal programming concerns, the parallel programmer has to deal with the added complexity brought about by multiple threads of control: managing their creation and destruction, and orchestrating their interactions via synchronization and communication. Parallel programs must also manage a richer set of resources than sequential programs, controlling, for example, the mapping and scheduling of computation onto multiple processors.

As in sequential programming, complexity in program development can be managed by providing appropriate programming language constructs. Language constructs can help both by supporting encapsulation, so as to prevent unwanted interactions between program components, and by providing higher-level abstractions that leverage programmer effort by allowing compilers to handle mundane, error-prone aspects of parallel program implementation. For example, the various languages that have been developed to support data-parallel programming achieve both these goals, albeit for a restricted class of programs [Chapman et al. 1992; Fox et al. 1990; Koelbel et al. 1994]. Data-parallel programs exploit the parallelism inherent in applying the same operation to all or most elements of large data struc-

tures. Data-parallel languages avoid unwanted interactions by enforcing sequential semantics. They use data distribution statements to provide a high-level, abstract syntax for specifying data placement, freeing the programmer from the labor of partitioning computation and translating between global and local addresses.

Our research goal is to develop language constructs and associated tools to support the more general class of *task-parallel* applications, in which multiple unrelated activities can take place concurrently. Task parallelism arises in time-dependent problems such as discrete-event simulation, in irregular problems such as those involving sparse matrices, and in multidisciplinary simulations coupling multiple, possibly data-parallel, computations. The challenge when developing language constructs for task-parallel programming is to provide the modularity and abstraction needed for ease of programming while maintaining the generality needed to support arbitrary parallel computations.

Compositionality has been proposed as a design principle for task-parallel programs [Chandy and Taylor 1991]. A compositional programming system is one in which properties of program components are preserved when those components are composed in parallel with other program components. That is, the behavior of the whole is a logical combination of the behavior of the parts. One property that we often want to be preserved in this way is determinism, so that programs constructed

from deterministic components can themselves be guaranteed to be deterministic. Compositionality can simplify program development by allowing program components to be developed and tested in isolation and then reused in any environment. However, not all parallel programming notations have this property. For example, notations based on shared variables tend not to be compositional: two deterministic procedures may be nondeterministic when executed concurrently, if both access the same variable.

In this paper, we describe various language constructs that have been proposed to support compositionality. We first use the examples of Strand [Foster and Taylor 1990] and PCN [Chandy and Taylor 1991] to show how the basic ideas of compositional programming can be supported by using a small number of simple concepts, namely, monotone operations on shared objects, a uniform addressing mechanism, and parallel composition. Then, we use the large body of practical experience that has been gained from the use of Strand and PCN to evaluate the approach. We identify specific strengths and distinguish those associated with compositionality from those due to other language features. We also identify key weaknesses and note that these are, for the most part, associated with the use of specialized languages. This observation motivates us to consider extensions to existing sequential languages as a means of providing a more flexible and accessible implementation of the ideas.

We develop requirements that we believe such language extensions should satisfy, and we review two languages—Compositional C++ [Chandy and Kesselman 1993] and Fortran M [Chandy and Foster 1995]—that meet these requirements.

2. STRAND

One particularly elegant and satisfying approach to compositional task-parallel programming is to define a simple language that provides just the essential elements required to support this programming style [Foster et al. 1990]. This language can be used both as a language in its own right and as a *coordination language* [Carriero and Gelernter 1989; Cole 1989; Kelly 1989; Lucco and Sharp 1991] providing a parallel superstructure for existing sequential code. These dual roles require a simple, uniform, highly parallel programming system in which

—the structure of the computation, the number of concurrently executing threads of control, and the placement of these threads can vary dynamically during program execution;

—communication and synchronization operations are introduced into a program via high-level abstractions that can be implemented efficiently by the language compiler;

—patterns of communication can change dynamically;

—the functional behavior of parallel program modules is independent of the scheduling or processor allocation strategy used;

—arbitrary parallel modules can be combined and will function correctly; and

—modules written in other languages can be incorporated.

These goals motivate the design of the Strand, PCN, CC++, and Fortran M languages described below.

2.1 Strand Design

The Strand language integrates ideas from earlier work in parallel logic programming [Clark and Gregory 1981], dataflow computing [Ackerman 1982], and imperative programming [Hoare 1978] to provide a simple task-parallel programming language based on four related ideas:

—single-assignment variables,

—a global, shared namespace,

—parallel composition as the only method of program composition, and

—a foreign language interface.

Single-assignment variables provide a unified mechanism for both synchronization and communication. All variables in Strand follow the single-assignment rule [Ackerman 1982]: a variable is set at most once and subsequently cannot change. Any

attempt by a program component to read a variable before it has been assigned a value will cause the program component to block. All synchronization operations are implemented via reading and writing these variables. New variables can be introduced by writing recursive procedure definitions.

Strand variables also define a global namespace. A variable can refer to any object in the computation, even another variable. The location of the variable or object being referenced does not matter. Thus, Strand does not require explicit communication operations; processes can communicate simply by reading and writing shared variables.

Unlike most programming languages, which support only the sequential composition of program components, Strand supports only parallel composition. A parallel composition of program components executes as a *concurrent interleaving* of the components, with execution order constrained only by availability of data, as determined by the single-assignment rule.

The combination of single-assignment variables, a global namespace, and parallel composition means that the behavior of a Strand program is invariant to the placement and scheduling of computations. One consequence of this invariance is that Strand programs are compositional: a program component will function correctly in any environment. Another consequence is that the specification of the location

of a computation is orthogonal to the specification of the computation. To exploit these features, Strand provides a mapping operator that allows the programmer to control the placement of a computation on a parallel computer.

By allowing modules written in sequential languages to be integrated into Strand computations, the foreign language interface supports the use of Strand as a coordination language. Sequential modules that are to be integrated in this way must implement pure functions. The interface supports communication between foreign modules and Strand by providing routines that allow foreign language modules to access Strand variables passed as arguments.

2.2 Strand Language

This summary of Strand language concepts is not intended to be comprehensive; for details, see [Foster and Taylor 1990]. The syntax of Strand is similar to that of the logic programming language Prolog. A program consists of a set of procedures, each defined by one or more *rules*. A rule has the general form

$$H \text{ :- } G_1,\ G_2,\ ...,\ G_m \mid B_1,\ B_2,\ ...,\ B_n. \qquad m, n \geq 0,$$

where the rule head H is a function prototype consisting of a name and zero or more arguments, the G_i are guard tests, "|" is the commit operator, and the B_j are body processes: calls to Strand, C, or Fortran procedures, or to the assignment

operator ":=". If $m = 0$, the "|" is omitted. Procedure arguments may be variables (distinguished by an initial capital letter), strings, numbers, or lists. A list is a record structure with a *head* and a *tail* and is denoted [*head*|*tail*].

A procedure's rules define the actions that the process executing that procedure can perform. The head and guard of the rule define the conditions under which an action can take place; the body defines the actions that are to be performed. When a procedure executes, the conditions defined by the various heads and guards are evaluated in parallel. Nonvariable terms in a rule head must match corresponding process arguments, and guard tests must succeed. If the conditions specified by a single rule hold, this rule is selected for execution, and new processes are created for the procedures in its body. If two or more rules could apply, one is selected nondeterministically. It suffices to ensure that conditions are mutually exclusive to avoid nondeterministic execution. If no condition holds, an error is signaled. For example, the following procedure defines a `consumer` process that executes either `action1` or `action2`, depending on the value of variable `X`.

```
consumer(X) :- X ==  "msg" | action1(X).

consumer(X) :- X =\= "msg" | action2(X).
```

In this procedure, `X` is a variable, `"msg"` is a string, and `==` and `=\=` represent equality and inequality tests, respectively. Notice that this procedure is deterministic.

2.2.1 *Communication and Synchronization.* As noted above, all Strand variables are single-assignment variables. A shared single-assignment variable can be used both to communicate values and to synchronize actions. For example, consider concurrently executing `producer` and `consumer` processes that share a variable `X`:

```
producer(X), consumer(X)
```

The producer may assign a value to `X` (e.g., `"msg"`) and thus communicate this value to the consumer:

```
producer(X) :- X := "msg".
```

As shown above, the `consumer` procedure may receive the value and use it in subsequent computation. The concept of synchronization is implicit in this model. The comparisons `X == "msg"` and `X =\= "msg"` can be made only if the variable `X` is defined. Hence, execution of `consumer` is delayed until `producer` executes and makes the value available.

The single-assignment variable would have limited utility in parallel programming if it could be used to exchange only a single value. In fact, processes that share a variable can use it to communicate a sequence or *stream* of values. This technique is achieved as follows. A recursively defined producer process incrementally constructs a list structure containing these values. A recursively defined

consumer process incrementally reads this same structure. Figure 1 illustrates this technique. The `stream_comm` procedure creates two processes, `stream_producer` and `stream_consumer`, that use the shared variable `X` to exchange `N` values. The producer incrementally defines `X` to be a list comprising `N` occurrences of the number `10`:

```
[10, 10, 10, ..., 10]
```

The statement `Out := [10|Out1]`, which defines the variable `Out` to be a list with head `10` and tail `Out1`, can be thought of as sending a message on `Out`. The new variable `Out1` is passed to the recursive call to `stream_producer`, which either uses it to communicate additional values or, if `N==0`, defines it to be the empty list `[]`.

The consumer incrementally reads the list `S`, adding each value received to the accumulator `Sum` and printing the total when it reaches the end of the list. The match operation `[Val|In1]` in the head of the first `stream_consumer` rule determines whether the variable shared with `stream_producer` is a list and, if so, decomposes it into a head `Val` and tail `In1`. This operation can be thought of as receiving the message `Val` and defining a new variable `In1` that can be used to receive additional messages.

2.2.2 *Foreign Interface.* "Foreign" procedures written in C or Fortran can be called in the body of a rule. A foreign procedure call suspends until all argu-

```
stream_comm(N) :-
    stream_producer(N, S),          % N is number of messages
    stream_consumer(0, S).          % Accumulator initially 0

stream_producer(N, Out) :-
  N > 0 |                           % More to send (N > 0):
    Out := [10|Out1],               %    Send message "10";
    N1 is N - 1,                    %    Decrement count;
    stream_producer(N1, Out1).      %    Recurse for more.
stream_producer(0, Out) :-          % Done sending (N == 0):
    Out := [].                      %    Terminate output.

stream_consumer(Sum, [Val|In1]) :-  % Receive message:
    Sum1 is Sum + Val,              %    Add to accumulator;
    stream_consumer(Sum1, In1).     %    Recurse for more.
stream_consumer(Sum, []) :-         % End of list (In == []):
    print(Sum).                     %    Print result.
```

Fig. 1. Producer/Consumer Program

ments are defined and then executes atomically, without suspension. This approach achieves a clean separation of concerns between sequential and parallel programming, provides a familiar notation for sequential concepts, and enables existing sequential code to be reused in parallel programs.

2.2.3 *Mapping.* The Strand compiler does not attempt to map processes to processors automatically. Instead, the Strand language provides constructs that allow mapping strategies to be specified by the programmer. This approach is possible because the Strand language is designed so that mapping affects only performance, not correctness. Hence, a programmer can first develop a program and then explore alternative mapping strategies by changing annotations. This technique is

```
augcgagucuauggcuucggccauggcggacggcucauu
augcgagucuaugguuucggccauggcggacggcucauu
augcgagucuauggacuucggccauggcggacggcucagu
augcgagucaaggggcucccuuggggcaccggcgcacggcucagu
```

(a)

```
augcgagucuauggc----uucg----gccauggcggacggcucauu
augcgagucuauggu----uucg----gccauggcggacggcucauu
augcgagucuauggac---uucg----gccauggcggacggcucagu
augcgaguc-aaggggcucccuuggggcaccggcgcacggcucagu
```

(b)

Fig. 2. RNA Sequence Alignment

illustrated below.

2.3 Programming Examples

We use a sequence alignment program developed by Ross Overbeek and his coworkers [Butler et al. 1989] to illustrate the use of Strand. The goal is to line up RNA sequences from separate but closely related organisms, with corresponding sections directly above one another and with *indels* (dashes) representing areas in which characters must be inserted or deleted to achieve this alignment. For example, Figure 2 shows (a) a set of four short RNA sequences and (b) an alignment of these sequences.

Overbeek et al.'s alignment algorithm uses a divide-and-conquer strategy that, in simplified terms, works as follows. First, "critical points"—short subsequences that are unique within a sequence—are identified for each sequence. Second, "pins"—

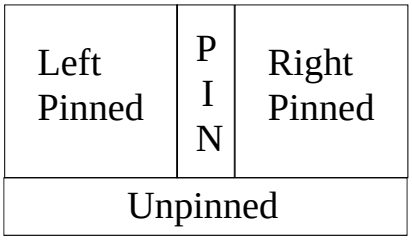

Fig. 3. Splitting Sequences Using a Pin

critical points that are common to several sequences—are identified. Third, the longest pin is used to partition the problem of aligning the sequences into three smaller alignment problems, corresponding to (a) the subsequences to the left of the pin in the pinned sequences, (b) the subsequences to the right of the pin, and (c) the unpinned sequences (Figure 3). Fourth, these three subproblems are solved by applying the alignment algorithm in a recursive fashion. Fifth, the three subalignments are combined to produce a complete alignment.

This is a complex algorithm that happens to exhibit many opportunities for parallel execution. For example, critical points can be computed in parallel for each sequence, and each alignment subproblem produced during the recursive application of the algorithm can be solved concurrently. The challenge is to formulate this algorithm in a way that does not obscure the basic algorithm structure and that allows alternative parallel execution strategies to be explored without substantial changes to the program. The Strand implementation achieves this goal. The pro-

```
align_chunk(Sequences,Alignment) :-
    pins(Chunks,BestPin),
    divide(Sequences,BestPin,Alignment).

pins(Chunk,BestPin) :-
    cps(Chunk,CpList),
    c_form_pins(CpList,PinList),
    best_pin(Chunk,PinList,BestPin).

cps([Seq|Sequences],CpList) :-
    CpList := [CPs|CpList1],
    c_critical_points(Seq,CPs),
    cps(Sequences,CpList1).
cps([],CpList) :- CpList := [].

divide(Seqs,Pin,Alignment) :-
  Pin =\= [] |
    split(Seqs,Pin,Left,Right,Rest),
    align_chunk(Left,LAlign) @ random,
    align_chunk(Right,RAlign) @ random,
    align_chunk(Rest,RestAlign) @ random,
    combine(LAlign,RAlign,RestAlign,Alignment).
divide(Seqs,[],Alignment) :-
    c_basic_align(Seqs,Alignment).
```

Fig. 4. Genetic Sequence Alignment Algorithm

cedures in Figure 4 implement the top level of the algorithm. The `align_chunk` procedure calls `pins` to compute critical points for each sequence in a set of sequences (a "chunk"), form a set of pins, and select the best pin. If a pin is found (`Pin =\= []`), `divide` uses it to split the chunk into three subchunks. Recursive calls to `align_chunk` align the subchunks. If no pin is found (`Pin == []`), an alternative procedure, `c_basic_align`, is executed.

This example illustrates three important characteristics of the Strand language.

First, programs can exploit high-level logic programming features to simplify the specification of complex algorithms. These features include the use of list structures to manage collections of data and a rule-based syntax that provides a declarative reading for program components. Second, programs can call routines written in sequential languages to perform operations that are most naturally expressed in terms of imperative operations on arrays. In the example, three C-language procedures (distinguished here by a "`c_`" prefix) are called in this way. This multilingual programming style permits rapid prototyping of algorithms without compromising performance. (The absence of an array data type means that code for manipulating arrays would be both clumsy and inefficient if written in Strand.) Third, alternative parallel implementation strategies can be explored simply by annotating the program text with different process mapping directives. For example, in Figure 4 annotations `@ random` are placed on the recursive calls to `align` to specify that these calls are to execute on randomly selected processors. Alternatively, annotations `@ elsewhere` could be used to specify that these calls are to be scheduled to idle processors by using a load-balancing strategy. As communication and synchronization are specified in terms of operations on shared single-assignment variables, no other change to the program text is required: the Strand compiler translates these operations into either low-level message-passing or shared-data access operations,

as required.

A second example illustrates the use of Strand to implement distributed algorithms. Figure 5 provides a complete implementation of a manager/worker load-balancing scheduler. As illustrated in Figure 6, request streams from different "worker" processes (`W`) are combined by a special process called a merger to yield a single stream. (The merger is Strand's second nondeterministic construct, the first being guards that are not mutually exclusive.) Each worker repeatedly sends a request for a task, waits for a response, executes the task that it receives, and terminates when no more tasks are available. A "manager" process (`M`) matches requests with tasks received on a separate stream, and signals termination when all tasks have been scheduled.

Mapping constructs are used to control the placement of worker processes on physical processors. The first statement in the program indicates that the programmer wants to think of the computer as a virtual ring. The ring virtual computer supports mapping annotations `@ fwd` and `@ bwd`, which specify that a process is to execute on the "next" or "previous" node in this ring, respectively. In the example, the recursive call in the `workers` procedure is annotated so that the `worker` processes are placed on successive virtual processors.

```
-machine(ring).                          % Virtual computer.

scheduler(NumWorkers, Tasks) :-          % Create processes:
    manager(Tasks, Requests),            %     Manager;
    merger(Reqs,Requests),               %     Merger;
    workers(Reqs).                       %     Workers.

manager([Task|Tasks], [Req|Requests]) :- % Serve request.
    Req := Task, manager(Tasks,Requests).
manager([], [Req|Requests]) :-           % Signal done.
    Req := "halt", manager([],Requests).
manager([],[]).                          % Terminate.


workers(NumWorkers,Reqs) :-              % Create workers, each
  NumWorkers > 0 |                       % on different node.
    NumWorkers1 is NumWorkers - 1,
    Reqs := [merge(R)|Reqs1],            % Register with merger.
    worker(R),                           % Create worker; then
    workers(NumWorkers1,Reqs1) @ fwd.    % move to next node.
workers(0,Reqs) :- Reqs := [].

worker(Reqs) :-                          % Worker:
    Reqs := [Request|Reqs1],             %     Request task;
    worker1(Reqs1,Request,"done").       %     Process task.

worker1(Reqs,Request,"done") :-          % Process task.
  Request =\= "halt" |                   %     Not halt; so:
    Reqs := [NewReq|Reqs1],              %     Request next task;
    execute(Request,Done),               %     Execute task;
    worker1(Reqs1,NewReq,Done).          %     Repeat process.
worker1(Reqs,"halt","done") :- Reqs := [].
```

Fig. 5. Load-Balancing Library

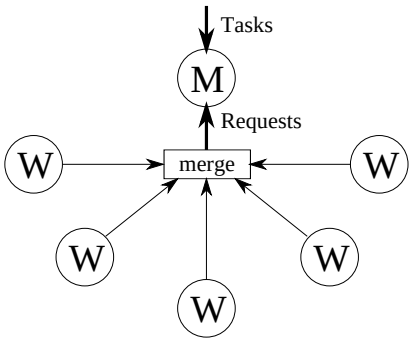

Fig. 6. Manager/Worker Scheduler Structure

2.4 Strand Toolkit

A small toolkit provides the essential utilities required for parallel application development. This comprises a compiler and runtime system, a linker for foreign code, a debugger, a parallel I/O library, and a performance profiler.

The compiler translates Strand programs into the instruction set of an abstract machine. A runtime system implements this abstract machine and provides communication, thread management, and memory management functions. Its implementation is designed for portability and is easily retargeted to new computers. The compiler and runtime system are designed to optimize the performance of programs that create many lightweight processes and that communicate by using recursive stream structures. For example, tail-recursion optimizations are applied to translate recursion into iteration and to reuse storage occupied by list cells,

hence avoiding the need for garbage collection in certain common cases. A garbage collector is nevertheless required in the general case. On distributed-memory computers, a shared variable is represented by a single occurrence and one or more *remote references* [Taylor 1989]; read and write operations on remote references are translated into communication operations. The garbage collector must also trace these interprocessor references; however, individual processors can reclaim storage independently, hence avoiding a need for global synchronization [Foster 1989].

The foreign code linker allows the programmer to define the data conversions that are to be performed when moving data between Strand, C, and Fortran; the linker generates the necessary conversion code. The debugger allows the programmer to trace program execution and to examine suspended processes in the event of deadlock.

Performance monitoring functions are integrated into the compiler and programming system [Kesselman 1991]. These functions allow information such as total procedure execution time, procedure execution frequencies, and communication volumes to be obtained on a per-processor basis. This information is collected by additional instructions inserted by the compiler; the cost of these instructions is almost always much less than 1 percent of total execution time [Kesselman 1991]. Since profiling is based on counters, rather than the logging of events, the amount

of data collected is independent of program execution time. Communication is required only upon program termination, to dump profile data collected on each processor. A graphical analysis tool called Gauge permits interactive exploration of this data.

2.5 Strand Critique

Unlike many parallel programming systems developed in a research environment, Strand has been used extensively for application development in areas as diverse as computational biology [Butler et al. 1989], discrete event simulation [Xu and Turner 1990], telephone exchange control [Armstrong and Virding 1989], automated theorem proving, and weather modeling. This work provides a broad base of practical experience on which we can draw when evaluating the strengths and weaknesses of the Strand approach. Analysis of this experience indicates three particular strengths of the Strand constructs:

—The use of parallel composition and a high-level, uniform communication abstraction simplifies development of task-parallel applications featuring dynamic creation and deletion of threads, complex scheduling algorithms, and dynamic communication patterns. Complex distributed algorithms can often be expressed in a few lines of code using Strand constructs.

—Parallel composition and single-assignment variables also enforce and expose the benefits of a compositional programming model. Program development, testing, and debugging, and the reuse of program components are simplified.

—The recursively defined data structures and rule-based syntax that Strand borrows from logic programming are useful when implementing symbolic applications, for example in computational biology.

This same analysis also reveals four significant weaknesses that limit the utility of the Strand system, particularly for larger scientific and engineering applications.

—While the use of a separate coordination language for parallel computation is conceptually economical, it is not universally popular. Writing even a simple program requires that a programmer learn a completely new language, and the logic-based syntax is unfamiliar to many.

—The foreign language interface is often too restrictive for programmers intent on reusing existing sequential code in a parallel framework. In particular, it is difficult to convert sequential code into single program/multiple data (SPMD) libraries, since this typically requires the ability to embed parallel constructs in existing sequential code, something that Strand does not support. As a consequence, combining existing program modules with Strand can require significant restructuring of those modules.

—The Strand abstractions provide little assistance to the programmer applying domain decomposition techniques to regular data structures. In these applications, the principal difficulties facing the programmer are not thread management or scheduling, but translating between local and global addresses, problems that have been addressed in data-parallel languages.

—The use of a new language means that program development tools such as debuggers and execution profilers have to be developed from scratch; it also hinders the application of existing sequential development tools to sequential code modules.

3. PROGRAM COMPOSITION NOTATION

Motivated in part by experiences with Strand, Program Composition Notation (PCN) was developed at Caltech and Argonne [Chandy and Taylor 1991; Foster et al. 1992]. This second-generation compositional language extends the basic Strand ideas of lightweight processes, logical variables, declarative programming, and multilingual programming in three ways. First, it integrates declarative and imperative programming without compromising compositional properties. Second, it provides a richer and more flexible syntax. Third, it supports the implementation and use of reusable parallel modules.

3.1 PCN Language

PCN syntax is similar to that of the C programming language. A program is a set of procedures, each with the following general form ($k, l \geq 0$).

```
name(arg_1,...,arg_k)
declaration_1; ...; declaration_l;
block
```

A `block` is a call to a PCN procedure (or to a procedure in a sequential language such as Fortran or C), a composition, or a primitive operation such as assignment. A composition is written {`op block`$_1$`, ..., block`$_m$}, $m > 0$, where `op` is one of "||" (parallel), ";" (sequential), or "?" (choice), indicating that the blocks `block`$_1$, ..., `block`$_m$ are to be executed concurrently, in sequence, or as a set of guarded commands, respectively. In the latter case, each block is a *choice* with the form `guard -> block`, where `guard` is a conjunction of boolean tests and `block` can be executed only if `guard` evaluates to true. If two or more guards evaluate to true, one is selected nondeterministically, as in Strand.

A parallel composition specifies opportunities for parallel execution but does not indicate how the composed blocks (which can be thought of as lightweight processes) are to be mapped to processors. As in Strand, mapping is specified by annotations.

```
stream_comm(n)
{|| stream_producer(n,x),                    % Execute in parallel
    stream_consumer(x)
}

stream_producer(n,out)
{ ? n > 0 ->                                 % If n > 0:
    {|| out = [10|out1],                     % Send message;
       stream_producer(n-1, out1)            % Recurse for more.
    },
    n == 0 -> out = []                       % If n == 0: stop
}

stream_consumer(sum, in)
{ ? in ?= [val|in1] ->                       % If message: receive;
       stream_consumer(sum+val,in1),         % Recurse for more.
    in ?= [] ->                              % If done:
       stdio:printf("Sum=%d\n",{sum},_)      % Print sum.
}
```

Fig. 7. PCN Producer/Consumer

In PCN, annotations can name arbitrary user-defined functions.

Any Strand program can be rewritten directly as a PCN program that uses only parallel composition, choice composition, and single-assignment variables and that uses PCN's definition statement ("=") in place of Strand's assignment statement (":="). For example, Figures 7 and 8 are direct translations of Figures 1 and 4.

3.1.1 *Imperative Constructs.* PCN programs can also use imperative constructs. Conventional, or *mutable*, scalar and array variables of type integer, double-precision real, and character can be created. (These are distinguished from single-assignment variables by the fact that they are explicitly declared; single-assignment variables

```
align_chunk(sequences,alignment)
{|| pins(chunks,bestpin),
    divide(sequences,bestpin,alignment)
}

pins(chunk,bestpin)
{|| cps(chunk,cplist),
    c_form_pins(cplist,pinlist),
    best_pin(chunk,pinlist,bestpin)
}

cps(sequences,cplist)
{ ? sequences ?= [seq|sequences1] ->
    {|| cplist = [cps|cplist1],
        c_critical_points(seq,cps),
        cps(sequences1,cplist1)
    },
    sequences ?= [] -> cplist = []
}

divide(seqs,pin,alignment)
{ ? pin != [] ->
    {|| split(seqs,pin,left,right,rest),
        align_chunk(left,lalign),
        align_chunk(right,ralign),
        align_chunk(rest,restalign),
        combine(lalign,ralign,restalign,alignment)
    },
    pin == [] ->
        c_basic_align(seqs,alignment)
}
```

Fig. 8. PCN Version of Figure 4

are not. Hence, a variable declaration serves not only to indicate the type of the variable, but also the fact that the variable is mutable.) Mutable variables, like variables in C or Fortran, have an initial arbitrary value that can be modified many times by using an assignment statement (":="). For example, Figure 9 shows PCN, C, and Fortran programs for computing the inner product of two double-precision arrays `array1` and `array2`. All assume that their arguments are passed by reference and use an iteration statement to accumulate the values `array1[i]*array2[i]` in the mutable variable `sum`.

The three procedures in Figure 9 can be called interchangeably by PCN programs. PCN semantics ensure that updates to mutable variables within `inner_product` do not result in race conditions in a parallel program. In particular, they prohibit updates to mutable variables shared by processes in a parallel block, and require the compiler to *copy* the value of mutables and definitions when they occur on the right-hand side of definition and assignment statements, respectively. In this way, the two worlds of parallel/declarative and sequential/imperative programming are able to coexist without the possibility of nondeterministic interactions [Chandy and Taylor 1991].

Figure 10 shows a program that receives arrays of double-precision values `a1` and `a2` on two input streams `in1` and `in2`, calls one of the `inner_product` routines

```
inner_product(n,array1,array2,sum)
double sum;
{ ; sum := 0.0,
    { ; i over 0..n-1 ::
            sum := sum + array1[i]*array2[i]
    }
}

inner_product(n,array1,array2,sum)
int *n;
double array1[], array2[], *sum;
{  int i;
   *sum = 0.0;
   for(i=0; i<*n; i++)
       *sum = *sum + array1[i]*array2[i];
}

SUBROUTINE INNER_PRODUCT(N,ARRAY1,ARRAY2,SUM)
INTEGER N
DOUBLE PRECISION ARRAY1(N), ARRAY2(N), SUM
INTEGER I
SUM = 0.0
DO I=1,N
   SUM = SUM + ARRAY1(I)*ARRAY2(I)
ENDDO
END
```

Fig. 9. Inner Product in PCN, C, and Fortran

```
f(in1,in2,out)
double sum;
{ ? in1 ?= [a1|in1a], in2 ?= [a2|in2a] ->
    { ? length(a1) == length(a2) ->
        { ; inner_product(length(a1),a1,a2,sum),
            out = [sum|out1],
            f(in1a,in2a,out1)
        },
        default ->
        {|| out = ["error"|out1],
            f(in1a,in2a,out1)
        }
    },
    default -> out = []
}
```

Fig. 10. PCN Program That Calls Inner Product

to compute the inner product, and sends the result (`sum`) on an output stream `out`. Notice that the mutable variable `sum` is used only within a sequential block. Furthermore, the compiler makes a copy of `sum` when creating the list structure `[sum|out1]`, hence ensuring that the process that receives the message `out` sees a single-assignment value.

3.1.2 *Modules and Templates.* PCN supports the application of modular programming techniques. A PCN process can encapsulate subprocesses and internal communication channels but need not encapsulate processor numbers or other physical names. Hence, a process can be thought of as a module and can be reused easily in different circumstances. A module may also be parameterized with the code executed at each node in a parallel structure, in which case we call it a *template*.

A distributed array of single-assignment variables (declared as an array of type "`port`") can be used as an interface, avoiding the contention that would occur if processes interacted via a centralized data structure [Foster et al. 1992]. PCN programmers regularly reuse modules and templates implementing parallel program structures such as pipelines and butterflies, distributed data structures such as arrays and dictionaries, and load balancing algorithms.

The PCN procedure `module_example` in Figure 11 composes a ring-pipeline template (`ring`), a reduction module (`maximum`), and an output module (`display`). Each module is parameterized with the number of processors on which it is to execute (`n`) and defines its own internal process and communication structure. As illustrated in Figure 12, the modules interact via distributed arrays of single-assignment variables `p1` and `p2`.

Figure 11 also shows an implementation of the `ring` template and a function prototype for the `ringnode` procedure invoked by this template in `module_example`. The syntax "`{|| i over 0..n-1 ::`" is a parallel enumerator, used here to create `n` instances of the process with name given by the variable `op` (the backquotes denote a higher-order call). As in Strand, a mapping annotation (`@ node(i)`) is used to indicate the processor on which each process is to execute. Each process is passed five variables as arguments: a threshold value and communication streams from the

```
module\_example(n, threshold)
port p1[n], p2[n];
{|| maximum(n, p1),
    ring(n, ringnode(), threshold, p1, p2),
    display(n, p2)
}

ring(n, op, threshold, I, O)
port S[n], I[], O[];
{|| i over 0..n-1 ::
        `op`(threshold,
             S[i], S[(i+1)%n], I[i], O[i]
            ) @ node(i)
}

ringnode(threshold, fr_nbr, to_nbr, in, out)
{|| ... }
```

Fig. 11. Template Use and Definition

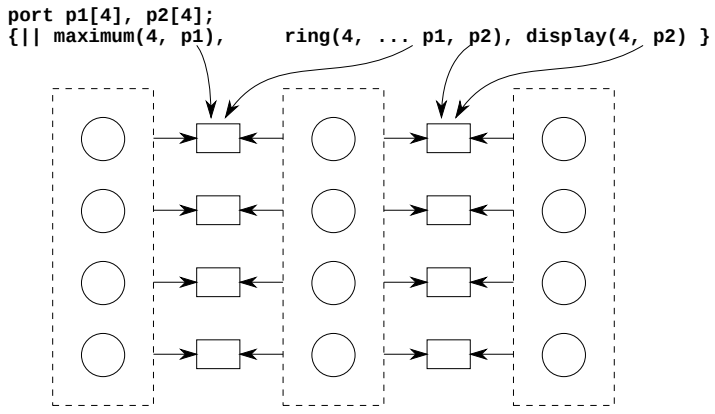

Fig. 12. An illustration of the program structures involved in the ring-pipeline example described in the text. Three modules, each comprising four processes, are composed. The circles represent processes and the squares the single-assignment variables comprising the interfaces between the modules.

left neighbor, to the right neighbor, and to and from the interface, respectively.

3.2 PCN Critique

PCN has also been used in a broad range of substantial application projects. In many cases, these have been numeric problems involving irregular, adaptive computation, distributed data structures, or reactive (data-driven) computations [Foster et al. 1992; Harrar et al. 1991]. Again, these experiences provide a solid basis for evaluation.

PCN's major contribution from a language design viewpoint was to show how a programming model based on single-assignment variables and concurrent composition could be integrated with the conventional world of "multiple-assignment" variables and sequential composition. At the implementation level, this integration was also pursued aggressively, with the result that PCN's foreign language interface was significantly more sophisticated and seamless than that used in Strand.

These various innovations certainly made the language easier to use, particularly for programming problems involving multiple languages. However, our analysis is that while PCN addressed some Strand deficiencies, these were probably not the important ones. PCN still suffers from the four essential weaknesses identified in Section 2.5.

4. LANGUAGE EXTENSIONS: CC++ AND FM

Strand and PCN have proven to be useful parallel programming languages, particularly for applications that can exploit their unique mix of declarative and imperative capabilities. As discussed above, their weaknesses appear to derive in large part from the use of a new language to express parallel computation. This observation suggests an alternative approach to compositional programming in which traditional languages, such as C++ and Fortran, are extended in ways that provides compositionality and high-level specification of communication and synchronization. (Support for symbolic applications appears less fundamental.) In principle, these language extensions can address Strand and PCN's weaknesses by providing a common framework for parallel and sequential programming and simplifying the integration of existing code. It would also be desirable for these extensions to support the specification of SPMD computations.

The design of a language extension that supports compositional parallel programming requires some analysis of what makes a programming language "compositional." Compositionality in Strand and PCN is achieved by using three mechanisms. Single-assignment variables provide both an interaction mechanism based on monotonic operations on shared state, and a uniform address space; parallel composition provides a concurrent interleaving. (State changes on single-assignment

variables are monotonic in that the value of a variable cannot be changed once written [Chandy and Kesselman 1992].) Together, these mechanisms ensure that neither the order in which program components execute nor the location of this execution affect the result computed. Other mechanisms can provide the same capabilities. For example, nonblocking send and blocking receive operations on a virtual channel data type are also monotonic and can form the basis for a compositional programming language [Chandy and Foster 1995].

These various consideration lead to the following additional design goals for compositional programming languages; these supplement those developed in Section 2.

—A language should define just a small set of new language constructs; these new constructs should be compatible with the basic concepts of the sequential base language.

—The new constructs should provide monotonic operations on shared program state, so as to support compositionality.

—The new constructs should be easily embedded in existing sequential code, so as to facilitate the development of parallel SPMD libraries.

—The language should retain support for flexible communication and synchronization structures, and a data-driven execution model.

—The language should support interoperability, both with other compositional

languages and with data-parallel languages.

In the following, we briefly review two parallel languages that adopt this language extension approach to compositional programming.

4.1 Compositional C++

Chandy and Kesselman's Compositional C++ (CC++) [Chandy and Kesselman 1993] is a general-purpose parallel programming language that extends C++ with six new keywords. It is not a purely compositional programming language. In order to guarantee compositionality, unacceptable restrictions would have to be made on the C++ constructs that are available in CC++. Thus, CC++ provides constructs that enable rather than guarantee the construction of compositional modules. In most cases, compositional modules can be obtained by following simple programming conventions [Chandy and Kesselman 1992].

CC++ provides three different mechanisms for creating threads of control: the parallel block, the parallel loop, and spawned functions. The first two have a parbegin/parend semantics, while the spawned function creates an independent thread.

As in Strand and PCN, single-assignment variables are used for synchronization. In CC++, a single-assignment variable is called a synchronization, or `sync` variable,

and is distinguished by the type modifier `sync`. A CC++ program can contain both `sync` and regular C++ variables. Programs that contain only `sync` variables are compositional. To support the development of compositional programs containing regular C++ variables, CC++ introduces atomic functions. Within an instance of a given C++ class, only one atomic function is allowed to execute at a time. The operations specified in the body of an atomic function execute without interference. Thus, an atomic function is like a monitor [Hoare 1974]. If all accesses to a shared C++ variable takes place within the body of an atomic function, then the resulting program is compositional.

The remaining aspects of C++ deal with the methods used to map computation to processors and to access data on different processors. The problem of dealing with global and static data is addressed by introducing a structure called a *processor object*, a virtual processor containing a private copy of all global and static data. Like other C++ objects, a processor object has a type declared by a class definition, encapsulates functions and data, and can be dynamically created and destroyed. Each instance of a processor object contains an address space from which regular objects can be allocated. As in Strand and PCN, the functional behavior of the program is independent of where processor objects are placed.

CC++ distinguishes between interprocessor object and intraprocessor object ref-

erences: a pointer that can refer to an object in another processor object must be declared to be `global`. Global pointers provide CC++ with a global name space, as in Strand and PCN, while also providing a two-level locality model that can be manipulated directly by a program. Dereferencing a global pointer causes an operation to take place in the processor object referenced by that global pointer. Thus in CC++, communication abstractions are provided by operations on `global` pointers, while synchronization abstractions are provided by `sync` pointers.

In summary, CC++ integrates parallel composition with sequential execution. It uses global pointers to provide a uniform global address space and `sync` variables and atomic functions to implement compositional interactions between program components.

4.2 Fortran M

Fortran M (FM) [Chandy and Foster 1995] is a small set of extensions to Fortran 77 for task-parallel programming. Although simple, the FM extensions provide the essential mechanisms required for compositional programming. Program components can encapsulate arbitrary concurrent computations and can be reused in any environment.

Concepts such as pointers and dynamic memory allocation are foreign to Fortran 77. Hence, the FM design bases its communication and synchronization con-

structs on an existing concept: file I/O. FM programs can dynamically create and destroy *processes*, single-reader/single-writer virtual files (*channels*), and multiple-writer, single-reader virtual files (*mergers*). Processes can encapsulate state and communicate by sending and receiving messages on channels and mergers; references to channels, called *ports*, can be passed as arguments or transferred between processes in messages, providing a restricted global address space.

FM processes are created by process block and process do-loop constructs with parbegin/parend semantics. Arguments passed to a process are copied in on call and back on return; common blocks are local to each process. A channel is a typed, first-in/first-out message queue with a single sender and a single receiver; the merger is similar but allows for multiple senders. FM constructs allow the programmer to control process placement by specifying the mapping of processes to *virtual computers*: arrays of virtual processors. Mapping decisions do not affect program semantics. Even complex programs can be guaranteed to be deterministic [Chandy and Foster 1995].

In summary, FM integrates parallel composition with sequential execution. It uses channels both to provide a simple form of uniform global address space and to implement compositional interactions between program components.

5. RELATED WORK

The Strand design builds on work in concurrent logic programming at Imperial College [Clark and Gregory 1981; Gregory 1987; Ringwood 1988], the Weizmann Institute [Mierowsky et al. 1985; Shapiro 1987; Taylor 1989], and elsewhere. Concurrent logic programming itself has intellectual roots in logic programming [Clocksin and Mellish 1981; Kowalski 1979], functional programming [Kahn and MacQueen 1977; McLennan 1990], guarded commands [Dijkstra 1975], and CSP [Hoare 1978]. However, Strand omits many characteristic features of logic programming languages, such as unification and backtracking, in order to focus on issues relevant to parallel programming. This strategy yields a dramatically simplified language that can be implemented efficiently on sequential and parallel computers. Strand's simplicity enables numerous compiler optimizations. In addition, Strand introduces constructs that support multilingual programming, allowing its use as a coordination language.

A promising alternative approach to achieving compositionality and determinism in parallel programs is to exploit parallelism while preserving sequential semantics. This approach is taken in parallel dataflow, logic, and functional languages, which exploit parallelism implicit in declarative specifications [Cann et al. 1990; Lusk et al. 1988; McLennan 1990]; in data-parallel languages, which exploit the paral-

lelism available when the same operation is applied to many elements of a data structure [Fox et al. 1990; Koelbel et al. 1994]; and in Jade, which allows programmers to identify statements that are independent and hence can be executed concurrently [Rinard et al. 1993]. Adherence to sequential semantics has important software engineering advantages. However, not all parallel algorithms are easily expressed in sequential terms. For example, the load-balancing algorithm of Figure 6 is an explicitly parallel algorithm, with no sequential equivalent.

Other explicitly parallel approaches include Linda and message-passing libraries such as p4, PVM, and MPI. Linda extends sequential languages with operations for creating processes and for manipulating a shared associative store called tuple space [Carriero and Gelernter 1989]. Like Strand and PCN, Linda utilizes a data-driven execution model in which the actions of "sending" and "receiving" data are decoupled and processes execute when data are available. A significant advantage of Linda is that the programmer need learn only a small set of tuple space operations. On the other hand, the global tuple space makes it difficult to develop modules that encapsulate internal communication operations: Linda is not "compositional."

The p4 [Butler and Lusk 1994] and PVM [Sunderam 1990] libraries extend sequential languages with functions for sending and receiving messages. Advantages include simplicity and portability, and the efficiency that can be achieved by ac-

cessing directly the low-level communication mechanisms of a message-passing computer. These features make them well suited for scientific and engineering applications, particularly when communication costs dominate performance. In other classes of problems, the low-level nature of these libraries can be a disadvantage. Applications that communicate complex data structures or that use dynamic process and communication structures are more easily expressed by using higher-level languages such as Strand and PCN. MPI [Gropp et al. 1995] provides a similar model and also introduces support for the modular construction of parallel programs.

6. SUMMARY

We have reviewed and evaluated several different approaches to the realization of the concept of "compositionality" in parallel programming languages. In Strand and PCN, concurrent composition and single-assignment variables are integrated into simple new languages that can be used either alone or as coordination languages for sequential languages. Experience with large-scale applications show that both systems have important strengths, but that the use of a specialized language can be a significant weakness. Conceptually hard things (such as specifying complex distributed algorithms or building reusable parallel modules) often become very easy in the compositional framework, but apparently easy things (such as imple-

menting SPMD programs, or generating a profile) become surprisingly difficult in the context of specialized languages.

Compositional C++ and Fortran M represent an alternative approach to compositional programming based on simple extensions to sequential languages. The resulting languages are considerably richer and more complex than Strand and PCN, but retain important compositional properties.

Acknowledgments

The Strand system was designed in collaboration with Steve Taylor. The initial PCN design was developed by Mani Chandy and Steve Taylor, who also developed the first compiler. Steve Tuecke contributed in many ways to the development of PCN. I am grateful to Carl Kesselman for his insightful comments on the topics covered in this paper. This work was supported in part by the National Science Foundation's Center for Research in Parallel Computation under Contract NSF CCR-8809615 and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

REFERENCES

ACKERMAN, W. B. 1982. Data flow languages. *Computer 15,* 2 (Feb.), 15–25.

ARMSTRONG, J. AND VIRDING, S. 1989. Programming telephony. In *Strand: New Concepts in Parallel Programming*. Prentice Hall.

BUTLER, R. ET AL. 1989. Aligning genetic sequences. In *Strand: New Concepts in Parallel Programming*. Prentice Hall.

BUTLER, R. AND LUSK, E. 1994. Monitors, message, and clusters: The p4 parallel programming system. *Parallel Computing 20*, 547–564.

CANN, D., FEO, J., AND DEBONI, T. 1990. Sisal 1,2: High performance applicative computing. In *Proc. Symp. Parallel and Distributed Processing*. IEEE Computer Science Press, 612–616.

CARRIERO, N. AND GELERNTER, D. 1989. Linda in context. *Commun. ACM 32,* 4 (Apr.), 444–458.

CHANDY, K. M. AND FOSTER, I. 1995. A deterministic notation for cooperating processes. *IEEE Trans. Parallel and Distributed Systems 6,* 8, 863–871.

CHANDY, K. M. AND KESSELMAN, C. 1992. The derivation of compositional programs. In *Proc. 1992 Joint Intl Conf. and Symp. on Logic Programming*. MIT Press.

CHANDY, K. M. AND KESSELMAN, C. 1993. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Object Oriented Programming*. MIT Press.

CHANDY, K. M. AND TAYLOR, S. 1991. *An Introduction to Parallel Programming*. Jones and Bartlett.

CHAPMAN, B., MEHROTRA, P., AND ZIMA, H. 1992. Programming in Vienna Fortran. *Scientific Programming 1,* 1, 31–50.

CLARK, K. AND GREGORY, S. 1981. A relational language for parallel programming. In *Proc. 1981 ACM Conf. on Functional Programming Languages and Computer Architectures*. 171–178.

CLOCKSIN, W. AND MELLISH, C. 1981. *Programming in Prolog*. Springer-Verlag.

Cole, M. 1989. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press.

Dijkstra, E. 1975. Guarded commands, nondeterminacy and the formal derivation of programs. *Commun. ACM 18*, 453–7.

Foster, I. 1989. A multicomputer garbage collector for a single-assignment language. *Intl J. Parallel Programming 18,* 3, 181–203.

Foster, I., Kesselman, C., and Taylor, S. 1990. Concurrency: simple concepts and powerful tools. *Computer Journal 33,* 6, 501–507.

Foster, I., Olson, R., and Tuecke, S. 1992. Productive parallel programming: The PCN approach. *Scientific Programming 1,* 1, 51–66.

Foster, I. and Taylor, S. 1990. *Strand: New Concepts in Parallel Programming.* Prentice-Hall.

Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C., and Wu, M. 1990. Fortran D language specification. Tech. Rep. TR90-141, Dept. of Computer Science, Rice University. Dec.

Gregory, S. 1987. *Parallel Logic Programming in PARLOG.* Addison-Wesley.

Gropp, W., Lusk, E., and Skjellum, A. 1995. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press.

Harrar, D., Keller, H., Lin, D., and Taylor, S. 1991. Parallel computation of Taylor-vortex flows. In *Proc. Conf. on Parallel Computational Fluid Dynamics.* Elsevier Science Publishers B.V.

Hoare, C. 1974. Monitors: An operating system structuring concept. *Commun. ACM 17,* 10 (Oct.), 549–557.

Hoare, C. 1978. Communicating sequential processes. *Commun. ACM 21,* 8, 666–677.

Kahn, G. and MacQueen, D. 1977. Coroutines and networks of parallel processes. In *Information Processing 77: Proc. IFIP Congress.* North-Holland, 993–998.

Kelly, P. 1989. *Functional Programming for Loosely-Coupled Multiprocessors.*

Kesselman, C. 1991. Integrating performance analysis with performance improvement in parallel programs. Ph.D. thesis, UCLA.

Koelbel, C., Loveman, D., Schreiber, R., Steele, G., and Zosel, M. 1994. *The High Performance Fortran Handbook.* MIT Press.

Kowalski, R. 1979. *Logic for Problem Solving.* North-Holland.

Lucco, S. and Sharp, O. 1991. Parallel programming with coordination structures. In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages.* Orlando, FL.

Lusk, E. et al. 1988. The Aurora or-parallel Prolog system. In *Proc. Fifth Generation Computer Systems Conference.* 819–830.

McLennan, B. 1990. *Functional Programming: Practice and Theory.* Addison-Wesley.

Mierowsky, C., Taylor, S., Shapiro, E., Levy, J., and Safra, S. 1985. Design and implementation of Flat Concurrent Prolog. Tech. Rep. CS85-09, Weizmann Institute of Science, Rehovot, Israel.

Rinard, M., Scales, D., and Lam, M. 1993. Jade: A high-level machine-independent language for parallel programming. *Computer 26,* 6, 28–38.

Ringwood, G. 1988. PARLOG86 and the dining logicians. *Commun. ACM 31*, 10–25.

Shapiro, E. 1987. *Concurrent Prolog: Collected Papers.* MIT Press.

SUNDERAM, V. 1990. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience 2,* 4, 315–339.

TAYLOR, S. 1989. *Parallel Logic Programming Techniques.* Prentice-Hall.

XU, M. AND TURNER, S. 1990. A multi-level time warp mechanism. In *Proc. 1990 Summer Computer Simulation Conf.* Society for Computer Simulation, 165–170.

Received January 1993; revised September 1993; accepted January 1996