Strand and PCN: Two Generations of Compositional Programming Languages

Ian Foster

CRPC-TR93446
1993

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Updated May, 1994.
To be published in *Communications of the ACM,* 1994.

Strand and PCN: Two Generations of Compositional Programming Languages

Ian Foster
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
foster@mcs.anl.gov

Abstract

Two parallel languages and their associated programming systems are reviewed and evaluated. Both Strand and PCN are designed to facilitate parallel program development by providing an expressive high-level notation; by integrating tools for debugging, performance analysis, etc.; and by providing portability across different parallel computers. Both provide explicit parallel constructs and use single-assignment variables as an abstraction for communication and synchronization. They support a compositional approach to program design, in which programs composed from simpler components inherit the properties of these components. Strand, developed in 1988, is a first-generation system specialized for symbolic applications. PCN, developed in 1990, is a second-generation system that supports both symbolic and numeric computing. Both systems are available on a range of parallel computers, have been widely distributed, and are in use in many applications. This article summarizes their principal features, describes representative applications, and evaluates their strengths and weaknesses for parallel programming.

1 Introduction

Parallel programming is often considered difficult. This may seem understandable: after all, the parallel programmer must control hundreds of processors, while the sequential programmer controls but one. Yet sequential programs manage millions of bytes of memory and execute billions of machine instructions. Why should additional processors complicate things so tremendously? One reason is that properties that we take for granted in sequential programming, such as deterministic execution, information hiding, modularity,

and virtual resources, are lacking in most parallel programming libraries and languages. In effect, most parallel programs are written in machine language.

One property that appears fundamental to good parallel programming practice is what Mani Chandy has termed "compositionality": the ability to develop programs by composing simpler components, in such a way that the resulting programs inherit the properties of the components from which they are constructed, even when executing concurrently. In particular, programs constructed from deterministic compositions can themselves be guaranteed to be deterministic, meaning that the result computed never depends on the order in which components are scheduled for execution. Compositionality permits modular design, allows components to be developed and tested separately, simplifies verification, and encourages reuse of code. However, not all parallel programming notations provide 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.

Compositionality and determinism can be achieved in a variety of ways: a number of approaches are reviewed below. The two parallel languages described in this article, Strand and PCN, realize compositionality by requiring that concurrently-executing components interact by reading and writing single-assignment or *definitional* variables [14, 7]. A definitional variable is initially undefined and can be assigned at most a single value. If a component attempts to read an undefined variable, execution of that component is suspended until the variable is defined. Hence, the result of a computation cannot depend on the time at which read operations occur. The race conditions that bedevil many parallel programs rarely occur in Strand and PCN programs, and then only if specialized constructs are employed.

Strand and PCN also share other properties that simplify parallel program development. A data-driven, lightweight-process execution model decouples the notion of "process" and "processor" and permits automatic overlapping of computation and communication. Sophisticated toolkits provide parallel debuggers, profilers, compilers, etc., for both parallel and networked computers. Programs can coordinate the execution of sequential code written in C, Fortran, etc., allowing reuse of sequential code in a parallel environment. Process mapping is specified by program annotations that affect performance but not semantics, allowing alternative mapping strategies to be explored without modifying program logic. Finally, support for symbolic computing — in particular, recursively-defined data structures and procedures similar to those employed in Lisp and Prolog — enable a high-level, declarative approach to parallel programming that is useful both for symbolic problems and for the prototyping of concurrent algorithms.

Strand is a first-generation system that grew out of early work in concurrent logic programming. It is a high-level, declarative language primarily used in symbolic and distributed-computing applications. Strand has been commercially supported since 1989 and is in use at over 300 sites in 21 countries. PCN is a second-generation system that integrates imperative constructs and facilities for reusing parallel code. It is used for both symbolic and numeric applications. The public-domain PCN system has been available since 1991 and has been distributed to several hundred sites. Both systems have been implemented on a wide range of parallel computer systems, made available to a large community of users, and used in substantial programming projects in industry and academia.

2 Strand

Strand's origins in logic programming are evident in its syntax, which has a distinctly declarative, symbolic flavor. Nevertheless, Strand is first and foremost a parallel language, and it retains logic programming concepts only when they are also useful for parallel programming. Interfaces to C and Fortran make it possible to integrate imperative computation into Strand programs. In fact, Strand is commonly used as a *coordination language*: that is, as a framework for coordinating the concurrent execution of sequential code modules.

Strand compilers for a range of parallel and networked computers are distributed under the tradename STRAND88 by Strand Software Technologies Inc. (Electronic mail: `will@sstl.uucp` and `strand@ppg.strand.com`).

2.1 Strand Language

This summary of Strand language concepts is not intended to be comprehensive; for details, see [14]. 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.

Communication and Synchronization. Strand variables are single-assignment, or *definitional*, variables. The value of such a variable is initially undefined, can be defined at most once, and subsequently cannot be changed. A process that requires the value of a variable waits until the variable is defined.

A shared definitional 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 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` which can be used to receive additional messages.

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 arguments 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.

```
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.
```

Figure 1: Producer/Consumer Program.

```
augcgagucuauggcuucggccauggcggacggcucauu
augcgagucuaugguuucggccauggcggacggcucauu
augcgagucuauggacuucggccauggcggacggcucagu
augcgagucaaggggcucccuuggggcaccggcgcacggcucagu
```

(a)

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

(b)

Figure 2: RNA Sequence Alignment

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 illustrated below.

2.2 Programming Examples

We use a sequence alignment program developed by Ross Overbeek and his coworkers [3] 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 utilizes a divide-and-conquer strategy which, in simplified terms, works as follows. First, "critical points" — short subsequences that are unique within a sequence — are identified for each sequence. Second, "pins" — 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

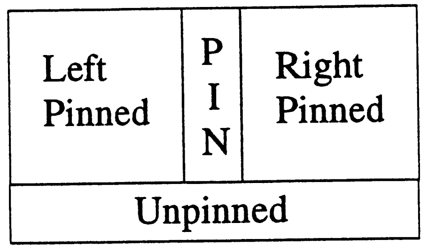

Figure 3: Splitting Sequences Using a Pin

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 has this property. The procedures 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 definitional 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

```
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).
```

Figure 4: Genetic Sequence Alignment Algorithm

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` that 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.

This example also illustrates the code reuse that can be achieved with Strand. The load-balancing library can be used in any Strand program that adheres to its interface. This interface is defined by the call to `scheduler` (its two arguments are an integer specifying a number of workers and a list of tasks) and the call to `execute`, which takes a task as an argument and invokes the appropriate procedure, defining the variable `Done` when this completes.

2.3 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, a performance profiler, and an X-Windows interface.

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* [32]; 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.

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. These functions, designed by Carl Kesselman [21], allow information such as total procedure execution time, procedure execution frequencies, and communication volumes

```
-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 := [].
```

Figure 5: Load-Balancing Library

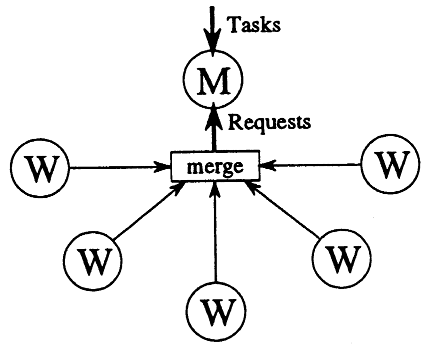

Figure 6: Manager/Worker Scheduler Structure

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 one per cent of total execution time [21]. As 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.

2.4 Strand Experiences

Strand has been used in companies, research laboratories, and academic institutions to program hypercubes, shared memory machines, transputer surfaces, and workstation networks. Universities in Europe, the U.S., and Japan use Strand to teach courses in parallel computing, distributed computing, parallel algorithms, parallel operating systems, and related topics.

Strand applications range from the purely symbolic through mixed symbolic/numeric to the purely numeric, from reactive to transformational, and from sequential through parallel to distributed. The most successful seem to be those with a strong symbolic or reactive component; these take advantage of Strand's high-level, declarative features and/or its lightweight processes and data-driven execution model. Most are multilingual, with a few tens, hundreds, or thousands of lines of Strand coordinating the execution of sequential code written in C or Fortran. For example:

Parallel Computer Algebra. Researchers at the Research Institute for Symbolic Computation in Linz, Austria, have developed ||Maple||, a parallel implementation of the computer algebra system Maple [30]. In the United States, a telecommunications company has developed an interface between Strand and the Mathematica computer algebra system. Both systems use Strand to coordinate execution of symbolic algebra computations on parallel computers and workstation networks.

Discrete Event Simulation. Researchers at the University of Exeter have used Strand to develop a parallel discrete event simulation system that implements a variant of the

Timewarp simulation algorithm [34]. This system is used on both transputer networks and the Kendall Square Research KSR-1.

Telephone Exchange Control. A Swedish telecommunications company has used Strand to develop telephone exchange control software. Strand is used to manage the data-driven execution of multiple threads of control, each representing a physical resource or telephone call [1].

Multiuser Databases. Strand Software Technologies has developed a system that allows multiple users to interact with a single Strand service in a workstation network. This facility has been used to develop a multiuser spreadsheet, a multiuser project management system, and concurrency control mechanisms for distributed databases.

Other representative applications include the prototyping of new concurrent file system structures, database management in the banking industry, planning, protein structure determination, finite element analysis, mesoscale weather modeling, implementation of logic programming systems, and parallel theorem provers.

2.5 Evaluation

As a high-level, declarative language, Strand inherits many of the strengths of sequential declarative languages such as Lisp and Prolog. In particular, complex concurrent algorithms can be implemented quickly and modified easily; symbolic algorithms can be expressed succinctly as recursively defined procedures that operate on recursively defined data structures; and programs have a naturally modular structure and a useful declarative reading. In addition, Strand provides portability across parallel computers, a data-driven execution model, and, through its foreign interface, a mechanism for migrating sequential code into a parallel environment.

Strand's weaknesses are also linked with its logic programming origins. Its syntax — in particular, its lack of iterative and block structuring constructs — is an obstacle to most programmers. In addition, the encapsulation of imperative constructs behind the foreign interface works well only when an application is primarily symbolic or when imperative components are easily isolated. Programs that are primarily numeric often become convoluted, due to the need to convert data repeatedly between symbolic and imperative representations.

In summary, Strand has proven to be a powerful symbolic and distributed programming language, but is ill suited for numeric problems. Its unfamiliar syntax poses a significant barrier to the casual user.

3 Program Composition Notation

Program Composition Notation (PCN) is a high-level parallel language and programming toolkit developed at Argonne National Laboratory and the California Institute of Technology [7, 13]. The language design extends the basic Strand ideas of lightweight processes, logical variables, declarative programming, and multilingual programming in three important 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.

The PCN software is in the public domain and is obtainable by anonymous ftp from `info.mcs.anl.gov`, in directory `pub/pcn`. It has been installed on a wide range of parallel and networked parallel computers; as system dependencies are isolated in the implementation, porting to a new computer system is normally straightforward.

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(arg₁,...,argₖ)
declaration₁; ...; declarationₗ;
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. 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 definitional 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.

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 definitional variables by the fact that they are declared.) 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

```
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.
}
```

Figure 7: PCN Producer/Consumer

```
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)
}
```

Figure 8: PCN Version of Figure 4

```
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
```

Figure 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 = []
}
```

Figure 10: PCN Program That Calls Inner Product.

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.

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 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 definitional value.

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 definitional variables can be used as an interface, avoiding the contention that would occur if processes interacted via a centralized data structure [13]. 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

```
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)
{|| ... }
```

Figure 11: Template Use and Definition.

own internal process and communication structure. The modules interact via distributed arrays of definitional variables `p1` and `p2`, declared using the syntax `port`.

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

3.2 PCN Toolkit

The PCN toolkit includes a compiler, linker, debugger, profiler, trace analyzer, I/O library, and mapping library. The *compiler* translates PCN programs to a machine-independent, low-level form (PCN object code). An interface to the C preprocessor allows macros, conditional compilation constructs, and the like to be used in PCN programs. A programmable transformation system integrated with the compiler allows programmer-specified transformations to be applied to programs [15]. Otherwise, the compilation and runtime techniques are similar to those employed in Strand.

The *linker* combines PCN object code (PCN compiler output), foreign object code that is called from PCN (C or Fortran compiler output), libraries, and the PCN run-time system into a single executable program. This permits C and Fortran procedures to be integrated seamlessly into PCN programs. A set of *standard libraries* provides access to Unix facilities (e.g., I/O) and other capabilities.

The *symbolic debugger*, PDB, includes specialized support for debugging of concurrent programs. PDB allows the programmer to detect deadlocked programs, examine the process pool, and set breakpoints in individual processes.

The *execution profiling* package incorporates run-time system support for collecting and saving execution profiles. As in Strand, the X-windows tool Gauge is provided for interactive exploration of profile data. A snapshoting facility allows multiple profiles to be created during a single program execution. The *trace analysis* package incorporates run-time system support for collecting and saving event traces, and two graphical tools, Upshot and PADL, for interactive exploration of trace data [23]. Gauge is illustrated in Figure 12, which shows two views of a profile from an execution of a parallel weather model. These are histograms showing message counts and communication volume per processor, respectively; each horizontal line (a pixel wide) represents a single processor. Total message counts and message volumes are also provided.

3.3 PCN Experiences

Like Strand, PCN has been applied to a broad range of different problems. However, there is a definite slant towards more numeric, scientific applications, particularly numeric problems that involve irregular, adaptive computation, distributed data structures, or reactive (data-driven) computations. For example, PCN has been used to develop a massively parallel implementation of a mesoscale meteorological model, for use in weather prediction and studies of regional impacts of global change. Here, PCN describes a logical mesh structure corresponding to a two-dimensional decomposition of the model data structures; during execution, nodes in this logical mesh may be moved between processors for load-balancing purposes, and new nodes may be created as the mesh is refined. Other mesh-based applications include a computational fluid dynamics code developed by Harrar et al. for computing Taylor-vortex flows, based on a torus structure [18] (5300 lines Fortran, 900 lines PCN); climate modeling codes based on icosahedral and overlapping stereographic grids (3800 lines C, 640 lines PCN); and a finite-element code for simulating flow in Titan rocket engines (9000 lines Fortran, 180 lines PCN).

Load-balancing libraries similar to that shown in Figure 5 have been used in a range of applications, including the computation of phylogenetic trees in computational biology, the prediction of protein structure, and computer graphics. Other PCN applications include circuit simulation, oil reservoir modeling, Hartree Fock quantum chemistry, molecular dynamics, genetic algorithms, computational fluid dynamics, and parallel theorem provers.

3.4 Evaluation

Like Strand, PCN provides portability, flexibility, reuse of sequential code, and symbolic processing capabilities. In addition, it provides a more convenient block-oriented syntax, integrates imperative constructs and, as a second generation system, has been able to cor-

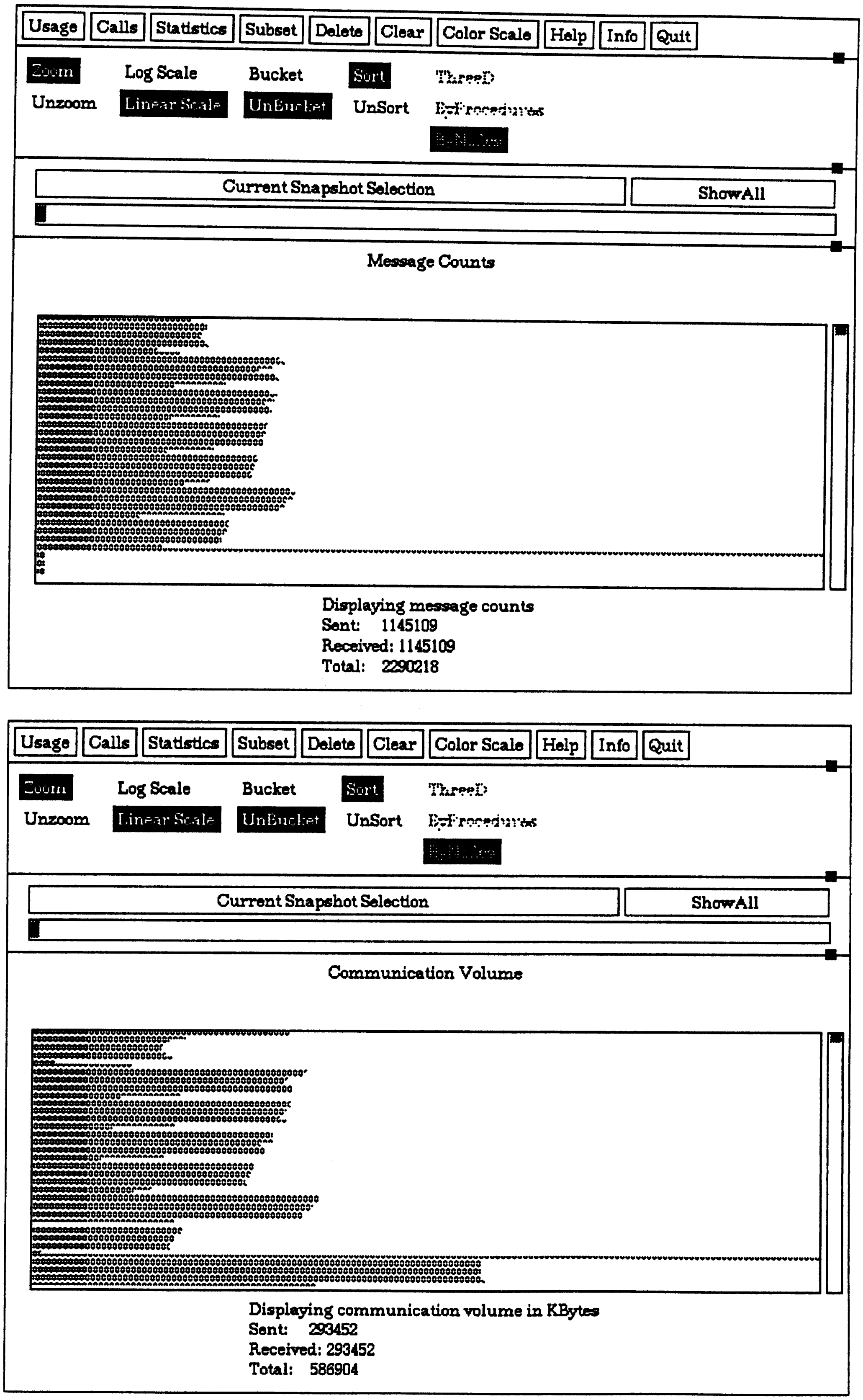

Figure 12: Message Count and Communication Volume Data from Mesoscale Model (see text for description)

rect many of Strand's "rough edges," particularly in the compiler, debugger, and foreign interface. It is our experience that most programmers prefer PCN to Strand for these reasons.

On the downside, PCN is not yet commercially supported. Although technical support is provided by its developers on a best-efforts basis, resources for this activity are necessarily limited. In addition, the PCN compiler does not currently optimize the performance of imperative code, which may execute 10–20 times more slowly than equivalent Fortran or C. (However, typical PCN applications spend much of their time executing Fortran or C, in which case this is not a problem.) Finally, while PCN syntax is more familiar to most sequential programmers than that of Strand, it remains a new language that must be mastered before parallel programs can be written.

In summary, PCN like Strand is an excellent language for prototyping and implementing scalable concurrent algorithms for parallel and distributed computer systems. It is better suited than Strand for applications that combine symbolic and numeric computation, and is supported by a richer program development toolkit.

4 Historical Notes

The Strand system was designed in 1988 by the author and Steve Taylor, and the first compiler was released by Strand Software Technologies in 1989. In 1990, Strand was awarded the British Computer Society's Award for Technical Innovation.

The Strand design builds on work in concurrent logic programming at Imperial College [8, 11, 17, 28], the Weizmann Institute [26, 29, 32], and elsewhere. Concurrent logic programming itself has intellectual roots in logic programming [9, 22], functional programming [20, 25], guarded commands [10], and CSP [19]. 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 yields a dramatically simplified language that can be implemented efficiently on sequential and parallel computers. The compiler incorporates numerous optimizations that take advantage of Strand's simplicity. In addition, Strand introduces constructs that support multilingual programming, allowing its use as a coordination language.

The initial PCN design was developed by Mani Chandy and Steve Taylor in 1990 [7]. Important innovations included the integration of declarative programming (as in Strand) and imperative programming (as in C and Fortran), and a syntax that is both more flexible and closer to sequential programming practice than that of Strand. Chandy and Taylor also developed a PCN to Strand translator that was used for early programming experiments. Subsequent development introduced extensible process mapping constructs and syntactic support for defining reusable templates, and developed implementation techniques for parallel computers [13, 15].

5 Related Work

Many parallel languages and libraries have been developed over the years with the goal of making parallel programming easier. Only a few of these systems have seen widespread use. We compare and contrast some of these approaches with Strand and PCN.

One promising 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 [4, 24, 25]; in data-parallel languages, which exploit the parallelism available when the same operation is applied to many elements of a data structure [33, 16]; and in Jade, which allows programmers to identify statements which are independent and hence can be executed concurrently [27]. 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. In the interests of generality, Strand and PCN provide explicit parallel constructs.

Other explicitly-parallel approachs include Linda and message-passing libraries such as p4 and PVM. Linda extends sequential languages with operations for creating processes and for manipulating a shared associative store called tuple space [5]. 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 use of a global tuple space for communication makes it difficult to develop modules that encapsulate internal communication operations: Linda is not "compositional."

p4 and PVM extend sequential languages with functions for sending and receiving messages [2, 31]. Advantages include simplicity and portability, and the efficiency that can be achieved by accessing 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 utilize dynamic process and communication structures are more easily expressed using higher-level languages such as Strand and PCN.

6 Future Directions

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. Future research directions enabled or suggested by the availability of these systems include the following.

- How can we best integrate declarative and imperative language compiler technology to construct optimizing compilers for languages like Strand and PCN?
- Can software reuse become the norm rather than the exception in parallel programming? What parallel modules and templates are needed to make this happen? How should these be organized to facilitate reuse?

- Are compositional programming ideas also valuable in applications that do not require the symbolic constructs of Strand or PCN? What language constructs are needed in these cases? (Initial studies suggest that only small extensions to sequential languages are required [6, 12].)

Acknowledgments

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 Office of Scientific Computing, U. S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] Armstrong, J., and Virding, S., Programming telephony, Chapter 13 in [14].

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

[3] Butler, R., et al., Aligning genetic sequences, Chapter 11 in [14].

[4] Cann, D. C., J. T. Feo, and T. M. DeBoni, Sisal 1,2: High Performance Applicative Computing, *Proc. Symp. Parallel and Distributed Processing*, IEEE CS Press, Los Alamitos, Calif., 1990, 612–616.

[5] Carriero, N., and Gelernter, D., *How to Write Parallel Programs*, MIT Press, 1990.

[6] Chandy, K. M., and Kesselman, C., Compositional parallel programming in CC++, Technical Report, Caltech, 1992.

[7] Chandy, K. M., and Taylor, S., *An Introduction to Parallel Programming*, Jones and Bartlett, 1991.

[8] Clark, K. and Gregory, S., A relational language for parallel programming, *Proc. 1981 ACM Conf. on Functional Programming Languages and Computer Architectures*, 171-178, 1981.

[9] Clocksin, W., and Mellish, C., *Programming in Prolog*, Springer-Verlag, 1981.

[10] Dijkstra, E.W., Guarded commands, nondeterminacy and the formal derivation of programs, *CACM*, 18, 453-7, 1975.

[11] Foster, I., *Systems Programming in Parallel Logic Languages*, Prentice Hall, 1990.

[12] Foster, I., and Chandy, K.M., Fortran M: A language for modular parallel programming, Preprint MCS-P327-0992, Mathematics and Computer Science Division, Argonne National Lab, Argonne, Ill. 60439, 1992.

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

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

[15] Foster, I., and Taylor, S., A compiler approach to scalable concurrent program design, *ACM Trans. Prog. Lang. Syst.*, (to appear).

[16] Fox, G., S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, M.Wu, Fortran D Language Specification, Technical Report TR90-141, Computer Science, Rice Univ., Houston, TX, 1990.

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

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

[19] Hoare, C., Communicating Sequential Processes, *CACM*, 21(8), 666–677, 1978.

[20] Kahn, G., and MacQueen, D., Coroutines and networks of parallel processes, *Information Processing 77: Proc. IFIP Congress*, B. Gilchrist (Ed.), 993–998, North-Holland, 1977.

[21] Kesselman, C., *Integrating Performance Analysis with Performance Improvement in Parallel Programs*, Ph.D. thesis, UCLA.

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

[23] Lusk, E., Performance visualization for parallel programs, *Theoretica Chimica Acta*, 1992.

[24] Lusk, E. et al., The Aurora Or-parallel Prolog system, *Proc. Fifth Generation Computer Systems Conference,* Tokyo, 819–830, 1988.

[25] McLennan, B. J., *Functional Programming: Practice and Theory*, Addison-Wesley, Reading, Mass. 1990

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

[27] Rinard, M. C., Scales, D. J., and Lam, M. S., Jade: A high-level machine-independent language for parallel programming, *Computer*, 26(6), 28–38, June 1993.

[28] Ringwood, G., PARLOG86 and the dining logicians, *CACM*, 31(1), 10–25, 1988.

[29] Shapiro, E. (Ed.), *Concurrent Prolog: Collected Papers*, MIT Press, 1987.

[30] Siegl, K., Gröbner bases computation in Strand: A case study for concurrent symbolic computation in logic programming languages, Tech. Report 90-54, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990.

[31] Sunderam, V., PVM: A Framework for Parallel Distributed Computing, *Concurrency Practice and Experience*, 2, 1990, 315–339.

[32] Taylor, S., *Parallel Logic Programming Techniques*, Prentice-Hall, 1989.

[33] Thinking Machines Corporation, *CM Fortran Reference Manual*, Thinking Machines, Cambridge, Mass., 1989.

[34] Xu, M., and Turner, S., A multi-level time warp mechanism, *Proc. 1990 Summer Computer Simulation Conf.*, 165–170, Society for Computer Simulation, 1990.