Strand: a Practical Parallel Programming Language

Ian Foster
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois, USA
itf@mcs.anl.gov

Stephen Taylor
Department of Computer Science
California Institute of Technology
Pasadena, California, USA
steve@vlsi.caltech.edu

Abstract

Effective parallel programming requires tools that facilitate the design and implementation of parallel algorithms and provide portability across parallel computers. *Strand* is the first such tool to be available on a broad range of multiprocessor architectures, including hypercubes, meshes, computing surfaces, banyan networks, shared-memory machines and local area networks. *Strand* is a new concurrent logic programming language. Its design is motivated by extensive experience with previous languages of this type. This experience indicated that for most practical parallel programming requirements, these research languages were too complex, inefficient and difficult to implement. *Strand* applies a broad variety of simplifications to achieve a practical parallel programming tool. This paper describes the issues surrounding the *Strand* design and reports on initial experiences with the language.

1 Introduction

Multiprocessor architectures imply a need for tools that provide portability across wide classes of parallel architecture. The state of the art in the development of such tools is not very advanced. If (as seems reasonable) we restrict attention to tools that have been used to implement real applications on a variety of machines, we find surprisingly few serious proposals. Parallel extensions to existing sequential languages (Fortran, C, Lisp) have been used on shared memory architectures. However, these do not provide portable programs. Macro packages and libraries that implement particular parallel programming models (e.g. message passing, tuple space) have been applied on other architectures [2, 18, 3]. However, these lack the convenience of a true parallel language. It is clear that more powerful parallel programming formalisms are required if we are to fully exploit the power of parallel computers. We believe that *concurrent logic programming* represents an attractive basis for such formalisms.

Concurrent logic programming languages have been an active area of research for several years. This research has provided a broad base of experience in concurrent logic programming. It has also led to a large number of language proposals [4, 19, 5, 15, 23, 17, 9, 16]. These are all based on a few key concepts: concurrent processes, communication via single-assignment variables and dataflow synchronization. In addition, most also employ a variety of features which reflect their logic programming origins, notably unification and or-parallelism. These inherited features complicate implementation and programming to little advantage; the languages have much to offer as parallel programming tools without them.

Approaching language design from this perspective makes it possible to apply a broad set of simplifications and performance optimizations. The result is *Strand*[1] [10]: a minimal language that incorporates only the vital concepts of research languages. Initial experiences suggest that *Strand* can be implemented efficiently on a broad range of multiprocessor architectures and that it is sufficiently expressive for a wide range of practical parallel programming problems.

Research in concurrent logic programming has led and will no doubt continue to lead in several directions. *Strand* represents the branch that aims for minimality in order to achieve conceptual simplicity, ease of programming and efficient parallel implementation.

2 Concurrent Logic Programming Languages

Concurrent logic programming languages differ in three dimensions: their use of static or dynamic dataflow, their variable binding scheme and their process organi-

[1] *Strand* is a registered trademark of Artificial Intelligence Limited.

zation. In each dimension a variety of possibilities arise ranging from simple, efficient schemes to complex extravagant ones. In this section we review representative proposals. Space does not permit a complete survey.

2.1 Language Overview

A concurrent logic program is a set of rules of the form:

$$H \ :- \ G_1,\ G_2,\ ...,\ G_m \mid B_1,\ B_2,\ ...\ B_n. \qquad (m,\ n \geq 0)$$

where H is the rule *head*, ' :- ' denotes implication, the G's are the rule *guard*, '|' is the commit operator and the B's are the rule *body*. A rule head has the form:

$$p(A_1,\ A_2,\ ...,\ A_n) \qquad (n \geq 0)$$

where p names the rule and A_1, ... A_n are its arguments. Rules with the same name and number of arguments are grouped to form a *process definition*.

Each rule specifies a possible process behavior. The head and guard collectively specify a set of preconditions under which a process may execute. A rule whose body is empty specifies that if the preconditions are satisfied then the process *terminates*. A rule of the form:

$$H \ :- \ G_1,\ G_2,\ ...,\ G_m \mid B_1.$$

specifies that if the preconditions are satisfied then the process *changes state* to the process B_1. A general rule of the form:

$$H \ :- \ G_1,\ G_2,\ ...,\ G_m \mid B_1,\ B_2,\ ...,\ B_n.$$

specifies that if the preconditions are satisfied then the process changes state to B_1 and simultaneously *forks* the processes B_2, ..., B_n.

2.2 Dataflow

Computation in concurrent logic programming languages involves the reduction of a *pool* of processes using process definitions. Processes communicate by reading and writing shared variables and synchronize using dataflow constraints.

To reduce a process, the heads of rules in the process definition are matched with process arguments and guard processes are evaluated. Dataflow can occur in two directions during matching; *input* causes variables in the rule head to be assigned values in the process; alternatively, *output* causes variables in the process to be assigned values in the rule head.

In languages based on *static dataflow* (such as Parlog [5]) the direction of dataflow in a process is determined entirely by the process definition, which specifies that certain arguments are to be used for input and others for output. For example, in the Parlog program:

$$mode\ f(?, \uparrow)$$
$$f(a, b).$$

the mode declaration specifies that the first argument is used for input and hence *reads* the corresponding process argument (it checks that it is the value a); the second argument is used for output and hence attempts to *write* a value to the corresponding process argument (the value b).

In languages based on dynamic dataflow (such as Concurrent Prolog [19]), the direction of dataflow is determined by the call to the procedure. A read-only annotation applied to variables is used to indicate process arguments that should be used for testing purposes. Thus in the case of the process $f(X?, Y)$, dataflow is as described above for the Parlog procedure. In the process $f(X, Y?)$, dataflow is in the opposite direction.

Dynamic dataflow permits procedures to be called with different argument patterns. However, static dataflow permits efficient compilation of test operations on process arguments; studies show that this results in significant performance gains [9].

2.3 Variable Binding

Processes communicate by assigning values to shared variables. This assignment can be achieved in a number of ways. The simplest is that used in early versions of Parlog [5]: an *assignment operator* ':=' is used to assign a value to a variable. Attempts to assign to a variable that already has a value result in run-time errors. GHC [23] and more recent versions of Parlog [12, 16] and Flat Parlog [9] employ a more complex *unification* procedure to assign values to variables. This attempts to make two terms identical, assigning values to variables in either term if required.

In GHC and Parlog, unification occurs in the body of rules. Other languages, in particular Concurrent Prolog and Flat Concurrent Prolog (FCP) [15], allow unification operations to be specified in a rule guard as preconditions for process reduction. If a rule defines several unification operations, these must occur as an atomic action: if one fails, the others must not be seen to occur [22].

Atomic unification permits elegant solutions to certain programming problems; however, the mechanisms required to support it are a source of run-time overhead and complexity [9], particularly in a multiprocessor [22].

2.4 Process Structures

Each rule in a process definition specifies a set of zero or more guard processes which must be solved before that rule can be used to reduce a process. In principle, the guards of all rules comprising a process definition should be evaluated concurrently (in *Or-parallel*). However, as at most one rule can be used to reduce a process, true Or-parallel evaluation requires that an implementation arbitrate between rules competing to reduce a process.

If rule guards are allowed to include calls to arbitrary procedures, process evaluation can generate process hierarchies [12]. In general, management of process hierarchies requires complex run-time mechanisms [8]. In contrast, *flat languages* [15, 9] restrict guard processes to predefined system primitives. No support for process hierarchies is required; this gives a simple operational model in which the state of a computation is represented by a process pool. Furthermore, Or-parallel evaluation can be efficiently simulated using sequential search through rules in a process definition.

3 *Strand*

Strand employs static dataflow, simple assignment to variables and a flat process structure. It also places restrictions on matching and assignment operations involving two variables: a test operation $X == X$ suspends until X is assigned a value and self-referential and cyclic structures are forbidden.

Strand programs are expressed using the notation presented in Section 2, with two restrictions. First, no variable may occur more than once in the head of a rule. (In fact, we can and do define syntactic sugar that permits this [10]; however, this restriction is made here to simplify presentation). Second, the rule guard contains only predefined test processes.

A *Strand* computation corresponds to a pool of concurrently executing processes. A computation step involves removing a process from the pool. If the process is a *predefined* process it is executed immediately; otherwise a reduction attempt is made. This involves selecting a rule from the program, *matching* the process to the rule head and executing the rule *guard*. If the preconditions specified by the head and guard are satisfied then the process *commits*. This causes new instances of the processes defined in the rule body to be added to the process pool. Processes can be selected for reduction in any order or in parallel.

Matching. The *matching algorithm* used to compare a rule head with a process can be defined informally as follows. It is applied left to right textually and depth first in term structures and may *succeed*, *fail* or *suspend*. If it succeeds it generates a (possibly empty) set of assignments to variables in the rule head. For future reference, we will denote this set by the symbol Θ.

Integers, reals and strings match only if they are identical in both type and value. Structures match if they are the same size and corresponding arguments match. A variable V in a rule matches any term T in a process (including a variable); in addition, the variable assignment $V := T$ is added to Θ. An attempt to match a variable in a process with a non-variable in a rule head causes a match to suspend. All other matches fail.

Matching is used for three purposes. First, to check that input data to a process has required values. Second, to pull apart or *destructure* the process state. (For example, to separate a list structure into its head and tail.) Finally, to synchronize process actions. Note that matching can never assign a value to a variable in a process. Another way of saying this is that matching only *reads* the data state; it does not modify or side effect it.

Guard Execution. *Strand* provides a variety of predefined guard tests which can be used to perform type checking, arithmetic comparison and term comparison. Like matching, these tests serve to constrain when a rule may be used to execute a process. Any number of tests can be written in a rule guard. They are executed from left to right textually after matching is complete. Each test may *succeed*, *fail* or *suspend*. If all tests in a guard *succeed* then the guard as a whole *succeeds*; if any test *fails* or *suspends* then the guard as a whole *fails* or *suspends* respectively. Most tests *suspend* if they encounter a variable during evaluation.

Assignment. The predefined assignment process ':=' is used to assign values to variables. A call $X := Y$ assigns the value Y to the variable X. If Y is a variable at the time of call, X and Y become aliases. Variables have the single assignment property: they can only be assigned a value once; an attempt to assign a value to a non-variable term is signaled as an error.

3.1 Operational Model

We now present a more abstract model of *Strand* program execution. The form of this presentation is derived from [11]. First we define *composite matching*.

Composite Matching. The procedure *CMatch* takes as arguments a process p and a rule copy of the form: $C = H :- G \mid B$. It executes the match algorithm *followed by* the guard execution and returns:

- Θ if match(p,H)=$\Theta \wedge$ G Θ = true.
- *fail* if either match or guard execution returns *fail*.
- *suspend* otherwise.

States. The state of a computation is a multi-set of processes, S. Every process in S is either an assignment process of the form $X := Y$ or is not an assignment process.

Transitions. A transition rule specifies a mapping between states. Execution of a program P may include the following transitions:

Reduction: $\{p_1, \ldots, p_i, \ldots, p_n\} \rightarrow \{p_1, \ldots, B\Theta, \ldots, p_n\}$
If: a. $p_i \neq X := Y$
b. $R' = H :- G \mid B$ is a *fresh copy* of a rule $R \in P$, $CMatch(p_i, R') = \Theta$

Note that if B is the empty body then no processes are added to the new state.

Assignment: $\{p_1, \ldots, X := Y, \ldots, p_n\} \rightarrow \{(p_1, \ldots, p_n)[X/Y]\}$
If: X is a variable and does not occur in Y.

Suspension: $S \rightarrow$ *suspend*
If: $\forall p \in S$
a. $p \neq X := Y$
b. $\forall R \in P$ $CMatch(p,R) \neq \Theta$

Computations: A computation is a sequence of transitions that ends in a terminal state (i.e. one in which no further rules can be applied). A computation that ends in the state $\{\}$ is called a *successful* computation. A computation that ends in the state *suspend* is termed a *suspending* computation.

3.2 Other Features

Strand's basic model incorporates the key features of concurrent logic programming: concurrent processes, single-assignment variables and dataflow synchronization. To this model we add features that make *Strand* a general purpose parallel programming language. As we currently only have informal characterizations for these features, we do not include them in the language definition proper.

Mergers. A *merger* is a predefined process that consumes a stream of messages on an input stream and forwards them on an output stream. A message of the form *merge*(S) is *not* output; instead, S is viewed as an additional input stream and messages arriving on S are also output. A merger guarantees that any message placed on any of its input streams will eventually appear on its output stream and that the order of elements in each input stream is preserved in the output. If all input streams are closed then the merger closes the output stream and terminates.

Mergers have been widely described (e.g. see [1]); they were first specified for concurrent logic programming by Safra and Shapiro [20].

Otherwise. The predefined guard test *otherwise* can be used to establish default rules in process definitions. A guard test *otherwise* succeeds if all textually previous rules in the process definition fail.

Known and Unknown. A guard test $known(X)$ succeeds if the value of X is currently available and fails otherwise. If applied repeatedly to the same variable, it is guaranteed to succeed eventually if this is assigned a value. Similarly, a test $unknown(X)$ succeeds if the value of X is not currently available and is guaranteed to fail eventually if X is assigned a value. These tests permit efficient polling in multiprocessor architectures. They are not essential to the language, as polling can always be implemented using other means if a merger is available.

Support for Integration. Support for processes written in other languages (Fortran, C, etc.) and user-defined data types is an integral part of the *Strand* language. These features support reuse of existing sequential code in a parallel environment [10].

Programming in the Large. Language features support the definition and use of modules. These features are described in detail in [10].

4 *Strand*: a Practical Language

We now explore the application and implementation of *Strand*. We show that the language permits convenient expressions of important parallel programming paradigms. We also outline implementation techniques for multiprocessors and explain how the language's design simplifies distributed implementation and encourages portability.

4.1 Basic Techniques

Over the years, researchers working with concurrent logic languages have explored a wide range of language features and programming techniques. We ourselves have written a large body of concurrent logic programming code, ranging over operating systems problems, parallel algorithms and various applications [7, 22]. Careful study of this and other codes indicates that programs can in practice be expressed using a small number of simple programming paradigms. In particular, six fundamental techniques appear repeatedly as basic building blocks for parallel programs: producer/consumers, incomplete messages, bounded buffers, difference lists, short circuits and blackboards. We show here how these techniques are expressed in *Strand*.

Producer/Consumers. The producer/consumer interprocess communication protocol [14] allows a producer process to send many messages to a number of consumer processes using a single shared variable. For example, consider the processes:

$$producer(Xs),\ consumer(Xs)$$

The producer process can use the shared variable *Xs* to communicate a *stream* of messages to the consumer. This is achieved by assigning an incomplete structure to the shared variable *Xs*; variables in this structure can be used for subsequent communication. By convention, a list is used as the incomplete structure; the head of the list contains the message and the tail contains the variable used to send further messages.

Figure 1 illustrates this protocol; it defines a producer that generates a stream of *N* messages (*thanks*) and a consumer that consumes these messages. The following initial process pool can be used to execute this program; this causes 3 *thanks* messages to flow from *producer* to *consumer*:

$$producer(3,Xs),\ consumer(Xs)$$

Incomplete Messages. The incomplete message protocol [19] allows two-way interprocess communication. In essence, a consumer process uses variables contained in a message to communicate values back to the producer of the message. Figure 2 illustrates this; it shows a variant of Figure 1 in which messages have the form $\{thanks,R\}$ and the consumer acknowledges receipt of each message by binding its *R* variable to the string "*you're welcome*". The producer creates an *appreciate* process to process each reply.

An important application of the incomplete message protocol is to implement *monitors*. It commonly happens that streams of requests generated by several consumers must be combined and passed to a monitor process encapsulating a resource. This can be achieved using a combination of this protocol and mergers. The merger combines request streams from many producers; the monitor is able to respond to requests without knowing the identity of the producers. This permits a single monitor to communicate with a varying number of producers.

```
producer(N, Out) :-                   % Rule 1.
    N > 0 |                           % Still generating?
        N1 is N - 1,                  %   One less to generate.
        Out := [thanks | Out1],       %   Generate a message.
        producer(N1, Out1).           %   Recurse to generate more.
producer(0, Out) :-                   % Rule 2 — all done?
        Out := [].                    %   Close stream

consumer([thanks | Ms]) :-            % Rule 3 — wait for thanks.
        consumer(Ms).                 %   Consume rest.
consumer([]).                         % Rule 4 — terminate.
```

Figure 1: The Producer/Consumer Protocol

```
producer(N, Out) :-                   % Rule 1.
    N > 0 |                           %   Still generating?
        N1 is N - 1,                  %   One less to generate.
        Out := [thanks, R | Out1],    %   Generate a message.
        appreciate(R),                %   Process reply.
        producer(N1, Out1).           %   Recurse to generate more.
producer(0, Out) :-                   % Rule 2. — all done?
        Out := [].                    %   Close stream

consumer([thanks, R | Ms]) :-         % Rule 3 — wait for thanks.
        R := "you're welcome",        %   Acknowledge thanks.
        consumer(Ms).                 %   Consume rest.
consumer([]).                         % Rule 4 — terminate.
```

Figure 2: The Incomplete Message Protocol

```
producer(N, [M | Ms]) :-            % Rule 1 — wait for buffer.
    N > 0 |                         % Still generating?
        N1 is N - 1,                % One less to generate.
        M := thanks,                % Generate a message.
        producer(N1, Ms).           % Recurse to generate more.
producer(0, [M | Ms]) :-            % Rule 2. - all done?
        M := done.                  % Close stream

consumer([thanks | Ms],Buffer) :-   % Rule 3 — wait for thanks.
        Buffer := [X | Bs],         % Extend buffer.
        consumer(Ms, Bs).           % Consume rest.
consumer([done | Ms], _).           % Rule 4 — terminate.
```

Figure 3: The Bounded Buffer Protocol

Bounded Buffers. The bounded buffers protocol is in practice rarely used. However, it is required for certain operating system problems. The producer/consumers protocol permits the producer to produce a large number of messages before they are consumed by the consumer. If too many messages are generated, the space available in a *Strand* system for storing messages may be exceeded. The bounded buffer protocol can be used to avoid this problem when it arises: it permits the number of outstanding (unreceived) messages to be bounded.

Unlike the previous protocols, this organization requires that the *consumer* generate a stream of unbound variables. The producer fills in these variables with messages. A number of stream elements are initially made available to the producer for sending messages; these form an initial message buffer. E.g.:

$$\mathit{Buffer} := [\ \mathit{M1},\ \mathit{M2},\ \mathit{M3}\ \ldots\ |\ \mathit{Mn}]$$

Figure 3 illustrates the application of the bounded-buffer protocol. The following call to this program creates an initial buffer containing five elements; this ensures that no more than five "*thanks*" can be outstanding (unreceived) at any one time:

$$\mathit{Buffer} := [\ \mathit{M1},\ \mathit{M2},\ \mathit{M3}\ \ldots\ |\ \mathit{Mn}],$$
$$\mathit{producer}(\mathit{Buffer}),\ \mathit{consumer}(\mathit{Buffer},\mathit{Mn})$$

Difference Lists. A difference list [6] is a representation of a list. The technique for manipulating this representation allows it to be constructed in parallel by a set of processes. In contrast to ordinary lists, the tail of a difference list is maintained as a variable. This variable tail is kept accessible. Two difference lists can be concatenated in constant time by assigning the head of one to the tail of the other.

The following process definition uses a difference list to collect a list of integer leaves in a binary tree of numbers. The first rule creates processes to collect the left and right branches of a tree represented by a term $\{Left,Right\}$; the difference list is split between the two processes. The second rule specifies that a process adds an integer leaf to its component of the difference list. The third rule states that a process adds *nothing* to its component of the difference list if applied to a real.

$$collect(T, O) :- collect(T, O, []).$$

$$\begin{array}{l} collect(\{L, R\}, Ob, Oe) :- collect(L, Ob, Om),\ collect(Om, Oe). \\ collect(I, Ob, Oe) :- integer(I)\ \mid\ Ob := [I \mid Oe]. \\ collect(R, Ob, Oe) :- real(R)\ \mid\ Ob := Oe. \end{array}$$

Short Circuits. The Short-Circuit technique [21] is used to detect that a collection of processes have terminated. When a network of processes is created, each process is spawned with two ends of a variable chain represented by variables. If a process forks, the chain is broken among the forked processes. E.g.:

$$\begin{array}{l} process(..., Left, Right) :- \\ \qquad process1(..., Left, Middle1), \\ \qquad process2(..., Middle1, Middle2), \\ \qquad process3(..., Middle2, Right). \end{array}$$

If a process terminates, it *closes* its portion of the circuit by assigning its left hand variable to its right. E.g.:

$$process1(..., Left, Right) :- Right := Left.$$

Now if the initial process places a constant (e.g. *done*) on the left of the chain:

$$process(...., done, AllDone)$$

then when all processes have terminated, the right end of the chain (*AllDone*) will eventually be bound to the same constant. This signifies that all processes have terminated. The *Strand* language and the distributed execution algorithm that supports it ensures that it is possible to detect the termination of every fundamental operation in the language. The short-circuit technique can hence always be used to detect termination of any program [10].

Blackboards. A blackboard is a mutable data structure maintained by a set of processes. The technique for organizing this structure allows it to be updated atomically; many processes can both read and write information on the blackboard.

Strand's single assignment property means that variables cannot be used directly to represent mutable data. Yet many programming problems require the ability to represent information that changes over time. This is achieved in *Strand* by encapsulating this data in a process to which other processes send read and write requests [10]. A merger can be used to combine streams of requests generated by a varying number of processes. This technique is very simple, so we do not illustrate it with an example.

4.2 Implementation Issues

The simplicity of the *Strand* language encourages efficient and portable implementations. In this section, we outline techniques that have proved useful when implementing *Strand* on a range of multiprocessor architectures.

Abstract Implementation Scheme. Current *Strand* implementations are based on a simple abstract machine. This consists of a *reduction* and a *communication* component located at every processor. In addition, data structures maintained at each processor represent local components of the global process pool. Each reduction component selects and executes processes from the local pool. Its instruction set supports the basic operations required by *Strand*'s operational model: test and assignment operations on process arguments and process creation and termination. The communication component processes messages that arrive at a processor.

A global address space is implemented by representing each variable by a single occurrence, located on a single processor, and by multiple *remote references* to that occurrence. A remote reference specifies a processor and an address within that processor. Recall that *Strand*'s operational model involves two main transition rules. The *reduction* rule *reads* values for matching and guard execution. The *assignment* rule *writes* values. Both read and write operations may generate communication if they encounter references to remote terms (i.e. remote references). The language implementation incorporates two algorithms, *read* and *assign*, that translate read and write operations on remote terms into messages that request other processors to perform read and write operations.

The *assign* algorithm is invoked when an assign operation encounters a remote reference. It generates an *Assign* message to the processor referenced by the remote reference. This contains an address and a representation of a *Strand* term. A processor receiving an *Assign* message binds the variable at the specified address to the supplied term.

The *read* algorithm is invoked when a read operation encounters a remote reference. It generates a *Read* message to the remote processor. This contains the address of the term required by the read operation and a return address for the value. A processor receiving a *Read* message returns the value of the referenced term immediately if it is a non-variable, by generating a *Value* message. Otherwise the remote read operation is suspended, to be completed if and when the variable in question is assigned a value.

These are the only mechanisms required to support distributed execution of *Strand*: no complex locking algorithms or reading and unification protocols are required. We now point to those features of *Strand* that have permitted us to reduce the distributed implementation to these few simple ideas.

1. The restriction of guards to primitive test operations avoids the need to support a process hierarchy [12].

2. No special actions need be performed when committing to the use of a particular rule. In contrast, other proposals require that Or-parallel computations be aborted [8] or that variable bindings created during guard execution be made public [22].

3. The clean distinction made between reading and writing of variables means that there is no need for moded unification [24] or open-coding of read- and write-mode alternatives [22].

4. The lack of atomic unification means that no special mechanisms are required to trail bindings [15] or to prevent deadlock and starvation in a distributed implementation [22].

5. The reading algorithm may simply delay at the remote site indefinitely since $X == X$ suspends until X is assigned a value. This avoids a need for complex reading protocols to support variable equality [7].

6. As assignment operations of the form $X := X$ are forbidden there is no need to dereference a remote term occurring on the right hand side of an assignment [7].

In principle, many of these optimizations can be applied to appropriate subsets of other concurrent logic languages. However, it is unlikely that a compiler can always determine when these optimizations can be applied. In addition, the extra machinery required to support the more complex features must still be provided in an implementation. We hence argue that *Strand*'s simplicity has significant implementation advantages. Initial experimental studies appear to support this contention. Existing *Strand* implementations are significantly faster than previously reported implementations of related languages. For example, the following table shows uniprocessor timings for three benchmarks (described in [9]). These were obtained using a prototype implementation of the *Strand* abstract machine written in 'C'. The timings were generated on Sun 3/260 and Sun 4/110 machines and are in reductions per second (RPS). Naive reverse is a tight computational loop which uses extensive tail recursion. The other programs create large numbers of process suspensions and spawn large numbers of processes.

Program	Sun 3/260 (RPS)	Sun 4/110 (RPS)
Reverse	49980	59540
Quicksort	21450	28910
Takeuchi	11300	15210

The compilation techniques used to achieve these results were extremely naive; we expect *Strand*'s simplicity to facilitate the development of excellent optimizing compilers. In particular, as the head and guard only read a process data state, it is easy to compile an entire process definition into a discrimination network.

Portability. *Strand* has been successfully ported to a wide range of multiprocessor architectures: the language is in active use on hypercubes, meshes, transputer computing surfaces, banyan networks, shared-memory machines and local area networks. Applications developed using *Strand* on any one of these architectures can be ported to the others with little or no reprogramming.

The key to the portability of the *Strand* implementation is the simplicity of the abstract implementation scheme described previously. Recall that this provides a reduction and communication component at each processor. The reduction component, if implemented in a high-level language, is in principle portable to any machine that supports a compiler for that language. The communication component is more machine specific but only requires the ability to send and receive messages. Implementations on different architectures differ chiefly in the techniques used to implement these send and receive operations.

On message passing machines (such as the Intel hypercube and the Symult mesh) facilities provided by the underlying operating system are used to achieve relatively efficient interprocessor communication. On local area networks, network protocols are used for the same purpose, though obviously at substantially greater cost. On shared memory machines such as Sequent, Encore and Butterfly, monitors and message queues located in shared memory are used to provide cheap interprocess communication. Careful design minimizes contention for shared resources. In principle, a *Strand* implementation specialized for shared memory machines can avoid simulated message passing. However, it is not clear that this provides performance benefits for realistic applications.

The keys to the portability of application programs written in *Strand* are the uniform nature of the language's computational model and an environment that provides transparent access to machine resources via a standard interface. The environment includes parallel programming libraries that provide programmers with a variety of virtual machines and processor scheduling tools [22]. Virtual machines permit a user to create applications that execute on a ring, torus, hypercube, etc.; libraries provide appropriate embeddings of these virtual topologies on a particular physical machine. Processor scheduling tools include manager/worker schedulers which distribute tasks generate by a programmer to idle processors [10]. The environment also provides support for the management of computations distributed over many processors [7].

Applications. *Strand* has already been used to develop a broad range of applications. For example: at Argonne National Laboratory, *Strand* is a principal implementation language in a project developing tools for DNA sequencing. These tools are written in a mixture of *Strand* and C and have been ported to several multiprocessors with encouraging results. At Aerospace Corporation, *Strand* is being used to investigate parallel algorithms for circuit verification. At CalTech, the language has been used to develop parallel implementations of fluid flow modeling codes written in Fortran. At Ellemtel in Sweden, it has been employed to develop a prototype telephone exchange control system. At the University of Chicago, *Strand* has been used to develop parallel versions of a high performance theorem prover [13]. At Strand Technologies Inc., it has been used to implement compilers, multiprocessor programming environments and distributed simulation tools. Several of these applications are reported in [10].

5 Conclusion

Strand represents the first practical programming tool to be available on a wide variety of parallel architectures. The motivation for *Strand* is our previous experience with the implementation and application of concurrent logic programming languages. This experience includes the development of several implementations, the coding of large systems and applications, and comparative studies of language implementations [9, 7, 22]. This experience has convinced us that previous research languages are too complex for effective implementation and programming. We have thus designed a minimal language that makes the key concepts of concurrent logic programming accessible to parallel programmers in a simpler framework.

Practical experiences with *Strand* suggest that the quest for simplicity has paid off. It has proved possible to achieve efficient implementations on a broad range of multiprocessor architectures. In addition, a wide variety of applications have already been developed by industrial partners and collaborators.

6 Acknowledgements

It is with pleasure that the authors acknowledge the excellent efforts of the research and development team at Strand Technologies Inc., Watford, England. They helped make *Strand* a reality. Michael Codish, Vijay Saraswat and Will Winsborough contributed to the definition of *Strand*'s operational model. This research was supported in part by the Applied Mathematical Sciences subprogram of the office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.

References

[1] Arvind, Gostelow, K.P. and Plouffe, W., Indeterminancy, monitors and dataflow. In *Proc. 6th ACM Symp. on Operating System Principles*, OS Review, 11(5), 159-169, 1977.

[2] Boyle, J. et al., *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, New Tork, 1987.

[3] Carriero, N. and Gelernter, D., Applications experience with Linda, *Proc. ACM/SIGPLAN PPEALS 1988*, 173-187.

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

[5] Clark, K. and Gregory, S., Parlog: parallel programming in logic, *ACM Trans. on Programming Languages and Systems* 8(1) 1986, 1-49.

[6] Clark, K.L. and Tarnlund, S.A., A first order theory of data and programs, *Information Processing 77; Proc. IFIP Congress 77*, B. Gilchrist (Ed.), Amsterdam: Elsevier/North Holland, 939-944.

[7] Foster, I.T., *Systems Programming in Parallel Logic Languages*, Prentice-Hall, London, 1989.

[8] Foster, I.T., Gregory, S. Ringwood, G. and Satoh, K., A sequential implementation of Parlog, *Proc. 3rd Intl Logic Programming Conference*, London, Springer-Verlag LNCS, 1986.

[9] Foster, I.T. and Taylor, S., Flat Parlog: a basis for comparison, *Intl Journal of Parallel Programming*, 16(2), 1988.

[10] Foster, I.T. and Taylor, S., *Strand*: *New Concepts in Parallel Programming*, Prentice-Hall, Englewood Cliffs, N.J. 1989.

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

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

[13] Jindal, A. Exploitation of parallel processing for implementing high performance deduction systems. PhD thesis, University of Illinois at Chicago, 1989.

[14] 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.

[15] Mierowsky, C., Taylor, T., Shapiro, E., Levy, J. and Safra, M., Design and implementation of Flat Concurrent Prolog, Technical Report CS85-09, Dept of Computer Science, Weizmann Institute of Science, Rehovot, Israel, 1985.

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

[17] Saraswat, V., Concurrent constraint programming languages. PhD thesis, Computer Science Dept, Carnegie-Mellon University, 1989.

[18] Seitz, C., The cosmic cube, *CACM* 28(1), 22-33.

[19] Shapiro, E., Concurrent Prolog: a progress report, *IEEE Computer*, 1986.

[20] Shapiro, E. and Safra, M., Multiway merge with constant delay in Concurrent Prolog, *New Generation Computing*, 4(2), 211-216.

[21] Takeuchi, A., How to solve it in Concurrent Prolog, unpublished note, ICOT, 1983.

[22] Taylor, S., *Parallel Logic Programming Techniques*, Prentice-Hall, Englewood Cliffs, N.J., 1989.

[23] Ueda, K., *Guarded Horn Clauses*, Eng.D. thesis, University of Tokyo, 1986.

[24] Warren,D.H.D., Applied logic — its use and implementation as a programming tool, SRI International Tech. Rep. 290, 1983.