Concurrency: Simple Concepts and Powerful Tools*

Ian Foster
Argonne National Laboratory

Carl Kesselman
The Aerospace Corporation

Stephen Taylor
California Institute of Technology

Abstract

Stepwise refinement is a central program development methodology that has been applied extensively to the design of sequential and parallel programs. In this methodology, a problem is successively decomposed into subproblems in order to untangle seemingly interdependent aspects of the design. To apply the methodology to parallel programs, one must be able to separate and reason about issues such as partitioning and mapping.

This paper describes programming language concepts that we have found useful in applying stepwise refinement to parallel programs. The concepts allow decisions concerning program structure to be delayed until late in the design process. This capability permits rapid experimentation with alternative structures and leads to both portable and scalable code. Although simple, the concepts form a sufficient basis for the construction of powerful programming tools. Both concepts and tools have been applied successfully in a wide variety of applications and are incorporated in a commercial concurrent programming system, *Strand*[1].

1 Parallel Program Design

Parallel program design can be *easy* if conducted via the appropriate methodology. The key idea is to separate concerns so that each aspect of a program design can be addressed in isolation. Fortunately, the stepwise refinement methodology used in the design of sequential programs [18] is equally applicable to parallel programming [2]. In this methodology, a program is *refined* incrementally from an initial specification; parallel aspects of the design such as partitioning and mapping can be introduced in distinct refinement steps.

*To appear in: *Computer Journal*
[1] *Strand* is a trademark of Artificial Intelligence Limited

Decisions concerning parallel aspects of a program design should be delayed until late in the refinement process. This approach encourages problem formulations in which much of a program's structure is independent of these decisions; thus, alternatives can be examined without substantial program modification. In addition, it is possible to achieve a degree of architectural independence: a program can be retargeted for a different architecture by modifying only the latter refinement stages.

Design decisions can be delayed only if preceding design steps do not commit the design to a specific architecture. For example, an early commitment to a globally shared data structure, as an means of communication between subprograms, may hinder subsequent partitionings for execution on multicomputers. Early commitments in the design can be avoided by adopting an abstract, architecturally independent view of communication, synchronization, and concurrent execution. Surprisingly, this architectural independence can be achieved by using a programming model based on only four simple concepts: monotone variables, concurrent interleaving, non-deterministic choice, and separation of sequential code. Furthermore, these ideas provide a foundation on which powerful programming tools can be built to support each step of the program development cycle. The tools include portable programming environments, low-overhead performance analyzers, support for the integration of existing sequential code, and compilers that allow parallel program structures to be reused.

This paper describes the framework that we have developed to support the application of stepwise refinement in parallel programs. The ideas and tools that we describe have been used to develop portable applications on a wide range of hardware platforms including multicomputers, networks of workstations, and shared-memory machines. The exposition in this paper uses the syntax of the concurrent programming language *Strand* [9]; however, the underlying concepts are language independent. Some of the ideas have been useful in recent research concerned with the central notion of program composition [3].

2 Simple Concepts

Four simple ideas are sufficient to express a wide variety of concurrent computations. The notion of *monotonicity* provides an abstract model of communication and synchronization. Programs are constructed by a *concurrent interleaving* of component programs. *Non-deterministic choice* is used to select between alternative program actions. Finally, *separation of sequential code* simplifies the introduction of state change and sequencing.

Monotonicity. Components of a parallel program may exchange information via shared monotone variables. A monotone variable is initially undefined; it can be assigned at most a single value and subsequently does not change. A program that requires the value of a variable waits until the variable is defined.

A shared monotone variable can be used to both communicate values and synchronize actions. For example, consider two programs producer and consumer that share a variable X:

$$\mathsf{producer(X),\ consumer(X)}$$

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

$$\mathsf{producer(X)}\ :-\ \mathsf{X}\ := \text{"}\mathsf{msg}\text{"}.$$

The consumer program may receive the value and use it in subsequent computation:

$$\mathsf{consumer(X)}\ :-\ \mathsf{X} == \text{"}\mathsf{msg}\text{"}\ \mid \mathsf{use(X)}.$$

The concept of *synchronization* is implicit in this model. In the example, the comparison X == "msg" can be made only if the variable X is defined. Hence, execution of the consumer is delayed until the value is available.

Monotonicity is valuable for two reasons. First, a program can be understood in isolation: choices made on the basis of monotone variables cannot change. This attribute eases the understanding of concurrent programs and avoids errors caused by time-dependent interactions. Second, the concept is trivial to implement efficiently: it maps directly to pointers within a single computer and to message passing between computers. Once available, the value of a variable can be propagated throughout a parallel machine without concern for consistency of copies [16]. Hence, programs can operate on distributed shared data without locking protocols or complex synchronization schemes.

Interleaving. A program component is able to execute when its data is available; if the data is available, the program is guaranteed to execute eventually. The order in which programs execute is not otherwise constrained. In particular, programs can be executed in parallel.

A consequence of monotonicity and interleaving is that it is not important where and when program components execute. Hence, decisions concerning partitioning, mapping, and granularity can be isolated from the rest of the program design process.

Choice. Programs must inevitably choose between alternative actions; this choice is based on the values of variables. We adopt a simple method of specifying program actions that makes such choices explicit and avoids overspecification [6]. The following program, which computes the maximum of two numbers, illustrates the concept.

$$\begin{array}{l} \mathsf{max(X,Y,Z)}\ :-\ \mathsf{X} >= \mathsf{Y}\ \mid \mathsf{Z} := \mathsf{X}. \\ \mathsf{max(X,Y,Z)}\ :-\ \mathsf{X} =< \mathsf{Y}\ \mid \mathsf{Z} := \mathsf{Y}. \end{array}$$

Informally, the two rules in this program specify two alternative actions, each with an associated condition. If X$\geq$Y, the output Z is defined to be X. Alternatively, if X$\leq$Y, Z is defined to be Y. If X=Y, either action can be performed. The program can be understood in terms of pre- and postconditions: if X>Y holds, Z=X will hold eventually, while X<Y leads to the postcondition Z=Y and X=Y to the postcondition X=Y=Z.

This intuitive understanding of the program is valid because of monotonicity and interleaving. The monotonicity of X and Y ensures that the preconditions are also monotone. For example, once X$\geq$Y, this condition holds forever and cannot be affected by actions performed by other programs. Interleaving ensures that once a precondition is satisfied, a valid postcondition will eventually be reached.

Separation of Sequential Code. State change and sequencing are familiar concepts from sequential programming. State change permits efficient management of memory via destructive operations to storage locations; sequencing permits state changes to be organized without the overhead of explicit synchronization operations on each access to data [11]. We choose to make these concepts available via simple interfaces to conventional languages such as C and Fortran. These interfaces allow a sequential program segment to be treated as an atomic black box that computes an input/output relation. Hence, it can be characterized in terms of pre- and postconditions in the same way as parallel program components.

This approach has a number of benefits: It achieves a clean separation of concerns between sequential and parallel programming; it provides a familiar notation for sequential concepts; and it enables existing sequential code to be reused in parallel programs.

Summary. Monotonicity, interleaving, and choice provide an abstract method for specifying parallel computations. Sequential components of these computations can be expressed with existing languages. The notion of monotonicity is at the heart of concurrent logic programming [4, 13], functional programming [12], and object-oriented programming [1]; the other ideas have also been examined extensively. Unfortunately, the ideas are frequently embedded in complex language designs and are obscured by the terminology associated with a particular programming paradigm. We consider the ideas to be a sufficient basis for parallel programming in and of themselves. Moreover, we find that they can be implemented efficiently on a wide variety of architectures.

3 Simple Programming Techniques

Experience in the use of the basic concepts listed previously has resulted in the isolation of six simple programming techniques [9]. Applications are constructed by the repeated use of these techniques in different combinations and guises. Understanding the programming concepts described earlier amounts to no more than an appreciation of the structure and application of the techniques.

The first three techniques are used to express various stream-based interprocess communication protocols. The *producer/consumer* protocol allows unbounded communication between a single producer and one or more consumers. The *incomplete message* protocol allows two-way unbounded communication. Finally, the *bounded buffer* protocol organizes communication so as to bound the number of unreceived messages.

The latter three techniques comprise the difference list [5], short circuit [15], and monitor. The *difference list* is a representation of a list that allows it to be constructed in parallel by many producers. The *short circuit* is used to detect termination of a set of program components. Finally, the *monitor* (or blackboard) allows concurrent but atomic access to a shared data structure.

We outline here the *most complex* technique, the bounded buffer. This organizes communication between a producer and a consumer so as to bound the number of unreceived messages, by using a message buffer via which all communication is conducted. The producer generates messages only if there is space in the buffer; the consumer is responsible for creating space when it removes elements.

```
producer(N,[M | Ms]) :-           % R1. Wait for buffer
    N > 0 |                       % Still generating?
        N1 is N - 1,              % One less to generate
        M := "msg",               % Generate a message
        producer(N1,Ms).          % Recurse to generate more
producer(0,[M | Ms]) :-           % R2. All done?
        M := "done".              % Close stream

consumer(["msg" | Ms],End) :-     % R3. Wait for "msg"
        End := [X | End1],        % Extend buffer
        consumer(Ms,End1).        % Consume rest
consumer(["done" | Ms],_).        % R4. Terminate
```

Figure 1: The Bounded Buffer Protocol

Figure 1 illustrates the implementation of this protocol. The following program creates an initial buffer represented by a list of five unbound variables. The producer is given access to the beginning of the buffer; the consumer is given access to both the beginning and end.

```
go() :-
      Buffer := [M1,M2,M3,M4,M5 | End],
      producer(100,Buffer),
      consumer(Buffer,End)
```

The producer waits for an element of the buffer to be available (R1) and then assigns this element a value representing a message (M := "msg"). The producer may terminate the protocol by adding a "done" message to the buffer (R2). If no element is available, the producer suspends. The consumer suspends until messages are available and then consumes them from the beginning of the buffer (R3). Each time it receives a message, the consumer creates space in the buffer by appending an unbound variable to the end (End := [X | End1]). When the consumer receives a "done" message it terminates (R4). Hence, execution of the initial set of processes causes 100 "msg" messages to flow from the producer to the consumer; no more than five messages are outstanding at any one time.

4 Powerful Tools

The concepts developed in previous sections permit the design of powerful programming tools that aid in every step of the program development cycle. Here we describe a representative selection of the tools that we have implemented. A *portable programming environment* provides an architecturally independent implementation of the programming concepts. Low-overhead measurement techniques provide a basis for *performance analysis tools*. A *foreign language interface* enables the integration of sequential code. Finally, compiler tools support *algorithmic abstractions*, allowing the reuse of program structures in different applications and on different architectures.

4.1 Portable Programming Environment

Portability has always been a central issue in computing, as it protects long-term investment in software. On sequential machines, standard operating systems and utilities have evolved that permit a significant degree of portability. Unfortunately, no clear standards are available for parallel machines either in software or hardware. Moreover, new and improved parallel machines will undoubtedly continue to appear. Hence, an important requirement for a parallel program development environment is that it provide a uniform interface on a variety of parallel machines.

There are three aspects to such an interface. Application codes must be able to execute on different architectures without modification (*application portability*). It must be possible to achieve efficient implementations of the environment on a wide variety of architectures (*system portability*). Finally, application codes must be able to utilize an increasing number of computers as additional hardware becomes available (*scalability*). These three goals can be achieved by using three simple concepts: an abstract machine, servers, and virtual machines.

Abstract Machine. An abstract machine is an architecture-independent compiler target language [17]. The use of an abstract machine greatly reduces the cost of porting a programming system to a new parallel computer: only a small machine-dependent back-end and/or run-time system need be developed to support a specific architecture. Our experience with *Strand* shows that this machine-specific code often comprises less than one hundred lines of C source [10].

On a parallel computer, an abstract machine must incorporate the central notions of communication and synchronization [16]. Recall that these notions can be expressed in terms of monotonicity. We find that monotonicity can be implemented by using just two generic operations: unblocked send and receive. These operations can be implemented readily on architectures as diverse as multicomputers (using message-passing), networks (using low-level network protocols) and shared-memory machines (using pointers) [10].

Servers. A server is a program that *encapsulates* and provides a uniform interface to system facilities such as device I/O. Servers are designed to invoke system facilities in response to messages received on an input stream; the messages originate in application codes. Servers can be implemented easily by using the concepts of monotonicity and concurrent interleaving [7].

A server may route a request to another, remote server if a required facility is not available locally. Hence, application codes can be provided with uniform access to system-wide facilities such as user interfaces and concurrent file systems.

Virtual Machines. A virtual machine provides an abstract architecture over which programs can be mapped using simple annotations [16]. A simple example of such a machine is a linear array of computers; a program can be mapped recursively to successive computers in this array by using an annotation `@fwd`. Mesh and tree topologies can also be supported. It is easy to build sophisticated program structures on top of these simple facilities.

Figure 2 illustrates these ideas using a simple matrix-multiply program. This program computes the matrix multiplication $C = A \times B^{-1}$. A is represented by a list of row vectors; the transposed matrix B is represented by a list of column vectors. The program is invoked with a call of the form `mm(A,B,C)` and executes `mm_row` processes on successive computers in a linear array.

A virtual machine may be arbitrarily large and can be automatically *embedded* in a smaller physical machine. This approach simplifies the task of mapping a program to an architecture by reducing the problem to two separate activities: mapping a program to a virtual machine and mapping a virtual machine to an architecture. The first of these activities can be performed easily, as the programmer can select a virtual machine that is

```
mm([A | As],Bs,Cs) :-
      mm_row(A,Bs,C),
      Cs := [C | Cs1],
      mm(As,Bs,Cs1)@fwd.
mm([],_,Cs) :- Cs := [].
```

Figure 2: Matrix Multiply

convenient for the problem structure. The second can be completely automated with compiler tools. Virtual machines also provide portability and scalability [9].

It is important to note that a virtual machine is simply a mechanism for directing program mapping: it does *not* restrict possible communication patterns. The underlying run-time system ensures that any two programs sharing a monotone variable can communicate directly irrespective of their location.

Summary. Sophisticated programming environments can be designed based on only the core concepts and implemented by using only the basic programming techniques. These environments allow programmers to design portable, scalable applications. Moreover, the environments themselves are both portable and scalable.

4.2 Performance Analysis Tools

All programs constructed with the simple concepts that we have emphasized have a common form: a program is a collection of choices expressed by rules; computation consists of repeated selection and execution of these choices. This regularity of structure permits the use of low-overhead performance monitoring techniques. These techniques heavily emphasize the use of compile-time information to reduce the amount of data that must be collected at run-time [14].

The expected cost to execute each choice in a program is determined by a static analysis of the program text in combination with architecture-specific calibration data. Dynamic counter values are collected during a particular program execution by instrumentation inserted by the compiler. The counters measure execution frequencies for particular choices; this information is sufficient, when combined with the static information, to compute a variety of performance metrics. These include execution and idle time on a per-program basis, processor utilization, and communication traffic.

Experience indicates that performance estimates obtained by using these techniques are within 10% of their actual value. The run-time overhead for instrumentation is less than 3%; hence, performance analysis has become an integral part of the program development cycle. In addition, the techniques extend naturally to integrate timing information for foreign language program components, collected by explicit timers.

A visualization tool promotes the exploration of the performance data by providing interactive statistical analysis and data summary facilities. An example processor-utilization display from this tool is shown in Figure 3. Processors are listed horizontally and programs vertically; the darkness of a square in the display indicates the level of activity. The program illustrated is a similarity search program that seeks to find close matches for a DNA sequence in a large database. This example illustrates how the tool can provide insights into the global operation of an application. Processor 1 executes relatively infrequently; it is responsible for coordinating concurrent operations on the database. The remainder of the nodes perform similar activities, although some load imbalance is apparent on processors 13 and 15. Closer investigation showed that this imbalance was associated with initial ramp-up and final ramp-down periods in the computation during which not all processors were busy.

4.3 Foreign Language Interface

Recall that monotonicity avoids the time-dependent errors that are often introduced when concurrent programs modify shared data; it also simplifies parallel implementation. However, exclusive use of monotonicity hinders the efficient implementation of algorithms which can be conveniently expressed in terms of sequential updates to data structures. We recognize the importance of sequentiality and updates. However, for simplicity, we choose to isolate the use of these concepts in distinct program components expressed in existing languages such as C or Fortran.

We achieve a clean separation of concerns between sequential and parallel components of programs by requiring that sequential components adopt a *monotone interface*. That is, we disguise programs that use modification to look like programs that use monotonicity. For example, consider the problem of computing the input-output relation N1=N+1. A monotone implementation will construct a new value N1 corresponding to the sum of N and 1. An alternative implementation may update N in place and return the result as N1; this may be safely used if no other concurrent program can access N. The safety property can be guaranteed trivially by encapsulating N inside a program.

An example of this use of modification can be seen in Figure 2. The `mm_row` routine called to compute a row of the output matrix can be implemented in a low-level language such as Fortran; moreover, if the programmer is able to verify that the input row `A` is unused in subsequent computation, the routine can reuse the storage occupied by `A` to hold the result row `C`.

This approach encourages the introduction of updating and sequencing in the final stages of program development. It permits local refinements that

Figure 3: Example Performance Display

enhance the efficiency of low-level operations. It also allows the integration of segments of existing code into parallel programs, providing a transition path from sequential to parallel programming. The technique is particularly beneficial in numeric computations involving grids, since the storage used to hold a grid point can be reused in successive iterations.

4.4 Algorithmic Abstractions

In the long term, with the availability of many thousands of computers, the structural organization of computations must become a dominant theme in parallel programming. It is not practical to manage at a microscopic level the detailed operation of thousands of cooperating computers. In consequence, more abstract views of concurrent computation are required.

Many of the problems that we face in parallel programming have already been tackled in the area of VLSI design. The central concept in this discipline is the use of hierarchy to control complexity: large circuits are built by replicating and combining smaller units. The combining functions have well-understood properties, permitting the internal details of the resulting forms to be ignored in subsequent stages of the hierarchy. Moreover, both subunits and combining forms can be archived and applied in alternative design problems.

In the same way, we now seek to structure parallel computations by combining well-understood program units. As in the case of VLSI, we focus on the individual units and the methods by which they are combined.

Reflecting upon a number of parallel applications that we have developed, we observe that the same parallel structures are used again and again, albeit in different guises. For example, many numerical simulations such as those occurring in climate modeling and fluid dynamics are structured as a grid of programs; each node executes the same basic cycle involving nearest-neighbor communication, computation, and global exchange. Another example where we have noticed commonality of structure is in simple load-balancing and bin-packing strategies. We have applied the same basic strategies repeatedly to problems as diverse as DNA matching, protein structure prediction, game trees, and molecular dynamics. Finally, we observe that these different computations often involve common subunits such as spanning trees for broadcasting information and combining global metrics.

In order to reuse these program structures and to combine them in program hierarchies, we seek to achieve an abstract, application-independent formulation. To illustrate how this may be achieved, we consider the problem of implementing a generic "grid computation" abstraction. In this abstraction, a set of identical programs cooperate to compute an approximation to a differential equation. The program in Figure 3 provides the top-level structure of this abstraction. It is parameterized with the size of the grid (Nodes), a termination condition (Threshold), and four functions. These will be used to initialize a subgrid (InitFn), extract and store boundary data for nearest neighbor communication (GetFn, PutFn), and advance the state of a subgrid computation (ComputeFn). Calls to these functions in the program are distinguished by an initial upper-case letter; calls to subprograms that form part of the abstraction have an initial lower-case letter.

```
grid_solve(Nodes,Threshold,InitFn,GetFn,PutFn,ComputeFn) :-
    grid_compute(Nodes,Threshold,InitFn,GetFn,PutFn,ComputeFn).

solve(Id,Streams,Threshold,InitFn,GetFn,PutFn,ComputeFn) :-
    InitFn(Id,State),
    InitialError is Threshold + 1.0,
    iterate(InitialError,Threshold,GetFn,PutFn,ComputeFn,State,Streams).

iterate(Error,Threshold,GetFn,PutFn,ComputeFn,State,Streams) :-
  Error > Threshold |
    grid_exchange(GetFn,PutFn,State,State1,Streams,Streams1),
    ComputeFn(State1,State2,MyError),
    global_combine(MyError,MaxError,Streams1,Streams2),
    iterate(MaxError,Threshold,GetFn,PutFn,ComputeFn,State2,Streams2).
iterate(Error,Threshold,_,_,_,_,Streams) :-
  Error =< Threshold | close(Streams).
```

Figure 4: Grid Computation Abstraction

The abstraction assumes a particular grid structure created by the subprogram grid_compute. This subprogram partitions the grid into subgrids, maps the subgrids to computers, establishes communication streams between nearest neighbors, creates a spanning tree for global operations, and invokes a solve program at each subgrid. Note that each of these operations is independent of both the application and the abstraction.

The solve program first initializes its local data state using the supplied InitFn. It then iterates until a grid-wide error is less than the specified threshold. The iterate subprogram is defined in terms of three other subprograms. The grid_exchange subprogram uses the supplied GetFn and PutFn to extract and store boundary values communicated with nearest neighbors. The supplied ComputeFn performs the subgrid computation, producing a new state. Finally, the global_combine subprogram uses the spanning tree established by grid_compute to obtain a global error value.

This program may be reused to obtain parallel implementations of any problem that can be formulated in terms of this particular grid structure and the four function arguments. However, in its present form it is both clumsy and inefficient. The arguments introduced to represent function names and communication streams obscure the program structure. In addition, both the passing of these arguments and the repeated higher-order calls are likely to be sources of run-time overhead. Hence, we prefer in practice to use compiler tools to combine the four application-specific programs with the abstraction [8]. This approach permits a more succinct representation of the abstraction that does not require the functional arguments. The compiler tools allow the linking of the application code with the abstraction to be achieved via an automatic source-to-source transformation. A powerful metalanguage allows the transformation to be specified in a few lines of code.

5 Conclusions

The perceived difficulty of programming parallel computers has often led to an unfortunate emphasis on language design. However, languages do not solve problems: they merely provide a means of describing solutions.

The design and implementation of problem solutions are primarily a programming task and must be supported by an appropriate methodology. Standard stepwise refinement techniques are ideally suited for parallel computers; however, additional refinement steps are required to introduce design decisions concerned with partitioning and mapping. We have emphasized the importance of delaying these decisions until late in the design process. The key feature that we look for in a language is that its concepts support a separation of concerns between algorithmic specification and parallel implementation.

We have described a *core* set of simple concepts that achieves this separation: monotonicity, interleaving, choice, and separation of sequential code. These are not necessarily the only concepts that could be employed ,but we have found them particularly useful and pervasive. We are convinced that they are a *sufficient* basis for practical parallel programming. Powerful programming tools have been built with the concepts; several of these tools are described in this paper.

The concepts and tools have been incorporated in a commercially available concurrent programming system called *Strand*. This programming system operates on a wide variety of parallel architectures. It has been used to develop substantial applications in areas as diverse as molecular dynamics, computational biology, climate modeling, fluid dynamics, and telephone exchange control.

The tools are now available and will naturally improve over time: the emphasis must now turn to application development. The challenge is to develop experience in solving significant scientific problems on large parallel computers. Armed with this experience, we can hope to build the next generation of advanced programming tools. We expect these tools to be based on the use of abstractions [8] as a means of organizing concurrent computations and on methods that will allow programmers to understand programs constructed by using these abstractions [3].

Acknowledgments

The exposition of the basic concepts and our appreciation of their relevance to computer science have been inspired by insightful discussions with K. M. Chandy.

This research, in the case of the first author, was supported by the Applied Mathematical Sciences subprogram of the office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. The third author was supported by ONR and AFOSR under Contract N00014-89-J-2301.

References

[1] Agha, G., *Actors*, MIT Press, 1986.

[2] Chandy, K. M. and Misra, J. *Parallel Program Design*, Addison-Wesley, Reading, Mass., 1988.

[3] Chandy, K. M. and Taylor, S., Composing parallel programs, *Beauty Is Our Business*, Springer-Verlag, 1990.

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

[5] Clark, K. and Tarnlund, S.A., A first order theory of data and programs, *Information Processing 77; Proc. IFIP Congress 77*, North Holland, 1977, 933–944.

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

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

[8] Foster, I., Automatic generation of self-scheduling programs, *IEEE Transactions on Parallel and Distributed Computing* (to appear).

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

[10] Foster, I. and Taylor, S., Strand: A practical parallel programming tool, *Proc. North American Conf. on Logic Programming*, MIT Press, 497–512, 1989.

[11] Gajski, D., Padua, D., Kuck, D., and Kuhn, R. A second opinion on dataflow machines and languages, *IEEE Computer*, 15(2), 58–69, 1982.

[12] Henderson, P., *Functional Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1980.

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

[14] Kesselman, C., *Integrating Performance Analysis with Performance Improvement in Parallel Programs*, Ph.D. thesis, UCLA (in preparation).

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

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

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

[18] Wirth, N., Program development by stepwise refinement, *CACM,* 14, 1971, 221–227.