CONF-9008181--1

DE91 006030

# Bilingual Parallel Programming

Ian Foster and Ross Overbeek
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

## 1 Introduction

Numerous experiments have demonstrated that computationally intensive algorithms support adequate parallelism to exploit the potential of large parallel machines. Yet successful parallel implementations of serious applications are rare. The limiting factor is clearly programming technology. None of the approaches to parallel programming that have been proposed to date — whether parallelizing compilers, language extensions, or new concurrent languages — seem to adequately address the central problems of portability, expressiveness, efficiency, and compatibility with existing software.

In this paper, we advocate an alternative approach to parallel programming based on what we call *bilingual programming*. We present evidence that this approach provides an effective solution to parallel programming problems.

The key idea in bilingual programming is to construct the upper levels of applications in a high-level language while coding selected low-level components in low-level languages. This approach permits the advantages of a high-level notation (expressiveness, elegance, conciseness) to be obtained without the cost in performance normally associated with high-level approaches. In addition, it provides a natural framework for reusing existing code.

The roots and motivations for bilingual programming predate parallel computers; they grow naturally out of fundamental issues associated with the programming task on *any* computer. Hence, we first review some of the lessons that have been learned during the past thirty years of programming uniprocessors. This background allows us to present the central tenets of bilingual programming as logical developments of fundamental concepts in programming methodology, rather than ad-hoc attempts to address issues that arise with concurrency. We then argue that the additional complexity of parallel programming makes the bilingual approach particularly attractive on parallel computers.

In the latter part of the paper, we introduce a particular bilingual approach with which we have considerable experience. The programming language Strand is used as the high-level concurrent language; lower-level routines are coded in C or Fortran. We summarize our experiences developing large parallel applications in computational biology, weather modeling, and automated reasoning. Finally, we review other approaches to parallel programming in the light of our analysis of the bilingual approach.

## 2 The Program Development Process

The program development process can be viewed as the formulation and implementation of a set of *design decisions* concerning algorithms and data representations. Research in program development methodologies and computer languages has produced effective techniques for reducing the number of design decisions associated with a program, limiting the effect of individual decisions, and simplifying the expression of decisions.

**Program Development Methodologies.** Two fundamental program design methodologies — stepwise refinement [20] and modular decomposition [15] — seek to reduce development and maintenance costs by encouraging programmers to respectively *defer* and *localize* design decisions.

The key idea in *stepwise refinement* is to tackle a task by repeatedly dividing it into smaller and smaller subtasks. The refinement process starts with an abstract specification for an algorithm and proceeds via a series of refinement steps to obtain an executable program. Each refinement step involves a number of design decisions concerning how a particular task and its data are to be implemented. Design decisions concerning representational details are deferred for as long as possible.

A deficiency of stepwise refinement is that it does not encourage the recognition of commonalities in a design. Application of stepwise refinement alone can result in similar problems being solved many times in the context of different subtasks. This complicates development and makes subsequent modification of a program more difficult.

An alternative methodology, *modular decomposition*, is used in large programs to address the deficiencies of stepwise refinement. The key idea in this methodology is to start the design process by identifying components of a program that will be common to several tasks or likely to change. The program is then developed as a set of modules, each encapsulating one or more such components. Duplication of effort is avoided, and subsequent modification of the program is easier because, in general, only a small number of modules are affected by any one change.

In summary, we see that the program development process has both top-down and bottom-up aspects. However, for our purposes we find it most natural to understand the end product in terms of a sequence of refinements of the original program specification (i.e., a top-down representation). Each step in the sequence introduces implementation commitments. The final program represents the outcome of an extended sequence of such commitments.

**High-Level Languages.** High-level languages reduce programming effort by specifying standardized implementation decisions for certain abstractions. They permit programmers to truncate the development process at an earlier stage than would otherwise be possible. For example, LISP provides list manipulation and memory management facilities; hence, a design that is expressed in terms of a "list" data type requires no further refinement if LISP is used as an implementation language. In contrast, an implementation in a lower-level language would require additional design decisions to produce a representation of lists and an implementation of the associated operations.

Unfortunately, standardized implementation decisions provided by high-level languages are unlikely to be optimal for all situations. A programmer can generally improve perfor-

mance by making additional design decisions that exploit application-specific knowledge. For example, the abstract data type "list of elements" may be implemented as a singly linked list, a doubly linked list, or an array; each implementation will be effective in different circumstances. No existing high-level language compiler is able to determine which strategy is optimal for all possible situations. Hence, there is necessarily a trade-off between implementation effort and efficiency.

# 3 Bilingual Programming

When developing programs, we are frequently interested in *minimizing* development and maintenance costs and *maximizing* performance. The cost of a program is closely related to the number of design decisions (commitments) made during development. Each decision leads to a development cost (incurred when the decision is implemented) and a possible maintenance cost (incurred when changing circumstances lead to backtracking on the original decision). Hence, one way of reducing costs is to adopt the standardized implementation decisions offered by a high-level language. Another is to adopt decisions encapsulated in existing code.

The use of a high-level language reduces costs but, as noted previously, adversely affects performance. Optimization of performance requires the substitution of application- and environment-specific design decisions for the standardized decisions encapsulated in high-level languages. Fortunately, it is generally the case that only a small proportion of the total design need be considered when seeking to optimize performance. Hence, it is usually possible to construct a program that provides "almost" optimal performance but that retains a significant portion of its logic in a high-level language. Such a bilingual program retains the advantages of a high-level language program without sacrificing performance.

The bilingual approach can be seen as a logical outgrowth of developments in methodologies and languages. Yet clearly it is not widely used in sequential programming today. We attribute this fact to accidental rather than intrinsic factors. In particular, the lack of standard interfaces between languages has hindered the development of portable bilingual programs. However, we see more promising possibilities on parallel computers, where standards remain to be defined and new problems demand new solutions.

# 4 Bilingual Parallel Programming

The design and development process for parallel programs is in many respects similar to that for sequential programs. However, the need to manage multiple processors introduces additional complexity: Problems concerned with concurrent execution, communication, synchronization, partitioning, mapping, load balancing, and data distribution must be addressed. Low-level solutions to these problems often compromise two highly desirable properties in a parallel program: scalability and portability. Hence, the developer of parallel programs is faced with both additional complexity and pressing reasons for deferring and localizing commitment to the design decisions required to address this complexity. This suggests that high-level languages may have a particularly important role to play.

At the same time, the primary motivation for parallel computation is performance. This point requires emphasis: the view that only algorithms are important, and that one can thus disregard the "small constant factors" introduced by the use of a high-level language, is folly. The effort required to develop a concurrent program can only be justified if an application requires substantial performance, and in most applications performance requires both good algorithms and low-level optimizations. Hence, we expect bilingual programming techniques to be particularly useful on parallel computers.

A high-level language to support parallel programming must provide linguistic support for important concurrent programming concepts: process management, communication, and synchronization. It should encourage portability by allowing programmers to express concurrent algorithms in a machine-independent way and by minimizing the effort (if any) required to specialize machine-independent programs for a particular computer. Finally, it should encourage the development of scalable applications by supporting separate specification of concurrency on the one hand and partitioning and mapping on the other.

It is presumably possible to design a language that provides these features and also supports the efficient implementation of low-level sequential algorithms. However, we believe that it is advantageous to work with a small, clean high-level language with a simple concurrent semantics. This simplifies understanding, analysis, and transformation of concurrent programs. The design space for high-level languages with these properties is presumably large. However, we have had good success with one particular language: the concurrent logic programming language Strand. We will restrict subsequent discussion to this particular context.

The decision to work with Strand represents an initial design decision that restricts the class of parallel algorithms that can be expressed in two ways. First, we are committed to MIMD rather than SIMD as the architectural model. This represents a personal preference. Second, we are committed to working within the message-passing model on which concurrent logic programming languages are based. Space does not permit a detailed justification of this decision. However, we point out that message-passing models have proved adequate for the vast majority of concurrent applications. Furthermore, they tend to simplify application development by reducing opportunities for unexpected interactions between concurrent processes.

# 5 Bilingual Programming in Practice

We now summarize the key features of an approach to bilingual parallel programming with which we have had considerable experience.

## 5.1 Specifying Concurrent Computation

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

practical parallel programming tool. Here, we provide a brief introduction to Strand and the parallel programming abstractions that it supports. A more complete description may be found in [11].

Computation in Strand is performed by sets of cooperating lightweight processes. These processes communicate and synchronize by reading and writing shared, single-assignment variables. A rule-based notation is used to specify process behavior. A program is a set of *guarded rules* with the form

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

where $H$ is the head of the rule, the $G$'s are its guard, and the $B$'s are its body. The head, guard, and body are all processes; a process has the form $F(T_1, ..., T_k)$, $k \geq 0$, where the $T$'s are terms. A term is a list structure (denoted $[Head|Tail]$), a tuple (denoted $\{T_1, ..., Tj\}$), a variable (denoted by a string starting with an uppercase letter), or a constant (denoted by itself).

Each rule specifies a set of preconditions that must be satisfied for a process to execute, and the actions to be performed if these preconditions are satisfied. Preconditions are expressed by non-variable terms in the head of a rule (which must match corresponding process arguments) and guard tests. For example, the rule

$$\mathsf{db([lookup(K,V) \mid In],Db)} \ :- \ \mathsf{K} \neq 0 \ \mid \mathsf{search(Db,K,V),\ db(In,Db).}$$

specifies that a process db can be replaced by two new processes search and db if its first argument has the form [lookup(K,V) | In] and $\mathsf{K} \neq 0$.

Strand variables have the single assignment property: Their value is initially undefined and once defined cannot be modified. Head matching and guard tests correspond to read operations on variables. An attempt to read a variable for which no value has been defined causes a process to suspend; this is Strand's synchronization mechanism. Variables are written by using a predefined *assign* process: a process $X := T$ assigns the value $T$ to the variable $X$.

Shared variables can be used to express a wide variety of communication patterns. A commonly used structure is the *stream*. A stream producer and one or more stream consumers initially share a single variable. The producer instantiates this variable to a list structure containing a message in its head and a new variable in its tail that can be used for further communication, for example:

$$\mathsf{X := [msg1 \mid X1],\ X1 := [msg2 \mid X2],\ ...}$$

## 5.2 Partitioning and Mapping

A Strand program indicates opportunities for concurrent execution but does not specify how these opportunities are to be exploited. The partitioning of the set of concurrently executing processes into tasks and the mapping of these tasks to the nodes of a particular parallel computer represent separate design decisions that need not be made until after a program has been developed. These decisions effect the efficiency of the final program but

do not in general effect its correctness. Tools provided with Strand systems reduce the effort required to implement these decisions [11, 8]. Hence, programs are tend to be easily portable: little or no effort is required to adapt a program for a new parallel computer.

The extent to which programs can be specialized to employ different partitioning and mapping decisions depends in part on the structure of the original code. For example, a program designed to solve a grid problem will probably be developed with a particular domain decomposition in mind, and will explicitly create the process network required to handle this decomposition on a parallel computer. This program has already been substantially specialized by the programmer and requires significant rewriting before an alternative decomposition can be employed.

On the other hand, consider the following program that encodes the top level of a state-space search problem. A tree is searched by first exploring a fixed number of nodes with a depth first strategy, then splitting any remaining subtree into a number of simpler trees, each to be searched in turn. This program specifies opportunities for concurrent execution in the state-space search (each subtree can be searched concurrently) but says nothing about how processes are to be mapped to processors.

```
search(Params,[Prob | Probs],Solns,Solns2) :-      % To search nodes,
    search_subtree(Params,Prob,Solns,Solns1),      %   search one node, &
    search(Params,Probs,Solns1,Solns2).            %   and search rest.
search(_,[],Solns,Solns1) :- Solns := Solns1.      % Search done.

search_subtree({M,N},Prob,Solns,Solns2) :-         % To search single node,
    process_prob(Prob,M,NewProb,Solns,Solns1),     %   expand node,
    split_prob(NewProb,N,Probs),                   %   split subtree.
    search({M,N},Probs,Solns1,Solns2).             %   and continue.
```

This problem can be partitioned and mapped in several different ways: static embedding, dynamic load balancing, and random mapping are all possibilities. Once a design decision has been made, a more specialized program that implements the decision can be obtained from the code shown here. Strand's high-level nature makes it easy to implement these specializations as automatic source-to-source transformations that can be encapsulated in libraries [8, 10]. The important thing to note is that a completely different parallel code can be obtained by making a different design decision at this stage, and that the cost of making or backtracking on these decisions is small.

### 5.3 Interfacing to Foreign Code

Effective mixed-language programming requires a clean and simple interface between heterogeneous components. This is achieved in Strand with two mechanisms, *user-defined operations* and *user-defined data types*. These are used to encapsulate foreign code and foreign data, respectively.

A user-defined operation invokes a foreign procedure. Strand ensures that a user-defined operation is scheduled only when data that it requires to execute is available.

The foreign procedure can then perform a finite amount of sequential execution. Upon completion, values computed for output arguments are returned to the concurrent component, which may pass them as arguments when invoking other user-defined operations. Note that user-defined operations are not coroutined on a single processor; instead, the execution of each operation is viewed as an indivisible action.

The Strand compiler automatically generates code to perform type conversion between Strand and other foreign representations of simple data types such as integers, reals, and strings. More complex foreign data structures (e.g., arrays) can be encapsulated in a special *user-defined data type*. Strand programs can pass user data from one user process to another but cannot examine their contents. As user data may be migrated from one processor to another, such data cannot contain addresses. This is the only restriction placed on their contents.

Foreign procedures can also be passed Strand data structures directly. This capability is useful when data is naturally represented in terms of Strand record structures (lists or tuples). Macro libraries are provided that allow the foreign procedure to access the Strand structures as abstract data types.

The foreign interface imposes a certain discipline on the programmer, as it requires that all communication between user processes be achieved by argument passing; common areas and other forms of global variables cannot (in general) be employed. This restriction has two important benefits. First, individual foreign procedures can be developed and debugged independently. Second, bilingual programs can be executed on both a single processor and multiple processors without modification.

## 5.4 Mutable Data

Recall that Strand variables have the single assignment property: their value is initially undefined and, once defined, cannot subsequently be modified. This property is important for several reasons. First, it avoids the race conditions that can arise when several concurrent processes read and write the same variable. Second, it permits a number of important optimizations in a parallel implementation. In particular, it permits non-variable data structures to be copied between processors without concern for the consistency of copies.

In contrast, sequential languages such as C and Fortran presuppose mutable data structures. Indeed, it is the ability to modify large data structures in place that permits these languages to provide succinct and efficient implementations of many algorithms.

We wish to maintain the single-assignment property in the concurrent component of bilingual programs, while permitting updates in foreign procedures. We achieve this by imposing the following restrictions on the operations that can be performed by programs. The user must be able to demonstrate that a bilingual program satisfies these requirements. Static analysis tools can assist the detection of improper programs [12].

1. *Updates are encapsulated in user-defined operations.* An operation that modifies a data structure (say $D$) must return as an output argument a *new reference* ($D'$) to that same data structure.

2. *Data structures are single-threaded.* At most one update operation can be applied to a particular reference to a data structure.

3. *Reads precede updates.* All read operations applied to a particular reference to a data structure must complete before any update operation is applied.

In these requirements, the term a "particular reference" is used to mean the "set of references that are equivalent modulo aliasing".

For example, assume that two user operations inc_vector(V,V1) and sum_vector(V,Sum) have been defined to increment and sum the elements of a vector (an integer array, represented as a user data type). The first operation returns a new reference V1 to the input vector V; the second computes the sum of its elements. Then the following set of processes correctly increments the vector V twice before summing its elements to yield S.

```
inc_vector(V,V1), inc_vector(V1,V2), sum_vector(V2,S)
```

On the other hand, the following set of processes would be illegal, as V would be updated by two processes.

```
inc_vector(V,V1), inc_vector(V,V2)
```

## 5.5 Abstract Data Types

Strand programs frequently need to deal with complex data types (such as arrays and record structures) constructed by foreign procedures. As these data do not correspond to any primitive Strand type, they will be passed to Strand as user data types. A closer integration of Strand and foreign components of a bilingual program can be achieved by extending a Strand system with user-defined operations to define appropriate abstract data types. For example, a one-dimensional integer array type may be implemented in Strand as a user data type (used to represent the array) as well as operations to read and write array elements:

- new_array(Size?,Array↑): returns an initialized Array of the designated size.
- size(Array?,Size↑): returns the Size of Array.
- get_int(Index?,Array?,Item↑): returns Item, the contents of element Index in Array.
- set_int(Index?,Array?,Item?,NewArray↑): updates element Index in Array to contain Item, and returns a new reference to the array, NewArray.

These operations allow Strand programs to access arrays created by foreign language procedures directly. For example, the following program creates a new array initialized to zero. Observe that this program obeys the requirements stated previously.

```
zero(Size,NewArray) :-
    new_array(Size,Array), zero_1(1,Size,Array, NewArray).

zero_1(I,Max,A,NewA) :-
  I ≤ Max | set_int(I,A,0,A1), I1 is I+1, zero_1(I1,Max,A1,NewA).
zero_1(I,Max,A,NewA) :- I > Max | NewA := A.
```

# 6 Experiences

We have provided a detailed account of some of our experiences developing bilingual parallel codes elsewhere [9]. Here we summarize our results and the conclusions reached.

In collaboration with colleagues, we have developed bilingual programs in computational biology, weather modeling, and automated reasoning. Each of these codes is a substantial application, used by scientists on a daily basis to support their research. In the following table, we characterize the codes in terms of the foreign language used and their code size.

| Program | Language | Foreign (lines) | Strand (lines) |
|---|---|---|---|
| Weather | Fortran | 25000 | 250 |
| Dynamics | Fortran | 4000 | 120 |
| Prover | C | 5000 | 400 |
| SimSearch | C | 500 | 210 |
| Alignment | C | 1200 | 1700 |

The codes cover a wide spectrum of applications and parallel algorithms. *Weather* is a large "dusty deck" numeric modeling code; it was parallelized using domain decomposition techniques. *Dynamics* is a molecular dynamics code; its performs a state-space search to find configurations of molecules that satisfy certain criteria. The top level of this code is essentially the fragment given in Section 5.2; it was parallelized using a dynamic load-balancing strategy. *Prover* implements a parallel theorem-proving algorithm based on a software pipeline. *SimSearch* solves a database search problem; a manager/worker structure is used to allocate parts of the database to idle processors. Finally, *Alignment* implements an algorithm for computing "alignments" of sequences of genetic material from different organisms; it was parallelized using functional decomposition and dynamic load-balancing techniques.

The first three applications use substantial amounts of pre-existing code. In these examples, the Strand component is used principally to coordinate the execution of the low-level sequential components. In contrast, *SimSearch* and *Alignment* were developed from scratch as bilingual applications. In the latter, the Strand component implements all but the most computationally intensive components of the algorithm.

All five applications went through several revisions in the course of their development. In each case, we found that the bulk of the modifications occurred in the high-level component of the code. The existence of a concise, high-level specification greatly simplified the exploration of alternative algorithms.

Our experience suggests that the bilingual approach permits the benefits of a high-level language to be attained without the performance degradation normally associated with high-level approaches. For example, in *SimSearch*, *Weather*, and *Dynamics*, we observed no significant difference in uniprocessor performance between the bilingual code and an equivalent sequential code written entirely in C or Fortran. Parallel programs coded using low-level facilities were not available for comparison with the bilingual programs on multiprocessors. However, performance studies suggest that communication time was not a dominant factor in any application.

The bilingual programs that we developed also proved to be highly portable. For example, *SimSearch* was run on eight different parallel computers. Porting was problematic only when low-level sequential code used non-portable constructs or excessive local memory.

We found that linguistic support for concurrent execution encouraged a modular approach to parallel programming and the encapsulation of parallel algorithms in libraries. Library code was frequently reused in other applications. For example, the state-space search code developed in *SPS* was used to develop parallel implementations of two different applications, simply by substituting alternative definitions for two low-level procedures. No changes to the concurrent component were required. At a lower level, both *SimSearch* and *SPS* used the same scheduler library code.

A final and important point is that we did not experience any particular difficulties developing the bilingual applications. We attribute this to the simple interface between the sequential and concurrent components. This permitted sequential procedures and concurrent programs to be developed and tested independently. Furthermore, the bilingual program could generally be tested on a single processor.

# 7 Related Approaches

It is instructive to review other approaches to parallel programming in the light of the analysis of program development methodologies presented in Section 2.

**Parallelizing Compilers.** For several years, a number of respected researchers have argued that the key to successful exploitation of parallel processors was advanced compilers [7, 14, 17]. They suggest that programmers should write standard Fortran, which compilers would automatically restructure to take advantage of parallel hardware.

This approach has proved successful when applied to fine-grained parallelism: vectorizing and trace-scheduling compilers give excellent results on certain codes. However, we argue that the approach is seriously flawed as a technique for exploiting large-grained parallelism, as it requires that programs be refined in an inappropriate sequence. Some of the refinements used to target a program toward a specific architecture must be achieved at a level well above that of a Fortran program. In particular, the refinement that com-

mits to an implementation based upon processes that communicate via message passing is naturally thought of as occurring before those that commit to specific implementations of abstract data types. The consequence of making these refinements out of sequence is that systematic analysis of the actual code becomes difficult if not impossible. Properties that could have been stated and verified when made about abstract data types before committing to a specific implementation (with the corresponding details introduced by memory management, etc.) become hard to express and verify.

**High-Level Languages.** Another group of respected researchers has also argued for compilers but in the context of high-level languages [16, 19]. They suggest that programmers should only need to write a high-level, declarative description of an algorithm; the compiler (and run-time system) will produce code appropriate to specific computational environments.

We believe that this argument is also flawed, due to the fact that automated refinement, while not necessarily sacrificing performance theoretically, often does cost performance in practice. Performance is sacrificed because refinements made by a human who understands the peculiarities of an algorithm are almost always capable of attaining greater performance than generalized transformations included in any actual compiler. This is the essence of the point discussed above in Section 2, and illustrated with the "list of elements" example. The refinements required to effectively map processes to processors and to balance load are areas in which completely automatic choice of optimal strategies seems particularly difficult.

**Language Extensions and Layering.** Most parallel programming to date has used lower-level languages extended with parallel processing constructs [1, 2, 6, 13, 18]. If properly designed, these extensions can maintain portability without sacrificing substantial performance. A number of packages offering such extensions exist; we have been involved in several efforts based on this approach. The principal drawback of the approach is the loss of notational elegance and freedom from detail associated with high level languages.

Advocates of languages such as C++ argue that layering of software can provide many of the benefits of a high-level language and in addition allow exploitation of the additional options offered by a lower-level language. For example, one can easily envision a set of routines that supports list processing, memory management, and synchronization facilities similar to those provided by Strand. This sort of layering can dramatically reduce complexity. However, experience suggests that the support and maintenance of lower layers introduces significant intellectual overhead. There is a substantial difference between an algorithm coded with C list-processing routines and an equivalent algorithm coded in Strand. Details introduced by the lower-level context must be remembered, and the lack of notational elegance (e g., the syntax of a list) produces a marked loss of clarity.

Similar arguments are advanced by proponents of high-level languages that support low-level features such as arrays and assignment (e.g., Common LISP). The low-level features permit efficient implementation of low-level algorithms, and hence seem to permit the benefits of the bilingual approach to be achieved in a single language. However, we believe that the introduction of low-level features tends to compromise the semantic elegance of the high-level language. For example, unrestricted use of updates in Strand would prevent the automatic transformation of programs to incorporate load balancing.

The use of a distinct high-level language permits a clean separation of concerns between high-level and low-level components.

**Coordination Languages.** Many of the arguments we have advanced in this paper have also been put forward by advocates of "coordination languages" [3]. Indeed, we are in basic agreement with the essential positions taken by advocates of coordination languages, and we have in the past discussed the use of Strand as a mechanism for coordinating computations written in a lower-level notations [9]. However, our current position views the boundary between the languages as determined essentially by performance considerations, rather than by a division between a sequential component and a coordination component.

# 8 Conclusions

Bilingual programming has a long, if not noble, history. It has been used since the earliest experiments with high level languages as a mechanism for mitigating the performance penalty associated with notations that offer standardized implementations of common abstract data types. Its value as a programming methodology is a direct consequence of the trade-offs between ease of expression and performance.

We have argued in this paper that bilingual programming should not be viewed as an ad-hoc solution to transient engineering problems but instead as a logical development of fundamental concepts in programming methodologies. The use of a high-level language permits programmers to adopt standardized implementation decisions when performance is not critical. This minimizes development costs and maximizes flexibility. At the same time, aceess to low-level languages permits the programmer to perform additional refinement steps when necessary. This permits efficient implementation of critical components.

The bilingual approach is applicable in many areas. However, we believe that it is particularly appropriate in parallel programming, where the need to specify not only sequential execution but also partitioning, scheduling, communication, and synchronization introduces additional complexity. High-level concurrent languages can reduce this complexity by providing standard implementations for common concurrent program structures and by permitting separate specification of program logic, partitioning, and scheduling. At the same time, the efficiency provided by low-level languages is particularly important on parallel computers.

We have accumulated considerable experience in bilingual programming using the high-level language Strand and the lower-level languages C and Fortran. We have developed major applications in several application areas using this technology. We have found that the bilingual approach encourages the development of parallel programs that perform well, are portable, and are easy to maintain. We advocate the adoption of bilingual programming as a technology for programming parallel processors.

# Acknowledgments

This research 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.

## References


[1] Babb, R., Parallel processing with large grain data flow techniques, *IEEE Computer*, 17, 55–61, 1984.

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

[3] Carriero, N. and Gelernter, D., Coordination languages and their significance. Technical report YALEU/DCS/RR-716, Yale University, 1989.

[4] Chandy, M. and Taylor, S., Program composition, *Proc. Supercomputing '89*, Reno, Nevada, 1989.

[5] 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.

[6] Dongarra, J. and Sorenson, D., Schedule: Tools for developing and analyzing parallel Fortran programs, *The Characteristics of Parallel Algorithms*, MIT Press, 1987.

[7] Fischer, J., Ellis, J., Ruttenberg, J., and Nicolau, A., Parallel processing: a smart compiler and a dumb machine, *Proc. SIGPLAN '84 Symp. on Compiler Construction*, ACM, 37–47, 1984.

[8] Foster, I., Automatic generation of self-scheduling programs, Preprint MCS-P124-0190, Argonne National Laboratory, 1990.

[9] Foster, I. and Overbeek, R., Experiences with bilingual parallel programming, *Proc. 5th Distributed Memory Comp. Conf.*, IEEE Press, 1990.

[10] Foster, I. and Stevens, R., Parallel programming with algorithmic motifs, *Proc. 1990 Intl Conf. on Parallel Processing*, Penn State University Press, 1990.

[11] Foster, I. and Taylor, S., *Strand: New Concepts in Parallel Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1990.

[12] Foster, I. and Winsborough, W., Unpublished information.

[13] Halstead, R.H., MultiLisp - A language for concurrent symbolic computation, *ACM Trans. Prog. Lang. and Syst.*, 7(4), 1985, 501-538.

[14] Padua, D.A., Kuck, D.J., and Lawrie, D.H., High-speed multiprocessors and compilation techniques, *IEEE Trans. on Computers*, C-29(9), 1980.

[15] Parnas, D., On the criteria to be used in decomposing systems into modules, *CACM*, 15, 1972, 330–336.

[16] Peyton Jones, S., *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

[17] Polychronopoulos, C., *Parallel Programming and Compilers*, Kluwer Academic, Boston, Mass., 1988.

[18] Seitz, C., The cosmic cube, *CACM* 28(1), 22-33, 1985.

[19] Warren, D.H.D., Or-parallel execution models of Prolog, *TAPSOFT'87, The 1987 Intl Joint Conf. on Theory and Practice of Software Development*, Springer-Verlag, 243-259, 1987.

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

# END

# DATE FILMED

02 / 05 / 91