Parallel Programming Languages:
Do They Meet the Needs of
Scientists and Engineers?

Technical Report CSE-89-03

Cherri M. Pancake
email: pancake@ducvax.auburn.edu

Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849 - 5347

July, 1989
Parallel Programming Languages: Do They Meet the Needs of Scientists and Engineers?

Technical Report CSE-89-03

Cherri M. Pancake
Department of Computer Science and Engineering
Auburn University
email: pancake@ducvax.auburn.edu

Donna Bergmark
Cornell National Supercomputer Facility
Cornell University
email: f71@cornellf.tn.cornell.edu

Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849 - 5347

July, 1989
Parallel Programming Languages: Do They Meet the Needs of Scientists and Engineers?

Technical Report CSE-89-03

Cherri M. Pancake
Department of Computer Science and Engineering
Auburn University
email: pancake@ducvax.auburn.edu

Donna Bergmark
Cornell National Supercomputer Facility
Cornell University
email: f7f@cornellf.tn.cornell.edu

Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849 - 5347

July, 1989
Parallel Programming Languages: 
Do They Meet the Needs of Scientists and Engineers?

Cherri M. Pancake  
Department of Computer Science and Engineering  
Auburn University  
Auburn, Alabama 36849

Donna Bergmark  
Cornell National Supercomputer Facility  
Cornell University  
Ithaca, New York 14853

Abstract

Although computer scientists use parallel programming languages to model concurrency, scientists and engineers approach parallelism with a different objective. The physical world they model is inherently parallel, but scientific programmers have become accustomed to using sequential techniques for its study. It is the ability to handle massive amounts of data that motivates their conversion to parallelism.

This report considers the current state of parallel programming languages from the perspective of scientists and engineers. It reviews the problem-solving strategies commonly employed by scientific programmers and compares them with the conceptual models for parallelism supported by high-level programming languages available to the general user community. The fact that existing languages were designed and implemented by members of the computer science profession is reflected in the prevalence of features representing that community's view of parallelism. Consequently, scientific programmers are unlikely to find language models that map cleanly to their problem-solving approaches.

Keywords: parallel programming, parallel programming languages, program development, programming languages, scientific programming
Introduction

For computer scientists, interest in parallel programming began with the need to describe nondeterministic sequencing. The earliest parallel programs evolved in the area of operating systems, where multiple program segments executed independently in real or simulated parallelism. Today, most efforts are directed at the exploitation of concurrency as a property of computational systems. From the modeling of networks to the development of parallel algorithms, the common denominator is the use of parallelism to describe inherently concurrent situations. Consequently, parallelism is integrated from the earliest stages of program design. Important issues in the selection of a programming language include the range of parallel features available, the accuracy with which they reflect the underlying machine architecture, and the degree of flexibility provided.

Scientists and engineers approach parallel programming with a different objective. Their interest in parallelism evolved from the desire to improve the performance of sequential algorithms applied to large-scale numerical computation. These users are concerned with the exploitation of distributed processing rather than concurrency per se. Although the physical world they model is inherently parallel, scientific programmers have become accustomed to using sequential techniques for its study. It is the ability to handle massive amounts of data within a reasonable period of time that motivates their conversion to parallelism. Rather than integrating parallelism into the design process, they incorporate it after the fact to speed up applications that have been tested and debugged in a sequential environment.

That scientific users view the future of parallel computing with optimism is demonstrated by their enthusiastic response to the increasing availability of parallel facilities. At the Cornell National Supercomputer Facility, for example, parallel processing capability tripled during the last three years, while parallel processing usage (in terms of CPU hours) increased 22 times. Nevertheless, it is estimated that only one in twenty user programs executed on the facility’s extensive network is parallel. Many more applications could take advantage of multiprocessor capabilities were it not for the difficulty of reformulating sequential code and the lack of program development tools applicable to a parallel environment. Vectorizing compilers offer a means of achieving a minimal degree of concurrency, but the true potential of parallelism remains unrecognized and/or underutilized by a large majority of the user community.

It is not clear how easily or effectively parallel techniques can be integrated into the scientific programming process. Critical concerns include the extent to which existing sequential programs can be converted to parallel form, how reusable the results are, and how well they will adapt to changes in the underlying architecture.

A portion of this research was conducted using the Cornell National Supercomputer Facility, a resource of the Center for Theory and Simulation in Science and Engineering (Cornell Theory Center), which receives major funding from the National Science Foundation and IBM Corporation, in addition to support from New York State and members of the Corporate Research Institute.
This article considers the current state of programming language support for parallelism from the perspective of scientists and engineers. Do existing languages provide the features needed for typical research applications? Are features accessible at appropriate levels? What kind of learning curve might be expected?

To understand the needs of the user community, we must first explore the nature of parallel programming. Most descriptions of parallel systems focus on architectural elements, such as the number of processors, access to memory, and communication topology. For our purposes, however, an abstract conceptualization is more appropriate. As the parallel user community grows, the average programmer will have little knowledge of — or interest in — the distinctions between multiprocessors and multicomputers, shared-memory versus message passing systems, and so on. Our discussion of parallelism will make use of logical concepts, therefore, without regard to the number or type of physical processors, links, or memory elements. Finally, attention will be concentrated on those programming languages that are currently available to significant portions of the general user community rather than experimental systems.

**Programming as a Problem-Solving Activity**

Scientists and engineers employ computation as a tool for solving problems related to the physical world. Like other tools, the computer imposes restrictions on the way the problem is formulated for solution and the types of solutions possible. How well parallel programming languages fill user needs cannot be measured without a clear understanding of the role played by language in determining the effectiveness of programming activities. The development of a computational solution involves a series of steps: delineation of the problem domain, selection of a problem-solving strategy, formulation of an algorithmic solution, implementation using a programming language, translation of the program into executable form, and preparation for execution. In effect, problem solution comprises four systems operating at different levels of abstraction (Figure 1). Each system defines its own collection of objects, a set of operations or manipulations applicable to those objects, and domains containing the values each object may legitimately assume.

At the highest level is the *conceptual solution*. Here, the problem is expressed in the abstract terms of human reasoning and human perception of the laws governing nature. Entities within the problem domain are portrayed in terms of logical characteristics, associations with other entities, and the attribution of meaningful values. The description is generally in the form of natural language text or diagrams.

Below this is the *algorithmic solution*, which specifies the steps required to solve the problem. Actions are formulated in terms of abstract operations carried out on representations of conceptual solution entities. The choice of notational form is usually on the basis of appropriateness to the logical model rather than any relationship to the computing environment where the solution will be carried out.
Although scientific models, like the physical world they represent, are inherently parallel, the formulation of algorithmic solutions typically involves a procedural approach. Steps are expressed as "find x such that y" rather than "accomplish actions x, y, z within time w".

The physical solution occupies the lowest stratum. It is at this level that the problem is actualized in the form of an executing process. The nature of computation requires that all problem components be expressed in the vocabulary of the machine, binary object code. Data objects thus occur as storage locations, operations as machine instructions, and values as bit patterns that can be recognized and manipulated by the processing unit. In parallel environments, the necessity of arranging processors in an interconnected topology imposes further restrictions: objects may require replication or distribution for storage in separate memories; instructions must be scheduled for execution on multiple processors; values may need to be mutually acceptable to a variety of processing units; and the activities of individual processors must be coordinated.

Between the algorithmic and physical levels is the implementation solution, which serves to bridge the gap between the representation of the problem as abstract manipulations and as the physical operations to compute those manipulations. Here, the problem is re-expressed in the terms supported by a programming language: data structures, primitive operations, and basic values. Since the program must be suitable for automatic translation to machine code, the programming language imposes a rigorous formalism; both syntax (structure) and semantics (meaning) are severely restricted as compared with the algorithmic system. At the same time, the language may be far removed from the functional capabilities of the physical system in order to provide expressiveness, generality, and portability.

It is important to note that multiple restructurings are required: the abstract model for problem solution is reformulated as an algorithmic solution, then transformed into program code, then translated into executable code. In addition to imposing development overhead, each restructuring is a source of potential error and distortion. The term "program development" is sometimes associated with just the manual transformation from algorithm to program. In this discussion, however, the automated transformation from programming language to machine instructions is also considered an integral part of the development process.

The way in which program transformations are effected and the difficulties they introduce vary according to the programming paradigm employed. Since traditional problem solving is described and carried out sequentially, we first consider a sequential paradigm. The high-order transformation is carried out by the programmer, who reformulates the logical model into a program written in the formal notation of a particular programming language. Since both versions are sequential, methodological revisions are required only when an abstract operation cannot conveniently or correctly be translated into code. Once the programmer is satisfied that the program accurately represents the conceptual model, the second
Figure 1. Problem solution systems
transformation can be performed automatically by systems programs (compiler, optimizer, linker, etc.). The current state of technology makes this translation straightforward. Errors that occur under a sequential paradigm therefore tend to be due to faulty solution logic or to misunderstandings of the mapping from algorithmic description to program, rather than to problems in the second transformation. Furthermore, sequential debugging activities, which usually involve clarification of logical processes, are reasonably well understood and are supported by a wide range of tools.

Methods for improving the efficiency of sequential program development reflect the multilayered organization of programming activities. As Klerer notes, "minimizing the linguistic difference between the specification [of the algorithm] and the computer language not only maximizes the economics of the programming process, it also increases the reliability of the software that is written."¹ The application of formal design methodologies represents an attempt to structure the conceptualization process so that conversion to programming language constructs will be more straightforward. This is the equivalent of moving the algorithmic solution closer to the implementation. Programming language designers often take the opposite approach; by introducing features with a higher level of abstraction, they effectively raise the level of the implementation solution by shifting transformation responsibilities from the programmer to the compiler.

Such fine-tuning of a programming system is possible precisely because of the clear delineation between the two levels of transformation. This property, referred to by database designers as logical independence, represents a commitment to maintaining a separation between machine-dependent and machine-independent factors. With few exceptions, today’s sequential programming languages shield the programmer from implementation details. At the same time, increasing efforts are devoted to support for the user’s conceptual models through provision of such features as abstract data types, reusable module definitions, and predefined collections of common data structures. The lessons of three decades of sequential program development are clear: programmer effectiveness is improved when language structures are moved away from physical issues and toward conceptual models.

The introduction of parallelism seriously complicates the program development process. The programmer is now concerned, not just with data objects and operations, but also with the interaction and relative timing of many operational entities: What activities can be carried out in parallel? What data must be shared among different activities? How are the activities to be coordinated when timing is unpredictable? As one source notes, "a conventional program consists of one executing process, of a single point in computational time-space, but a parallel program consists of many, and to the extent that we have to worry about the relationship among these points in time and space, the mood turns nasty."²

Although we look forward to the day when real-world problems may be mapped to parallel hardware seamlessly and automatically, the fact remains that
parallel paradigms are still in the embryonic stage. Little is known about effective
techniques for conceptualizing and formalizing parallel strategies. In spite of
criticisms of "von Neumann programming," computation is still viewed by a large
majority of the user community as a sequence of transformations on data. Parallel
processing is correspondingly visualized as the simultaneous execution of sequential
procedures. This "extended sequential" view of parallelism has been advocated by
some authors, but it is responsible for a number of misconceptions among scientific
users. First, it implies that if a sequential problem is partitioned into subproblems
for execution on parallel processors, results will be reliable and repeatable. That
this is not the case is the motivation for current efforts to develop quality debugging
tools for parallel systems. Misconceptions about the nature of parallelism also lead
to confusion when parallelization does not achieve a speedup proportional to the
number of processors. Finally, the failure to understand the extreme dependence of
performance on network topology leads to problems with reusability and portability.

Strategies for Parallel Problem-Solving

Programmers have long been aware that language features have significant
impact on how easily an algorithm can be transformed into workable code. Few programmers would elect to implement list-processing software in Fortran
or computationally intensive matrix algorithms in Lisp, and even the so-called
general-purpose languages are recognized as being suited to certain problem-solving
approaches. This is not to say that it is impossible to construct an accurate
implementation using an inappropriate language. The transformation process is
more tedious and error-prone, however, when the conceptual models supported by
the language are related only peripherally to the problem-solving model of the
programmer.

To be effective in bridging the "semantic gap" between algorithm and program,
a conceptual model must relate both to the language and to the initial logic of
problem solution. Therefore, a survey of typical problem-solving strategies is in
order before examining the models supported by parallel languages. The following
approaches were identified among scientists and engineers at the Cornell National
Supercomputer Facility (CNSF) and similar centers nationwide. The order in
which they are described reflects increasing complexity in terms of requirements for
parallelism. Although the list is not exhaustive, it is representative of the program
types implemented on typical large-scale parallel systems.

1. Domain decomposition. This strategy involves the replication of operations
or sequences of operations for application to extremely large quantities of data.
The data is divided or partitioned and the same procedure is carried out on each
subgroup. In sequential programs this is often implemented as a DO loop; in parallel
systems, the corresponding approach is referred to as Single Program, Multiple Data
(SPMD). Many numerical experiments employ this strategy, including atmospheric
studies of turbulence mixing, where the domain under study — space — is subdivided into independent units. A single, computationally intensive sequence of operations is then applied across the domain in autonomous iterations.

2. Monte Carlo simulation. Conceptually, this approach employs a random walk through a global value space, typically involving millions of "snapshots". Occasional synchronization is used to determine when the desired probability density has been satisfied, or to identify when an optimal solution has been found. Parallelism is incorporated by performing individual simulations concurrently. One example is the study of the three-dimensional structures of polypeptides and proteins, where up to ten million random configurations of particles are computed, and the energy of each configuration determined.

3. Convergence approach. Large amounts of data are again involved, but here computations are performed in multiple passes over the data in order to apply decreasing grain or grid sizes. Each pass thus involves the parallel execution of iterations at a particular granularity. Synchronization occurs at the end of each pass so that the results of analysis can be used to increase the accuracy of the subsequent pass. This strategy has been used to simulate world economics. National-level models, each consisting of several hundred nonlinear equations, are first solved in parallel to determine factors such as import and export prices. The results are then combined in a global simulation where where convergence or relaxation methods are used to obtain a stable model.

4. Systolic system. This strategy is applied to situations where the output from one set of calculations must serve as input to the next. Conceptually, each set may be carried out in parallel, with initialization and termination staggered to accommodate coordination requirements. One example is the study of diffusion through fluid mixtures; in this case a three-dimensional lattice is analyzed one plane at a time, with the results used to compute the subsequent plane. A related approach is used for studies of ion scattering, where an extensive sequence of differential equations must be applied successively to the trajectory of each incoming particle.

5. Time evolution approach. Here, computations are performed progressively for a series of time steps. This approach combines aspects of convergence and systolic systems. Synchronization activities and data exchange occur at the end of each time step, with the output of one set of computations used as input for the next. In this case, however, the computational scale required at each step precludes the simple pipelining of processors; the effect of pipelining must be achieved by periodically redistributing work as well as data. Examples include models of the earth's substructure surface using seismic data or the study of black hole formation, where the first hours of the universe are simulated.

6. Search space pruning. In this approach, an extremely large search space is traversed to find an optimal (or acceptable) solution path. In principle, all paths can be searched in parallel. Interaction at periodic intervals allows the abandoning
of fruitless paths or the premature termination of the search when a solution is found. Such an approach is thought to be a promising one for studies in protein folding. A particular conformation of atoms can be reconfigured into several others, but in a structure containing hundreds of atomic groups only some conformations are feasible according to laws of molecular dynamics.

7. Pattern convolution. Successive transformations are performed in an attempt to approximate experimental data; alternatively, new values are calculated until acceptable statistics can be achieved. Data distribution must be reconfigured dynamically, with corresponding requirements for synchronization and communication. An example of this approach is the study of automata behavior in order to understand brain evolution as a function of elementary activities in the nervous system. The action of a neuron depends on its inputs, or stimuli, from nearby cells; this action in turn becomes a stimulus for neighboring cells. Arbitrary cell networks are specified and stimuli introduced as input data.

To compare the strategies, it is useful to envision parallelism as a three-dimensional space (see Figure 2). The dimensions reflect three salient features of parallel processing situations: work distribution, data distribution, and level of interaction. Each axis represents the degree of interrelationship among concurrently executing program units, with the origin indicating complete independence.

Position along the work distribution axis describes how much overlap exists among the activities performed by individual processors. At the origin each

![Figure 2. Parallelism as a three-dimensional space](image)
processor executes a disjoint sequence of identical operations. This corresponds to the Single Program, Multiple Data (SPMD) approach and is typified by a Monte Carlo simulation. A Multiple Instruction, Multiple Data (MIMD) program represents the opposite extreme. In this case, each processor executes a functionally distinct set of operations which must be coordinated in order to solve the problem; a discrete event simulation is an example. Programming languages typically support this aspect of parallelism in their provision for process specification (e.g., task definition versus fork/join versus array operations), where constructs focus on when activities are to be executed serially and when in parallel.

The data distribution axis indicates the degree to which data is shared among processors. Independence occurs when each processor has access to a disjoint set of data, as in a program to calculate vector minima or polynomial coefficients. Total interdependence reflects a sharing of all data by all processors, as in programs that attempt to derive models approximating experimental data. Language support includes provision for shared access to data via locks, semaphores, critical sections, or data replication mechanisms.

The extent to which processors must be interconnected is reflected by the interaction axis. In some cases, all processors may execute in complete independence. This situation is typified by programs whose asynchronous units exercise a sequence of solution paths and then self-terminate. Complete interdependence presupposes direct or indirect links among all processors so that the fullest level of cooperation can be achieved, as in the case of a program to model dynamically varying topologies. Programming languages provide explicit control over interaction through facilities for (1) communication, or the exchange of data information, and (2) synchronization, the exchange of control information. Typical mechanisms include message-passing, events, barriers, and rendezvous.

<table>
<thead>
<tr>
<th>Processor Interdependence</th>
<th>Work</th>
<th>Data</th>
<th>Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent data systems</td>
<td>low</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>Monte Carlo simulation</td>
<td>low</td>
<td>low</td>
<td>low-med</td>
</tr>
<tr>
<td>Convergence approach</td>
<td>low</td>
<td>low</td>
<td>low-med</td>
</tr>
<tr>
<td>Systolic system</td>
<td>low-med</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>Time evolution approach</td>
<td>low-med</td>
<td>low</td>
<td>low-med</td>
</tr>
<tr>
<td>Search space pruning</td>
<td>low</td>
<td>med</td>
<td>low-med</td>
</tr>
<tr>
<td>Pattern convolution</td>
<td>low-med</td>
<td>med</td>
<td>low-med</td>
</tr>
</tbody>
</table>
Table 1 uses this concept of parallel space in summarizing the processing characteristics of each problem-solving approach. Well-defined patterns emerge from the comparison. First, unlike parallel problems in computer science, which generally involve a number of functionally distinct processes, scientific applications are likely to execute the same function repeatedly on a variety of data. Consequently, the strategies cluster along the low end of the work distribution axis.

Second, from atmospheric dynamics to the structural properties of cardboard, scientific and engineering problems typically involve enormous amounts of data. Nevertheless, because the physical world demonstrates a high degree of locality, computational solutions to real-world problems do as well. Data granularity (the size of data manipulated by a single processor) varies substantially among scientific programs, but whatever the granularity, items to be processed together tend to be near each other both logically and physically. As a result, the representative strategies also cluster along the low end of the data distribution axis.

Finally, most scientific strategies demonstrate significantly less need for interaction than might be expected. This is to some extent attributable to the granularity of problem-solving activities. A recent sample of parallel scientific applications at the CNSF, for example, indicated average synchronization intervals of 350 CPU seconds on an IBM 3090-600E. Furthermore, since scientific programming relies heavily on the use of arrays and other regular structures, communication can often be limited to the exchange of results with neighboring processors. In areas as diverse as high-energy physics, theoretical chemistry, seismology, and electronics, the remarkable speedup achieved through parallelization has been attributed primarily to the limited nature of processor interaction.\(^5\) This is reflected as a clustering along the low end of the interaction axis.

Figure 3 depicts the composite domain of scientific problem-solving approaches in parallel space. What sort of conceptual models might be expected to correlate well with these programming needs? There is a clear requirement for models that facilitate the description, management, and partitioning of large quantities of data. Alternatives for specifying a range of processing entities and coordinating their interaction, on the other hand, are less important.

**Language Support for Conceptual Models**

Existing programming languages support parallelism in a variety of ways (see the appendix for a description). The specific facilities provided reflect the language designer’s concept of parallel processing and how this might best be integrated into high-level programs. A survey of currently available languages reveals that in most cases direct support is limited to a single conceptual model, with subsidiary features added to achieve some degree of flexibility. Furthermore, although syntax and naming conventions differ significantly from one language to another, the range of conceptual models supported is quite limited. In keeping with our spatial
representation scheme, the models can be divided into three general categories according to the structural focus employed to incorporate parallelism: task-oriented, data-oriented, or interaction-oriented. As shall be seen, the classification is analogous to the use of functional decomposition, data decomposition, or entity decomposition as the basis for program development.

The earliest models for parallelism were task-oriented, reflecting the viewpoint of operating systems designers who saw parallelism as the concurrent execution of distinct programs. In this category, parallelization is achieved by partitioning the tasks to be performed into a collection of sequential processes that will be carried out concurrently in relative autonomy. Coordination is achieved by a variety of simple mechanisms for pairwise interprocess communication and synchronization, but the primary emphasis is on facilities for defining and creating independent processes. One process cannot be said to strictly dominate another, and there is no real mechanism for global control of activities. The cooperating processes, pipeline, and monitor/task models are common examples from this category. Many commercially available parallel programming languages limit their direct support to task-oriented models (e.g., Ada, occam, PL/I, Algol 68, Smalltalk, Concurrent Pascal, Mesa, Modula, Concurrent Euclid).

The common theme of the task-oriented models is that a single task is assigned to each logical processor. The basic units of program development, processes, thus correspond to functional entities in the problem domain. Since concurrency is introduced by simultaneously executing two or more processes, parallelism occurs as multiple threads of control. The order of execution within a process is deterministic,
but the unit may be created dynamically, undergo dormant periods while awaiting the completion of another process, and be terminated on an individual basis. Task-oriented models therefore assume that the global execution of multiple processes is nondeterministic.

Data-oriented models provide a contrasting approach, in which parallelism occurs as the simultaneous performance of a common operation on multiple data elements. Whenever the size of the data warrants parallelization, activities are replicated for application across data subsets. In effect, program execution is viewed as a single thread of control which temporarily diverges into parallel action sequences. The methods for distributing data and synchronizing reconvergence become more important than the definition of activities per se. The most common examples are the concurrent loop and master/slave, which reflect the data parallelism inherent in vectors and are supported by the parallel Fortran, Parallel Pascal, and some parallel C libraries. The so-called domain decomposition model, which corresponds to other topological distributions of data, is typically limited to parallel versions of functional and logic languages (e.g., Parlog, Concurrent Prolog, *Lisp, and MultiLisp).

In data-oriented models, then, data rather than tasks are partitioned and distributed across logical processors. This corresponds to the use of data decomposition as the methodology for program development. Parallel control threads exist, but they are ephemeral, varying in number according to the requirements of data subsets. The overhead for thread creation is typically less than for full-scale task management. Some data-oriented models are analogous in many ways to task-oriented versions, but the source of nondeterminism is reversed: processing of blocks of data at the local level is nondeterministic, while major program operations proceed deterministically.

Among interaction-oriented models the distinction between data and manipulation becomes blurred. Here, parallel entities represent the components of a modularly decomposed system. Unlike the results of functional decomposition or data decomposition, entities do not uniformly represent subsets of activities or data. Instead, decomposition is based on the notion of orderly interaction via a well-defined communication system. What is important is the establishing of a message transmission protocol among independent self-contained entities; the question of whether an entity contains data, operations, or both is immaterial. The object-oriented, transaction, and actor models are included in this category. Although a number of parallel languages supporting such models have been proposed (e.g., ABCL, Concurrent Smalltalk, Orient/84, PLITS, Act3), they are still in the experimental phase.

The rationale behind interaction-oriented models is that logical processors should be capable of representing both physical and conceptual objects within the problem domain. No real distinction is drawn between data entities and functional entities, so requirements for data and control memory may vary significantly
from one processor to another. Processors are black boxes which emit and receive messages. An interesting consequence of this approach is that the issue of determinism becomes largely irrelevant.

**Alternatives for the Scientific Programmer**

It is clearly possible to implement a variety of conceptual models using any parallel programming language. A clever programmer can build a monitor system in parallel Fortran or a concurrent loop model in Ada, even though the languages provide no direct facilities for their implementation. Program development becomes significantly easier and more reliable, however, when the language supports the desired model directly. If an appropriate high-level structure is available, programmers can take full advantage of the compiler's semantic checking as a safeguard against many forms of runtime errors. In addition, the use of clearly defined language structures makes it possible to achieve an acceptable degree of independence from the underlying architecture. This allows the porting of programs from one system to another, as well as accommodating system upgrades or other modifications. Finally, when language syntax corresponds closely to the problem domain, program structure can more visibly reflect logical concepts. The resulting comprehensibility facilitates debugging activities and improves the likelihood of reusability.

The question to be posed is how well the conceptual models established by language designers match those employed by physical scientists and engineers. From this viewpoint, it is particularly unfortunate that existing languages are so restrictive in terms of the number and range of models supported. No single conceptual model is likely to provide enough generality for all common problem-solving strategies. What's more, experience supports the notion that program development benefits most when different models are applied to different types of problems.\(^7\) If language support for parallelism is to be effective from the user perspective, it is critical that a wider choice of models and problem decomposition alternatives be made available.

Figure 4 relates current language support to the three-dimensional space of parallel problem domains. Task-oriented structures are distributed across a variety of general-purpose programming languages as well as more recent parallel languages; related routines are included in many system libraries. Collectively, the conceptual models are suited to describing a wide range of activities. This is reflected as a general dispersal along the work distribution axis.

The situation for data-oriented models is less encouraging, even though these are more likely to be of use to the scientific user community. Integrated language support is generally limited to array operations. Language extensions and libraries may incorporate data-oriented features, but their use is clumsy and restricted to a limited range of granularity. Little or no provision is made for the dynamic
Figure 4. Parallel domain of language support

redistribution of data or for abstract data types suited to the management of massive amounts of data. In terms of the data distribution axis, then, language-supported models cluster along the low end.

Available languages do not provide direct support for managing the interaction of parallel program components as abstract entities. Instead, communication and synchronization mechanisms have been developed as necessary offshoots of task- and data-oriented approaches. True interaction-oriented models are not yet available in production-level systems. Language support thus clusters along the low end of the interaction axis in the parallel problem domain. (It should be noted that interaction-oriented approaches to parallelism — like their sequential counterparts, the object-oriented programming languages — require new and innovative techniques for program development. It is difficult to conceive of the scientific community embracing such a radical departure from traditional incremental programming methods.)

A comparison of Figures 3 and 4 reveals that scientific and engineering programmers are unlikely to find language models that map cleanly to their problem-solving approaches. This is disturbing in view of the fact that these users are precisely those who should benefit most from the error detection, machine-independence, and comprehensibility provided by appropriate models. Of even more concern is the fact that the program entities crucial to scientific programming are supported at an embarrassingly low level of abstraction. Data sharing and distribution must be specified in terms of primitives such as locks and messages. Consequently, programmers are forced to juggle inadequately described
and potentially dangerous operations. The current state of parallel programming support effectively represents a leap backward in time; as in the early days of Fortran, program features resemble symbolic assembly code more than high-level structures.

In addition, the primitives used to describe parallel features are very closely tied to the underlying system. It has been clearly demonstrated that management of the architectural configuration determines to a great extent the efficiency, effectiveness, and reliability of parallel implementations.\textsuperscript{8,9} Unfortunately, it is equally clear that full responsibility lies with the programmer, who must now be concerned with the optimal scheduling and binding of logical to physical processors, network topology specifications, and the distribution of data to memory locations.\textsuperscript{10,11} Unlike the sequential programming environment — where scientists and engineers learned to develop applications — parallel systems lack the buffering effect of logical independence. As a recent author points out, “when multiprocessors do not conform to the concept of a single logical model, but rather must be viewed as a dynamic pool of processing, storage, and connection resources, the control in software over communication and synchronization becomes a truly formidable task.”\textsuperscript{12}

All in all, the present level of language support for parallel programming requires that the user expend more effort in managing the problem-solving resource than actually solving the problem. This may be a positive factor in some cases. As Paul Hudak points out, “we as programmers usually have more to say about a problem than just the answer. We typically have a specific data representation in mind and we might know a better way to run the program on a particular architecture.”\textsuperscript{13} The key here is that while computing professionals should be capable of applying configuration-specific expertise, it is counterproductive to expect the same of the general user community. Support for scientific applications is inadequate as long as parallelism must be expressed in terms of the underlying architecture and operating system. Scientists and engineers turn to parallel processing in an attempt to understand the physical world, typically through modeling and simulation. They need language features that improve the comprehensibility and accuracy of parallel programs. All this presupposes some degree of logical independence from the architectural configurations on which programs will execute. Parallelism should be incorporated at a reasonable level of abstraction rather than simply providing a notationally convenient way of specifying what are, in fact, low-level operations.

If these arguments sound suspiciously like the justifications for high-level programming languages and structured programing methodologies we heard two decades ago, that is because parallel programming today faces similar problems. In reflecting a low-level view of concurrent execution, parallel programming languages do the scientific and engineering communities a disservice. This approach not only increases the frustration and expense of program development, but also raises questions about the reliability of program results. Furthermore, it is shortsighted to tie program development so closely to specific machine configurations.
and control mechanisms. By implementing parallelism in machine- and translator-specific fashion, we limit the program's transportability and reusability as well as its applicability to the improved systems that surely lie ahead.

In summary, the fact that existing languages were designed and implemented by members of the computer science profession is reflected in the prevalence of features representing that community's view of parallelism. Most high-level structures are devoted to the textual delineation of functional entities. Mechanisms for synchronizing their activities are cruder, making use of interrupts (real or simulated) and inter-process procedure calls. Communication is similarly limited to common storage and inter-process parameter passing. Conspicuously absent are structures designed to support efficient dynamic distribution of large quantities of data. Although programming language designers have spent more than two decades developing structured alternatives for supporting concurrency, the choices available to the scientific community are still primitive and error-prone.

New Directions

The plight of the scientific programmer has not gone unnoticed. Several new systems for parallel programming have recently emerged from the experimental stage and will soon be in production. The Linda-based systems developed at Yale² appear to have gained acceptance by computer manufacturers. Cogent, for example, recently announced development of C++ and Fortran versions to be compatible with (and portable to) such competitive products as Sequent's Balance and Symmetry, Encore's Multimax, and Intel's iPSC computers. This is a major step forward, since Linda's goal is to provide flexibility and expressiveness for a variety of programming paradigms through machine-independent constructs. Although Linda's approach is interaction-oriented, it appears to be scaled at a level that scientific programmers will find workable: "We believe that tightly-bound collections of synchronous or quasi-synchronous activities tend to force programmers to think in simultaneities. Great simplification of the potentially formidable task of parallel programming is possible ... if concurrent processes are so loosely bound that each can be developed independently..."¹⁴ Even more importantly, many forms of data distribution are inherent in Linda and thus eliminate the need to assimilate new programming techniques in order to deal with certain data-oriented strategies.

In a totally different sphere, supercomputer manufacturers and researchers have joined forces in the Parallel Computing Forum to define a parallel version of Fortran which will be implemented on all major systems. The result, a proposed standard for PCF-Fortran, was released in August, 1988, for public review and comment. Since the standard goes well beyond the normal conception of language "extension" to introduce a number of sophisticated constructs for managing concurrency, it remains to be seen how well scientific programmers will respond. It is also difficult to predict how much time will elapse before the first PCF-Fortran compilers are
available.

An important factor in both developments is that a concerted effort is being made to build on language constructs already familiar to scientific programmers so that the transition will not require total re-orientation. The challenge facing the parallel programmer is learning to use parallelism effectively in problem-solving efforts. For programming language designers, the challenge is to provide high-level features which reflect the conceptual models of the scientific user community. This will not be easy. Recent studies indicate that abstraction mechanisms may themselves compromise the correct functioning of concurrent programs. Nevertheless, language support for parallelism will impede rather than facilitate program development unless it can move beyond its present limitations to provide convenient structures for describing the applications of scientists and engineers.
Appendix:

Programming Language Support for Parallelism

Programming languages support multiprocessing by incorporating parallel features as integral parts of the language structure; by adding parallel extensions to a basically sequential language; or by providing interfaces to parallel routines stored in system libraries. It is also possible to incorporate parallelism into a purely sequential language by means of a parallelizing compiler. A further distinction may be made according to the granularity, or level at which parallelism is applied: in macrotasking, entire program units are executed concurrently; microtasking refers to the concurrent execution of loop iterations or other blocks of statements; and vectorization provides concurrent execution of a single operation across elements of an array.

(1) Parallel Programming Languages. Integrating parallelism directly into the design of a programming language offers the best chance for clear and unified support of conceptual models. The development of good debugging tools also depends on the availability of well defined high-level language facilities. Unfortunately, experience has shown that it is extremely difficult to design features that are both generally applicable and conducive to clean object code (compare the variety of approaches presented in recent surveys16;17). Furthermore, application costs are not limited to the acquisition of a suitable compiler. Typically, the programmer must learn a new philosophy of program development as well as a new language structure, and any existing programs must be reformulated as well as recompiled. Occam, Ada, Concurrent Pascal, and Modula2 are examples of languages where a significant number of structures are devoted to supporting parallelism; others such as PL/I, Mesa, and Algol68 integrate parallel features in the original language definition, but on a much simpler scale.

(2) Parallel Extensions to Sequential Languages. The addition of parallel features to a familiar language by means of macros or other pre-processable extensions clearly facilitates the parallelization of existing programs. Programmers need only assimilate a few structures and learn to identify the situations for which they are appropriate. On the other hand, it is extremely difficult to integrate parallel constructs cleanly and logically (see, for example, comparative examples of parallel Fortrans18). Other problems include a strong machine- and dialect-dependence, resulting in decreased portability, plus the fact that extensions may interfere with existing compiler optimizations. Parallel Pascal, Concurrent C, MultiLisp, and most of the parallel Fortrans fit into this category.

(3) Parallel Runtime Libraries. Library routines can provide flexible and language-independent support for parallelism. Carefully designed libraries can obviate the rewriting and recompilation of programs when the hardware system is modified. However, the use of runtime routines is typically awkward and error-prone (Ranka et al. provide a good example of this8). Bulky parameter lists may
be needed to compensate for the fact that library elements execute in isolation from
general program context. Debugging becomes more complex since there is no clearly
defined relationship with program structure or semantics. Another disadvantage is
that although the library approach appears to provide an easy means of integrating
portability — just recode the library routines and leave the invoking programs alone
— this is not always true. Parallel libraries often have such a close relationship
with the underlying architecture that porting programs to other machines results
in inefficient or unreliable performance. Several parallel versions of Fortran and C
are examples of high-level interfaces to libraries.

(4) Parallelizing Compilers. Parallel language features such as those already
described are explicit; that is, the programmer specifies when concurrent execution
is appropriate. Parallelization is said to be implicit when the compiler is capable of
recognizing potentially concurrent portions of a sequential program and generating
parallel code (this is differentiated from vectorization, which generates parallel code
for predefined array operations only). Implicit parallelization requires extensive
analysis of the dependencies among data items and cannot guarantee an optimal
solution. Although parallelizing compilers are widely cited as the most promising
direction for future developments, the versions currently in production possess only
limited capabilities.19;20
References


