Models and Languages for Parallel Computation

DAVID B. SKILLICORN

Queen's University

DOMENICO TALIA

Universitá della Calabria

We survey parallel programming models and languages using six criteria to assess their suitability for realistic portable parallel programming. We argue that an ideal model should be easy to program, should have a software development methodology, should be architecture-independent, should be easy to understand, should guarantee performance, and should provide accurate information about the cost of programs. These criteria reflect our belief that developments in parallelism must be driven by a parallel software industry based on portability and efficiency. We consider programming models in six categories, depending on the level of abstraction they provide. Those that are very abstract conceal even the presence of parallelism at the software level. Such models make software easy to build and port, but efficient and predictable performance is usually hard to achieve. At the other end of the spectrum, low-level models make all of the messy issues of parallel programming explicit (how many threads, how to place them, how to express communication, and how to schedule communication), so that software is hard to build and not very portable, but is usually efficient. Most recent models are near the center of this spectrum, exploring the best tradeoffs between expressiveness and performance. A few models have achieved both abstractness and efficiency. Both kinds of models raise the possibility of parallelism as part of the mainstream of computing.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: D.1 [**Programming Techniques**]; D.3.2 [**Programming Languages**]: Language Classifications

General Terms: Languages, Performance, Theory

Additional Key Words and Phrases: General-purpose parallel computation, logic programming languages, object-oriented languages, parallel programming languages, parallel programming models, software development methods, taxonomy

INTRODUCTION

Parallel computing is about 20 years old, with roots that can be traced back to the CDC6600 and IBM360/91. In the years since then, parallel computing has permitted complex problems to be solved and high-performance applications to be implemented both in traditional areas, such as science and engineering, and in new application areas,

Authors' addresses: D. B. Skillicorn, Computing and Information Science, Queen's University, Kingston K7L 3N6, Canada; email: ⟨skill@qucis.queensu.ca⟩; D. Talia, ISI-CNR, c/o DEIS, Universitá della Calabria, 87036 Rende (CS), Italy; email: ⟨talia@si.deis.unical.it⟩.

such as artificial intelligence and finance. Despite some successes and a promising beginning, parallel computing has not become a major methodology in computer science, and parallel computers represent only a small percentage of the computers sold over the years. Parallel computing creates a radical shift in perspective, so it is perhaps not surprising that it has yet to become a central part of practical applications of computing. Given that opinion over the past 20 years has oscillated between wild optimism ("whatever the question, parallelism is the answer") and extreme pessimism ("parallelism is a declining niche market"), it is perhaps a good time to examine the state of parallel computing. We have chosen to do this by an examination of parallel programming models. Doing so addresses both software and development issues, and hardware and performance issues.

We begin by discussing reasons why parallel computing is a good idea, and suggest why it has failed to become as important and central as it might have been. In Section 2, we review some basic aspects of parallel computers and software. In Section 3, we discuss the concept of a programming model and list some properties that we believe models of parallel programming ought to have if they are to be useful for software development and also for effective implementation. In Section 4, we assess a wide spectrum of existing parallel programming models, classifying them by how well they meet the requirements we have suggested.

Here are some reasons why parallelism has been a topic of interest.

—The real world is inherently parallel, so it is natural and straightforward to express computations about the real world in a parallel way, or at least in a way that does not preclude parallelism. Writing a sequential program often involves imposing an order on actions that are independent and could be executed concurrently. The particular order in which they are placed is arbitrary and becomes a barrier to understanding the program, since the places where the order is significant are obscured by those where it is not. Arbitrary sequencing also makes compiling more difficult, since it is much harder for the compiler to infer which code movements are safe. The nature of the real world also often suggests the right level of abstraction at which to design a computation.

—Parallelism makes available more computational performance than is available in any single processor, although getting this performance from parallel computers is not straightforward. There will always be applications that are computationally bounded in science (the grand challenge problems), and in engineering (weather forecasting). There are also new application areas where large amounts of computation can be put to profitable use, such as data mining (extracting consumer spending patterns from credit card data) and optimization (just-in-time retail delivery).

—There are limits to sequential computing performance that arise from fundamental physical limits such as the speed of light. It is always hard to tell how close to such limits we are. At present, the cost of developing faster silicon and gallium arsenide processors is growing much faster than their performance and, for the first time, performance increases are being obtained by internal use of parallelism (superscalar processors), although at a small scale. So it is tempting to predict that performance limits for single processors are near. However, optical processors could provide another large jump in computational performance within a few decades, and applications of quantum effects to processors may provide another large jump over a longer time period.

—Even if single-processor speed improvements continue on their recent historical trend, parallel computation

is still likely to be more cost-effective for many applications than using leading-edge uniprocessors. This is largely because of the costs of designing and fabricating each new generation of uniprocessors, which are unlikely to drop much until the newer technologies, such as optical computation, mature. Because the release of each new faster uniprocessor drives down the price of previous generations, putting together an ensemble of older processors provides cost-effective computation if the cost of the hardware required to connect them is kept within reasonable limits. Since each new generation of processors provides a decimal order of magnitude increase in performance, modestly sized ensembles of older processors are competitive in terms of performance. The economics of processor design and production favor replication over clever design. This effect is, in part, responsible for the popularity of networks of workstations as low-cost supercomputers.

Given these reasons for using parallelism, we might expect it to have moved rapidly into the mainstream of computing. This is clearly not the case. Indeed, in some parts of the world parallel computing is regarded as marginal. We turn now to examining some of the problems and difficulties of using parallelism that explain why its advantages have not (yet) led to its widespread use.

—Conscious human thinking appears to us to be sequential, so that there is something appealing about software that can be considered in a sequential way—a program is rather like the plot of a novel, and we have become used to designing, understanding, and debugging it in this way. This property in ourselves makes parallelism seem difficult, although of course much human cognition does take place in a parallel way.

—The theory required for parallel computation is immature and was developed after the technology. Thus the theory did not suggest directions, or even limits, for technology. As a result, we do not yet know much about abstract representations of parallel computations, logics for reasoning about them, or even parallel algorithms that are effective on real architectures.

—It is taking a long time to understand the balance necessary between the performance of different parts of a parallel computer and how this balance affects performance. Careful control of the relationship between processor speed and communication interconnect performance is necessary for good performance, and this must also be balanced with memory-hierarchy performance. Historically, parallel computers have failed to deliver more than a small fraction of their apparently achievable performance, and it has taken several generations of using a particular architecture to learn the lessons on balance.

—Parallel computer manufacturers have designated high-performance scientific and numerical computing as their market, rather than the much larger high-effectiveness commercial market. The high-performance market has always been small, and has tended to be oriented towards military applications. Recent world events have seen this market dwindle, with predictable consequences for the profitability of parallel computer manufacturers. The small market for parallel computing has meant that parallel computers are expensive, because so few of them are sold, and has increased the risk for both manufacturers and users, further dampening enthusiasm for parallelism.

—The execution time of a sequential program changes by no more than a constant factor when it is moved from one uniprocessor to another. This is not true for a parallel program, whose execution time can change by an order of magnitude when it is moved

across architecture families. The fundamental nonlocal nature of a parallel program requires it to interact with a communication structure, and the cost of each communication depends heavily on how both program and interconnect are arranged and what technology is used to implement the interconnect. Portability is therefore a much more serious issue in parallel programming than in sequential. Transferring a software system from one parallel architecture to another may require an amount of work up to and including rebuilding the software completely. For fundamental reasons, there is unlikely ever to be one best architecture family, independent of technological changes. Therefore parallel software users must expect continual changes in their computing platforms, which at the moment implies continual redesign and rebuilding of software. The lack of a long-term growth path for parallel software systems is perhaps the major reason for the failure of parallel computation to become mainstream.

Approaches to parallelism have been driven either from the bottom, by the technological possibilities, or from the top, by theoretical elegance. We argue that the most progress so far, and the best hope for the future, lies in driving developments from the middle, attacking the problem at the level of the model that acts as an interface between software and hardware issues.

In the next section, we review basic concepts of parallel computing. In Section 3, we define the concept of a model and construct a checklist of properties that a model should have to provide an appropriate interface between software and architectures. In Section 4, we assess a large number of existing models using these properties, beginning with those that are most abstract and working down to those that are very concrete. We show that several models raise the possibility of both long-term portability and performance. This suggests a way to provide the missing growth path for parallel software development and hence a mainstream parallel computing industry.

2. BASIC CONCEPTS OF PARALLELISM

In this section we briefly review some of the essential concepts of parallel computers and parallel software. We begin by considering the components of parallel computers.

Parallel computers consist of three building blocks: processors, memory modules, and an interconnection network. There has been steady development of the sophistication of each of these building blocks, but it is their arrangement that most differentiates one parallel computer from another. The processors used in parallel computers are increasingly exactly the same as processors used in single-processor systems. Present technology, however, makes it possible to fit more onto a chip than just a single processor, so there is considerable investigation into what components give the greatest added value if included on-chip with a processor. Some of these, such as communication interfaces, are relevant to parallel computing.

The interconnection network connects the processors to each other and sometimes to memory modules as well. The major distinction between variants of the multiple-instruction multiple-data (MIMD) architectures is whether each processor has its own local memory, and accesses values in other memories using the network; or whether the interconnection network connects all processors to memory. These alternatives are called *distributed-memory MIMD* and *shared-memory MIMD,* respectively and are illustrated in Figure 1.

Distributed-memory MIMD architectures can be further differentiated by the total capacity of their interconnection networks, that is, the total volume of data that can be in transit in the network at any one time. For example, an architecture whose processor-mem-

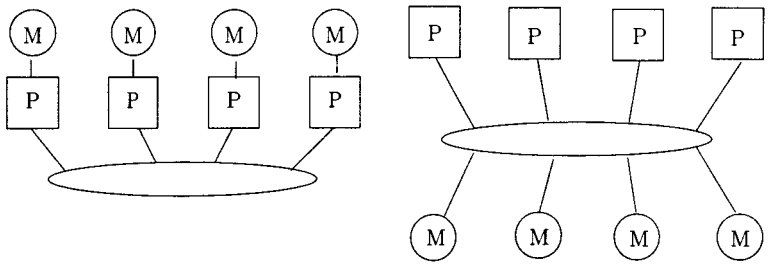

Fig. 1. Distributed-memory MIMD and shared-memory MIMD architectures.

ory pairs (sometimes called *processing elements*) are connected by a mesh requires the same number of connections to the network for each processor, no matter how large the parallel computer of which it is a member. The total capacity of the network grows linearly with the number of processors in the computer. On the other hand, an architecture whose interconnection network is a hypercube requires the number of connections per processor to be a logarithmic function of the total size of the computer. The total network capacity grows faster than linearly in the number of processors.

Another important style of parallel computer is the *single-instruction multiple-data* (SIMD) class. Here a single processor executes a single instruction stream, but broadcasts each instruction to be executed to a number of data processors. These data processors interpret the instruction's addresses either as local addresses in their own local memories or as global addresses, perhaps modified by adding a local base address to them.

We now turn to the terminology of parallel software. The code executing in a single processor of a parallel computer is in an environment that is quite similar to that of a processor running in a multiprogrammed single-processor system. Thus we speak of processes or tasks to describe code executing inside an operating-system-protected region of memory. Because many of the actions of a parallel program involve communicating with remote processors or memory locations, which takes time, most processors execute more than one process at a time. Thus all of the standard techniques of multiprogramming apply: processes become descheduled when they do something involving a remote communication, and are made ready for execution when a suitable response is received. A useful distinction is between the *virtual parallelism* of a program, the number of logically independent processes it contains, and the *physical parallelism,* the number of processes that can be active simultaneously (which is, of course, equal to the number of processors in the executing parallel computer).

Because of the number of communication actions that occur in a typical parallel program, processes are interrupted more often than in a sequential environment. Process manipulation is expensive in a multiprogrammed environment so, increasingly, parallel computers use threads rather than processes. Threads do not have their own operating-system-protected memory region. As a result there is much less context to save when a context switch occurs. Using threads is made safe by making the compiler responsible for enforcing their interaction, which is possible because all of the threads come from a single parallel program.

Processes communicate in a number of different ways, constrained, of course, by what is possible in the executing architecture. The three main ways are as follows.

—*Message passing.* The sending process packages the message with a header indicating to which processor and process the data are to be routed, and inserts them into the interconnection network. Once the message has been passed to the network, the sending process can continue. This kind of send is called a *nonblocking send.* The receiving process must be aware that it is expecting data. It indicates its readiness to receive a message by executing a receive operation. If the expected data have not yet arrived, the receiving process suspends (blocks) until they do.

—*Transfers through shared memory.* In shared-memory architectures, processes communicate by having the sending process place values in designated locations, from which the receiving process can read them. The actual process of communication is thus straightforward. What is difficult is detecting when it is safe either to put a value into the location or to remove it. Standard operating-system techniques such as semaphores or locks may be used for this purpose. However, this is expensive and complicates programming. Some architectures provide full/empty bits associated with each word of shared memory that provide a lightweight and high-performance way of synchronizing senders and receivers.

—*Direct remote-memory access.* Early distributed-memory architectures required the processor to be interrupted every time a request was received from the network. This is poor use of the processor and so, increasingly, distributed-memory architectures use a pair of processors in each processing element. One, the application processor, does the program's computation; the other, the messaging processor, handles traffic to and from the network. Taken to the limit, this makes it possible to treat message passing as direct remote memory access to the memories of other processors. This is a hybrid form of communication in that it applies to distributed-memory architectures but has many of the properties of shared memory.

These communication mechanisms need not correspond directly to what the architecture provides. It is straightforward to simulate message passing using shared memory, and possible to simulate shared memory using message passing (an approach known as *virtual shared memory*).

The interconnection network of a parallel computer is the mechanism that allows processors to communicate with one another and with memory modules. The *topology* of this network is the complete arrangement of individual links between processing elements. It is naturally represented as a graph. The diameter of the interconnection network's topology is the maximum number of links that must be traversed between any pair of processing elements. It forms a lower bound for the worst-case *latency* of the network, that is, the longest time required for any pair of processing elements to communicate. In general, the observed latency is greater than that implied by the topology because of congestion in the network itself.

The performance of a parallel program is usually expressed in terms of its execution time. This depends on the speed of the individual processors, but also on the arrangement of communication and the ability of the interconnection network to deliver it. When we speak of the cost of a program, we usually mean its execution-time cost. However, in the context of software development, cost may also include the resources necessary to develop and maintain a program.

3. MODELS AND THEIR PROPERTIES

A *model of parallel computation* is an interface separating high-level properties from low-level ones. More concretely, a model is an abstract machine providing certain operations to the pro-

gramming level above and requiring implementations for each of these operations on all of the architectures below. It is designed to separate software-development concerns from effective parallel-execution concerns and provides both abstraction and stability. Abstraction arises because the operations that the model provides are higher-level than those of the underlying architectures, simplifying the structure of software and reducing the difficulty of its construction. Stability arises because software construction can assume a standard interface over long time frames, regardless of developments in parallel computer architecture. At the same time, the model forms a fixed starting point for the implementation effort (transformation system, compiler, and run-time system) directed at each parallel computer. The model therefore insulates those issues that are the concern of software developers from those that are the concern of implementers. Furthermore, implementation decisions, and the work they require, are made once for each target rather than once for each program.

Since a model is just an abstract machine, models exist at many different levels of abstraction. For example, every programming language is a model in our sense, since each provides some simplified view of the underlying hardware. This makes it hard to compare models neatly because of the range of levels of abstraction involved and because many high-level models can be emulated by other lower-level models. There is not even a one-to-one connection between models: a low-level model can naturally emulate several different higher-level ones, and a high-level model can be naturally emulated by different low-level ones. We do not explicitly distinguish between programming languages and more abstract models (such as asynchronous order-preserving message passing) in what follows.

An executing parallel program is an extremely complex object. Consider a program running on a 100-processor system, large but not unusual today. There are 100 active threads at any given moment. To conceal the latency of communication and memory access, each processor is probably multiplexing several threads, so the number of active threads is several times larger (say, 300). Any thread may communicate with any of the other threads, and this communication may be asynchronous or may require a synchronization with the destination thread. So there are up to 300^2 possible interactions "in progress" at any instant. The state of such a program is very large. The program that gives rise to this executing entity must be significantly more abstract than a description of the entity itself if it is to be manageable by humans. To put it another way, a great deal of the actual arrangement of the executing computation ought to be implicit and capable of being inferred from its static description (the program), rather than having to be stated explicitly. This implies that models for parallel computation require high levels of abstraction, much higher than for sequential programming. It is still (just) conceivable to construct modestly sized sequential programs in assembly code, although the newest sequential architectures make this increasingly difficult. It is probably impossible to write a modestly sized MIMD parallel program for 100 processors in assembly code in a cost-effective way.

Furthermore, the detailed execution behavior of a particular program on an architecture of one style is likely to be very different from the detailed execution on another. Thus abstractions that conceal the differences between architecture families are necessary.

On the other hand, a model that is abstract is not of great practical interest if an efficient method for executing programs written in it cannot be found. Thus models must not be so abstract that it is intellectually, or even computationally, expensive to find a way to execute them with reasonable efficiency on a large number of parallel architec-

tures. A model, to be useful, must address both issues, abstraction and effectiveness, as summarized in the following set of requirements [Skillicorn 1994b]. A good model of parallel computation should have the following properties.

(1) *Easy to Program*. Because an executing program is such a complex object, a model must hide most of the details from programmers if they are to be able to manage, intellectually, the creation of software. As much as possible of the exact structure of the executing program should be inserted by the translation mechanism (compiler and run-time system) rather than by the programmer. This implies that a model should conceal the following.

—*Decomposition* of a program into parallel threads. The program must be divided up into the pieces that will execute on distinct processors. This requires separating the code and data structures into a potentially large number of pieces.

—*Mapping* of threads to processors. Once the program has been divided into pieces, a choice must be made about which piece is placed on which processor. The placement decision is often influenced by the amount of communication taking place between each pair of pieces, ensuring that pieces that communicate a lot are placed near each other in the interconnection network. It may also be necessary to ensure that particular pieces are mapped to processors that have some special hardware capability, for example, a high-performance floating-point functional unit.

—*Communication* among threads. Whenever nonlocal data are required, a communication action of some kind must be generated to move the data. Its exact form depends heavily on the designated architecture, but the processes at both ends must arrange to treat it consistently so that one process does not wait for data that never come.

—*Synchronization* among threads. There will be times during the computation when a pair of threads, or even a larger group, must know that they have jointly reached a common state. Again the exact mechanism used is target-dependent. There is enormous potential for deadlock in the interaction between communication and synchronization.

Optimal decomposition and mapping are known to be exponentially expensive to compute. The nature of communication also introduces complexity. Communication actions always involve dependencies, because the receiving end must be prepared to block and the sender often does too. This means that a chain of otherwise unexceptional communications among n processes can be converted to a deadlocked cycle by the actions of a single additional process (containing, say, a receiving action from the last process in the chain before a sending action to the first). Hence correctly placing communication actions requires awareness of an arbitrarily large fraction of the global state of the computation. Experience has shown that deadlock in parallel programs is indeed easy to create and difficult to remove. Requiring humans to understand programs at this level of detail effectively rules out scalable parallel programming.

Thus models ought to be as abstract and simple as possible. There should be as close a fit as possible between the natural way in which to express the program and that demanded by the programming language. For many programs, this may mean that parallelism is not even made explicit in the program text. For applications that are naturally expressed in a concurrent way, it implies that the apparent parallel structure of the program need not be related to the actual way in which parallelism is exploited at execution.

(2) *Software Development Methodology*. The previous requirement implies a large gap between the information pro-

vided by the programmer about the semantic structure of the program, and the detailed structure required to execute it. Bridging it requires a firm semantic foundation on which transformation techniques can be built. Ad hoc compilation techniques cannot be expected to work on problems of this complexity.

There is another large gap between specifications and programs that must also be addressed by firm semantic foundations. Existing sequential software is, with few exceptions, built using standard building blocks and algorithms. The correctness of such programs is almost never properly established; rather they are subjected to various test regimes, designed to increase confidence in the absence of disastrous failure modes. This methodology of testing and debugging does not extend to portable parallel programming for two major reasons: the new degree of freedom created by partitioning and mapping hugely increases the state space that must be tested—debugging thus requires interacting with this state space, in which even simple checkpoints are difficult to construct; and the programmer is unlikely to have access to more than a few of the designated architectures on which the program will eventually execute, and therefore cannot even begin to test the software on other architectures. There are also a number of more mundane reasons: for example, reproducibility on systems whose communication networks are effectively nondeterministic, and the sheer volume of data that testing generates. Verification of program properties after construction also seems too unwieldy for practical use. Thus only a process aiming to build software that is correct by construction can work in the long term. Such calculational approaches have been advocated for sequential programming, and they are not yet workable in the large there. Nevertheless, they seem essential for parallel programming, even if they must remain goals rather than practice in the medium term.

A great deal of the parallel software produced so far has been of a numerical or scientific kind, and development methodology issues have not been a priority. There are several reasons for this. Much numerical computing is essentially linear algebra, and so program structures are often both regular and naturally related to the mathematics from which they derive. Thus such software is relatively simpler than newer commercial applications. Second, long-term maintainability is emphasized less because of the research nature of such software. Many programs are intended for short-term use, generating results that make themselves obsolete by suggesting new problems to be attacked and new techniques to be used. By contrast, most commercial software must repay its development costs out of the business edge it creates.

(3) *Architecture-Independent.* The model should be architecture-independent, so that programs can be migrated from parallel computer to parallel computer without having to be redeveloped or indeed modified in any nontrivial way. This requirement is essential to permit a widespread software industry for parallel computers.

Computer architectures have comparatively short life spans because of the speed with which processor and interconnection technology are developing. Users of parallel computing must be prepared to see their computers replaced, perhaps every five years. Furthermore, it is unlikely that each new parallel computer will much resemble the one that it replaces. Redeveloping software more or less from scratch whenever this happens is not cost-effective, although this is usually what happens today. If parallel computation is to be useful, it must be possible to insulate software from changes in the underlying parallel computer, even when these changes are substantial.

This requirement means that a model must abstract from the features of any

particular style of parallel computer. Such a requirement is easy to satisfy in isolation, since any sufficiently abstract model satisfies it, but is more difficult with the other requirements.

(4) *Easy to Understand.* A model should be easy to understand and to teach, since otherwise it is impossible to educate existing software developers to use it. If parallelism is to become a mainstream part of computing, large numbers of people have to become proficient in its use. If parallel programming models are able to hide the complexities and offer an easy interface they have a greater chance of being accepted and used. Generally, easy-to-use tools with clear goals, even if minimal, are preferable to complex ones that are difficult to use.

These properties ensure that a model forms an effective target for software development. However, this is not useful unless, at the same time, the model can be implemented effectively on a range of parallel architectures. Thus we have some further requirements.

(5) *Guaranteed Performance.* A model should have guaranteed performance over a useful variety of parallel architectures. Note that this does not mean that implementations must extract every last ounce of performance out of a designated architecture. For most problems, a level of performance as high as possible on a given architecture is unnecessary, especially if obtained at the expense of much higher development and maintenance costs. Implementations should aim to preserve the order of the apparent software complexity and keep constants small.

Fundamental constraints on architectures, based on their communication properties, are now well understood. Architectures can be categorized by their power in the following sense: an architecture is powerful if it can execute an arbitrary computation with the expected performance, given the text of the program and basic properties of the designated architecture. The reason for reduced performance, in general, is congestion, which arises not from individual actions of the program but from the collective behavior of these actions. Powerful architectures have the resources, in principle at least, to handle congestion without having an impact on program performance; less powerful ones do not.

The most powerful architecture class consists of shared-memory MIMD computers and distributed-memory MIMD computers whose total interconnection network capacity grows faster than the number of processors, at least as fast as $p \log p$, where p is the number of processors. For such computers, an arbitrary computation with parallelism p taking time t can be executed in such a way that the product pt (called the *work*) is preserved [Valiant 1990b]. The apparent time of the abstract computation cannot be preserved in a real implementation since communication (and memory access) imposes latencies, typically proportional to the diameter of the interconnection network. However, the time dilation that this causes can be compensated for by using fewer processors, multiplexing several threads of the original program onto each one, and thus preserving the product of time and processors. There is a cost to this implementation, but it is an indirect one—there must be more parallelism in the program than in the designated architecture, a property known as *parallel slackness.*

Architectures in this richly connected class are powerful, but do not scale well because an increasing proportion of their resources must be devoted to interconnection network hardware. Worse still, the interconnection network is typically the most customized part of the architecture, and therefore by far the most expensive part.

The second class of architectures are distributed-memory MIMD computers whose total interconnection network capacity grows only linearly with the number of processors. Such computers are scalable because they require only a constant number of communication

links per processor (and hence the local neighborhoods of processors are unaffected by scaling) and because a constant proportion of their resources is devoted to interconnection network hardware. Implementing arbitrary computations on such machines cannot be achieved without loss of performance proportional to the diameter of the interconnection network. Computations taking time t and p processors have an actual work cost of ptd (where d is the diameter of the interconnection network). What goes wrong in emulating arbitrary computations on such architectures is that, during any step, each of the p processors could generate a communication action. Since there is only total capacity proportional to p in the interconnection network, these communications use its entire capacity for the next d steps in the worst case. Communication actions attempted within this window of d steps can be avoided only if the entire program is slowed by a factor of d to compensate. Thus this class necessarily introduces reduced performance when executing computations that communicate heavily. Architectures in this class are scalable, but they are not as powerful as those in the previous class.

The third class of architectures are SIMD machines which, though scalable, emulate arbitrary computations poorly. This is because of their inability to do more than a small constant number of different actions on each step [Skillicorn 1990].

Thus scalable architectures are not powerful and powerful architectures are not scalable. To guarantee performance across many architectures, these results imply that

—the amount of communication allowed in programs must be reduced by a factor proportional to the diameter of realistic parallel computers (i.e., by a factor of $\log p$ or $\sqrt{p}$); and

—computations must be made more regular, so that processors do fewer different operations at each moment, if SIMD architectures are considered as viable designated architectures.

The amount of communication that a program carries out can be reduced in two ways: either by reducing the number of simultaneous communication actions, or by reducing the distance that each travels. It is attractive to think that distance could always be reduced by clever mapping of threads to processors, but this does not work for arbitrary programs. Even heuristic algorithms whose goal is placement for maximum locality are expensive to execute and cannot guarantee good results. Only models that limit the frequency of communication, or are restricted enough to make local placement easy to compute, can guarantee performance across the full range of designated parallel computers.

(6) *Cost Measures.* Any program's design is driven, more or less explicitly, by performance concerns. Execution time is the most important of these, but others such as processor utilization or even cost of development are also important. We describe these collectively as the cost of the program.

The interaction of cost measures with the design process in sequential software construction is a relatively simple one. Because any sequential machine executes with speed proportional to any other, design decisions that change the asymptotic complexity of a program can be made before any consideration of the computer on which it will eventually run. When a target decision has been made, further changes can be made, but they are of the nature of tuning, rather than algorithm choice. In other words, the construction process has two phases: decisions are made about algorithms and the asymptotic costs of the program are affected; and decisions are made about arrangements of program text and only the constants in front of the asymptotic costs are affected [Knuth 1976].

This neat division cannot be made for parallel software development, because

small changes in program text and choice of designated computer are both capable of affecting the asymptotic cost of a program. If real design decisions are to be made, a model must make the cost of its operations visible during all stages of software development, before either the exact arrangement of the program or the designated computer has been decided. Intelligent design decisions rely on the ability to decide that Algorithm A is better than Algorithm B for a particular problem.

This is a difficult requirement for a model, since it seems to violate the notion of an abstraction. We cannot hope to determine the cost of a program without *some* information about the computer on which it will execute, but we must insist that the information required be minimal (since otherwise the actual computation of the cost is too tedious for practical use). We say that a model has cost measures if it is possible to determine the cost of a program from its text, minimal designated computer properties (at least the number of processors it has), and information about the size, but not the values, of its input. This is essentially the same view of cost that is used in theoretical models of parallel complexity such as the PRAM [Karp and Ramachandran 1990]. This requirement is the most contentious of all of them. It requires that models provide predictable costs and that compilers do not optimize programs. This is not the way in which most parallel software is regarded today, but we reiterate that design is not possible without it. And without the ability to design, parallel software construction will remain a black art rather than an engineering discipline.

A further requirement on cost measures is that they be well behaved with respect to modularity. Modern software is almost always developed in pieces by separate teams and it is important that each team need only know details of the interface between pieces. This means that it must be possible to give each team a resource budget such that the overall cost goal is met if each team meets its individual cost allocation. This implies that the cost measures must be compositional, so that the cost of the whole is easily computable from the cost of its parts, and convex, so that it is not possible to reduce the overall cost by increasing the cost of one part. Naive parallel cost measures fail to meet either of these requirements.

3.1 Implications

These requirements for a model are quite demanding and several subsets of them are strongly in tension with each other. For example, abstract models make it easy to build programs but hard to compile them to efficient code, whereas low-level models make it hard to build software but easy to implement it efficiently. We use these requirements as a metric by which to classify and assess models.

The level of abstraction that models provide is used as the primary basis for categorizing them. It acts as a surrogate for simplicity of the model, since in an abstract model less needs to be said about details of thread structure and points at which communication and synchronization take place. Level of abstraction also correlates well with quality of software development methodology since calculations on programs are most powerful when their semantics is clean.

The extent to which the structure of program implementations is constrained by the structure of the program text is closely related to properties such as guaranteed performance and cost measures. There are three levels of constraint on program structure.

—Models that allow dynamic process or thread structure cannot restrict communication. Because any new thread can generate communication, even models that restrict communication within a particular syntactic block cannot limit it over the whole program. Thus such models cannot guarantee that the communication gener-

ated by a program will not overrun the total communication capacity of a designated parallel computer. Because dynamic process creation involves decisions by the run-time system, it is also impossible to define cost measures that can be used during design. (This does not mean, of course, that it is impossible to write efficient programs in models with dynamic process structure, or that it is impossible to improve efficiency by good design or clever compilation techniques. All it means is that some programs that can be written in the model will perform poorly, and it is not straightforward to detect which ones. Such programs could potentially be avoided by a programming discipline, but they cannot be ruled out on the model's own terms.)

—Models that have a static process or thread structure but do not impose syntactic limits on communication permit programs to be written that can perform badly because of communication overruns. On the other hand, it is usually possible to define cost measures for them, since the overruns can be statically predicted from the program structure.

—Models that have a static process or thread structure can guarantee performance by suitably limiting the frequency with which communication actions can be written in the program. It follows that it is also possible to define cost measures for them.

In summary, only when the structure of the program is static, and the amount of communication is bounded, does a model both guarantee performance and allow that performance to be predicted by cost measures.

We use control of structure and communication as the secondary basis for categorizing models. This choice of priorities for classification reflects our view that parallel programming should aim to become a mainstream part of computing. In specialized areas, some of these requirements may be less important. For example, in the domain of high-performance numerical computing, program execution times are often quadratic or even worse in the size of the problem. In this setting, the performance penalty introduced by execution on a distributed-memory MIMD computer with a mesh topology, say, may be insignificant compared to the flexibility of an unrestricted-communication model. There will probably never be a model that satisfies all potential users of parallelism. However, models that satisfy many of the preceding requirements are good candidates for general-purpose parallelism, the application of parallelism across wide problem domains [McColl 1993a].

4. OVERVIEW OF MODELS

We now turn to assessing existing models according to the criteria outlined in the previous section. Most of these models were not developed with the ambitious goal of general-purpose parallelism, so it is not a criticism to say that some of them fail to meet all of the requirements. Our goal is to provide a picture of the state of parallel programming today, but from the perspective of seeing how far towards general-purpose parallelism it is reasonable to get.

We have not covered all models for parallel computation, but we have tried to include those that introduce significant ideas, together with some sense of the history of such models. We do not give a complete description of each model, but instead concentrate on the important features and provide comprehensive references. Many of the most important papers on programming models and languages have been reprinted in Skillicorn and Talia [1994].

Models are presented in decreasing order of abstraction, in the following categories.

(1) Models that abstract from parallelism completely. Such models describe only the purpose of a program and not how it is to achieve this

purpose. Software developers do not even need to know if the program they build will execute in parallel. Such models are necessarily abstract and relatively simple, since programs need be no more complex than sequential ones.

(2) Models in which parallelism is made explicit, but decomposition of programs into threads is implicit (and hence so is mapping, communication, and synchronization). In such models, software developers are aware that parallelism will be used and must have expressed the potential for it in programs, but they do not know how much parallelism will actually be applied at run-time. Such models often require programs to express the maximal parallelism present in the algorithm and then the implementation reduces that degree of parallelism to fit the designated architecture, at the same time working out the implications for mapping, communication, and synchronization.

(3) Models in which parallelism and decomposition must both be made explicit, but mapping, communication, and synchronization are implicit. Such models require decisions to be made about the breaking up of available work into pieces, but they relieve the software developer of the implications of such decisions.

(4) Models in which parallelism, decomposition, and mapping are explicit, but communication and synchronization are implicit. Here the software developer must not only break the work up into pieces, but must also consider how best to place the pieces on the designated processor. Since locality often has a marked effect on communication performance, this almost inevitably requires an awareness of the designated processor's interconnection network. It becomes hard to make such software portable across different architectures.

(5) Models in which parallelism, decomposition, mapping, and communication are explicit, but synchronization is implicit. Here the software developer is making almost all of the implementation decisions, except that fine-scale timing decisions are avoided by having the system deal with synchronization.

(6) Models in which everything is explicit. Here software developers must specify all of the detail of the implementation. As we noted earlier, it is extremely difficult to build software using such models, because both correctness and performance can only be achieved by attention to vast numbers of details.

Within each of these categories, we present models according to their degree of control over structure and communication, in these categories:

—models in which thread structure is dynamic;
—models in which thread structure is static but communication is not limited; and
—models in which thread structure is static and communication is limited.

Within each of these categories we present models based on their common paradigms. Figures 2 and 3 show a classification of models for parallel computation in this way.

4.1 Nothing Explicit

The best models of parallel computation for programmers are those in which they need not be aware of parallelism at all. Hiding all the activities required to execute a parallel computation means that software developers can carry over their existing skills and techniques for sequential software development. Of course, such models are necessarily abstract, which makes the implementer's job difficult since the transformation, compilation, and run-time systems must infer all of the structure of the eventual program. This means deciding how the

- Nothing Explicit, Parallelism Implicit
  - Dynamic Structure
    - Higher-order Functional-Haskell
    - Concurrent Rewriting—OBJ, Maude
    - Interleaving—Unity
    - Implicit Logic Languages—PPP, AND/OR, REDUCE/OR, Opera, Palm, concurrent constraint languages
  - Static Structure
    - Algorithmic Skeletons—P3L, Cole, Darlington
  - Static and Communication-Limited Structure
    - Homomorphic Skeletons—Bird-Meertens Formalism
    - Cellular Processing Languages—Cellang, Carpet, CDL, Ceprol
    - Crystal
- Parallelism Explicit, Decomposition Implicit
  - Dynamic Structure
    - Dataflow—Sisal, Id
    - Explicit Logic Languages—Concurrent Prolog, PARLOG, GHC, Delta-Prolog, Strand
    - Multilisp
  - Static Structure
    - Data Parallelism Using Loops—Fortran variants, Modula 3*
    - Data Parallelism on Types—pSETL, parallel sets, match and move, Gamma, PEI, APL, MOA, Nial and AT
  - Static and Communication-Limited Structure
    - Data-Specific Skeletons—scan, multiprefix, paralations, dataparallel C, NESL, CamlFlight
- Decomposition Explicit, Mapping Implicit
  - Dynamic Structure
  - Static Structure
    - BSP, LogP
  - Static and Communication-Limited Structure.
- Mapping Explicit, Communication Implicit
  - Dynamic Structure
    - Coordination Languages—Linda, SDL
    - Non-message Communication Languages—ALMS, PCN, Compositional C++
    - Virtual Shared Memory
    - Annotated Functional Languages—Paralf
    - RPC—DP, Cedar, Concurrent CLU, DP
  - Static Structure
    - Graphical Languages—Enterprise, Parsec, Code
    - Contextual Coordination Languages—Ease, ISETL-Linda, Opus
  - Static and Communication-Limited Structure
    - Communication Skeletons
- Communication Explicit, Synchronization Implicit
  - Dynamic Structure
    - Process Networks—Actors, Concurrent Aggregates, ActorSpace, Darwin
    - External OO—ABCL/1, ABCL/R, POOL-T, EPL, Emerald, Concurrent Smalltalk
    - Objects and Processes—Argus, Presto, Nexus
    - Active Messages—Movie
  - Static Structure
    - Process Networks—static dataflow
    - Internal OO—Mentat
  - Static and Communication-Limited Structure
    - Systolic Arrays—Alpha
- Everything Explicit
  - Dynamic Structure
    - Message Passing—PVM, MPI
    - Shared Memory—FORK, Java, thread packages
    - Rendezvous—Ada, SR, Concurrent C
  - Static Structure
    - Occam
  - PRAM

Figure 2. Classification of models of parallel computation.

specified computation is to be achieved, dividing it into appropriately sized pieces for execution, mapping those pieces, and scheduling all of the communication and synchronization among them.

At one time it was widely believed that automatic translation from abstract program to implementation might be effective starting from an ordinary sequential imperative language. Although a great deal of work was invested in parallelizing compilers, the approach was defeated by the complexity of determining whether some aspect of a program was essential or simply an artifact of its sequential expression. It is now acknowledged that a highly automated translation process is only practical if it begins from a carefully chosen model that is both abstract and expressive.

Inferring all of the details required for an efficient and architecture-independent implementation is possible, but it is difficult and, at present, few such models can guarantee performance.

We consider models at this high level of abstraction in subcategories: those that permit dynamic process or thread structure and communication, those that have static process or thread struc-

ture but unlimited communication, and those that also limit the amount of communication in progress at any given moment.

4.1.1 *Dynamic Structure.* A popular approach to describing computations in a declarative way, in which the desired result is specified without saying how that result is to be computed, is to use a set of functions and equations on them. The result of the computation is a solution, usually a least fixed point, of these equations. This is an attractive framework in which to develop software, for such programs are both abstract and amenable to formal reasoning by equational substitution. The implementation problem is then to find a mechanism to solve such equations.

Higher-order functional programming treats functions as λ-terms and computes their values using reduction in the λ-calculus, allowing them to be stored in data structures, passed as arguments, and returned as results. An example of a language that allows higher-order functions is Haskell [Hudak and Fasel 1992]. Haskell also includes several typical features of functional programming such as user-defined types, lazy evaluation, pattern matching, and list comprehensions. Furthermore, Haskell has a parallel functional I/O system and provides a module facility.

The actual technique used in higher-order functional languages for computing function values is called *graph reduction* [Peyton-Jones and Lester 1992]. Functions are expressed as trees, with common subtrees for shared subfunctions (hence graphs). Computation rules select graph substructures, reduce them to simpler forms, and replace them in the larger graph structure. When no further computation rules can be applied, the graph that remains is the result of the computation.

It is easy to see how the graph-reduction approach can be parallelized in principle—rules can be applied to nonoverlapping sections of the graph independently and hence simultaneously. Thus multiple processors can search for reducible parts of the graph independently and in a way that depends only on the structure of the graph (and so does not have to be inferred by a compiler beforehand). For example, if the expression (exp1 ∗ exp2), where exp1 and exp2 are arbitrary expressions, is to be evaluated, two threads may independently evaluate exp1 and exp2, so that their values are computed simultaneously.

Unfortunately, this simple idea turns out to be quite difficult to make work effectively. First, only computations that contribute to the final result should be executed, since doing others wastes resources and alters the semantics of the program if a nonessential piece fails to terminate. For example, most functional languages have some form of conditional like

```
if b(x) then
   f(x)
else
   g(x)
```

Clearly exactly one of the values of f(x) or g(x) is needed, but which one is not known until the value of b(x) is known. So evaluating b(x) first prevents redundant work, but on the other hand lengthens the critical path of the computation (compared to evaluating f(x) and g(x) speculatively). Things are even worse if, say, f(x) fails to terminate, but only for values of x for which b(x) is false. Now evaluating f(x) speculatively causes the program not to terminate, whereas the other evaluation order does not.

It is difficult to find independent program pieces that are known to be required to compute the final result without quite sophisticated analysis of the program as a whole. Also, the actual structure of the graph changes dramatically during evaluation, so that it is difficult to do load-balancing well and to handle the spawning of new subtasks and communication effectively. Parallel graph reduction has been a limited success for shared-memory distributed

computers, but its effectiveness for distributed-memory computers is still unknown.[1] Such models are simple and abstract and allow software development by transformation, but they do not guarantee performance. Much of what happens during execution is determined dynamically by the run-time system so that cost measures (in our sense) cannot practically be provided.

Concurrent rewriting is a closely related approach in which the rules for rewriting parts of programs are chosen in some other way. Once again, programs are terms describing a desired result. They are rewritten by applying a set of rules to subterms repeatedly until no further rules can be applied. The resulting term is the result of the computation. The rule set is usually chosen to be both terminating (there is no infinite sequence of rewrites) and confluent (applying rules to overlapping subterms gets the same result in the end), so that the order and position where rules are applied makes no difference to the final result. Some examples of such models are OBJ [Goguen and Winkler 1988; Goguen et al. 1993, 1994], a functional language whose semantics is based on equational logic, and Maude.[2] An example, based on one in Lincoln et al. [1994], gives the flavor of this approach. The following is a functional module for polynomial differentiation, assuming the existence of a module that represents polynomials and the usual actions on them. The lines beginning with eq are rewrite rules, which should be familiar from elementary calculus. The line beginning with ceq is a conditional rewrite rule.

```
fmod POLY-DER is
  protecting POLYNOMIAL .
  op der : Var Poly → Poly .
  op der : Var Mon → Poly .
  var A : Int .
  var N : NzNat .
  vars P Q : Poly .
  vars U V : Mon .
  eq der(P + Q) = der(P) + der(Q) .
  eq der(U . V) = (der(U) . V) + (U .
    der(V)) .
  eq der(A * U) = A * der(U) .
  ceq der(X ^ N) = N * (X ^ (N - 1)) if
    N > 1 .
  eq der(X ^ 1) = 1 .
  eq der(A) = 0 .
endfm
```

An expression such as

```
der(X ^ 5 + 3 * X ^ 4 + 7 * X ^ 2)
```

can be computed in parallel because there are soon multiple places where a rewrite rule can be applied. This simple idea can be used to emulate many other parallel computation models.

Models of this kind are simple and abstract and allow software development by transformation, but again they cannot guarantee performance and are too dynamic to allow useful cost measures.

Interleaving is a third approach that derives from multiprogramming ideas in operating systems via models of concurrency such as transition systems. If a computation can be expressed as a set of subcomputations that commute, that is, it can be evaluated in any order and repeatedly, then there is considerable freedom for the implementing system to decide on the actual structure of the executing computation. It can be quite hard to express a computation in this form, but it is made considerably easier by allowing each piece of the computation to be protected by a *guard,* that is, a Boolean-valued expression. Informally speaking, the semantics of a program in this form is that all the guards are evaluated and one or more subprograms whose guards are true are then evaluated. When they have completed, the whole process begins again. Guards could determine the whole sequence of the computation, even sequentializing it by having guards of the form step = i, but the intent of the model is rather to use the weakest guards, and therefore,

[1] See Darlington et al. [1987], Hudak and Fasel [1992], Kelly [1989], Peyton-Jones et al. [1987], Rabhi and Manson [1991], and Thompson [1992].
[2] See Lechner et al. [1995], Meseguer [1992], Meseguer and Winkler [1991], and Winkler [1993].

say, the least about how the pieces are to be fit together.

This idea lies behind UNITY[3] and an alternative that considers independence of statements more: action systems.[4] UNITY (unbounded nondeterministic iterative transformations) is both a computational model and a proof system. A UNITY program consists of declaration of variables, specification of their initial values, and a set of multiple-assignment statements. In each step of execution, some assignment statement is selected nondeterministically and executed. For example, the following program,

```
Program P
  initially x=0
  assign x:= a(x) || x:= b(x) || x:= c(x)
end {P}
```

consists of three assignments that are selected nondeterministically and executed. The selection procedure obeys a fairness rule: every assignment is executed infinitely often.

Like rewriting approaches, interleaving models are abstract and simple, but guaranteed performance seems unlikely and cost measures are not possible.

Implicit logic languages exploit the fact that the resolution process of a logic query contains many activities that can be performed in parallel [Chassin de Kergommeaux and Codognet 1994]. The main types of inherent parallelism in logic programs are OR parallelism and AND parallelism. OR parallelism is exploited by unifying a subgoal with the head of several clauses in parallel. For instance, if the subgoal to be solved is ?-a(x) and the matching clauses are

```
a(x):- b(x).  a(x):- c(x).
```

then OR parallelism is exploited by unifying, in parallel, the subgoal with the head of each of the two clauses. For an introduction to logic programming, see Lloyd [1984]. AND parallelism divides the computation of a goal into several threads, each of which solves a single subgoal in parallel. For instance, if the goal to be solved is

```
?- a(x), b(x), c(x).
```

The subgoals a(x), b(x), and c(x) are solved in parallel. Minor forms of parallelism are search parallelism and unification parallelism where parallelism is exploited, respectively, in the searching of the clause database and in the unification procedure.

Implicit parallel logic languages provide automatic decomposition of the execution tree of a logic program into a network of parallel threads. This is done by the language support both by static analysis at compile time and at run-time. No explicit annotations of the program are needed. Implicit logic models include PPP [Fagin and Despain 1990], the AND/OR process model [Conery 1987], the REDUCE/OR model [Kale 1987], OPERA [Briat et al. 1991], and PALM [Cannataro et al. 1991]. These models differ in how they view parallelism and their designated architectures are varied, but they are mainly designed to be implemented on distributed-memory MIMD machines [Talia 1994]. To implement parallelism these models use either thread-based or subtree-based strategies. In thread-based models, each single goal is solved by starting a thread. In subtree-based models, the search tree is divided into several subtrees, with one thread associated with each subtree. These two different approaches correspond to different grain sizes: in thread-based models the grain size is fine, whereas in the subtree-based models the parallelism grain size is medium or coarse.

Like other approaches discussed in this section, implicit parallel logic languages are highly abstract. Thus it is hard to guarantee performance, although good performance may sometimes be achieved. Cost measures can-

[3] See Bickford [1994], Chandy and Misra [1988], Gerth and Pneuli [1989], and Radha and Muthukrishnan [1992].
[4] See Back [1989a,b] and Back and Sere [1989; 1990].

not be provided because implicit logic languages are highly dynamic.

Constraint logic programming is an important generalization of logic programming aimed at replacing the pattern-matching mechanism of unification by a more general operation called *constraint satisfaction* [Saraswat et al. 1991]. In this environment, a constraint is a subset of the space of all possible values that a variable of interest can take. A programmer does not explicitly use parallel constructs in a program, but defines a set of constraints on variables. This approach offers a framework for dealing with domains other than Herbrand terms, such as integers and Booleans [Lloyd 1984]. In concurrent constraint logic programming, a computation progresses by executing threads that concurrently communicate by placing constraints in a global store and synchronizes by checking that a constraint is entailed by the store. The communication patterns are dynamic, so that there is no predetermined set of threads with which a given thread interacts. Moreover, threads correspond to goal atoms, so they are activated dynamically during program execution. Concurrent constraint logic programming models include *cc* [Saraswat et al. 1991], the CHIP CLP language [van Hentenryek 1989], and CLP [Jaffar and Lassez 1987]. As in other parallel logic models, concurrent constraint languages are too dynamic to allow practical cost measures.

4.1.2 *Static Structure.* One way to infer the structure to be used to compute an abstract program is to insist that the abstract program be based on fundamental units or components whose implementations are predefined. In other words, programs are built by connecting ready-made building blocks. This approach has the following natural advantages.

—The building blocks raise the level of abstraction because they are the fundamental units in which programmers work. They can hide an arbitrary amount of internal complexity.
—The building blocks can be internally parallel but composable sequentially, in which case programmers need not be aware that they are programming in parallel.
—The implementation of each building block needs to be done only once for each architecture. The implementation can be done by specialists, and time and energy can be devoted to making it efficient.

In the context of parallel programming, such building blocks have come to be called *skeletons* [Cole 1989], and they underlie a number of important models. For example, a common parallel programming operation is to sum the elements of a list. The arrangement of control and communication to do this is exactly the same as that for computing the maximum element of a list and for several other similar operations. Observing that these are all special cases of a reduction provides a new abstraction for programmer and implementer alike. Furthermore, computing the maximum element of an array or of a tree is not very different from computing it for a list, so that the concept of a reduction carries over to other potential applications. Observing and classifying such regularities is an important area of research in parallel programming today. One overview and classification of skeletons is the Basel Algorithm Classification Scheme [Burkhart et al. 1993].

For the time being, we restrict our attention to *algorithmic skeletons,* those that encapsulate control structures. The idea is that each skeleton corresponds to some standard algorithm or algorithm fragment, and that these skeletons can be composed sequentially. Software developers select the skeletons they want to use and put them together. The compiler or library writer chooses how each encapsulated algorithm is implemented and how intra- and inter-skeleton parallelism are exploited for each possible target architecture.

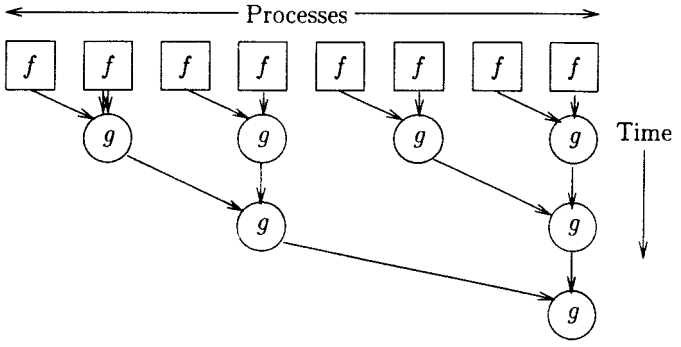

Figure 3. A skeleton for computing arbitrary list homomorphisms.

We briefly mention some of the most important algorithmic skeleton approaches. The Pisa Parallel Programming Language (P^3L)[5] uses a set of algorithmic skeletons that capture common parallel programming paradigms such as pipelines, worker farms, and reductions. For example, in P^3L worker farms are modeled by means of the farm constructor as follows.

```
farm P in (int data) out (int result)
  W in (data) out (result)
  result = f(data)
  end
end farm
```

When the skeleton is executed, a number of workers W are executed in parallel with the two P processes (the emitter and the collector). Each worker executes the function f( ) on its data partition. Similar skeletons were developed by Cole [1989, 1990, 1992, 1994], who also computed cost measures for them on a parallel architecture. Work of a similar sort, using skeletons for reduce and map over pairs, pipelines, and farms, is also being done by Darlington's group at Imperial College [Darlington et al. 1993].

Algorithmic skeletons are simple and abstract. However, because programs must be expressed as compositions of the skeletons provided, the expressiveness of the abstract programming language is an open question. None of the approaches previously described addresses this explicitly, nor is there any natural way in which to develop algorithmic skeleton programs, either from some higher-level abstraction or directly at the skeleton level. On the other hand, guaranteed performance for skeletons is possible if they are chosen with care, and because of this, cost measures can be provided.

4.1.3 *Static and Communication-Limited Structure.* Some skeleton approaches bound the amount of communication that takes place, usually because they incorporate awareness of geometric information.

One such model is *homomorphic skeletons* based on data types, an approach that developed from the Bird–Meertens formalism [Skillicorn 1994b]. The skeletons in this model are based on particular data types, one set for lists, one set for arrays, one set for trees, and so on. All homomorphisms on a data type can be expressed as an instance of a single recursive and highly parallel computation pattern, so that the arrangement of computation steps in an implementation needs to be done only once for each datatype.

Consider the pattern of computation and communication shown in Figure 3. Any list homomorphism can be computed by appropriately substituting for f and g, where g must be associative.

[5] See Baiardi et al. [1991] and Danelutto et al. [1991a,b; 1990].

For example,

sum	$f = id, g = +$
maximum	$f = id$, $g =$ binary max
length	$f = K_1, g = +$(where K_1 is the function that always returns 1)
sort	$f = id, g =$ merge

Thus a template for scheduling the individual computations and communications can be reused to compute many different list homomorphisms by replacing the operations that are done as part of the template. Furthermore, this template can also be used to compute homomorphisms on bags (multisets), with slightly weaker conditions on the operations in the g slots—they must be commutative as well as associative.

The communication required for such skeletons is deducible from the structure of the data type, so each implementation needs to construct an embedding of this communication pattern in the interconnection topology of each designated computer. Very often the communication requirements are mild; for example, it is easy to see that list homomorphisms require only the existence of a logarithmic-depth binary tree in the designated architecture interconnection network. All communication can then take place with nearest neighbors (and hence in constant time). Homomorphic skeletons have been built for most of the standard types: sets and bags [Skillicorn 1994b], lists [Bird 1987; Malcolm 1990; Spivey 1989], trees [Gibbons 1991], arrays [Banger 1992, 1995], molecules [Skillicorn 1994a], and graphs [Singh 1993].

The homomorphic skeleton approach is simple and abstract, and the method of construction of data type homomorphisms automatically generates a rich environment for equational transformation. The communication pattern required for each type is known as the standard topology for that type. Implementations with guaranteed performance can be built for any designated computers into whose interconnection topologies the standard topology can be embedded. Because the complete schedule of computation and communication is determined in advance by the implementer, cost measures can be provided [Skillicorn and Cai 1995].

Cellular processing languages are based on the execution model of cellular automata. A cellular automaton consists of a possibly infinite n-dimensional lattice of cells. Each cell is connected to a limited set of adjacent cells. A cell has a state chosen from a finite alphabet. The state of a cellular automaton is completely specified by the values of the variables at each cell. The state of a single cell is a simple or structured variable that takes values in a finite set. The states of all cells in the lattice are updated simultaneously in discrete time steps. Cells update their values by using a transition function that takes as input the current state of the local cell and some limited collection of nearby cells that lie within some bounded distance, known as a neighborhood. Simple neighborhoods of a cell (C) in a 2-D lattice are

```
 N     NNN   N N
NCN    NCN     C
 N     NNN   N N
```

Cellular processing languages, such as Cellang [Eckart 1992], CARPET [Spezzano and Talia 1997], CDL, and CEPROL [Seutter 1985], allow cellular algorithms to be described by defining the state of cells as a typed variable, or a record of typed variables, and a transition function containing the evolution rules of an automaton. Furthermore, they provide constructs for the definition of the pattern of the cell neighborhood. These languages implement a cellular automaton as an SIMD or SPMD program, depending on the designated architecture. In the SPMD (single program, multiple data) approach, cellular algorithms are implemented as a collection of medium-grain processes mapped onto different processing elements.

Each process executes the same program (the transition function) on different data (the state of cells). Thus all the processes obey, in parallel, the same local rule, which results in a global transformation of the whole automaton. Communication occurs only among neighboring cells, so the communication pattern is known statically. This allows scalable implementations with guaranteed performance, both on MIMD and SIMD parallel computers [Cannataro et al. 1995]. Moreover, cost measures can be provided.

Another model that takes geometric arrangement explicitly into account is Crystal.[6] Crystal is a functional language with added data types called *index domains* to represent geometry, that is, locality and arrangement. The feature distinguishing Crystal from other languages with geometric annotations is that index domains can be transformed and the transformations reflected in the computational part of programs. Crystal is simple and abstract, and possesses a transformation system based both on its functional semantics and transformations of index domains. Index domains are a flexible way of incorporating target interconnection network topology into derivations, and Crystal provides a set of cost measures to guide such derivations. A more formal approach that is likely to lead to interesting developments in this area is Jay's [1995] shapely types.

The key to an abstract model is that the details of the implementation must somehow be implicit in the text of each program. We have seen two approaches. The first relies on the run-time system to find and exploit parallelism and the second relies on predefined building blocks, knowledge of which is built into the compiler so that it can replace the abstract program operations by implementations, piece by piece.

[6] See Creveuil [1991] and Yang and Choo [1991; 1992a,b].

4.2 Parallelism Explicit

The second major class of models is that in which parallelism is explicit in abstract programs, but software developers do not need to be explicit about how computations are to be divided into pieces and how those pieces are mapped to processors and communicate. The main strategies for implementing decomposition both depend on making decomposition and mapping computationally possible and effective. The first is to renounce temporal and spatial locality and assume low-cost context switch, so that the particular decomposition used does not affect performance much. Because any decomposition is effective, a simple algorithm can be used to compute it. The second is to use skeletons that have a natural mapping to designate processor topologies, skeletons based on the structure of the data that the program uses.

4.2.1 Dynamic Structure. Dataflow [Herath et al. 1987] expresses computations as operations, which may in principle be of any size but are usually small, with explicit inputs and results. The execution of these operations depends solely on their data dependencies—an operation is computed after all of its inputs have been computed, but this moment is determined only at run-time. Operations that do not have a mutual data dependency may be computed concurrently.

The operations of a dataflow program are considered to be connected by paths, expressing data dependencies, along which data values flow. They can be considered, therefore, as collections of first-order functions. Decomposition is implicit, since the compiler can divide the graph representing the computation in any way. The cut edges become the places where data move from one processor to another. Processors execute operations in an order that depends solely on those that are ready at any given moment. There is therefore no temporal context beyond the execution of each single operation, and hence no

advantage to temporal locality. Because operations with a direct dependence are executed at widely different times, possibly even on different processors, there is no advantage to spatial locality either. As a result, decomposition has little direct effect on performance (although some caveats apply). Decomposition can be done automatically by decomposing programs into the smallest operations and then clustering to get pieces of appropriate size for the designated architecture's processors. Even random allocation of operations to processors performs well on many dataflow systems.

Communication is not made explicit in programs. Rather, the occurrence of a name as the result of an operation is associated, by the compiler, with all of those places where the name is the input of an operation. Because there are really no threads (threads contain only a single instruction), communication is effectively unsynchronized.

Dataflow languages have taken different approaches to expressing repetitive operations. Languages such as Id [Ekanadham 1991] and Sisal [McGraw et al. 1985; McGraw 1993; Skedzielewski 1991] are first-order functional (or single assignment) languages. They have syntactic structures looking like loops that create a new context for each execution of the "loop body" (so that they seem like imperative languages except that each variable name may be assigned to only once in each context). For example, a Sisal loop with single-assignment semantics can be written as follows.

```
for i in 1, N
   x := A[i] + B[i]
   returns value of sum x
end for
```

Sisal is the most abstract of the dataflow languages, because parallelism is only visible in the sense that its loops are expected to be opportunities for parallelism. In this example, all of the loop bodies could be scheduled simultaneously and then their results collected.

Dataflow languages are abstract and simple, but they do not have a natural software development methodology. They can provide guaranteed performance; indeed, Sisal performs competitively with the best FORTRAN compilers on shared-memory architectures [McGraw 1993]. However, performance on distributed-memory architectures is still not competitive. Because so much scheduling is done dynamically at runtime, cost measures are not possible.

Explicit logic languages are those in which programmers must specify the parallelism explicitly [Shapiro 1989]. They are also called concurrent logic languages. Examples of languages in this class are PARLOG [Gregory 1987], Delta-Prolog [Pereira and Nasr 1984], Concurrent Prolog [Shapiro 1986], GHC [Ueda 1985], and Strand [Foster and Taylor 1990].

Concurrent logic languages can be viewed as a new interpretation of Horn clauses, the process interpretation. According to this interpretation, an atomic goal ← C can be viewed as a process, a conjunctive goal ← C1, . . . , Cn as a process network, and a logic variable shared between two subgoals can be viewed as a communication channel between two processes. The exploitation of parallelism is achieved through the enrichment of a logic language like Prolog with a set of mechanisms for the annotation of programs. One of these mechanisms, for instance, is the annotation of shared logical variables to ensure that they are instantiated by only one subgoal. For example, the model of parallelism utilized by PARLOG and Concurrent Prolog languages is based on the CSP (communicating sequential processes) model. In particular, communication channels are implemented in PARLOG and Concurrent Prolog by means of logical variables shared between two subgoals (e.g., p(X,Y), q(Y,Z)). Both languages use the guard concept to handle nondeterminism in the same way as it is used in CSP to delay communication between parallel processes until a commitment is reached.

A program in a concurrent logic language is a finite set of guarded clauses,

$$H \leftarrow G1, G2, \ldots, Gn \mid B1, B2, \ldots, Bm. \quad n,m \geq 0,$$

where H is the clause head, the set Gi is the guard, and Bi is the body of the clause. Operationally the guard is a test that must be successfully evaluated with the head unification for the clause to be selected. The symbol |, the *commit* operator, is used as a conjunction between the guard and the body. If the guard is empty, the commit operator is omitted.

The declarative reading of a guarded clause is: H is true if both Gi and Bi are true. According to the process interpretation, to solve H it is necessary to solve the guard Gi, and if its resolution is successful, B1, B2, . . . , Bm are solved in parallel.

These languages require programmers to explicitly specify, using annotations, which clauses can be solved in parallel [Talia 1993]. For example, in PARLOG the '.' and ';' clause separators control the search for a candidate clause. Each group of '.' separated clauses are tried in parallel. The clauses following a ';' are tried only if all the clauses that precede the ';' have been found to be noncandidate clauses. For instance, suppose that a relation is defined by a sequence of clauses

C1. C2; C3.

The clauses C1 and C2 are tested for candidacy in parallel, but the clause C3 is tested only if both C1 and C2 are found to be noncandidate clauses. Although concurrent logic languages extend the application areas of logic programming from artificial intelligence to system-level applications, program annotations require a different style of programming. They weaken the declarative nature of logic programming by making the exploitation of parallelism the responsibility of the programmer. From the point of view of our classification, concurrent logic programs such as PARLOG programs define parallel execution explicitly by means of annotations. Furthermore, PARLOG allows dynamic creation of threads to solve subgoals but, because of the irregular structure of programs, it does not limit communication among threads.

Another symbolic programming language in which parallelism is made explicit by the programmer is Multilisp. The Multilisp [Halstead 1986] language is an extension of Lisp in which opportunities for parallelism are created using *futures.* In the language implementation, there is a one-to-one correspondence between threads and futures. The expression (future X) returns a suspension for the value of X and immediately creates a new process to evaluate X, allowing parallelism between the process computing a value and the process using that value. When the value of X is computed, the value replaces the future. Futures give a model that represents partially computed values; this is especially significant in symbolic processing where operations on structured data occur often. An attempt to use the result of a future suspends until the value has been computed. Futures are first-class objects and can be passed around regardless of their internal status. The future construct creates a computation style much like that found in the dataflow model. In fact, futures allow eager evaluation in a controlled way that fits between the fine-grained eager evaluation of dataflow and the laziness of higher-order functional languages. By using futures we have dynamic thread creation, and hence programs with dynamic structure; moreover, the communication of values among expressions is not limited.

4.2.2 *Static Structure.* Turning to models with static structure, the skeleton concept appears once more, but this time based around single data structures. At first glance it would seem that monolithic operations on objects of a data type, doing something to every item of a list or array, is a programming

model of limited expressiveness. However, it turns out to be a powerful way of describing many interesting algorithms.

Data parallelism arose historically from the attempt to use computational pipelines. Algorithms were analyzed for situations in which the same operation was applied repeatedly to different data and the separate applications did not interact. Such situations exploit vector processors to dramatically reduce the control overhead of the repetition, since pipeline stalls are guaranteed not to occur because of the independence of the steps. With the development of SIMD computers, it was quickly realized that vectorizable code is also SIMD code, except that the independent computations proceed simultaneously instead of sequentially. SIMD code can be efficiently executed on MIMD computers as well, so vectorizable code situations can be usefully exploited by a wide range of parallel computers.

Such code often involves arrays and can be seen more abstractly as instances of *maps*, the application of a function to each element of a data structure. Having made this abstraction, it is interesting to ask what other operations might be useful to consider as applied monolithically to a data structure. Thus data parallelism is a general approach in which programs are compositions of such monolithic operations applied to objects of a data type and producing results of that same type.

We distinguish two approaches to describing such parallelism: one based on (parallel) loops and the other based on monolithic operations on data types.

Consider FORTRAN with the addition of a **ForAll** loop, in which iterations of the loop body are conceptually independent and can be executed concurrently. For example, a **ForAll** statement such as

```
ForAll (I = 1:N, J = 1:M)
  A(I,J) = I * B(J)
```

on a parallel computer can be executed in parallel. Care must be taken to ensure that the loops do not reference the same locations; for example, by indexing the same element of an array via a different index expression. This cannot be checked automatically in general, so most FORTRAN dialects of this kind place the responsibility on the programmer to make the check. Such loops are maps, although not always over a single data object.

Many FORTRAN dialects such as FORTRAN-D [Tseng 1993] and High Performance Fortran (HPF) [High Performance FORTRAN Language Specification 1993; Steele 1993] start from this kind of parallelism and add more direct data parallelism by including constructs for specifying how data structures are to be allocated to processors, and operations to carry out other data-parallel operations, such as reductions. For example, HPF, a parallel language based on FORTRAN-90, FORTRAN D, and SIMD FORTRAN, includes the Align directive to specify that certain data are to be distributed in the same way as certain other data. For instance

```
!HPF$ Align X (:,:) with D (:,:)
```

aligns X with D, that is, ensures that elements of X and D with the same indices are placed on the same processor. Furthermore, the Distribute directive specifies a mapping of data to processors; for example,

```
!HPF$ Distribute D2 (Block, Block)
```

specifies that the processors are to be considered a two-dimensional array and the points of D2 are to associate with processors in this array in a blocked fashion. HPF also offers a directive to inform the compiler that operations in a loop can be executed independently (in parallel). For example, the following code asserts that A and B do not share memory space.

```
!HPF$ Independent
  Do I = 1, 1000
  A(I) = B(I)
  end Do
```

Other related languages are Pandore II [André et al. 1994; Andre and Thomas 1989; Jerid et al. 1994] and C** [Larus et al. 1992]. This work is beginning to converge with skeleton approaches: for example, Darlington's group has developed a FORTRAN extension that uses skeletons [Darlington et al. 1995]. Another similar approach is the latest language in the Modula family, Modula 3* [Heinz 1993]. Modula 3* supports **forall**-style loops over data types in which each loop body executes independently and the loop itself ends with a barrier synchronization. It is compiled to an intermediate language that is similar in functionality to HPF.

Data-parallel languages based on data types other than arrays have also been developed. Some examples are: parallel SETL [Flynn-Hummel and Kelly 1993; Hummel et al. 1991], parallel sets [Kilian 1992a,b], match and move [Sheffler 1992], Gamma [Banâtre and LeMetayer 1991; Creveuil 1991; Mussat 1991], and PEI [Violard 1994]. Parallel SETL is an imperative-looking language with data-parallel operations on bags. For example, the inner statement of a matrix multiplication looks like

$$c(i,j) := +/\{a(i,k) * b(k,j) : k \text{ over } \{1..n\}\}.$$

The outer set of braces is a bag comprehension, generating a bag containing the values of the expression before the colon for all values of k implied by the text after the colon. The +/ is a bag reduction or fold, summing these values. Gamma is a language with data-parallel operations on finite sets. For example, the code to find the maximum element of a set is

$$\text{maxM} := x{:}M,\ y{:}M \rightarrow x{:}M \leftarrow x \geq y$$

which specifies that any pair of elements x and y may be replaced in a set by the element x, provided the value in x is larger than the value in y.

There are also models based on arrays but which derive from APL rather than FORTRAN. These include Mathematics of Arrays (MOA) [Mullin 1988], and Nial and Array Theory [More 1986].

Data-parallel languages simplify programming because operations that require loops in lower-level parallel languages can be written as single operations (which are also more revealing to the compiler since it does not have to try to infer the pattern intended by the programmer). With a sufficiently careful choice of data-parallel operations, some program transformation capability is often achieved. The natural mapping of data-parallel operations to architectures, at least for simple types, makes guaranteed performance, and also cost measures, possible.

4.2.3 *Static and Communication-Limited Structure.* The data-parallel languages of the previous section were developed primarily with program construction in mind. There is another set of similar languages whose inspiration was primarily architectural features. Because of these origins, they typically pay more attention to the amount of communication that takes place in computing each operation. Thus their thread structures are fixed, communication takes place at well-defined points, and the size of data involved in communications is small and fixed.

A wide variety of languages was developed whose basic operations were data-parallel list operations, inspired by the architecture of the Connection Machine 2. These often included a map operation, some form of reduction, perhaps using only a fixed set of operators, and later scans (parallel prefixes) and permutation operations. In approximately chronological order, these models are: scan [Blelloch 1987], multiprefix [Ranade 1989], paralations [Goldman 1989; Sabot 1989], the C* data-parallel language [Hatcher and Quinn 1991; Quinn and Hatcher 1990], the scan-vector model and NESL [Blelloch 1990, 1993, 1996; Blelloch and Sabot 1988, 1990; Blelloch and Greiner 1994], and CamlFlight [Hains and Foisy 1993]. As

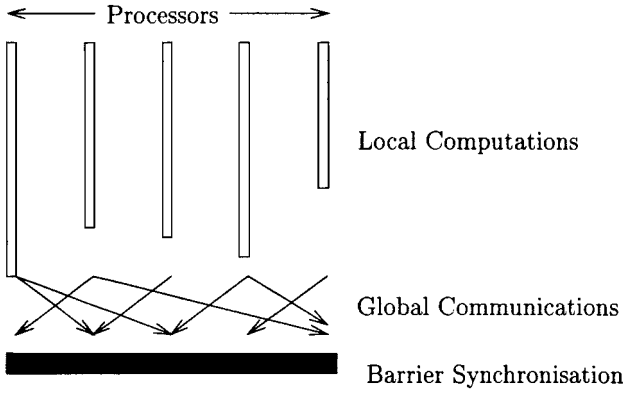

Figure 4. A BSP superstep.

for other data-parallel languages, these models are simple and fairly abstract. For instance, C* is an extension of the C language that incorporates features of the SIMD parallel model. In C*, data parallelism is implemented by defining data of a parallel kind. C* programs map variables of a particular data type, defined as parallel by the keyword poly, to separate processing elements. In this way, each processing element executes, in parallel, the same statement for each instance of the specified data type. At least on some architectures, data-parallel languages provide implementations with guaranteed performance and have accurate cost measures. Their weakness is that the choice of operations is made on the basis of what can be efficiently implemented, so that there is no basis for a formal software development methodology.

4.3 Decomposition Explicit

Models of this kind require abstract programs to specify the pieces into which they are to be divided, but the placement of these pieces on processors and the way in which they communicate need not be described so explicitly.

4.3.1 *Static Structure.* The only examples in this class are those that renounce locality, which ensures that placement does not matter to performance.

Bulk synchronous parallelism (BSP)[7] is a model in which interconnection network properties are captured by a few architectural parameters. A BSP abstract machine consists of a collection of p abstract processors, each with local memory, connected by an interconnection network whose only properties of interest are the time to do a barrier synchronization (l) and the rate at which continuous randomly addressed data can be delivered (g). These BSP parameters are determined experimentally for each parallel computer.

A BSP (abstract) program consists of p threads and is divided into *supersteps.* Each superstep consists of: a computation in each processor, using only locally held values; a global message transmission from each processor to any set of the others; and a barrier synchronization. At the end of a superstep, the results of global communications become visible in each processor's local environment. A superstep is shown in Figure 4. If the maximum local computation on a step takes time w and the maximum number of values sent by or received by any processor is h, then the total time for a superstep is given by

$$t = w + hg + l$$

[7] See McColl [1994a,b; 1993a], Skillicorn et al. [1996], and Valiant [1989; 1990a,b].

(where g and l are the network parameters), so that it is easy to determine the cost of a program. This time bound depends on randomizing the placement of threads and using randomized or adaptive routing to bound communication time.

Thus BSP programs must be decomposed into threads, but the placement of threads is then done automatically. Communication is implied by the placement of threads, and synchronization takes place across the whole program. The model is simple and fairly abstract, but lacks a software construction methodology. The cost measures give the real cost of a program on any architecture for which g and l are known.

The current implementation of BSP uses an SPMD library that can be called from C and FORTRAN. The library provides operations to put data into the local memory of a remote process, to get data from a remote process, and to synchronize. We illustrate with a small program to compute prefix sums:

```
int prefixsums(int x) {
   int i, left, right;
   bsp_pushregister(&left,sizeof(int));
   bsp_sync( );
   right = x;
   for (i=1;i<bsp_nprocs( );i*=2) {
      if (bsp_pid( )+i < bsp_nprocs( ))
      bsp_put(bsp_pid( )+i,&right,&left,
         0,sizeof(int));
      bsp_sync( );
      if (bsp_pid( )>=i) right = left +
         right;
   }
   bsp_popregister(&left);
   return right;
}
```

The bsp-pushregister and bsp-popregister calls are needed so that each process can refer to variables in remote processes by name, even though they might have been allocated in heap or stack storage.

Another related approach is LogP [Culler et al. 1993], which uses similar threads with local contexts, updated by global communications. However, LogP does not have an overall barrier synchronization. The LogP model is intended as an abstract model that can capture the technological reality of parallel computation. LogP models parallel computations using four parameters: the latency (L), overhead (o), bandwidth (g) of communication, and the number of processors (P). A set of programming examples has been designed with the LogP model and implemented on the CM-5 parallel machine to evaluate the model's usefulness. However, the LogP model is no more powerful than BSP [Bilardi et al. 1996], so BSP's simpler style is perhaps to be preferred.

4.4 Mapping Explicit

Models in this class require abstract programs to specify how programs are decomposed into pieces and how these pieces are placed, but they provide some abstraction for the communication actions among the pieces. The hardest part about describing communication is the necessity of labeling the two ends of each communication action to say that they belong together, and of ensuring that communication actions are properly matched. Given the number of communications in a large parallel program, this is a tedious burden for software developers. All of the models in this class try to reduce this burden by decoupling the ends of the communication from each other, by providing higher-level abstractions for patterns of communication, or by providing better ways of specifying communication.

4.4.1 *Dynamic Structure. Coordination languages* simplify communication by separating the computation aspects of programs from their communication aspects and providing a separate language in which to specify communication. This separation makes the computation and communication orthogonal to each other, so that a particular coordination style can be applied to any sequential language.

The best known example is Linda [Ahuja et al., 1994; Carriero 1987; Car-

riero and Gelernter 1988, 1993], which replaces point-to-point communication with a large shared pool into which data values are placed by processes and from which they are retrieved associatively. This shared pool is known as a *tuple space.* The Linda communication model contains three communication operations: *in,* which removes a tuple from tuple space, based on its arity and the values of some of its fields, filling in the remaining fields from the retrieved tuple; *read* (*rd*), which does the same except that it copies the tuple from tuple space; and *out,* which places a tuple in tuple space. For example, the *read* operation

```
rd("Canada", ?X, "USA")
```

searches the tuple space for tuples of three elements, first element "Canada," last element "USA," and middle element of the same type as variable *X*. Besides these three basic operations, Linda provides the eval(t) operation that implicitly creates a new process to evaluate the tuple and insert it in the tuple space.

The Linda operations decouple the send and receive parts of a communication—the "sending" thread does not know the "receiving" thread, not even if it exists. Although the model for finding tuples is associative matching, implementations typically compile these away, based on patterns visible at compile time. The Linda model requires programmers to manage the threads of a program, but reduces the burden imposed by managing communication. Unfortunately, a tuple space cannot necessarily be implemented with guaranteed performance, so that the model cannot provide cost measures—worse, Linda programs can deadlock. Another important issue is a software development methodology. To address this issue a high-level programming environment, called the Linda Program Builder (LPB), has been implemented to support the design and development of Linda programs [Ahmed 1994]. The LPB environment guides a user through program design, coding, monitoring, and execution of Linda software.

Nonmessage communication languages reduce the overheads of managing communication by disguising communication in ways that fit more naturally into threads. For example, ALMS [Arbeit et al. 1993; Peierls and Campbell 1991] treats message passing as if the communication channels were memory-mapped. Reference to certain message variables in different threads behaves like a message transfer from one to the others. PCN [Foster et al. 1992, 1991; Foster and Tuecke 1991] and Compositional C++ also hide communication by single-use variables. An attempt to read from one of these variables blocks the thread if a value has not already been placed in it by another thread. These approaches are similar to the use of full/empty bits on variables, an old idea coming back to prominence in multithreaded architectures.

In particular, the PCN (program composition notation) language is based on two simple concepts, concurrent composition and single-assignment variables. In PCN, single-assignment variables are called *definitional* variables. Concurrent composition allows parallel execution of statement blocks to be specified, without specifying, in the concurrent composition, how each of the pieces of the composition is mapped to processors. Processes that share a definitional variable can communicate with each other through it. For instance, in the parallel composition

```
{ || producer(X), consumer(X)}
```

the two processes producer and consumer can use X to communicate regardless of their location on the parallel computer. In PCN, the mapping of processes to processors is specified by the programmer either by annotating programs with location functions or by defining mappings of virtual to physical topologies.

The logical extension of mapping communication to memory is *virtual shared memory,* in which the abstraction pro-

vided to the program is of a single, shared address space, regardless of the real arrangement of memory. This requires remote memory references either to be compiled into messages or to be effected by messages at run-time. So far, results have not suggested that this approach is scalable, but it is an ongoing research area.[8]

Annotated functional languages make the compiler's job easier by allowing programmers to provide extra information about suitable ways to partition the computation into pieces and place them [Kelly 1989]. The same reduction rules apply, so that the communication and synchronization induced by this placement follow in the same way as in pure graph reduction.

An example of this kind of language is Paralf [Hudak 1986]. Paralf is a functional language based on lazy evaluation; that is, an expression is evaluated on demand. However, Paralf allows a user to control the evaluation order by explicit annotations. In Paralf, communication and synchronization are implicit, but it provides a mapping notation to specify which expressions are to be evaluated on which processor. An expression followed by the annotation $on proc is evaluated on the processor identified by proc. For example, the expression

```
(f(x) $on ($self+1)) * (h(x) $on ($self))
```

denotes the computation of the f(x) subexpression on a neighbor processor in parallel with the execution of h(x).

The *remote procedure call* (RPC) mechanism is an extension of the traditional procedure call. An RPC is a procedure call between two different processes, the caller and the receiver. When a process calls a remote procedure on another process, the receiver executes the code of the procedure and passes back to the caller the output parameters. Like rendezvous, RPC is a synchronous cooperation form. During the execution of the procedure, the caller is blocked and is reactivated by the arrival of the output parameters. Full synchronization of RPC might limit the exploitation of a high degree of parallelism among the processes that compose a concurrent program. In fact, when a process P calls a remote procedure r of a process T, the caller process P remains idle until the execution of r terminates, even if P could execute some other operation during the execution of r. To partially limit this effect, most new RPC-based systems use lightweight threads. Languages based on the remote procedure call mechanism are DP [Hansen 1978], Cedar [Swinehart et al. 1985], and Concurrent CLU [Cooper and Hamilton 1988].

4.4.2 *Static Structure.* *Graphical languages* simplify the description of communication by allowing it to be inserted graphically and at a higher, structured level. For example, the language Enterprise [Lobe et al. 1992; Szafron et al. 1991] classifies program units by type and generates some of the communication structure automatically based on type. The metaphor is of an office, with some program units communicating only through a "secretary," for example. Parsec [Feldcamp and Wagner 1993] allows program units to be connected using a set of predefined connection patterns. Code [Newton and Browne 1992] is a high-level dataflow language in which computations are connected together graphically and a firing rule and result-passing rule are associated with each computation. Decomposition in these models is still explicit, but communication is both more visible and simpler to describe. The particular communication patterns available are chosen for applicability reasons rather than efficiency, so performance is not guaranteed, nor are cost measures.

Coordination languages with contexts extend the Linda idea. One of the weaknesses of Linda is that it provides a single global tuple space and thus pre-

[8] See Chin and McColl [1994], Li and Hudak [1989], Raina [1990], and Wilkinson et al. [1992].

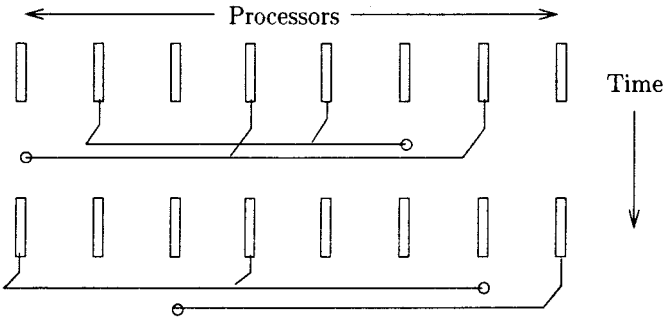

Figure 5. A communication skeleton.

vents modular development of software. A model that extends Linda by including ideas from Occam is the language Ease [Zenith 1991a,b,c, 1992]. Ease programs have multiple tuple spaces, which are called *contexts* and may be visible to only some threads. Because those threads that access a particular context are known, contexts take on some of the properties of Occam-like channels. Threads read and write data to contexts as if they were Linda tuple spaces, with associative matching for reads and inputs. However, they can also use a second set of primitives that move data to a context and relinquish ownership of the data, or retrieve data from a context and remove them from the context. Such operations can use pass-by-reference since they guarantee that the data will be referenced by only one thread at a time. Ease has many of the same properties as Linda, but makes it easier to build implementations with guaranteed performance. Ease also helps with decomposition by allowing process structuring in the style of Occam.

Another related language is ISETL-Linda [Douglas et al. 1995], which is an extension to the SETL paradigm of computing with sets as aggregates. It adds Linda-style tuple spaces as a data type and treats them as first-class objects. To put it another way, ISETL-Linda resembles a data-parallel language in which bags are a data type and associative matching is a selection operation on bags. Thus ISETL-Linda can be seen as extending SETL-like languages with a new data type or as extending Linda-like languages with skeletons.

A language of the same kind derived from FORTRAN is Opus [Mehrotra and Haines 1994], which has both task and data parallelism, but communication is mediated by shared data abstractions. These are autonomous objects that are visible to any subset of tasks but are internally sequential; that is, only one method within each object is active at a time. They are a kind of generalization of monitors.

4.4.3 *Static and Communication-Limited Structure. Communication skeletons* extend the idea of prestructured building blocks to communication [Skillicorn 1996]. A communication skeleton is an interleaving of computation steps, which consist of independent local computations, and communication steps, which consist of fixed patterns of communication in an abstract topology. These patterns are collections of edge-disjoint paths in an abstract topology, each of which functions as a broadcast channel. Figure 5 shows a communication skeleton using two computation steps, interleaved with two different communication patterns. This model is a blend of ideas from BSP and from algorithmic skeletons, together with concepts such as adaptive routing and broadcast that are supported by new architectural designs. The model

is moderately architecture-independent because communication skeletons can be built assuming a weak topology target and then embedding results can be used to build implementations for targets with richer interconnection topologies. It can provide guaranteed performance and does have cost measures.

4.5 Communication Explicit

Models in this class require communication to be explicit, but reduce some of the burden of synchronization associated with it. Usually this is done by having an asynchronous semantics: messages are delivered but the sender cannot depend on the time of delivery, and delivery of multiple messages may be out of order.

4.5.1 *Dynamic Structure. Process nets* resemble dataflow in the sense that operations are independent entities that respond to the arrival of data by computing and possibly sending on other data. The primary differences are that the operations individually decide what their response to data arrival will be, and individually decide to change their behavior. They therefore lack the global state that exists, at least implicitly, in dataflow computations.

The most important model in this class is actors [Agha 1986; Baude 1991; Baude and Vidal-Naquet 1991]. Actor systems consist of collections of objects called actors, each of which has an incoming message queue. An actor repeatedly executes the sequence: read the next incoming message, send messages to other actors whose identity it knows, and define a new behavior that governs its response to the next message. Names of actors are first-class objects and may be passed around in messages. Messages are delivered asynchronously and unordered. However, neither guaranteed performance nor cost measures for actors is possible because the total communication in an actor program may not be bounded. Moreover, because the Actor model is highly distributed, compilers must serialize execution to achieve execution efficiency on conventional processors. One of the most effective compiler transformations is to eliminate creation of some types of actors and to change messages sent to actors on the same processor into function calls [Kim and Agha 1995]. Thus the actual cost of executing an actor program is indeterminate without a specification of how the actors are mapped to processors and threads.

A different kind of process net is provided by the language Darwin [Eisenbach and Patterson 1993; Radestock and Eisenbach 1994], which is based on the π-calculus. The language provides a semantically well founded configuration subset for specifying how ordinary processes are connected and how they communicate. It is more like a process net than a configuration language since the binding of the semantics of communication to connections is dynamic.

One of the weaknesses of the actor model is that each actor processes its message queue sequentially and this can lead to bottlenecks. Two extensions of the model that address this issue have been proposed: Concurrent Aggregates [Chien 1991; Chien and Dally 1990] and ActorSpace [Agha and Callsen 1993]. Concurrent Aggregates (CA) is an object-oriented language well suited to exploit parallelism on fine-grain massively parallel computers. In it, unnecessary sources of serialization have been avoided. An aggregate in CA is a homogeneous collection of objects (called representatives) that are grouped together and referenced by a single aggregate name. Each aggregate is multi-access, so it may receive several messages simultaneously, unlike other object-oriented languages such as the actor model and ABCL/1. Concurrent Aggregates incorporates many other innovative features such as delegation, intra-aggregate addressing, first-class messages, and user continuations. Delegation allows the behavior of an aggregate to be constructed incrementally from that of many other aggregates. In-

tra-aggregate addressing makes cooperation among parts of an aggregate possible.

The ActorSpace model extends the actor model to avoid unnecessary serializations. An actor space is a computationally passive container of actors that acts as a context for matching patterns. In fact, the ActorSpace model uses a communication model based on destination patterns. Patterns are matched against listed attributes of actors and actor spaces that are visible in the actor space. Messages can be sent to one arbitrary member of a group or broadcast to all members of a group defined by a pattern.

We now turn to *external OO models.* Actors are regarded as existing regardless of their communication status. A superficially similar approach, but one which is quite different underneath, is to extend sequential object-oriented languages so that more than one thread is active at a time. The first way to do this, which we have called external object-orientation, is to allow multiple threads of control at the highest level of the language. Objects retain their traditional role of collecting code that logically belongs together. Object state can now act as a communication mechanism since it can be altered by a method executed by one thread and observed by a method executed as part of another thread. The second approach, which we call internal object orientation, encapsulates parallelism within the methods of an object, but the top level of the language appears sequential. It is thus closely related to data-parallelism. We return to this second case later; here we concentrate on external OO models and languages.

Some interesting external object-based models are ABCL/1 [Yonezawa et al. 1987], ABCL/R [Watanabe et al. 1988], POOL-T [America 1987], EPL [Black et al. 1984], Emerald [Black et al. 1987], and Concurrent Smalltalk [Yokote et al. 1987]. In these languages, parallelism is based on assigning a thread to each object and using asynchronous message passing to reduce blocking. EPL is an object-based language that influenced the design of Emerald. In Emerald, all entities are objects that can be passive (data) or active. Each object consists of four parts: a name, a representation (data), a set of operations, and an optional process that can run in parallel with invocations of object operations. Active objects in Emerald can be moved from one processor to another. Such a move can be initiated by the compiler or by the programmer using simple language constructs. The primary design principles of ABCL/1 (an object-based concurrent language) are practicality and clear semantics of message passing. Three types of message passing are defined: *past, now,* and *future.* The *now* mode operates synchronously, whereas the *past* and *future* modes operate asynchronously. For each of the three message-passing mechanisms, ABCL/1 provides two distinct modes, *ordinary* and *express,* which correspond to two different message queues. To give an example, *past* type message passing in *ordinary* and *express* modes is, respectively,

[Obj <= msg] and [Obj <<= msg],

where Obj is the receiver object and msg is the sent message.

In ABCL/1, independent objects can execute in parallel but, as in the actor model, messages are processed serially within an object. Although message passing in ABCL/1 programs may take place concurrently, no more than one message can arrive at the same object simultaneously. This limits the parallelism among objects. An extension of ABCL/1 is ABCL/R where reflection has been introduced.

Objects and processes exploit parallelism in external object-oriented languages in two principal ways: using the objects as the unit of parallelism by assigning one or more processes to each object, or defining processes as components of the language. In the first approach, languages are based on active objects and each process is bound to a

particular object for which it is created. In the latter approach, two different kinds of entities are defined, objects and processes. A process is not bound to a single object, but is used to perform all the operations required to satisfy an action. Therefore, a process can execute within many objects, changing its address space when an invocation to another object is made. Whereas the object-oriented models discussed before use the first approach, systems like Argus [Liskov 1987] and Presto [Bershad et al. 1988] use the second approach. In this case, languages provide mechanisms for creating and controlling multiple processes external to the object structure.

Argus supports coarse-grain and medium-grain objects and dynamic process creation. In Argus, *guardians* contain data objects and procedures. A guardian instance is created dynamically by a call to a creator procedure and can be explicitly mapped to a processor:

```
guardianType$creator(parameters)
    processor X.
```

The expense of dynamic process creation is reduced by maintaining a pool of unused processes. A new group of processes is created only when the pool is emptied. In these models, parallelism is implemented on top of the object organization and explicit constructs are defined to ensure object integrity. It is worth noticing that these models were developed for programming coarse-grain programs in distributed systems, not tightly coupled, fine-grain parallel machines.

Active messages is an approach that decouples communication and synchronization by treating messages as active objects rather than passive data. Essentially a message consists of two parts: a data part and a code part that executes on the receiving processor when the message has been transmitted. Thus a message changes into a process when it arrives at its destination. There is therefore no synchronization with any process at the receiving end, and hence a message "send" does not have a corresponding "receive." This approach is used in the Movie system [Faigle et al. 1993] and the language environments for the J-machine [Chien and Dally 1990; Dally et al. 1992; Noakes and Dally 1990].

4.5.2 *Static Structure. Internal object-oriented languages* are those in which parallelism occurs within single methods. The Mentat Programming Language (MPL) is a parallel object-oriented system designed for developing architecture-independent parallel applications. The Mentat system integrates a data-driven computation model with the object-oriented paradigm. The data-driven model supports a high degree of parallelism, whereas the object-oriented paradigm hides much of the parallel environment from the user. MPL is an extension of C++ that supports both intra- and interobject parallelism. The compiler and the run-time support of the language are designed to achieve high performance. The language constructs are mapped to the macrodataflow model that is the computation model underlying Mentat, which is a medium-grain data-driven model in which programs are represented as directed graphs. The vertices of the program graphs are computation elements that perform some function. The edges model data dependencies between the computation elements. The compiler generates code to construct and execute data dependency graphs. Thus interobject parallelism in Mentat is largely transparent to the programmer. For example, suppose that A, B, C, D, E, and M are vectors and consider the statements:

```
A = vect_op.add (B,C);
M = vect_op.add (A, vect_op.add
    (D,E));
```

The Mentat compiler and run-time system detect that the two additions (B + C) and (D + E) are not data-dependent on one another and can be executed in parallel. Then the result is automati-

cally forwarded to the final addition. That result is forwarded to the caller and associated with *M*. In this approach, the programmer makes granularity and partitioning decisions using Mentat class definition constructs, and the compiler and the run-time support manage communication and synchronization [Grimshaw 1993a,b, 1991; Grimshaw et al. 1991a, b].

4.5.3 *Static and Communication-Limited Structure. Systolic arrays* are gridlike architectures of processing elements or cells that process data in an *n*-dimensional pipelined fashion. By analogy with the systolic dynamics of the heart, systolic computers perform operations in a rhythmic, incremental, and repetitive manner [Kung 1982] and pass data to neighbor cells along one or more directions. In particular, each computing element computes an incremental result and the systolic computer derives the final result by interpreting the incremental results from the entire array. A parallel program for a systolic array must specify how data are mapped onto the systolic elements and the data flow through the elements. High-level programmable arrays allow the development of systolic algorithms by the definition of inter- and intracell parallelism and cell-to-cell data communication. Clearly, the principle of rhythmic communication distinguishes systolic arrays from other parallel computers. However, even if high-level programmability of systolic arrays creates a more flexible systolic architecture, penalties can occur because of complexity and possible slowing of execution due to the problem of data availability. High-level programming models are necessary for promoting widespread use of programmable systolic arrays. One example is the language Alpha [de Dinechin et al. 1995], where programs are expressed as recurrence equations. These are transformed into systolic form by regarding the data dependencies as defining an affine or vector space that can be geometrically transformed.

4.6 Everything Explicit

The next category of models is those that do not hide much detail of decomposition and communication. Most of the first-generation models of parallel computation are at this level, designed for a single architecture style, explicitly managed.

4.6.1 *Dynamic Structure.* Most models provide a particular paradigm for handling partitioning, mapping, and communication. A few models have tried to be general enough to provide multiple paradigms, for example, Pi [Dally and Wills 1989; Wills 1990], by providing sets of primitives for each style of communication. Such models can have guaranteed performance and cost measures, but they make the task of software construction difficult because of the amount of detail that must be given about a computation. Another set of models of the same general kind are the programming languages Orca [Bal et al. 1990] and SR [Andrews and Olsson 1993; Andrews et al. 1988]. Orca is an object-based language that uses shared data-objects for interprocess communication. The Orca system is a hierarchically structured set of abstractions. At the lowest level, reliable broadcast is the basic primitive so that writes to a replicated structure can take effect rapidly throughout a system. At the next level of abstraction, shared data are encapsulated in passive objects that are replicated throughout the system. Parallelism in Orca is expressed through explicit process creation. A new process can be created through the fork statement

```
fork proc_name (params)[on
    (cpu_number)].
```

The on part optionally specifies the processor on which to run the child process. The parameters specify the shared data objects used for communication between the parent and the child processes.

Synchronizing Resources (SR) is based on the *resource* concept. A re-

source is a module that can contain several processes. A resource is dynamically created by the *create* command and its processes communicate by the use of semaphores. Processes belonging to different resources communicate using only a restricted set of operations explicitly defined in the program as procedures.

There is a much larger set of models or programming languages based on a single communication paradigm. We consider three paradigms: message passing, shared memory, and rendezvous.

Message passing is the basic communication technology provided on distributed-memory MIMD architectures, and so message-passing systems are available for all such machines. The interfaces are low-level, using *send*s and *receive*s to specify the message to be exchanged, process identifier, and address.

It was quickly realized that message-passing systems look much the same for any distributed-memory architecture, so it was natural to build standard interfaces to improve the portability of message-passing programs. The most recent example of this is MPI (message passing interface) [Dongarra et al. 1995; Message Passing Interface Forum 1993], which provides a rich set of messaging primitives, including point-to-point communication, broadcasting, and the ability to collect processes in groups and communicate only within each group. MPI aims to become the standard message-passing interface for parallel applications and libraries [Dongarra et al. 1996]. Point-to-point communications are based on *send* and *receive* primitives

```
MPI_Send (buf, bufsize, datatype,
  dest, ..... )
MPI_Recv (buf, bufsize, datatype,
  source, ..... ).
```

Moreover, MPI provides primitives for collective communication and synchronization such as MPI_Barrier, MPI_Bcast, and MPI_Gather. In its first version, MPI does not make provision for process creation, but in the MPI2 version additional features for active messages, process startup, and dynamic process creation are provided.

More architecture-independent message-passing models have been developed to allow transparent use of networks of workstations. In principle, such networks have much unused compute power to be exploited. In practice, the large latencies involved in communicating among workstations make them low-performance parallel computers. Models for workstation message-passing include systems such as PVM,[9] Parmacs [Hempel 1991; Hempel et al. 1992], and p4 [Butler and Lusk 1992]. Such models are exactly the same as inter-multiprocessor message-passing systems, except that they typically have much larger-grain processes to help conceal the latency, and they must address heterogeneity of the processors. For example, PVM (parallel virtual machine) has gained widespread acceptance as a programming toolkit for heterogeneous distributed computing. It provides a set of primitives for process creation and communication that can be incorporated into existing procedural languages in order to implement parallel programs. In PVM, a process is created by the pvm_spawn( ) call. For instance, the statement

```
proc_num = pvm_spawn ("progr1",
  NULL, PVMTaskDefault, 0, n_proc)
```

spawns n_proc copies of the program progr1. The actual number of processes started is returned to proc_num. Communication between two processes can be implemented by the primitives

```
pvm_send (proc_id, msg) and pvm_rec
  (proc_id, msg).
```

For group communication and synchronization the functions pvm_bcast( ),

[9] See Beguelin et al. [1993; 1995; 1994], Geist [1993], and Sunderam [1990; 1992].

pvm_mcast( ), pvm_barrier( ) can be used.

Using PVM and similar models, programmers must do all of the decomposition, placement, and communication explicitly. This is further complicated by the need to deal with several different operating systems to communicate this information to the messaging software. Such models may become more useful with the increasing use of optical interconnection and high-performance networks for connecting workstations.

Shared-memory communication is a natural extension of techniques used in operating systems, but multiprogramming is replaced by true multiprocessing. Models for this paradigm are therefore well understood. Some aspects change in the parallel setting. On a single processor it is never sensible to busy-wait for a message, since this denies the processor to other processes; it might be the best strategy on a parallel computer since it avoids the overhead of two context switches. Shared-memory parallel computers typically provide communication using standard paradigms such as shared variables and semaphores. This model of computation is an attractive one since issues of decomposition and mapping are not important. However, it is closely linked to a single style of architecture, so that shared-memory programs are not portable.

An important shared-memory programming language is Java [Lea 1996], which has become popular because of its connection with platform-independent software delivery on the Web. Java is thread-based, and allows threads to communicate and synchronize using *condition variables.* Such shared variables are accessed from within synchronized methods. A critical section enclosing the text of the methods is automatically generated. These critical sections are rather misleadingly called monitors. However, notify and wait operations must be explicitly invoked within such sections, rather than being automatically associated with entry and exit. There are many other thread packages available providing lightweight processes with shared-memory communication.[10]

Rendezvous-based programming models are distributed-memory paradigms using a particular cooperation mechanism. In the rendezvous communication model, an interaction between two processes A and B takes place when A calls an *entry* of B and B executes an *accept* for that entry. An entry call is similar to a procedure call and an accept statement for the entry contains a list of statements to be executed when the entry is called. The best known parallel programming languages based on rendezvous cooperation are Ada [Mundie and Fisher 1986] and Concurrent C [Gehani and Roome 1986]. Ada was designed on behalf of the US Department of Defense mainly to program real-time applications both on sequential and parallel distributed computers. Parallelism in the Ada language is based on processes called *tasks.* A task can be created explicitly or can be statically declared. In this latter case, a task is activated when the block containing its declaration is entered. Tasks are composed of a specification part and a body. As discussed before, this mechanism is based on entry declarations, entry calls, and accept statements. Entry declarations are allowed only in the specification part of a task. Accept statements for the entries appear in the body of a task. For example, the following accept statement executes the operation when the entry *square* is called.

```
accept SQUARE (X: INTEGER; Y: out
  INTEGER) do
  Y := X * X;
end;
```

Other important features of Ada for parallel programming are the use of the *select* statement, which is similar to the

[10] See Buhr and Stroobosscher [1990], Buhr et al. [1991], Faust and Levy [1990], and Mukherjee et al. [1994].

CSP ALT command for expressing nondeterminism, and the exception-handling mechanism for dealing with software failures. On the other hand, Ada does not address the problem of mapping tasks onto multiple processors and does not provide conditions to be associated with the entry declarations, except for some special cases such as protected objects. Recent surveys of such models can be found in Bal et al. [1989] and Gottlieb et al. [1983a,b].

4.6.2 *Static Structure.* Most low-level models allow dynamic process creation and communication. An exception is Occam [Jones and Goldsmith 1988], in which the process structure is fixed and communication takes place along synchronous channels. Occam programs are constructed from a small number of primitive constructs: assignment, input (?), and output (!). To design complex parallel processes, primitive constructs can be combined using the parallel constructor

```
PAR
  Proc1
  Proc2
```

The two processes are executed in parallel and the PAR constructor terminates only after all of its components have terminated. An alternative constructor (ALT) implements nondeterminism. It waits for input from a number of channels and then executes the corresponding component process. For example, the following code

```
ALT
  request ? data
    DataProc
  exec ? oper
    ExecProc
```

waits to get a data request or an operation request. The process corresponding to the selected guard is executed.

Occam has a strong semantic foundation in CSP [Hoare 1985], so that software development by transformation is possible. However, it is so low-level that this development process is only practical for small or critical applications.

4.7 PRAM

A final model that must be considered is the PRAM model [Karp and Ramachandran 1990], which is the basic model for much theoretical analysis of parallel computation. The PRAM abstract machine consists of a set of processors, capable of executing independent programs but doing so synchronously, connected to a shared memory. All processors can access any location in unit time, but they are forbidden to access the same location on the same step.

The PRAM model requires detailed descriptions of computations, giving the code for each processor and ensuring that memory conflict is avoided. The unit-time memory-access part of the cost model cannot be satisfied by any scalable real machine, so the cost measures of the PRAM model are not accurate. Nor can they be made accurate in any uniform way, because the real cost of accessing memory for an algorithm depends on the total number of accesses and the pattern in which they occur. One attempt to provide some abstraction from the PRAM is the language FORK [Kessler and Seidl 1995].

A good overview of models aimed at particular architectures can be found in McColl [1993b].

5. SUMMARY

We have presented an overview of parallel programming models and languages, using a set of six criteria that an ideal model should satisfy. Four of the criteria relate to the need to use the model as a target for software development. They are: ease of programming and the existence of a methodology for constructing software that handles issues such as correctness, independence from particular architectures, and simplicity and abstractness. The remaining criteria address the need for execution of the model on real parallel machines:

guaranteed performance and the existence of costs that can be inferred from the program. Together these ensure predictable performance for programs.

We have assessed models by how well they satisfy these criteria, dividing them into six classes, ranging from the most abstract, which generally satisfy software development criteria but not predictable performance criteria, to very concrete models, which provide predictable performance but make it hard to construct software.

The models we have described represent an extremely wide variety of approaches at many different levels. Overall, some interesting trends are visible.

—Work on low-level models, in which the description of computations is completely explicit, has diminished significantly. We regard this as a good thing, since it shows that an awareness of the importance of abstraction has spread beyond the research community.

—There is a concentration on models in the middle range of abstraction, with a great deal of ingenuity being applied to concealing aspects of parallel computations while struggling to retain maximum expressiveness. This is also a good thing, since tradeoffs among expressiveness, software development complexity, and run-time efficiency are subtle. Presumably a blend of theoretical analysis and practical experimentation is the most likely road to success, and this strategy is being applied.

—There are some very abstract models that also provide predictable and useful performance on a range of parallel architectures. Their existence raises the hope that models satisfying all of the properties with which we began can eventually be constructed.

These trends show that parallel programming models are leaving low-level approaches and moving towards more abstract approaches, in which languages and tools simplify the task of designers and programmers. At the same time these trends provide for more robust parallel software with predictable performance.

This scenario brings many benefits for parallel software development. Models, languages, and tools represent an intermediate level between users and parallel architectures and allow the simple and effective utilization of parallel computation in many application areas. The availability of models and languages that abstract from architecture complexity has a significant impact on the parallel software development process and therefore on the widespread use of parallel computing systems.

Thus we can hope that, within a few years, there will be models that are easy to program, providing at least moderate abstraction, that can be used with a wide range of parallel computers, making portability a standard feature of parallel programming, that are easy to understand, and that can be executed with predictably good performance. It will take longer for software development methods to come into general use, but that should be no surprise because we are still struggling with software development for sequential programming. Computing costs for programs is possible for any model with predictable performance, but integrating such costs into software development in a useful way is much more difficult.

ACKNOWLEDGMENTS

We are grateful to Dave Dove and Luigi Palopoli for their comments on a draft of this article and to the anonymous referees for their helpful comments.

REFERENCES

AGHA, G. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA.

AGHA, G. AND CALLSEN, C. J. 1993. ActorSpace: An open distributed programming paradigm. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (May), 23–32.

AHMED, S., CARRIERO, N., AND GELERNTER, D.

1994. A program building tool for parallel applications. In *DIMACS Workshop on Specification of Parallel Algorithms* (Princeton University, May), 161–178.

AHUJA, S., CARRIERO, N., GELERNTER, D., AND KRISHNASWAMY, V. 1988. Matching languages and hardware for parallel computation in the Linda machine. *IEEE Trans. Comput. 37,* 8 (August), 921–929.

AMERICA, P. 1987. POOL-T: A parallel object-oriented language. In *Object-Oriented Concurrent Programming,* A. Yonezawa et al., Eds., MIT Press, Cambridge, MA, 199–220.

ANDRÉ, F. AND THOMAS, H. 1989. The Pandore System. In *IFIP Working Conference on Decentralized Systems* (Dec.) poster presentation.

ANDRÉ, F., MAHEO, Y., LE FUR, M., AND PAZAT, J.-L. 1994. The Pandore compiler: Overview and experimental results. Tech. Rep. PI-869, IRISA, October.

ANDREWS, G. R. AND OLSSON, R. A. 1993. *The SR Programming Language.* Benjamin/Cummings, Redwood City, CA.

ANDREWS, G. R., OLSSON, R. A., COFFIN, M. A., ELSHOFF, I., NILSEN, K., PURDIN, T., AND TOWNSEND, G. 1988. An overview of the SR language and implementation. *ACM Trans. Program. Lang. Syst. 10,* 1 (Jan.), 51–86.

ARBEIT, B., CAMPBELL, G., COPPINGER, R., PEIERLS, R., AND STAMPF, D. 1993. The ALMS system: Implementation and performance. Brookhaven National Laboratory, internal report.

BACK, R. J. R. 1989a. A method for refining atomicity in parallel algorithms. In *PARLE89 Parallel Architectures and Languages Europe* (June) LNCS 366, Springer-Verlag, 199–216.

BACK, R. J. R. 1989b. Refinement calculus part II: Parallel and reactive programs. Tech. Rep. 93, Åbo Akademi, Departments of Computer Science and Mathematics, SF-20500 Åbo, Finland.

BACK, R. J. R. AND SERE, K. 1989. Stepwise refinement of action systems. In *Mathematics of Program Construction,* LNCS 375, June, Springer-Verlag, 115–138.

BACK, R. J. R. AND SERE, K. 1990. Deriving an Occam implementation of action systems. Tech. Rep. 99, Åbo Akademi, Departments of Computer Science and Mathematics, SF-20500 Åbo, Finland.

BAIARDI, F., DANELUTTO, M., JAZAYERI, M., PELAGATTI, S., AND VANNESCHI, M. 1991. Architectural models and design methodologies for general-purpose highly-parallel computers. In *IEEE CompEuro 91—Advanced Computer Technology, Reliable Systems and Applications* (May).

BAL, H. E., STEINER, J. G., AND TANENBAUM, A. S. 1989. Programming languages for distributed computing systems. *Comput. Surv. 21,* 3 (Sept.), 261–322.

BAL, H. E., TANENBAUM, A. S., AND KAASHOEK, M. F. 1990. Orca: A language for distributed processing. *ACM SIGPLAN Not. 25* 5 (May), 17–24.

BANÂTRE, J. P. AND LE MÉTAYER, D. 1991. Introduction to Gamma. In *Research Directions in High-Level Parallel Programming Languages,* J. P. Banâtre and D. Le Métayer, Eds., LNCS 574, June, Springer-Verlag, 197–202.

BANGER, C. 1992. Arrays with categorical type constructors. In *ATABLE'92 Proceedings of a Workshop on Arrays* (June), 105–121.

BANGER, C. R. 1995. Construction of multidimensional arrays as categorical data types. Ph.D. Thesis, Queen's University, Kingston, Canada.

BAUDE, F. 1991. Utilisation du paradigme acteur pour le calcul parallèle. Ph.D. thesis, Université de Paris-Sud.

BAUDE, F. AND VIDAL-NAQUET, G. 1991. Actors as a parallel programming model. In *Proceedings of Eighth Symposium on Theoretical Aspects of Computer Science.* LNCS 480, Springer-Verlag.

BEGUELIN, A., DONGARRA, J., GEIST, A., MANCHEK, R., MOORE, K., AND SUNDERAM, V. 1993. PVM and HeNCE: Tools for heterogeneous network computing. In *Software for Parallel Computation, NATO ASI Series F.,* Vol. 106 J. S. Kowalik and L. Grandinetti, Eds. Springer-Verlag, 91–99.

BEGUELIN, A., DONGARRA, J., GEIST, A., MANCHEK, R., AND SUNDERAM, V. 1995. Recent enhancements to PVM. *Int. J. Supercomput. Appl. High Perform. Comput.*

BEGUELIN, A., DONGARRA, J. J., GEIST, G. A., MANCHEK, R., AND SUNDERAM, V. S. PVM software system and documentation. Email to netlib@ornl.gov.

BEGUELIN, A., DONGARRA, J., GEIST, A., MANCHEK, R., OTTO, S., AND WALPOLE, J. 1994. PVM: Experiences, current status and future direction. Tech. Rep. CS/E 94-015, Oregon Graduate Institute CS.

BERSHAD, B. N., LAZOWSKA, E., AND LEVY, H. 1988. Presto: A system for object-oriented parallel programming. *Softw. Pract. Exper.* (August).

BICKFORD, M. 1994. Composable specifications for asynchronous systems using UNITY. In *Proceedings International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Nov.), 216–227.

BILARDI, G., HERLEY, K. T., PIETRACAPRINA, A., PUCCI, G., AND SPIRAKIS, P. 1996. BSP vs. LogP. In *Proceedings of the Eighth Annual Symposium on Parallel Algorithms and Architectures* (June), 25–32.

BIRD, R. S. 1987. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design,* M. Broy, Ed., Springer-Verlag, 3–42.

BLACK, A., HUTCHINSON, N., JUL, E., LEVY, H., AND CARTER, L. 1987. Distribution and abstract types in Emerald. *IEEE Trans. Softw. Eng. 13,* 1 (Jan.), 65–76.

BLACK, A. ET AL. 1984. EPL programmers' guide. University of Washington, Seattle, June.

BLELLOCH, G. 1987. Scans as primitive parallel operations. In *Proceedings of the International Conference on Parallel Processing* (August), 355–362.

BLELLOCH, G. E. 1990. *Vector Models for Data-Parallel Computing.* MIT Press, Cambridge, MA.

BLELLOCH, G. E. 1993. NESL: A nested data-parallel language. Tech. Rep. CMU-CS-93-129, Carnegie Mellon University, April.

BLELLOCH, G. E. 1996. Programming parallel algorithms. *Commun. ACM 37,* 3 (March).

BLELLOCH, G. AND GREINER, J. 1994. A parallel complexity model for functional languages. Tech. Rep. CMU-CS-94-196, School of Computer Science, Carnegie Mellon University, October.

BLELLOCH, G. E. AND SABOT, G. W. 1988. Compiling collection-oriented languages onto massively parallel computers. In *Proceedings of the Second Symposium on the Frontiers of Massively Parallel Computation,* 575–585.

BLELLOCH, G. E. AND SABOT, G. W. 1990. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.,* 119–134.

BRIAT, J., FAVRE, M., GEYER, C., AND CHASSIN DE KERGOMMEAUX, J. 1991. Scheduling of OR-parallel Prolog on a scalable reconfigurable distributed memory multiprocessor. In *Proceedings of PARLE 91, Springer Lecture Notes in Computer Science 506,* Springer-Verlag, 385–402.

BUHR, P. A. AND STROOBOSSCHER, R. A. 1990. The μSystem: Providing light-weight concurrency on shared-memory multiprocessor computers running UNIX. *Softw. Pract. Exper. 20,* 9 (Sept.), 929–963.

BUHR, P. A., MACDONALD, H. I., AND STROOBOSSCHER, R. A. 1991. μSystem reference manual, version 4.3.3. Tech. Rep., Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, March.

BURKHART, H., FALCÓ KORN, C., GUTZWILLER, S., OHNACKER, P., AND WASER, S. 1993. BACS: Basel algorithm classification scheme. Tech. Rep. 93-3, Institut für Informatik der Universität Basel, March.

BUTLER, R. AND LUSK, E. 1992. User's guide to the p4 programming system. Tech. Rep. ANL-92/17, Argonne National Laboratory, Mathematics and Computer Science Division, October.

CANNATARO, M., DI GREGORIO, S., RONGO, R., SPATARO, W., SPEZZANO, G., AND TALIA, D. 1995. A parallel cellular automata environment on multicomputers for computational science. *Parallel Comput. 21,* 5, 803–824.

CANNATARO, M., SPEZZANO, G., AND TALIA, D. 1991. A parallel logic system on a multicomputer architecture. In *Fut. Gen. Comput. Syst. 6,* 317–331.

CARRIERO, N. 1987. Implementation of tuple space machines. Tech. Rep. YALEU/DCS/RR-567, Department of Computer Science, Yale University, December.

CARRIERO, N. AND GELERNTER, D. 1988. Application experience with Linda. In *ACM/SIGPLAN Symposium on Parallel Programming* (July), Vol. 23, 173–187.

CARRIERO, N. AND GELERNTER, D. 1993. Learning from our success. In *Software for Parallel Computation,* NATO ASI Series F, Vol. 106 J. S. Kowalik and L. Grandinetti, Eds. Springer-Verlag, 37–45.

CHANDY, K. M. AND MISRA, J. 1988. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA.

CHASSIN DE KERGOMMEAUX, J. AND CODOGNET, P. 1994. Parallel logic programming systems. *ACM Comput. Surv. 26,* 3, 295–336.

CHEN, M., CHOO, Y.-I., AND LI, J. 1991. Crystal: Theory and pragmatics of generating efficient parallel code. In *Parallel Functional Languages and Compilers,* B. K. Szymanski, Ed., ACM Press Frontier Series, New York, 255–308.

CHIEN, A. A. 1991. Concurrent aggregates: Using multiple-access data abstractions to manage complexity in concurrent programs. *OOPS Messenger 2,* 2 (April), 31–36.

CHIEN, A. A. AND DALLY, W. J. 1990. Concurrent Aggregates. In *Second SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Feb.), 187–196.

CHIN, A. AND MCCOLL, W. F. 1994. Virtual shared memory: Algorithms and complexity. *Inf. Comput. 113,* 2 (Sept.), 199–219.

COLE, M. 1989. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing, Pitman, London.

COLE, M. 1990. Towards fully local multicomputer implementations of functional programs. Tech. Rep. CS90/R7, Department of Computing Science, University of Glasgow, January.

COLE, M. 1992. Writing parallel programs in non-parallel languages. In *Software for Parallel Computers: Exploiting Parallelism through Software Environments, Tools, Algorithms*

and Application Libraries, R. Perrot, Ed., Chapman and Hall, London, 315–325.

COLE, M. 1994. Parallel programming, list homomorphisms and the maximum segment sum problem. In *Parallel Computing: Trends and Applications,* G. R. Joubert, Ed., North-Holland, 489–492.

CONERY, J. S. 1987. *Parallel Execution of Logic Programs.* Kluwer Academic.

COOPER, R. AND HAMILTON, K. G. 1988. Preserving abstraction in concurrent programming. *IEEE Trans. Softw. Eng. SE-14,* 2, 258–263.

CREVEUIL, C. 1991. Implementation of Gamma on the Connection Machine. In *Research Directions in High-Level Parallel Programming Languages,* J. P. Banâtre and D. Le Métayer, Eds., LNCS 574, June, Springer-Verlag, 219–230.

CULLER, D., KARP, R., PATTERSON, D., SAHAY, A., SCHAUSER, K. E., SANTOS, E., SUBRAMONIAN, R., AND VON EICKEN, T. 1993. LogP: Toward a realistic model of parallel computation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (May).

DALLY, W. J. AND WILLS, D. S. 1989. Universal mechanisms for concurrency. In *PARLE '89, Parallel Architectures and Languages Europe* (June), LNCS 365, Springer-Verlag, 19–33.

DALLY, W. J., FISKE, J. A. S., KEEN, J. S., LETHIN, R. A., NOAKES, M. D., NUTH, P. R., DAVISON, R. E., AND FYLER, G. A. 1992. The message-driven processor. *IEEE Micro* (April), 23–39.

DANELUTTO, M., DI MEGLIO, R., ORLANDO, S., PELAGATTI, S., AND VANNESCHI, M. 1991a. A methodology for the development and the support of massively parallel programs. *Fut. Gen. Comput. Syst.* Also appears as "The P^3L language: An introduction," Hewlett-Packard Rep. HPL-PSC-91-29, December.

DANELUTTO, M., DI MEGLIO, R., PELAGATTI, S., AND VANNESCHI, M. 1990. High level language constructs for massively parallel computing. Tech. Rep., Hewlett-Packard Pisa Science Center, HPL-PSC-90-19.

DANELUTTO, M., PELAGATTI, S., AND VANNESCHI, M. 1991b. High level languages for easy massively parallel computing. Tech. Rep., Hewlett-Packard Pisa Science Center, HPL-PSC-91-16.

DARLINGTON, J., CRIPPS, M., FIELD, T., HARRISON, P. G., AND REEVE, M. J. 1987. The design and implementation of ALICE: A parallel graph reduction machine. In *Selected Reprints on Dataflow and Reduction Architectures,* S. S. Thakkar, Ed., IEEE Computer Society Press, Los Alamitos, CA.

DARLINGTON, J., FIELD, A. J., HARRISON, P. G., KELLY, P. H. J., WU, Q., AND WHILE, R. L. 1993. Parallel programming using skeleton functions. In *PARLE93, Parallel Architectures and Languages Europe* (June).

DARLINGTON, J., GUO, Y., AND YANG, J. 1995. Parallel Fortran family and a new perspective. In *Massively Parallel Programming Models* (Berlin, Oct.), IEEE Computer Society Press, Los Alamitos, CA.

DE DINECHIN, F., QUINTON, P., AND RISSET, T. 1995. Structuration of the ALPHA language. In *Massively Parallel Programming Models* (Berlin, Oct.), IEEE Computer Society Press, Los Alamitos, CA, 18–24.

DONGARRA, J., OTTO, S. W., SNIR, M., AND WALKER, D. 1995. An introduction to the MPI standard. Tech. Rep. CS-95-274, University of Tennessee. Available at http://www.netlib.org/tennessee/ut-cs-95-274.ps, January.

DONGARRA, J. J., OTTO, S. W., SNIR, M., AND WALKER, D. 1996. A message passing standard for MPP and workstations. *Commun. ACM 39,* 7, 84–90.

DOUGLAS, A., ROWSTRON, A., AND WOOD, A. 1995. ISETL-Linda: Parallel programming with bags. Tech. Rep. YCS 257, Department of Computer Science, University of York, UK, September.

ECKART, J. D. 1992. Cellang 2.0: Reference manual. *SIGPLAN Not. 27,* 8, 107–112.

EISENBACH, S. AND PATTERSON, R. 1993. π-calculus semantics for the concurrent configuration language Darwin. In *Proceedings of 26th Annual Hawaii International Conference on System Science* (Jan.) Vol. II, IEEE Computer Society Press, Los Alamitos, CA.

EKANADHAM, K. 1991. A perspective on Id. In *Parallel Functional Languages and Compilers,* B. K. Szymanski, Ed., ACM Press, New York, 197–254.

FAGIN, B. S. AND DESPAIN, A. M. 1990. The performance of parallel Prolog programs. *IEEE Trans. Comput. C-39,* 12, 1434–1445.

FAIGLE, C., FURMANSKI, W., HAUPT, T., NIEMIC, J., PODGORNY, M., AND SIMONI, D. 1993. MOVIE model for open systems based high performance distributed computing. *Concurrency Pract. Exper. 5,* 4 (June), 287–308.

FAUST, J. E. AND LEVY, H. M. 1990. The performance of an object-oriented threads package. In *OOPSLA ECOOP '90 Conference on Object-Oriented Programming: Systems, Languages, and Applications,* (Ottawa, Canada, Oct.), 278–288, Published in *SIGPLAN Not. 25,* 10 (Oct.).

FELDCAMP, D. AND WAGNER, A. 1993. Parsec: A software development environment for performance oriented parallel programming. In *Transputer Research and Applications 6.* S. Atkins and A. Wagner, Eds., May, IOS Press, Amsterdam, 247–262.

FLYNN-HUMMEL, S. AND KELLY, R. 1993. A rationale for parallel programming with sets. *J. Program. Lang. 1,* 187–207.

FOSTER, I. AND TAYLOR, S. 1990. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, NJ.

FOSTER, I. AND TUECKE, S. 1991. Parallel programming with PCN. Tech. Rep. ANL-91/32, Rev. 1, Mathematics and Computer Science Division, Argonne National Laboratory, December.

FOSTER, I., OLSON, R., AND TUECKE, S. 1992. Productive parallel programming: The PCN approach. *Sci. Program. 1,* 1, 51–66.

FOSTER, I., TUECKE, S., AND TAYLOR, S. 1991. A portable run-time system for PCN. Tech. Rep. ALMS/MCS-TM-137. Argonne National Laboratory and CalTech, December. Available at ftp://info.mcs.anl.gov/pub/tech_reports/.

GEHANI, H. H. AND ROOME, W. D. 1986. Concurrent C. *Softw. Pract. Exper. 16,* 9, 821–844.

GEIST, G. A. 1993. PVM3: Beyond network computing. In *Parallel Computation,* J. Volkert, Ed., LNCS 734, Springer-Verlag, 194–203.

GERTH, R. AND PNUELI, A. 1989. Rooting UNITY. In *Proceedings of the Fifth International Workshop on Software Specification and Design* (Pittsburgh, PA, May).

GIBBONS, J. 1991. Algebras for tree algorithms. D. Phil. Thesis, Programming Research Group, University of Oxford.

GOGUEN, J. AND WINKLER, T. 1988. Introducing OBJ3. Tech. Rep. SRI-CSL-88-9, SRI International, Menlo Park, CA, August.

GOGUEN, J., KIRCHNER, C., KIRCHNER, H., MÉGRELIS, A., MESEGUER, J., AND WINKLER, T. 1994. An introduction to OBJ3. In *Lecture Notes in Computer Science,* Vol. 308, Springer-Verlag, 258–263.

GOGUEN, J., WINKLER, T., MESEGUER, J., FUTATSUGI, K., AND JOUANNAUD, J.-P. 1993. Introducing OBJ. In *Applications of Algebraic Specification using OBJ.* J. Gouguen, Ed., Cambridge.

GOLDMAN, K. J. 1989. Paralation views: Abstractions for efficient scientific computing on the Connection Machine. Tech. Rep. MIT/LCS/TM398, MIT Laboratory for Computer Science.

GOTTLIEB, A. ET AL. 1983a. The NYU Ultracomputer—designing an MIMD shared memory parallel computer. *IEEE Trans. Comput. C-32* (Feb.), 175–189.

GOTTLIEB, A., LUBACHEVSKY, B., AND RUDOLPH, L. 1983b. Basic techniques for the efficient coordination of large numbers of cooperating sequential processes. *ACM Trans. Program. Lang. Syst. 5,* 2 (April), 164–189.

GREGORY, S. 1987. *Parallel Logic Programming in PARLOG.* Addison-Wesley, Reading, MA.

GRIMSHAW, A. S. 1991. An introduction to parallel object-oriented programming with Mentat. Tech. Rep. 91-07, Computer Science Department, University of Virginia, April.

GRIMSHAW, A. 1993a. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Comput. 26,* 5 (May), 39–51.

GRIMSHAW, A. S. 1993b. The Mentat computation model: Data-driven support for object-oriented parallel processing. Tech. Rep. 93-30, Computer Science Department, University of Virginia, May.

GRIMSHAW, A. S., LOYOT, E. C., AND WEISSMAN, J. B. 1991a. Mentat programming language. MPL Reference Manual 91-32, University of Virginia, 3 November.

GRIMSHAW, A. S., LOYOT, E. C., SMOOTAND, S., AND WEISSMAN, J. B. 1991b. Mentat user's manual. Tech. Rep. 91-31, University of Virginia, 3 November 1991.

HAINS, G. AND FOISY, C. 1993. The data-parallel categorical abstract machine. In *PARLE93, Parallel Architectures and Languages Europe* (June), LNCS 694, Springer-Verlag.

HANSEN, P. B. 1978. Distributed processes: A concurrent programming concept. *Commun. ACM 21,* 11, 934–940.

HATCHER, P. J. AND QUINN, M. J. 1991. *Data-Parallel Programming on MIMD Computers.* MIT Press, Cambridge, MA.

HEINZ, E. A. 1993. Modula-3*: An efficiently compilable extension of Modula-3 for problem-oriented explicitly parallel programming. In *Proceedings of the Joint Symposium on Parallel Processing 1993* (Waseda University, Tokyo, May), 269–276.

HEMPEL, R. 1991. The ANL/GMD macros (PARMACS) in FORTRAN for portable parallel programming using the message passing programming model—users' guide and reference manual. Tech. Rep., GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, November.

HEMPEL, R., HOPPE, H.-C., AND SUPALOV, A. 1992. PARMACS-6.0 library interface specification. Tech. Rep., GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, December.

HERATH, J., YUBA, T., AND SAITO, N. 1987. Dataflow computing. In *Parallel Algorithms and Architectures* (May), LNCS 269, 25–36.

HIGH PERFORMANCE FORTRAN LANGUAGE SPECIFICATION 1993. Available by ftp from titan.rice.cs.edu, January.

HOARE, C. A. R. 1985. *Communicating Sequential Processes.* International Series in Computer Science, Prentice-Hall, Englewood Cliffs, NJ.

HUDAK, P. 1986. Para-functional programming. *IEEE Comput. 19,* 8, 60–70.

HUDAK, P. AND FASEL, J. 1992. A gentle introduction to Haskell. *SIGPLAN Not. 27,* 5 (May).

HUMMEL, R., KELLY, R., AND FLYNN-HUMMEL, S. 1991. A set-based language for prototyping parallel algorithms. In *Proceedings of the Computer Architecture for Machine Perception '91 Conference* (Dec.).

JAFFAR, J. AND LASSEZ, J.-L. 1987. Constraint logic programming. In *ACM Symposium on Principles of Programming Languages,* ACM Press, New York, 111–119.

JAY, C. B. 1995. A semantics for shape. *Sci. Comput. Program. 25,* (Dec.) 251–283.

JERID, L., ANDRE, F., CHERON, O., PAZAT, J. L., AND ERNST, T. 1994. HPF to C-PANDORE translator. Tech. Rep. 824, IRISA: Institut de Recherche en Informatique et Systemes Aleatoires, May.

JONES, G. AND GOLDSMITH, M. 1988. *Programming in Occam2.* Prentice-Hall, Englewood Cliffs, NJ.

HALSTEAD, R. H. JR. 1986. Parallel symbolic computing. *IEEE Comput. 19,* 8 (Aug.).

KALE, L. V. 1987. The REDUCE-OR process model for parallel evaluation of logic programs. In *Proceedings of the Fourth International Conference on Logic Programming* (Melbourne, Australia), 616–632.

KARP, R. M. AND RAMACHANDRAN, V. 1990. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science, Vol. A,* J. van Leeuwen, Ed., Elsevier Science Publishers and MIT Press.

KELLY, P. 1989. *Functional Programming for Loosely-Coupled Multiprocessors.* Pitman, London.

KEßLER, C. W. AND SEIDL, H. 1995. Integrating synchronous and asynchronous paradigms: The Fork95 parallel programming language. In *Massively Parallel Programming Models* (Berlin, Oct.), IEEE Computer Society Press, Los Alamitos, CA.

KILIAN, M. F. 1992a. Can O-O aid massively parallel programming? In *Proceedings of the Dartmouth Institute for Advanced Graduate Study in Parallel Computation Symposium* D. B. Johnson, F. Makedon, and P. Metaxas, Eds. (June), 246–256.

KILIAN, M. F. 1992b. Parallel sets: An object-oriented methodology for massively parallel programming. Ph.D. Thesis, Harvard University, 1992.

KIM, W.-Y. AND AGHA, G. 1995. Efficient support of location transparency in concurrent object-oriented programming languages. In *Supercomputing '95.*

KNUTH, D. E. 1976. Big omicron and big omega and big theta. *SIGACT 8,* 2, 18–23.

KUNG, H. T. 1982. Why systolic architectures? *IEEE Comput. 15,* 1, 37–46.

LARUS, J. R., RICHARDS, B., AND VISWANATHAN, G. 1992. C**: A large-grain, object-oriented, data-parallel programming language. Tech. Rep. TR1126, University of Wisconsin-Madison, November.

LEA, D. 1996. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, Reading, MA.

LECHNER, U., LENGAUER, C., AND WIRSING, M. 1995. An object-oriented airport: Specification and Refinement in Maude. In *Recent Trends in Data Type Specifications,* E. Astesiano, G. Reggio, and A. Tarlecki, Eds., LNCS 906, Springer-Verlag, Berlin, 351–367.

LI, K. AND HUDAK, P. 1989. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst. 7,* 4 (Nov.), 321–359.

LINCOLN, P., MARTÍ-OILIET, N., AND MESEGUER, J. 1994. Specification, transformation, and programming of concurrent systems in rewriting logic. Tech. Rep. SRI-CSL-94-11, SRI, May.

LISKOV, B. 1987. Implementation of Argus. In *Proceedings of the Eleventh Symposium on Operating Systems Principles,* ACM Press, New York, 111–122.

LLOYD, J. W. 1984. *Foundations of Logic Programming.* Springer-Verlag.

LOBE, G., LU, P., MELAX, S., PARSONS, I., SCHAEFFER, J., SMITH, C., AND SZAFRON, D. 1992. The Enterprise model for developing distributed applications. Tech. Rep. 92-20, Department of Computing Science, University of Alberta, November.

MALCOLM, G. 1990. Algebraic data types and program transformation. Ph.D. Thesis, Rijksuniversiteit Groningen, September.

MCCOLL, W. F. 1993a. General purpose parallel computing. In *Lectures on Parallel Computation,* A. M. Gibbons and P. Spirakis, Eds., Cambridge International Series on Parallel Computation, Cambridge University Press, Cambridge, 337–391.

MCCOLL, W. F. 1993b. Special purpose parallel computing. In *Lectures on Parallel Computation,* A. M. Gibbons and P. Spirakis, Eds., Cambridge International Series on Parallel Computation, Cambridge University Press, Cambridge, 261–336.

MCCOLL, W. F. 1994a. Bulk synchronous parallel computing. In *Second Workshop on Abstract Models for Parallel Computation,* Oxford University Press, New York, 41–63.

MCCOLL, W. F. 1994b. An architecture independent programming model for scalable parallel computing. In *Portability and Performance for Parallel Processors,* J. Ferrante and A. J. G. Hey, Eds., Wiley, New York.

MCGRAW, J., SKEDZIELEWSKI, S., ALLAN, S., OLDEHOEFT, R., GLAUERT, J., KIRKHAM, C., NOYCE, B., AND THOMAS, R. 1985. Sisal: Streams

and iteration in a single assignment language: Reference manual 1.2. Tech. Rep. M-146, Rev. 1, Lawrence Livermore National Laboratory, March.

MCGRAW, J. R. 1993. Parallel functional programming in Sisal: Fictions, facts, and future. In *Advanced Workshop, Programming Tools for Parallel Machines* (June). Also available as Laurence Livermore National Laboratories Tech. Rep. UCRL-JC-114360.

MEHROTRA, P. AND HAINES, M. 1994. An overview of the Opus language and runtime system. Tech. Rep. 94-39, NASA ICASE, May.

MENTAT 1994. Mentat tutorial. Available at ftp://uvacs.cs.virginia.edu/pub/mentat/tutorial.ps.Z.

MESEGUER, J. 1992. A logical theory of concurrent objects and its realization in the Maude language. Tech. Rep. SRI-CSL-92-08, SRI International, July.

MESEGUER, J. AND WINKLER, T. 1991. Parallel programming in Maude. In *Research Directions in High-Level Parallel Programming Languages,* J. P. Banâtre and D. Le Métayer, Eds., LNCS 574, June, Springer-Verlag, 253–293.

MESSAGE PASSING INTERFACE FORUM 1993. MPI: A message passing interface. In *Proceedings of Supercomputing '93,* IEEE Computer Society, Washington, DC, 878–883.

MORE, T. 1986. On the development of array theory. Tech. Rep., IBM Cambridge Scientific Center.

MUKHERJEE, B., EISENHAUER, G., AND GHOSH, K. 1994. A machine independent interface for lightweight threads. *ACM Oper. Syst. Rev. 28,* 1 (Jan.), 33–47.

MULLIN, L. M. R. 1988. A mathematics of arrays. Ph.D. Dissertation, Syracuse University, Syracuse, NY, December.

MUNDIE, D. A. AND FISHER, D. A. 1986. Parallel processing in Ada. *IEEE Comput. C-19,* 8, 20–25.

MUSSAT, L. 1991. Parallel programming with bags. In *Research Directions in High-Level Parallel Programming Languages,* J. P. Banâtre and D. Le Métayer, Eds., LNCS 574, June, Springer-Verlag, 203–218.

NEWTON, P. AND BROWNE, J. C. 1992. The CODE2.0 graphical parallel programming language. In *Proceedings of the ACM International Conference on Supercomputing* (July).

NOAKES, M. O. AND DALLY, W. J. 1990. System design of the J-Machine. In *Proceedings of the Sixth MIT Conference on Advanced Research in VLSI,* MIT Press, Cambridge, MA, 179–194.

PEIERLS, R. AND CAMPBELL, G. 1991. ALMS—programming tools for coupling application codes in a network environment. In *Proceedings of the Heterogeneous Network-Based Concurrent Computing Workshop* (Tallahassee, FL, Oct.). Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.91.

PEREIRA, L. M. AND NASR, R. 1984. Delta-Prolog: A distributed logic programming language. In *Proceedings of the International Conference on Fifth Generation Computer Systems* (Tokyo, Nov.), 283–291.

PEYTON-JONES, S. L., CLACK, C., AND HARRIS, N. 1987. GRIP—a parallel graph reduction machine. Tech. Rep., Department of Computer Science, University of London.

PEYTON-JONES, S. L. AND LESTER, D. 1992. *Implementing Functional Programming Languages.* International Series in Computer Science, Prentice-Hall, Englewood Cliffs, NJ.

QUINN, M. J. AND HATCHER, P. J. 1990. Data-parallel programming in multicomputers. *IEEE Softw.* (Sept.), 69–76.

RABHI, F. A. AND MANSON, G. A. 1991. Experiments with a transputer-based parallel graph reduction machine. *Concurrency Pract. Exper. 3,* 4 (August), 413–422.

RADESTOCK, M. AND EISENBACH, S. 1994. What do you get from a π-calculus semantics? In *PARLE'94 Parallel Architectures and Languages Europe,* C. Halatsis, D, Maritsas, G. Philokyprou, and S. Theodoridis, Eds., LNCS 817, Springer-Verlag, 635–647.

RADHA, S. AND MUTHUKRISHNAN, C. 1992. A portable implementation of unity on von Neumann machines. *Comput. Lang. 18,* 1, 17–30.

RAINA, S. 1990. Software controlled shared virtual memory management on a transputer based multiprocessor. In *Transputer Research and Applications 4,* D. L. Fielding, Ed., IOS Press, Amsterdam, 143–152.

RANADE, A. G. 1989. Fluent parallel computation. Ph.D. thesis, Yale University.

SÂBOT, G. 1989. *The Paralation Model: Architecture-Independent Parallel Programming.* MIT Press, Cambridge, MA.

SARASWAT, V. A., RINARD, M., PANANGADEN, P. 1991. Semantic foundations of concurrent constraint programming. In *Proceedings of the POPL '91 Conference,* ACM Press, New York, 333–352.

SEUTTER, F. 1985. CEPROL, a cellular programming language. *Parallel Comput. 2,* 327–333.

SHAPIRO, E. 1986. Concurrent Prolog: A progress report. *IEEE Comput. 19* (August), 44–58.

SHAPIRO, E. 1989. The family of concurrent logic programming languages. *ACM Comput. Surv. 21,* 3, 412–510.

SHEFFLER, T. J. 1992. Match and move, an approach to data parallel computing. Ph.D. The-

sis, Carnegie-Mellon, October. Appears as Rep. CMU-CS-92-203.

SINGH, P. 1993. Graphs as a categorical data type. Master's Thesis, Computing and Information Science, Queen's University, Kingston, Canada.

SKEDZIELEWSKI, S. K. 1991. Sisal. In *Parallel Functional Languages and Compilers,* B. K. Szymanski, Ed., ACM Press Frontier Series, New York, 105–158.

SKILLICORN, D. B. 1990. Architecture-independent parallel computation. *IEEE Comput. 23,* 12 (Dec.), 38–51.

SKILLICORN, D. B. 1994a. Categorical data types. In *Second Workshop on Abstract Models for Parallel Computation,* Oxford University Press, New York, 155–168.

SKILLICORN, D. B. 1994b. *Foundations of Parallel Programming.* Number 6 in Cambridge Series in Parallel Computation, Cambridge University Press, New York.

SKILLICORN, D. B. 1996. Communication skeletons. In *Abstract Machine Models for Parallel and Distributed Computing,* M. Kara, J. R. Davy, D. Goodeve, and J. Nash, Eds., (Leeds, April) IOS Press, The Netherlands, 163–178.

SKILLICORN, D. B. AND CAI, W. 1995. A cost calculus for parallel functional programming. *J. Parallel Distrib. Comput. 28,* 1 (July), 65–83.

SKILLICORN, D. B. AND TALIA, D. 1994. *Programming Languages for Parallel Processing.* IEEE Computer Society Press, Los Alamitos, CA.

SKILLICORN, D. B., HILL, J. M. D., AND MCCOLL, W. F. 1997. Questions and answers about BSP. *Sci. Program. 6,* 3, 249–274.

SPEZZANO, G. AND TALIA, D. 1997. A high-level cellular programming model for massively parallel processing. In *Proceedings of the Second International Workshop on High-Level Programming Models and Supportive Environments, HIPS'97,* IEEE Computer Society Press, Los Alamitos, CA, 55–63.

SPIVEY, J. M. 1989. A categorical approach to the theory of lists. In *Mathematics of Program Construction* (June), LNCS 375. Springer-Verlag, 399–408.

STEELE, G. L., JR. 1992. High Performance Fortran: Status report. In *Proceeding of a Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory Multiprocessors,* (Sept.) appeared as *SIGPLAN Not. 28,* 1 (Jan.), 1–4.

SUNDERAM, V. 1992. Concurrent computing with PVM. In *Proceedings of the Workshop on Cluster Computing* (Tallahassee, FL, Dec.). Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.92.

SUNDERAM, V. S. 1990. PVM: A framework for parallel distributed computing. Tech. Rep. ORNL/TM-11375, Department of Math and Computer Science, Emory University, Oak Ridge National Lab, February. Also *Concurrency Pract. Exper. 2* 4 (Dec.), 315–349.

SWINEHART, D. ET AL. 1985. The structure of Cedar. *SIGPLAN Not. 20,* 7 (July), 230–244.

SZAFRON, D., SCHAEFFER, J., WONG, P. S., CHAN, E., LU, P., AND SMITH, C. 1991. Enterprise: An interactive graphical programming environment for distributed software. Available by ftp from cs.ualberta.ca.

TALIA, D. 1993. A survey of PARLOG and Concurrent Prolog: The integration of logic and parallelism. *Comput. Lang. 18,* 3, 185–196.

TALIA, D. 1994. Parallel logic programming systems on multicomputers. *J. Program. Lang. 2,* 1 (March), 77–87.

THOMPSON, S. J. 1992. Formulating Haskell. Tech. Rep. No. 29/92, Computing Laboratory, University of Kent, Canterbury, UK.

TSENG, C.-W. 1993. An optimizing Fortran D compiler for MIMD distributed-memory machines. Ph.D. Thesis, Rice University, January. Also Rice COMP TR-93-199.

UEDA, K. 1985. Guarded Horn clauses. Tech. Rep. TR-103, ICOT, Tokyo.

VALIANT, L. G. 1989. Bulk synchronous parallel computers. Tech. Rep. TR-08-89, Computer Science, Harvard University.

VALIANT, L. G. 1990a. A bridging model for parallel computation. *Commun. ACM 33* 8 (August), 103–111.

VALIANT, L. G. 1990b. General purpose parallel architectures. In *Handbook of Theoretical Computer Science, Vol. A,* J. van Leeuwen, Ed., Elsevier Science Publishers and MIT Press.

VAN HENTENRYCK, P. 1989. Parallel constraint satisfaction in logic programming: Preliminary results of Chip within PEPSys. In *Proceedings of the Sixth International Congress on Logic Programming,* MIT, Cambridge, MA, 165–180.

VIOLARD, E. 1994. A mathematical theory and its environment for parallel programming. *Parallel Process. Lett. 4,* 3, 313–328.

WATANABE, T. ET AL. 1988. Reflection in an object-oriented concurrent language. *SIGPLAN Not. 23* 11 (Nov.), 306–315.

WILKINSON, T., STIEMERLING, T., OSMON, P., SAULSBURY, A., AND KELLY, P. 1992. Angel: A proposed multiprocessor operating system kernel. In *Proceedings of the European Workshops on Parallel Computing (EWPC'92),* (Barcelona, March 23–24) 316–319.

WILLS, D. S. 1990. Pi: A parallel architecture interface for multi-model execution. Tech. Rep. AI-TR-1245, MIT Artificial Intelligence Laboratory.

WINKLER, T. 1993. Programming in OBJ and Maude. In *Functional Programming, Concurrency, Simulation and Automated Reasoning,* P. E. Lauer, Ed., LNCS 693, Springer-Verlag, Berlin, 229–277.

YANG, A. AND CHOO, Y. 1991. Parallel-program transformation using a metalanguage. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*

YANG, A. AND CHOO, Y. 1992a. Formal derivation of an efficient parallel Gauss-Seidel method on a mesh of processors. In *Proceedings of the Sixth International Parallel Processing Symposium,* (March), IEEE Computer Society Press, Los Alamitos, CA.

YANG, A. AND CHOO, Y. 1992b. Metalinguistic features for formal parallel-program transformation. In *Proceedings of the Fourth IEEE International Conference on Computer Languages* (April), IEEE Computer Society Press, Los Alamitos, CA.

YOKOTE, Y. ET AL. 1987. Concurrent programming in Concurrent Smalltalk. In *Object-Oriented Concurrent Programming,* A. Yonezawa et al., Ed., MIT Press, Cambridge, MA, 129–158.

YONEZAWA, A. ET AL. 1987. *Object-Oriented Concurrent Programming.* MIT Press, Cambridge, MA.

ZENITH, S. E. 1991a. The axiomatic characterization of Ease. In *Linda-Like Systems and their Implementation,* Edinburgh Parallel Computing Centre, TR91-13, 143–152.

ZENITH, S. E. 1991b. A rationale for programming with Ease. In *Research Directions in High-Level Parallel Programming Languages,* J. P. Banâtre and D. Le Métayer, Eds., LNCS 574, June, Springer-Verlag, 147–156.

ZENITH, S. E. 1991c. Programming with Ease. Centre de Recherche en Informatique, École Nationale Supérieure des Mines de Paris, Sept. 20.

ZENITH, S. E. 1992. Ease: the model and its implementation. In *Proceeding of a Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory Multiprocessors,* (Sept.) appeared as *SIGPLAN Not. 28,* 1 (Jan.), 1993, 87.

Received April 1996; revised April 1997; accepted December 1997