Models and Languages for Parallel Computation

David B. Skillicorn
Computing and Information Science
Queen's University
Kingston
Canada
skill@qucis.queensu.ca

Domenico Talia
ISI–CNR
c/o DEIS
Universitá della Calabria
87036 Rende (CS)
Italy
talia@si.deis.unical.it

October 1996

Abstract

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 be efficiently implementable, 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 efficiency 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 trade-offs between expressiveness and efficiency. However, there are models that are both abstract and able to be implemented efficiently, opening the prospect of parallelism as part of the mainstream of computing, rather than a high-performance backwater.

ACM Classification: D.1 Programming Techniques, C.4 Performance of Systems, D.3.2 Language Classifications.

General Terms: Languages, Performance, Theory.

Other Keywords: Parallel programming models, parallel programming languages, general-purpose parallel computation, taxonomy, software development methods, object-oriented languages, logic programming languages.

1 Introduction

Parallel computing is about twenty 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 such as artificial intelligence and finance. Despite some successes and a promising beginning, parallel computing did 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 not yet become a central part of practical applications of computing. Given that opinion over the past twenty 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 performance and hardware 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 done. 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 hence 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 optimisation (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 very 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. These costs 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.

Given these reasons for using parallelism, we might expect parallelism to have rapidly moved 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 with using parallelism, which 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 a sequential 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, rather than suggesting directions, or at least 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 the way this balance has an effect on 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 targeted 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 cost of a sequential program changes by no more than a constant factor when it is moved from one uniprocessor to another. Unfortunately, this is not true for a parallel program, whose cost may change by an order of magnitude when it is moved across architecture families. The fundamental non-local nature of a parallel program requires it to interact with a communication structure, and the cost of this 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 of architecture, 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 has been made 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 the right kind of interface between software and architectures. In section 4 we then 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 work going on to decide 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 capacity of their interconnection networks. For example, an architecture whose processor-memory pairs (sometimes called *processing elements*) are connected by a mesh require 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

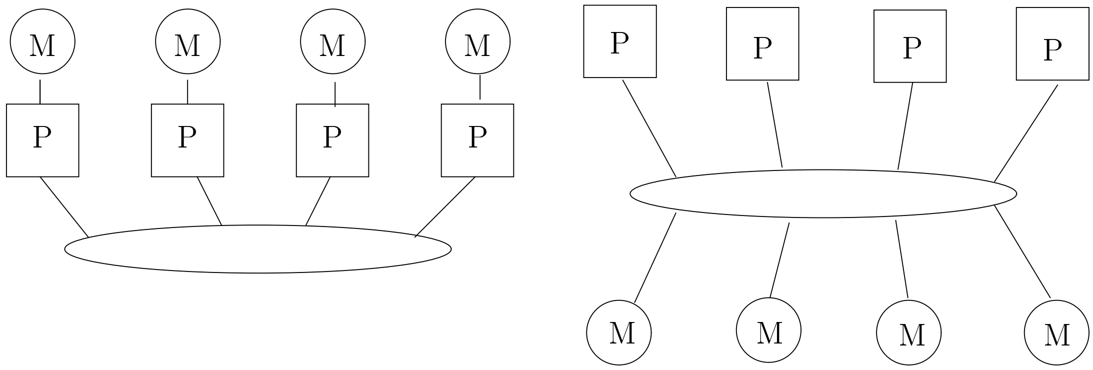

Figure 1: Distributed-Memory MIMD and Shared-Memory MIMD Architectures

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 the instruction to be executed to a number of data processors. These data processors may either interpret the instruction's addresses 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 talk 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. Often we talk about 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 safe because the contexts in a parallel program are all cooperating, and were consistently created by a compiler, which can be made responsible for enforcing their safe interaction.

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

- *Message passing.* The sending process packages the message with a header indicating to which processor and process the data is to be routed, and inserts it 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 *non-blocking 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 has not yet arrived, the receiving process suspends until it does.

- *Transfers through shared memory.* In shared-memory architectures, processes communicate by having the sending process places 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. These 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 very 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. In 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 do not have to 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*).

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 programming 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. It provides both abstraction and stability. Abstraction arises because the operations that the model provides are much 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 that remains stable 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 runtime 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 imply, 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 they each provide 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 necessary one-to-one connection between models: a low-level model may naturally emulate several different higher-level ones, and a high-level model may be naturally emulated by different low-level ones. We will 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 is running on a hundred-processor system, large but not unusual today. There are one hundred 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 virtual threads is several times larger (say 300). Any thread may communicate with any of the other virtual 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 increasing difficult. It is probably impossible to write a modestly-sized MIMD parallel program for one hundred 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 architectures. A model, to be useful, must address both issues, abstraction and effectiveness, which are summarized in the following set of requirements [181]. 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:

    - *Decomposition* of a program into parallel threads. A program must be divided up into the pieces that will execute on distinct processors. This requires separating the program 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 that takes place between each pair of pieces, so that pieces which 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 particular processors that may have some special hardware capability, for example a high-performance floating-point functional unit.

- *Communication* among threads. Whenever non-local data is required, a communication action of some kind must be generated to move the data. Its exact form will depend heavily on the target architecture, but the processes at both ends must arrange to treat it consistently, so that one process does not wait for data that will 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 will be target-dependent. There is enormous potential for deadlock in the interaction between communication and synchronization.

Decomposition and mapping are known to be exponentially expensive to compute optimally. Communication requires placing two ends of communication in the correct threads at the correct place in their respective sequences. Synchronization requires understanding the global state of the computation, which we have already observed is very large. 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 little coupling 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 means that the apparent parallel structure 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 provided 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 a further large gap between specifications and programs, which 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 will not extend to portable parallel programming for two reasons. First, 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. Second, the programmer is unlikely to have access to more than a few of the target architectures on which the program will eventually execute, and therefore cannot even begin to test the software on other architectures. Verification of program properties after construction also seem 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, but they seem essential for parallel programming.

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 non-trivial 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 need some further requirements:

5. **Efficiently Implementable.** A model should be efficiently implementable over a useful variety of parallel architectures. Note that efficiently implementable should not be taken to mean that implementations extract every last ounce of performance out of a target architecture. Parallel computation is useful over a large range of problems, not just the high-performance numerical computations that have historically formed its application domain. For most problems, a level of performance as high as possible on a given architecture is unnecessary, especially if it is 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 without inefficiency. The most powerful architecture class contains shared-memory MIMD computers and distributed-memory MIMD computers whose 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 and taking time t can be executed in such a way that the product pt (called the *work*) is preserved [198]. 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 on to 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 target architecture, a property known as *parallel slackness*.

Architectures in this 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 custom part of the architecture, and therefore by far the most expensive part.

The second class of architectures are distributed-memory MIMD computers whose 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 are devoted to interconnection network hardware. Implementing arbitrary computations on such machines cannot be achieved without loss of efficiency 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 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 only be avoided if the entire program is slowed by a factor of d to compensate.

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 very inefficiently. This is because of their inability to do more than a small constant number of different actions on each step [179].

Thus scalable architectures are not powerful and powerful architectures are not scalable. To achieve efficient implementation across many architectures, these results imply that we must

- reduce the amount of communication allowed in programs by a factor proportional to the diameter of realistic parallel computers (that is by a factor of $\log p$ or $\sqrt{p}$); and
- make computations more regular, so that processors do fewer different operations at each moment, if SIMD architectures are considered as viable target 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 for placement to maximize 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 be efficiently implemented across a full range of target 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 will 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 which computer it will eventually run

on. When a target decision has been made, further changes may be made, but they are of the nature of tuning, rather than algorithm choice. In other words, the construction process can be divided into two phases. In the first, decisions are made about algorithms and the asymptotic cost of the program may be affected; in the second, decisions are made about arrangements of program text, and only the constants in front of the asymptotic costs are affected [132].

This neat division cannot be made for parallel software development, because small changes in program text and choice of target 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 available during all stage of software development, before either the exact arrangement of the program or the target computer have 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 required information be as minimal as possible (since otherwise the actual computation of the cost will be too tedious for practical use). We will say that a model has cost measures if it is possible to determine the cost of a program from its text, minimal target 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 [128].

This requirement is the most contentious of all of them. It requires that models provide *predictable costs* and that compilers do not optimise 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 do design, parallel software construction will remain a black art rather than an engineering discipline.

A further requirement on cost measures is that they are 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 is 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. Abstract models make it easy to build programs but hard to compile them to efficient code, while low-level models make it hard to build software but easy to implement it efficiently. We will 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 with quality of software development methodology since abstract operations can typically only be mapped to implementations if they are semantically clean.

The extent to which the structure of program implementations is constrained by the structure of the program text acts as a surrogate for efficient implementation and the existence of cost measures. Efficient and predictable implementation depends on known placement of computations, and limitations on communication volume. A model that allows fully dynamic behavior by processes is not going to be efficiently implementable or possess cost measures because it has the potential to generate too much communication, and because the cost of communication depends on the interactions of processes whose existence and placement is not known until run-time. A model that does not allow dynamic creation of threads is more likely to permit predictable performance, but may still allow too much communication. Only when the structure of the program is static, and the amount of communication is bounded, can a model satisfy both of these requirements. We will 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 can 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 inefficiency 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 requirements above are good candidates for *general-purpose parallelism*, the application of parallelism across wide problem domains [144].

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 [184].

Models are presented in decreasing order of abstraction, in the following six 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 need to know even if the program they build will execute in parallel. Such models are necessary 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 do not know even 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 reduce that degree of parallelism to fit the target 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 about the breaking up of available work into pieces to be made, 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 target processor. Since locality will often have a marked effect on communication performance, this almost inevitably requires an awareness of the target processor's interconnection network. It becomes very 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. Such models cannot usually be either efficient, since they have no way to limit communication volume, and hence will overrun the communication capacity of some architectures. Nor can they have cost measures, since program costs depend on run-time decisions, and hence cannot be inferred during program design.

- Models that are static, but do not limit communication. Such models cannot be efficient, because again they have no way to prevent interconnection network capacity overruns. However, because they are static, it is possible for them to have cost measures.

- Models that are static and limit communication. Such models can suitably restrict communication, and so may be efficient, and may possess cost measures, because their execution-time structure is implicit in each program's structure.

Within each of these categories we present models based on their common paradigms. Tables 1 and 2 show a classification of models for parallel computation in this way.

Nothing Explicit, Parallelism Implicit
- *Dynamic*
  - Higher order functional–Haskell
  - Concurrent Rewriting–OBJ, Maude
  - Interleaving–Unity
  - Implicit Logic Languages–PPP, AND/OR, REDUCE/OR, Opera, Palm, concurrent constraint languages
- *Static*
  - Algorithmic Skeletons–P3L, Cole, Darlington
- *Static and Communication-Limited*
  - Homomorphic Skeletons–Bird-Meertens Formalism
  - Cellular Processing Languages–Cellang, Carpet, CDL, Ceprol
  - Crystal

Parallelism Explicit, Decomposition Implicit
- *Dynamic*
  - Dataflow–Sisal, Id
  - Explicit Logic Languages–Concurrent Prolog, PARLOG, GHC, Delta-Prolog, Strand
  - Multilisp
- *Static*
  - 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*
  - Data-Specific Skeletons–scan, multiprefix, paralations, dataparallel C, NESL, CamlFlight

Decomposition Explicit, Mapping Implicit
- *Dynamic*
- *Static*
  - BSP, LogP
- *Static and Communication-Limited*

Table 1: Classification of Models of Parallel Computation

Mapping Explicit, Communication Implicit

- *Dynamic*
  - 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*
  - Graphical Languages–Enterprise, Parsec, Code
  - Contextual Coordination Languages–Ease, ISETL-Linda, Opus
- *Static and Communication-Limited*
  - Communication Skeletons

Communication Explicit, Synchronization Implicit

- *Dynamic*
  - 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*
  - Process Networks–static dataflow
  - Internal OO–Mentat
- *Static and Communication-Limited*
  - Systolic Arrays–Alpha

Everything Explicit

- *Dynamic*
  - Message Passing–PVM, MPI
  - Shared Memory–FORK, Java, thread packages
  - Rendezvous–Ada, SR, Concurrent C
- *Static*
  - Occam

PRAM

Table 2: Classification of Models of Parallel Computation (cont.)

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 of the activities that are 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 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 the 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 of such models can guarantee efficient implementations.

We consider models at this high level of abstraction in subcategories: those that permit dynamic structure and communication, those that have static structure and communication, and those that also limit the amount of communication in progress at any given moment.

4.1.1 Dynamic.

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 using 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 for finding solutions to 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 [119]. Haskell also includes several typical features of functional programming such as user-defined types, lazy evaluation, pattern matching, and list comprehensions. Further, 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* [164]. 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 non-overlapping sections of the graph independently, and hence concurrently. 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 concurrently.

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 is wasteful of resources, and alters the semantics of the program if a non-essential piece fails to terminate. For example, most functional languages have some form of conditional like this

```
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 isn't 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. For now evaluating `f(x)` speculatively will cause the program not to terminate, while the other evaluation order does not.

It is quite 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 [67, 119, 129, 163, 166, 193]. Such models are simple and abstract, and allow software development by transformation, but they are not efficiently implementable and 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 [97–99], a functional language whose semantics is based on equational logic, and Maude [136, 151, 152, 203]. An example, based on one in [139], will give 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. 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 are hard to implement efficiently, 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 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 might 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 of 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 say the least, about how the pieces are to be fitted together.

This idea lies behind UNITY [29, 49, 95, 168], and an alternative that considers independence of statements more: action systems [12–15]. UNITY (Unbounded Nondeterministic Iterative Transformations) is both a computational model and a proof system. A UNITY program consists of a declaration of variables, a 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 efficient implementations seem 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 [71]. In particular, 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 we have to solve the subgoal `?-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. 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 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 [85], the AND/OR process model [58], the REDUCE/OR model [127], OPERA [39], and PALM [45]. These models differ in the way they view parallelism and their target architectures are varied, but they are mainly designed to be be implemented on distributed-memory MIMD machines [192]. 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 they are hard to to implement efficiently, although some of them exhibit good performance. Cost measures cannot 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* [172]. 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 explicit 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. In concurrent constraint logic programming a computation progresses by executing threads that concurrently communicate by placing constraints in a global store and synchronize by checking that a constraint is entailed by the store. The communication patterns are dynamic, so that there is no predetermined limited set of threads with which a given thread may interact. Moreover, threads correspond to goal atoms, so they are activated dynamically during program execution. Concurrent constraint logic programming models include *cc* [172], the CHIP CLP language [199], and CLP [122]. As in other parallel logic models, concurrent constraint languages are too dynamic to allow practical cost measures.

4.1.2 Static.

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 together 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 may hide an arbitrary amount of internal complexity.
- The building blocks can be internally parallel, but composable sequentially, in which case programmers do not need to 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* [54], 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. An overview and classification of skeletons can be found as part of the Basel Algorithm Classification Scheme [42].

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 the way in which each encapsulated algorithm is implemented and how parallelism intra– and inter–skeleton is exploited for each possible target architecture.

We briefly mention some of the most important algorithmic skeleton approaches. The Pisa Parallel Programming Language (P^3L) [16, 64–66] 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, who also computed cost measures for them on a parallel architecture [54–57]. 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 [68].

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 open to question. None of the approaches described above addresses this explicitly, and 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, efficient implementations for skeletons are possible, provided that they are chosen with care, and because of this, cost measures can be provided.

4.1.3 Static and Communication-Limited.

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 was developed from the Bird-Meertens formalism [181]. The skeletons in this model are based on particular data types, one set for lists, one set for arrays, one set from 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 only to be done once for each datatype.

Consider the pattern of computation and communication shown in Figure 2. Any list homomorphism can be computed by appropriately substituting for f and g, where g must be associative. 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 may 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 target 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 target architecture interconnection network. Then all communication can take place with nearest neighbors (and hence in constant time). Homomorphic skeletons have been built for most of the standard types: sets and bags [181], lists [31, 142, 186], trees [96], arrays [20, 21], molecules [180], and graphs [177].

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

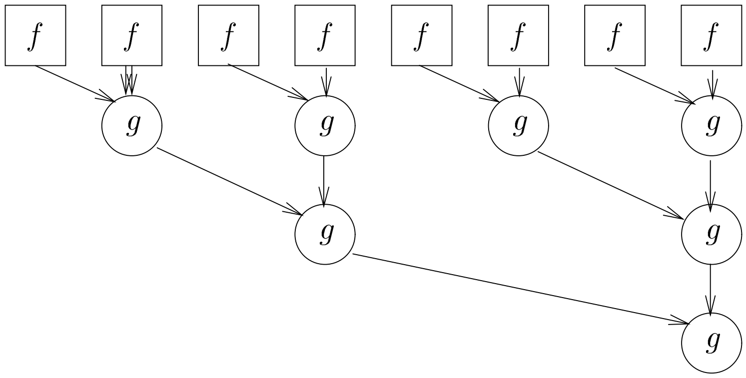

Figure 2: A Skeleton for Computing Arbitrary List Homomorphisms

type. Efficient implementations can be built for any target 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 [183].

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 a 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. The transition function 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 2D lattice are

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

Cellular processing languages, such as Cellang [75], CARPET [185], CDL, and CEPROL [173], 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. Further, 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 target 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 efficient and scalable implementations, both on MIMD and SIMD parallel computers [44]. Moreover, cost measures can be provided.

Another model that takes geometric arrangement explicitly into account is *Crystal* [50, 204–206]. Crystal is a functional language with added data types, called *index domains* to represent geometry, that is locality and arrangement. The feature which distinguishes 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 *shapely types* [123].

4.2 Parallelism Explicit.

The second major class of models are those 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. There are two main strategies for implementing decomposition, both depending 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 decomposition does not matter very much for performance. In this situation, any decomposition is effective, so a simple algorithm can be used to compute it. The second is to use skeletons that have a natural mapping to target processor topologies, skeletons based on the structure of the data that the program uses.

4.2.1 Dynamic.

Dataflow [115] 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 moves 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 target 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 operations execute only when all of their inputs are present, communication is always unsynchronized.

Dataflow languages have taken different approaches to expressing repetitive operations. Language such as Id [77] and Sisal [147, 148, 178] are first-order functional (or single assignment) languages. They have syntactic structures looking like loops, which create a new context for each execution of the 'loop body' (so that they seem like imperative languages except that each variable name may only be assigned to 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
```

In Sisal parallelism is not explicit at the source level. However, the language run-time system may exploit 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 be efficiently implemented; indeed Sisal performs competitively with the best Fortran compilers on shared-memory architectures [148]. However, performance on distributed-memory architectures is still not competitive. Because so much scheduling is done dynamically at run-time, cost measures are not possible.

Explicit logic languages are those in which programmers must specify the parallelism explicitly [175]. They are also called concurrent logic languages. Examples of languages in this class are PARLOG [102], Delta-Prolog [162], Concurrent Prolog [174], GHC[195], and Strand [90].

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 concurrency 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 non-determinism 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 <- G1, G2,..., Gn | B1, B2,..., Bm.` $n, m \geq 0$

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 `|` is called the *commit* operator, and it 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 [191]. 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 only tried 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` will be tested for candidacy in parallel but the clause `C3` will be 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.

Another symbolic programming language in which parallelism is made explicit by the programmer is Multilisp. The Multilisp [126] 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. A future applied to an expression creates a thread to evaluate in parallel that expression which begins immediately, that is eagerly. The expression `(future X)` immediately returns a suspension for the value of `X` and creates a thread to concurrently 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 very 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.

4.2.2 Static.

Turning to models with static structure and communication, we re-encounter the skeleton concept, but this time skeletons 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 very 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 analysed for situations in which the same operation applied repeatedly to different data and where 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 realised that vectorisable 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 vectorisable code situations can be usefully exploited by a wide range of parallel computers.

Such code situations often involve arrays, and can be seen more abstractly as instance 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, and many answers have been suggested. 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 parallelism: the first based on (parallel) loops, and the second 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 [194] and High Performance Fortran (HPF) [116, 187] 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. In particular, HPF is a parallel language based on Fortran–90, Fortran D, and SIMD Fortran. It 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`. Further, 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 offers also 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 [6–8, 124] and C** [134]. This work is beginning to converge with skeleton approaches: for example Darlington's group have developed a Fortran extension that uses skeletons [69]. Another similar approach is the latest language in the Modula family, Modula 3* [112]. 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 very similar in functionality to HPF.

Data-parallel languages based on data types other than arrays have also been developed. Some examples are: parallel SETL [120, 121], parallel sets [130, 131], match and move [176], Gamma [19, 60, 158], and PEI [200]. 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 over {1..n}}
```

Gamma is a language with data-parallel operations on finite sets. For example, the code to find the maximum element of a set is

$$\mathtt{max}\,\mathtt{M} := \mathtt{x}:\mathtt{M},\ \mathtt{y}:\mathtt{M} \rightarrow \mathtt{x}:\mathtt{M} \leftarrow \mathtt{x} \geq \mathtt{y}$$

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) [156], and Nial and Array Theory [154].

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 and infer what pattern was 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 efficient implementations, and also cost measures, possible.

4.2.3 Static and Communication-Limited.

The data-parallel languages of the previous section were developed with program construction primarily in mind. There are 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.

A wide variety of languages were 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 [32], multiprefix [170], paralations [100, 171], the C* data–parallel language [111, 165], the scan-vector model and NESL [33–38], and CamlFlight [109]. As for other data-parallel languages, these models are simple and fairly abstract. For instance, C* is an extension of the C language that incorporate features of the SIMD parallel model. In C* data parallelism is implemented by defining data of 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. Data–parallel languages usually provide efficient implementations, at least on some architectures, by design, and for the same reason 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.

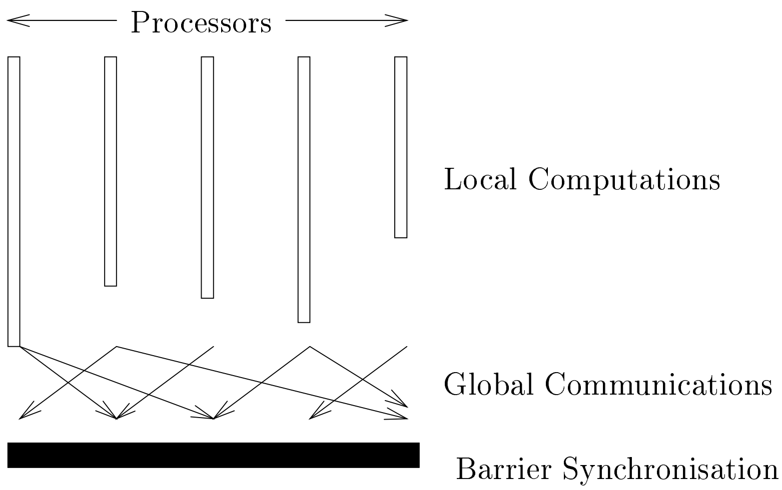

Figure 3: A BSP superstep

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 does not have to be described so explicitly.

4.3.1 Static.

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

Bulk synchronous parallelism (BSP) [143, 144, 146, 196–198] 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 randomly-addressed data can continuously 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 3. 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

$$t = w + hg + l$$

(where g and l are the network parameters above) 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, fairly abstract, but lacks a software construction methodology. The cost measures give the real cost of a program on any architecture, and implementations are as efficient as any BSP program could be (but there could be other programs in a different style that were more efficient).

The current implementation of BSP uses an SPMD library that can be invoked 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 [61], 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 to serve as an abstract model that is able to captures 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 have 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 [30], 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 to label the two ends of each communication action to say that they belong together, and to ensure that communication actions are properly matched. Given the number of communications in a large parallel program, this is a tedious burden to place on 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

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 [4, 46–48], 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" and 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 is not necessarily efficiently implementable, 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 [3]. The LPB environment guides a user through program design, coding, monitoring, and execution of Linda software.

Non-message communication languages reduce the overheads of managing communication by disguising communication in ways that fit more naturally into threads. For example, ALMS [11, 161] treats message passing as if the communication channels were memory mapped. Reference to certain message variables appearing in different threads behaves like a message transfer from one to the others. PCN [89, 91, 92] 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 very 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 how the composed blocks are to be mapped to processors. Processes that shares 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.

The logical extension of mapping communication to memory is *virtual shared memory*, in which the abstraction provided 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 [53, 138, 169, 201].

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 [129]. The same reduction rules apply, so that the communication and synchronization induced by this placement follows in the same way as in pure graph reduction.

An example of this kind of languages is Paralf [118]. 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 expression's are to be evaluated on which processor. An expression followed by the annotation `$on proc` will be 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)`.

Remote Procedure Call. 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 [110], Cedar [81], and Concurrent CLU [59].

4.4.2 Static

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 [141, 190] classifies program units by type and inserts 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 [88] allows program units to be connected using a set of predefined connection patterns. Code [159] 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 efficient implementations are not guaranteed, and 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 prevents modular development of software. A model that extends Linda by including ideas from Occam is the language Ease [207–210]. Ease programs have multiple tuple spaces, which are called *contexts* and may be visible only to some threads. Because those threads that may 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 may 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 it from the context. Such operations can use pass-by-reference since they guarantee that the data will only be referenced by one thread at a time. Ease has many of the same properties as Linda, but makes it easier to build efficient implementations. Ease also helps with decomposition by allowing process structuring in the style of Occam.

Another related language is ISETL-Linda [74], 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 [149]. It is a language with 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 which 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

Communication skeletons extend the idea of prestructured building blocks to communication [182]. 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 4 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 target topology, and then embedding results used to build implementations for targets with richer interconnection topologies. It can be efficiently implemented and does have cost measures.

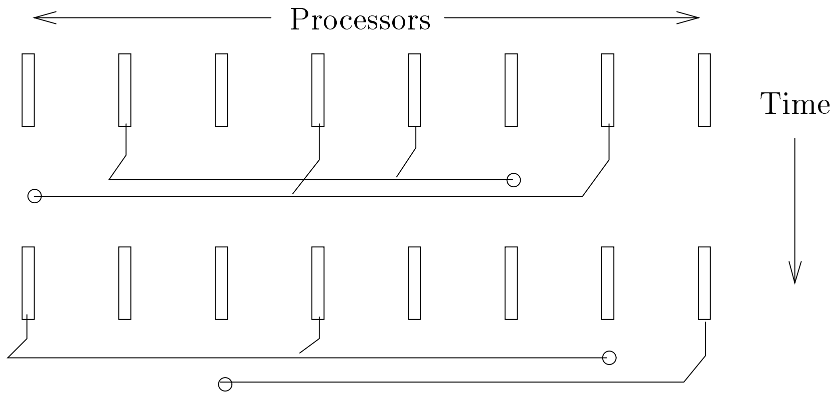

Figure 4: A Communication Skeleton

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 when it will happen, and delivery of multiple messages may be out of order.

4.5.1 Dynamic.

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 may individually decide what their response to data arrival will be, and may 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 [1, 22, 23]. Actor systems consist of collections of objects called actors, each of which has an incoming message queue. An actor repeatedly executes the following 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, efficient implementations of actors are not possible without restricting the total communication, and the distributed nature of the model makes this impossible to do. This and the nature of the communication delivery system makes cost measures impossible. The actor model is quite low level, but it is straightforward and modular.

A different kind of process net is provided by the language Darwin [76, 167] 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. Unlike most configuration languages, the binding of the semantics of communication to connections is dynamic.

One of the weaknesses of the actor model is that an 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 [51, 52] and ActorSpace [2]. 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 an homogeneous collection of objects (called representatives) that are grouped together and may be 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 like 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. Intra-aggregate addressing makes cooperation among parts of an aggregate possible.

The ActorSpace model extends the actor model to avoid unnecessary synchronizations. In the ActorSpace model, communications are asynchronous, so an actor sending a message need not block its execution until the recipient is ready to receive or process the message. Thus programmers are freed from explicitly specifying code to manage messages when an actor is not in a state to process them. By not creating unnecessary data dependencies, the message-driven approach of the ActorSpace model allows maximum concurrency to be exploited. 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.

External OO models. Actors are regarded as existing whether or not they are being communicated with. 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. There are two ways to do this. The first, 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 together 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, but here we concentrate on external OO models and languages.

Some interesting external object-based models are ABCL/1 [80], ABCL/R [82], POOL-T [5], EPL [78], Emerald [79], and Concurrent Smalltalk [83]. In these languages, parallelism is based on assigning a thread to each object, and asynchronous message passing is used to increase concurrency further. 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, like the actor model, messages are processed serially within an object. Though 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 concurrency between objects. An extension of ABCL/1 is ABCL/R where reflection has been introduced.

Objects and processes. Parallelism in external object-oriented languages can be exploited 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. 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 it 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 [140] and Presto [28] 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 it 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 both 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 [86], and the language environments for the J-machine [52, 62, 160].

4.5.2 Static.

Internal object-oriented languages. We now return to object-oriented languages in which parallelism occurs within single methods. The Mentat Programming Language (MPL) is an parallel object-oriented system designed to address the problems of 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, while the object-oriented paradigm hides much of the parallel environment from the user. MPL is an extension of C++ which supports both intra- and inter-object parallelism. The compiler and the run-time support of the language are designed to achieve high performance. The language constructs are mapped to the macro dataflow model that is the computation model underlying Mentat. It 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 performs 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 automatically forwarded to the final addition. That result will be forwarded to the caller and associated with M. In this approach the programmer makes granularity and partitioning decisions using Mentat class definition constructs, while the compiler and the run-time support manage communication and synchronization [103–108, 150].

4.5.3 Static and Communication-Limited.

Systolic arrays. A systolic array is a gridlike architecture 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 [133] 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. In particular high-level programmable arrays allow the developments of systolic algorithms by the definition of inter- and intra-cell concurrency, and cell-to-cell data communication. Clearly, the principle of rhythmic communication separates 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 [70], where programs are expressed as recurrence equations. These are transformed into systolic form by regarding the data dependencies as defining an affine space which can be geometrically transformed.

4.6 Everything Explicit.

The next category of models are 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.

Most models provide a particular paradigm for handling partitioning, mapping, and communication. There are a few models that have tried to be general enough to provide multiple paradigms, for example Pi [63, 202], by providing sets of primitives for each style of communication. Such models can be efficiently implemented and can have 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 [18] and SR [9, 10]. Orca is an object-based language which 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 rapidly take effect 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 that are used for communication between the parent and the child processes. Synchronizing Resources (SR) is based on the *resource* concept. A resource is a module that can contain several processes. A resource can be dynamically created by the *create* command and its processes can communicate by the use of semaphores. Processes belonging to different resources can communicate using only a restricted set of operations explicitly defined in the program as procedures.

There are 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 *receives* to specify the message to be exchanged, process identifier and address.

It was quickly realised 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*) [72, 153], 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 has been defined to become the standard message passing interface for parallel applications and libraries [73]. 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 start-up, 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 [24–27, 94, 188, 189], Parmacs [113, 114], and p4 [43]. 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()`, `pvm_mcast()`, `pvm_barrier()` can be used.

Using PVM and similar models, programmers must do all of the decomposition, placement, and communication explicitly. This may be 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 ATM 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 [135], 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 [40, 41, 87, 155].

Rendezvous. 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 [157] and Concurrent C [93]. Ada was designed on behalf of the U.S. 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 only allowed 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 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.

Recent surveys of such models can be found in [17, 84, 101].

4.6.2 Static.

Most low-level models allow dynamic process creation and communication. An exception is Occam [125], in which the process structure is fixed, and communication takes place across 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 terminate 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 [117], so that software development by transformation is possible. However, it is so low-level that this is only practical for small or critical applications.

4.7 PRAM.

A final model that must be considered is the PRAM model [128], 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 very 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 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 [137].

A good overview of models aimed at particular architectures can be found in [145].

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 be able to use the model as a target for software development. They are: ease of programming, the existence of a methodology for constructing software that handles issues such as correctness, independence from particular architectures, and simplicity and abstractness. The two remaining criteria address the need for execution of the model on real parallel machines. They are: efficient implementability, 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, that 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 very explicit, has diminished significantly. We regard this as a good thing, since it shows that the importance of abstraction is being realised by 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 the maximum expressiveness. This is also a good thing, since trade-offs among expressiveness, software development complexity, and runtime 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 will 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 make simpler 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 may 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 from there 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 efficiently. 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. Being able to compute 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.

Acknowledgement We are grateful to Luigi Palopoli for his comments on a draft of this paper, and to the anonymous referees for their helpful comments.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] G. Agha and C.J. Callsen. ActorSpace: An open distributed programming paradigm. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 23–32, May 1993.

[3] S. Ahmed, N. Carriero, and D. Gelernter. A program building tool for parallel applications. In *DIMACS Workshop on Specification of Parallel Algorithms*, pages 161–178, Princeton University, USA, May 1994.

[4] S. Ahuja, N. Carriero, D. Gelernter, and V. Krishnaswamy. Matching languages and hardware for parallel computation in the Linda machine. *IEEE Transactions on Computers*, 37, No.8:921–929, August 1988.

[5] P. America. POOL-T: A parallel object-oriented language. In A. Yonezawa et al., editor, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[6] F. André, O. Chéron, and J.-L. Pazat. Compiling sequential programs for distributed memory parallel computers with Pandore II. Preprint, May 1992.

[7] Francoise Andre, Yves Maheo Marc Le Fur, and Jean-Louis Pazat. The Pandore compiler: Overview and experimental results. Technical Report PI-869, IRISA, October 1994.

[8] Françoise André and Henry Thomas. The Pandore System. IFIP working conference on Decentralized Systems, December 1989. Poster Presentation.

[9] G.R. Andrews and R.A. Olsson. *The SR Programming Language.* Benjamin/Cummings, 1993.

[10] G.R. Andrews, R.A. Olsson, M.A. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[11] B. Arbeit, G. Campbell, R. Coppinger, R. Peierls, and D. Stampf. The ALMS system: implementation and performance. Brookhaven National Laboratory, Internal Report, (In preparation).

[12] R.J.R. Back. A method for refining atomicity in parallel algorithms. In *PARLE89 Parallel Architectures and Languages Europe*, Springer Lecture Notes in Computer Science 366, pages 199–216, June 1989.

[13] R.J.R. Back. Refinement calculus part II: Parallel and reactive programs. Technical Report 93, Åbo Akademi, Departments of Computer Science and Mathematics, SF-20500 Åbo, Finland, 1989.

[14] R.J.R. Back and K. Sere. Stepwise refinement of action systems. In *Mathematics of Program Construction*, Springer Lecture Notes in Computer Science 375, pages 115–138, June 1989.

[15] R.J.R. Back and K. Sere. Deriving an Occam implementation of action systems. Technical Report 99, Åbo Akademi, Departments of Computer Science and Mathematics, SF-20500 Åbo, Finland, 1990.

[16] F. Baiardi, M. Danelutto, M. Jazayeri, S. Pelagatti, and M. Vanneschi. Architectural models and design methodologies for general-purpose highly-parallel computers. In *IEEE CompEuro 91 – Advanced Computer Technology, Reliable Systems and Applications*, May 1991.

[17] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming languages for distributed computing systems. *Computing Surveys*, 21(3):261–322, September 1989.

[18] H.E. Bal, A.S. Tanenbaum, and M.F. Kaashoek. Orca: A language for distributed processing. *SIGPLAN Notices*, 25(5):17–24, May 1990.

[19] J.P. Banâtre and D. Le Métayer. Introduction to Gamma. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, pages 197–202. Springer Lecture Notes in Computer Science 574, June 1991.

[20] C. Banger. Arrays with categorical type constructors. In *ATABLE'92 Proceedings of a Workshop and Arrays*, pages 105–121, June 1992.

[21] C.R. Banger. *Construction of Multidimensional Arrays as Categorical Data Types*. PhD thesis, Queen's University, Kingston, Canada, 1995.

[22] F. Baude. *Utilisation du Paradigme Acteur pour le Calcul Parallèle*. PhD thesis, Université de Paris-Sud, 1991.

[23] F. Baude and G. Vidal-Naquet. Actors as a parallel programming model. In *Proceedings of 8th Symposium on Theoretical Aspects of Computer Science*. Springer Lecture Notes in Computer Science 480, 1991.

[24] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and V. Sunderam. PVM and HeNCE: Tools for heterogeneous network computing. In J.S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 91–99. Springer-Verlag, 1993.

[25] A Beguelin, J Dongarra, A Geist, R Manchek, and V Sunderam. Recent enhancements to PVM. *International Journal of Supercomputing Applications and High Performance Computing*, 95.

[26] A. Beguelin, J.J. Dongarra, G.A. Geist, R. Manchek, and V.S. Sunderam. PVM software system and documentation. Email to `netlib@ornl.gov`.

[27] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, Steve Otto, and Jon Walpole. PVM: Experiences, current status and future direction. Technical Report CS/E 94-015, Oregon Graduate Institute CS, 1994.

[28] B.N. Bershad, E. Lazowska, and H. Levy. Presto: a system for object-oriented parallel programming. *Software–Practice and Experience*, August 1988.

[29] Mark Bickford. Composable specifications for asynchronous systems using UNITY. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 216–227, November 1994.

[30] G. Bilardi, K.T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP vs LogP. In *Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures*, pages 25–32, June 1996.

[31] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987.

[32] G. Blelloch. Scans as primitive parallel operations. In *Proceedings of the International Conference on Parallel Processing*, pages 355–362, August 1987.

[33] G.E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[34] G.E. Blelloch and G.W. Sabot. Compiling collection-oriented languages onto massively-parallel computers. In *Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 575–585, 1988.

[35] G.E. Blelloch and G.W. Sabot. Compiling collection-oriented languages onto massively-parallel computers. *Journal of Parallel and Distributed Computing*, pages 119–134, 1990.

[36] Guy Blelloch and John Greiner. A parallel complexity model for functional languages. Technical Report CMU-CS-94-196, School of Computer Science, Carnegie Mellon University, October 1994.

[37] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, Carnegie Mellon University, April 1993.

[38] Guy E. Blelloch. Programming parallel algorithms. In *Proceedings of the 3rd Workshop on Parallel Algorithms (WOPA)*, May 1993.

[39] J. Briat, M. Favre, C. Geyer, and J. Chassin de Kergommeaux. Scheduling of OR-parallel prolog on a scalable reconfigurable distributed memory multiprocessor. In *Proceedings of PARLE 91, Springer Lecture Notes in Computer Science 506*, pages 385–402. Springer-Verlag, 1991.

[40] Peter A. Buhr, Hamish I. Macdonald, and Richard A. Stroobosscher. μSystem reference manual, version 4.3.3. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, March 1991.

[41] Peter A. Buhr and Richard A. Stroobosscher. The μSystem: Providing light-weight concurrency on shared-memory multiprocessor computers running UNIX. *Software Practice and Experience*, 20(9):929–963, September 1990.

[42] H. Burkhart, C. Falcó Korn, S. Gutzwiller, P. Ohnacker, and S. Waser. BACS: Basel Algorithm Classification Scheme. Technical Report 93–3, Institut für Informatik der Universität Basel, March 1993.

[43] R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report ANL-92/17, Argonne National Laboratory, Mathematics and Computer Science Division, October 1992.

[44] M. Cannataro, S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, and D. Talia. A parallel cellular automata environment on multicomputers for computational science. *Parallel Computing*, 21, No.5:803–824, 1995.

[45] M. Cannataro, G. Spezzano, and D. Talia. A parallel logic system on a multicomputer architecture. In *Future Generation Computer Systems*, volume 6, pages 317–331, 1991.

[46] N. Carriero. Implementation of tuple space machines. Technical Report YALEU/DCS/RR-567, Dept. of Computer Science, Yale University, December 1987.

[47] N. Carriero and D. Gelernter. Application experience with Linda. In *ACM/SIGPLAN Symposium on Parallel Programming*, volume 23, pages 173–187, July 1988.

[48] N. Carriero and D. Gelernter. Learning from our success. In J.S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 37–45. Springer-Verlag, 1993.

[49] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[50] M. Chen, Y.-I. Choo, and J. Li. Crystal: Theory and pragmatics of generating efficient parallel code. In B.K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 255–308. ACM Press Frontier Series, 1991.

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

[52] A.A. Chien and W.J. Dally. Concurrent Aggregates. In *Second SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–196, February 1990.

[53] Andrew Chin and W. F. McColl. Virtual shared memory: Algorithms and complexity. *Information and Computation*, 113(2):199–219, September 1994.

[54] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[55] M. Cole. Towards fully local multicomputer implementations of functional programs. Technical Report CS90/R7, Department of Computing Science, University of Glasgow, January 1990.

[56] M. Cole. Writing parallel programs in non-parallel languages. In R. Perrot, editor, *Software for Parallel Computers: Exploiting Parallelism through Software Environments, Tools, Algorithms and Application Libraries*, pages 315–325. Chapman and Hall, 1992.

[57] Murray Cole. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem. In G. R. Joubert, editor, *Parallel Computing: Trends and Applications*, pages 489–492. North-Holland, 1994.

[58] J.S. Conery. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, 1987.

[59] R. Cooper and K.G. Hamilton. Preserving abstraction in concurrent programming. *IEEE Transactions on Software Engineering*, SE-14 No.2:258–263, 1988.

[60] C. Creveuil. Implementation of Gamma on the Connection Machine. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, pages 219–230. Springer Lecture Notes in Computer Science 574, June 1991.

[61] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Toward a realistic model of parallel computation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[62] W.J. Dally, J.A.S. Fiske, J.S. Keen, R.A. Lethin, M.D. Noakes, P.R. Nuth, R.E. Davison, and G.A. Fyler. The message-driven processor. *IEEE Micro*, pages 23–39, April 1992.

[63] W.J. Dally and D.S. Wills. Universal mechanisms for concurrency. In *PARLE '89, Parallel Architectures and Languages Europe*, pages 19–33. Springer-Verlag Lecture Notes in Computer Science 365, June 1989.

[64] M. Danelutto, R. di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 1992. Also appears as "The P^3L language: an introduction", Hewlett-Packard Report HPL-PSC-91-29, December 1991.

[65] M. Danelutto, R. di Meglio, S. Pelagatti, and M. Vanneschi. High level language constructs for massively parallel computing. Technical report, Hewlett Packard Pisa Science Center, HPL-PSC-90-19, 1990.

[66] M. Danelutto, S. Pelagatti, and M. Vanneschi. High level languages for easy massively parallel computing. Technical report, Hewlett Packard Pisa Science Center, HPL-PSC-91-16, 1991.

[67] J. Darlington, M. Cripps, T. Field, P.G. Harrison, and M.J. Reeve. The design and implementation of ALICE: a parallel graph reduction machine. In S.S. Thakkar, editor, *Selected Reprints on Dataflow and Reduction Architectures*. IEEE Computer Society Press, 1987.

[68] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, Q. Wu, and R.L. While. Parallel programming using skeleton functions. In *PARLE93, Parallel Architectures and Languages Europe*, June 1993.

[69] J. Darlington, Y. Guo, and J. Yang. Parallel Fortran family and a new perspective. In *Massively Parallel Programming Models*, Berlin, October 1995, October 1995. IEEE Computer Society Press.

[70] F. de Dinechin, P. Quinton, and T. Risset. Structuration of the ALPHA language. In *Massively Parallel Programming Models*, Berlin, October 1995. IEEE Computer Society Press.

[71] J. Chassin de Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, 1994.

[72] J. Dongarra, S. W. Otto, M. Snir, and D. Walker. An introduction to the MPI standard. Technical Report CS-95-274, University of Tennessee, `http://www.netlib.org/tennessee/ut-cs-95-274.ps`, January 1995.

[73] J.J. Dongarra, S.W. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 39(7):84–90, 1996.

[74] A. Douglas, A. Rowstron, and A. Wood. ISETL-Linda: Parallel programming with bags. Technical Report YCS 257, Department of Computer Science, University of York, U.K., September 1995.

[75] J. Dana Eckart. Cellang 2.0: Reference manual. *ACM Sigplan Notices*, 27, No.8:107–112, 1992.

[76] S. Eisenbach and R. Patterson. π-calculus semantics for the concurrent configuration language Darwin. In *Proceedings of 26th Annual Hawaii International Conference on System Science*, volume II. IEEE Computer Society Press, January 1993.

[77] K. Ekanadham. A perspective on Id. In B.K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 197–254. ACM Press, 1991.

[78] A. Black et al. EPL programmers' guide. University of Washington, Seattle, June 1984.

[79] A. Black et al. Distribution and abstract types in emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[80] A. Yonezawa et al. *Object-Oriented Concurrent Programming*. MIT Press, 1987.

[81] D. Swinehart et al. The structure of Cedar. *SIGPLAN Notices*, 20(7):230–244, July 1985.

[82] T. Watanabe et al. Reflection in an object-oriented concurrent language. *SIGPLAN Notices*, 23(11):306–315, November 1988.

[83] Y. Yokote et al. Concurrent programming in Concurrent Smalltalk. In A. Yonezawa et al., editor, *Object-Oriented Concurrent Programming*, pages 129–158. MIT Press, 1987.

[84] A. Gottlieb *et al.* The NYU Ultracomputer – designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32:175–189, February 1983.

[85] B.S. Fagin and A.M. Despain. The performance of parallel Prolog programs. *IEEE Transactions on Computers*, C-39, No.12:1434–1445, 1990.

[86] C. Faigle, W. Furmanski, T. Haupt, J. Niemic, M. Podgorny, and D. Simoni. MOVIE model for open systems based high performance distributed computing. In *IEEE Symposium on High Performance Distributed Computing*, September 1992.

[87] John E. Faust and Henry M. Levy. The performance of an object-oriented threads package. In *OOPSLA ECOOP '90 Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 278–288, Ottawa, Canada, October 1990. Published in SIGPLAN Notices Vol.25, No.10. Oct.1990.

[88] D. Feldcamp and A. Wagner. Parsec: A software development environment for performance oriented parallel programming. In S. Atkins and A. Wagner, editors, *Transputer Research and Applications 6*, pages 247–262, Amsterdam, Oxford, Washington, Tokyo, May 1993. IOS Press.

[89] I. Foster, R. Olson, and S. Tuecke. Productive parallel programming: The PCN approach. *Scientific Programming*, 1(1):51–66, 1992.

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

[91] Ian Foster, Steve Tuecke, and Stephen Taylor. A portable run-time system for PCN. Technical report, Argonne National Laboratory and CalTech, December 1992. Only available online.

[92] Ian Foster and Steven Tuecke. Parallel Programming with PCN. Technical Report ANL-91/32, Rev.1, Mathematics and Computer Science Division, Argonne National Laboratory, December 1991.

[93] H.H. Gehani and W.D. Roome. Concurrent C. *Software–Practice and Experience*, 16 (9):821–844, 1986.

[94] G. A. Geist. PVM3: Beyond network computing. In J. Volkert, editor, *Parallel Computation*, Lecture Notes in Computer Science 734, pages 194–203. Springer, 1993.

[95] R. Gerth and A. Pnueli. Rooting UNITY. In *Proceedings Fifth International Workshop on Software Specification and Design*, Pittsburgh, Penn., May 1989.

[96] J. Gibbons. *Algebras for Tree Algorithms.* D.Phil. thesis, Programming Research Group, University of Oxford, 1991.

[97] Joseph Goguen, Claude Kirchner, Héléne Kirchner, Aristide Mégrelis, José Meseguer, and Tim Winkler. An introduction to OBJ3. In *Lecture Notes in Computer Science*, volume 308, pages 258–263, 1994.

[98] Joseph Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Menlo Park, CA, August 1988.

[99] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993. also to appear as Technical Report from SRI International.

[100] K.J. Goldman. Paralation views: Abstractions for efficient scientific computing on the Connection Machine. Technical Report MIT/LCS/TM398, M.I.T. Laboratory for Computer Science, 1989.

[101] A. Gottlieb, B. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of large numbers of cooperating sequential processes. *ACM Transactions of Programming Languages and Systems*, 5(2), April 1983.

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

[103] A. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer*, 26, No.5:39–51, May 1993.

[104] A. S. Grimshaw, E. C. Loyot, S. Smootand, and J. B. Weissman. Mentat user's manual. Technical Report 91-31, University of Virginia, 3 November 1991.

[105] A. S. Grimshaw, E. C. Loyot, and J. B. Weissman. Mentat programming language. MPL Reference Manual 91-32, University of Virginia, 3 November 1991.

[106] Andrew S. Grimshaw. An introduction to parallel object-oriented programming with Mentat. Technical Report TR-91-07, University of Virginia Computer Science, 1991.

[107] A.S. Grimshaw. An introduction to parallel object-oriented programming with Mentat. Technical Report 91-07, Computer Science Department, University of Virginia, April 1991.

[108] A.S. Grimshaw. The Mentat computation model: Data-driven support for object-oriented parallel processing. Technical Report 93-30, Computer Science Department, University of Virginia, May 1993.

[109] G. Hains and C. Foisy. The data-parallel categorical abstract machine. In *PARLE93, Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science 694. Springer-Verlag, June 1993.

[110] P. Brinch Hansen. Distributed processes: A concurrent programming concept. *Communications of the ACM*, 21(11):934–940, 1978.

[111] P.J. Hatcher and M.J. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.

[112] E.A. Heinz. 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*, pages 269–276, Waseda University, Tokyo, May 1993.

[113] R. Hempel. The ANL/GMD macros (PARMACS) in Fortran for portable parallel programming using the message passing programming model – Users' Guide and Reference Manual. Technical report, GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, November 1991.

[114] R. Hempel, H.-C. Hoppe, and A. Supalov. PARMACS-6.0 library interface specification. Technical report, GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, December 1992.

[115] J. Herath, T. Yuba, and N. Saito. Dataflow computing. In *Parallel Algorithms and Architectures*, Springer Lecture Notes in Computer Science 269, pages 25–36, May 1987.

[116] High performance Fortran language specification. Available by ftp from `titan.rice.cs.edu`, January 1993.

[117] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science, 1985.

[118] P. Hudak. Para-functional programming. *IEEE Computer*, 19, No.8:60–70, 1986.

[119] P. Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27, No.5, May 1992.

[120] R. Hummel, R. Kelly, and S. Flynn Hummel. A set-based language for prototyping parallel algorithms. In *Proceedings of the Computer Architecture for Machine Perception '91 Conference*, December 1991.

[121] S. Flynn Hummel and R. Kelly. A rationale for parallel programming with sets. *Journal of Programming Languages*, 1:187–207, 1993.

[122] J. Jaffar and J.-L Lassez. Constraint logic programming. In *ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM Press, 1987.

[123] C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25, Nos.2-3:251–283, December 1995.

[124] L. Jerid, F. Andre, O. Cheron, J. L. Pazat, and T. Ernst. HPF to C-PANDORE translator. Technical Report 824, IRISA: Institut de Recherche en Informatique et Systemes Aleatoires, May 1994.

[125] G. Jones and M. Goldsmith. *Programming in Occam2.* Prentice-Hall, 1988.

[126] R.H. Halstead Jr. Parallel symbolic computing. *IEEE Computer*, 19, No.8, August 1986.

[127] L.V. Kale. The REDUCE-OR process model for parallel evaluation of logic programs. In *Proceedings of the 4th International Conference on Logic Programming*, pages 616–632, Melbourne, Australia, 1987.

[128] R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*. Elsevier Science Publishers and MIT Press, 1990.

[129] P. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors.* Pitman, 1989.

[130] M.F. Kilian. Can O-O aid massively parallel programming? In D.B. Johnson, F. Makedon, and P. Metaxas, editors, *Proceedings of the Dartmouth Institute for Advanced Graduate Study in Parallel Computation Symposium*, pages 246–256, June 1992.

[131] M.F. Kilian. *Parallel Sets: An Object-Oriented Methodology for Massively Parallel Programming.* PhD thesis, Harvard University, 1992.

[132] D.E. Knuth. Big omicron and big omega and big theta. *SIGACT*, 8, No.2:18–23, 1976.

[133] H.T. Kung. Why systolic architectures? *IEEE Computer*, 15, No.1:37–46, 1982.

[134] J.R. Larus, B. Richards, and G. Viswanathan. C**: A large-grain, object-oriented, data-parallel programming language. Technical Report TR1126, University of Wisconsin-Madison, November 1992.

[135] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, 1996.

[136] U. Lechner, C. Lengauer, and M. Wirsing. An Object-Oriented Airport: Specification and Refinement in Maude. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications*, volume 906 of *Lecture Notes in Computer Science*, pages 351–367. Springer-Verlag, Berlin, 1995.

[137] C.W. Keßler and H. Seidl. Integrating synchronous and asynchronous paradigms: The Fork95 parallel programming language. In *Massively Parallel Programming Models*, Berlin, October 1995, October 1995. IEEE Computer Society Press.

[138] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[139] P. Lincoln, N. Martí-Oiliet, and J. Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. Technical Report SRI-CSL-94-11, SRI, May 1994.

[140] B. Liskov. Implementation of Argus. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 111–122. ACM Press, 1987.

[141] G. Lobe, P. Lu, S. Melax, I. Parsons, J. Schaeffer, C. Smith, and D. Szafron. The Enterprise model for developing distributed applications. Technical Report 92–20, Department of Computing Science, University of Alberta, November 1992.

[142] G. Malcolm. *Algebraic Data Types and Program Transformation.* PhD thesis, Rijksuniversiteit Groningen, September 1990.

[143] W. F. McColl. An architecture independent programming model for scalable parallel computing. In J. Ferrante and A. J. G. Hey, editors, *Portability and Performance for Parallel Processors.* Wiley, 1994.

[144] W.F. McColl. General purpose parallel computing. In A.M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, Cambridge International Series on Parallel Computation, pages 337–391. Cambridge University Press, Cambridge, 1993.

[145] W.F. McColl. Special purpose parallel computing. In A.M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, Cambridge International Series on Parallel Computation, pages 261–336. Cambridge University Press, Cambridge, 1993.

[146] W.F. McColl. Bulk synchronous parallel computing. In *Second Workshop on Abstract Models for Parallel Computation*. Oxford University Press, 1994.

[147] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. Sisal: Streams and iteration in a single assignment language: Reference manual 1.2. Technical Report M-146, Rev.1, Lawrence Livermore National Laboratory, March 1985.

[148] J.R. McGraw. Parallel functional programming in Sisal: Fictions, facts, and future. In *Advanced Workshop, Programming Tools for Parallel Machines*, June 1993. Also available as Laurence Livermore National Laboratories Technical Report UCRL-JC-114360.

[149] P. Mehrotra and M. Haines. An overview of the Opus language and runtime system. Technical Report 94-39, NASA ICASE, May 1994.

[150] Mentat. Mentat tutorial. ftp://uvacs.cs.virginia.edu/pub/mentat/tutorial.ps.Z, 1994.

[151] J. Meseguer and T. Winkler. Parallel programming in Maude. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, pages 253–293. Springer Lecture Notes in Computer Science 574, June 1991.

[152] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. Technical Report SRI-CSL-92-08, SRI International, July 1992.

[153] Message Passing Interface Forum. MPI: A message passing interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society, 1993.

[154] T. More. On the development of array theory. Technical report, IBM Cambridge Scientific Center, 1986.

[155] B. Mukherjee, G. Eisenhauer, and K. Ghosh. A machine independent interface for lightweight threads. *ACM Operating Systems Review*, 28(1):33–47, January 1994.

[156] L. M. R. Mullin. *A Mathematics of Arrays*. Doctoral dissertation, Syracuse University, Syracuse, N.Y., December 1988.

[157] D.A. Mundie and D.A. Fisher. Parallel processing in Ada. *IEEE Computer*, C-19 No.8:20–25, 1986.

[158] L. Mussat. Parallel programming with bags. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, pages 203–218. Springer Lecture Notes in Computer Science 574, June 1991.

[159] P. Newton and J.C. Browne. The CODE2.0 graphical parallel programming language. In *Proceedings of the ACM International Conference on Supercomputing*, July 1992.

[160] M.O. Noakes and W.J. Dally. System design of the J-Machine. In *Proceedings of the Sixth MIT Conference on Advanced Research in VLSI*, pages 179–194. MIT Press, 1990.

[161] Ronald Peierls and Graham Campbell. ALMS - programming tools for coupling application codes in a network environment. In *Proceedings of the Heterogeneous Network-Based Concurrent Computing Workshop*, Tallahassee, FL, October 1991. Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from `ftp.scri.fsu.edu` in directory `pub/parallel-workshop.91`.

[162] L.M. Pereira and R. Nasr. Delta-Prolog: A distributed logic programming language. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 283–291, Tokyo, November 1984.

[163] S.L. Peyton-Jones, C. Clack, and N. Harris. GRIP – a parallel graph reduction machine. Technical report, Department of Computer Science, University of London, 1987.

[164] S.L. Peyton-Jones and David Lester. *Implementing Functional Programming Languages.* Prentice-Hall International Series in Computer Science, 1992.

[165] M.J. Quinn and P.J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, pages 69–76, September 1990.

[166] F.A. Rabhi and G.A. Manson. Experiments with a transputer-based parallel graph reduction machine. *Concurrency Practice and Experience*, 3(4):413–422, August 1991.

[167] M. Radestock and S. Eisenbach. What do you get from a π-calculus semantics? In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe*, number 817 in Lecture Notes in Computer Science, pages 635–647. Springer-Verlag, 1994.

[168] S. Radha and C. Muthukrishnan. A Portable Implementation of Unity on Von Neumann Machines. *Computer Languages*, 18(1):17–30, 1992.

[169] S. Raina. Software controlled shared virtual memory management on a transputer based multiprocessor. In D. L. Fielding, editor, *Transputer Research and Applications 4*, pages 143–152, Amsterdam, 1990. IOS Press.

[170] A.G. Ranade. *Fluent Parallel Computation.* PhD thesis, Yale University, 1989.

[171] G. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming.* MIT Press, 1989.

[172] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of the POPL '91 Conference*, pages 333–352. ACM Press, 1991.

[173] F. Seutter. CEPROL, a cellular programming language. *Parallel Computing*, 2:327–333, 1985.

[174] E. Shapiro. Concurrent Prolog: A progress report. *IEEE Computer*, 19:44–58, August 1986.

[175] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[176] T.J. Sheffler. *Match and Move, an Approach to Data Parallel Computing.* PhD thesis, Carnegie-Mellon, October 1992. Appears as Report CMU-CS-92-203.

[177] P. Singh. Graphs as a categorical data type. Master's thesis, Computing and Information Science, Queen's University, Kingston, Canada, 1993.

[178] S.K. Skedzielewski. Sisal. In B.K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 105–158. ACM Press Frontier Series, 1991.

[179] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51, December 1990.

[180] D.B. Skillicorn. Categorical data types. In *Second Workshop on Abstract Models for Parallel Computation*, pages 155–168, Oxford University Press, 1994.

[181] D.B. Skillicorn. *Foundations of Parallel Programming.* Cambridge Series in Parallel Computation. Cambridge University Press, 1994.

[182] D.B. Skillicorn. Communication skeletons. In M. Kara, J.R. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 163–178, Leeds, April 1996. IOS Press.

[183] D.B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28(1):65–83, July 1995.

[184] D.B. Skillicorn and D. Talia. *Programming Languages for Parallel Processing.* IEEE Computer Society Press, 1994.

[185] G. Spezzano and D. Talia. CARPET: A programming language for parallel cellular processing. In *Proceedings European School on Parallel Programming Environments 96*, Alpe d'Huez, France, April 1996.

[186] J.M. Spivey. A categorical approach to the theory of lists. In *Mathematics of Program Construction*, pages 399–408. Springer-Verlag Lecture Notes in Computer Science 375, June 1989.

[187] G.L. Steele Jr. High Performance Fortran: Status report. In *Proceeding of a Workshop on Languages, Compilers and Run-Time Enviroments for Distributed Memory Multiprocessors, appeared as SIGPLAN Notices, Vol 28, No. 1, January 1993*, pages 1–4, September 1992.

[188] V. S. Sunderam. PVM: A framework for parallel distributed computing. Technical Report ORNL/TM-11375, Dept. of Math and Computer Science, Emory University, Oak Ridge National Lab, February 1990. Also *Concurrency: Practice and Experience*, 2(4):315–349, Dec. 1990.

[189] Vaidy Sunderam. Concurrent computing with PVM. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, December 1992. Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.92.

[190] D. Szafron, J. Schaeffer, P.S. Wong, E. Chan, P. Lu, and C. Smith. Enterprise: An interactive graphical programming environment for distributed software. Available by ftp from `cs.ualberta.ca`, 1991.

[191] D. Talia. A survey of PARLOG and Concurrent Prolog: The integration of logic and parallelism. *Computer Languages*, 18(3):185–196, 1993.

[192] D. Talia. Parallel logic programming systems on multicomputers. *Journal of Programming Languages*, 2(1):77–87, March 1994.

[193] S. J. Thompson. Formulating Haskell. Technical Report No. 29/92, Computing Laboratory, University of Kent, Canterbury, UK, 1992.

[194] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, January 1993. Also Rice COMP TR-93-199.

[195] K. Ueda. Guarded Horn clauses. Technical Report TR-103, ICOT, Tokyo, 1985.

[196] L.G. Valiant. Bulk synchronous parallel computers. Technical Report TR-08-89, Computer Science, Harvard University, 1989.

[197] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[198] L.G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*. Elsevier Science Publishers and MIT Press, 1990.

[199] P. van Hentenryck. Parallel constraint satisfaction in logic programming: Preliminary results of Chip within PEPSys. In *Proceedings of the 6th International Congress on Logic Programming*, pages 165–180. Mit Press, Cambridge, Mass., 1989.

[200] E. Violard. A mathematical theory and its environment for parallel programming. *Parallel Processing Letters*, 4, No.3:313–328, 1994.

[201] T. Wilkinson, T. Stiemerling, P. Osmon, A. Saulsbury, and P. Kelly. Angel: a proposed multiprocessor operating system kernel. In *Proceedings of the European Workshops on Parallel Computing (EWPC'92), 23-24 March 1992, Barcelona , Spain*, pages 316–319, 1992.

[202] D.S. Wills. Pi: A parallel architecture interface for multi-model execution. Technical Report AI-TR-1245, MIT Artificial Intelligence Laboratory, 1990.

[203] T. Winkler. Programming in OBJ and Maude. In P. E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, Lecture Notes in Computer Science 693, pages 229–277. Springer-Verlag, Berlin, 1993.

[204] Allan Yang and Young-il Choo. Parallel-program transformation using a metalanguage. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1991.

[205] Allan Yang and Young-il Choo. Formal derivation of an efficient parallel Gauss-Seidel method on a mesh of processors. In *Proceedings of the 6th International Parallel Processing Symposium*. IEEE Computer Society Press, March 1992.

[206] Allan Yang and Young-il Choo. Metalinguistic features for formal parallel-program transformation. In *Proceedings of the 4th IEEE International Conference on Computer Languages.* IEEE Computer Society Press, April 1992.

[207] S. Ericsson Zenith. The axiomatic characterization of Ease. In *Linda-Like Systems and their Implementation*, pages 143–152. Edinburgh Parallel Computing Centre, TR91-13, 1991.

[208] S. Ericsson Zenith. A rationale for programming with Ease. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, pages 147–156. Springer Lecture Notes in Computer Science 574, June 1991.

[209] S. Ericsson Zenith. Ease: the model and its implementation. In *Proceeding of a Workshop on Languages, Compilers and Run-Time Enviroments for Distributed Memory Multiprocessors, appeared as SIGPLAN Notices, Vol 28, No. 1, January 1993*, page 87, September 1992.

[210] Steven Ericsson Zenith. Programming with Ease. Centre de Recherche en Informatique, École Nationale Supérieure des Mines de Paris, September 20, 1991.