How to Write Parallel Programs: A Guide to the Perplexed

NICHOLAS CARRIERO AND DAVID GELERNTER

Department of Computer Science, Yale University, New Haven, Connecticut 06520

We present a framework for parallel programming, based on three conceptual classes for understanding parallelism and three programming paradigms for implementing parallel programs. The conceptual classes are result parallelism, which centers on parallel computation of all elements in a data structure; agenda parallelism, which specifies an agenda of tasks for parallel execution; and specialist parallelism, in which specialist agents solve problems cooperatively. The programming paradigms center on live data structures that transform themselves into result data structures; distributed data structures that are accessible to many processes simultaneously; and message passing, in which all data objects are encapsulated within explicitly communicating processes. There is a rough correspondence between the conceptual classes and the programming methods, as we discuss. We begin by outlining the basic conceptual classes and programming paradigms, and by sketching an example solution under each of the three paradigms. The final section develops a simple example in greater detail, presenting and explaining code and discussing its performance on two commercial parallel computers, an 18-node shared-memory multiprocessor, and a 64-node distributed-memory hypercube. The middle section bridges the gap between the abstract and the practical by giving an overview of how the basic paradigms are implemented.

We focus on the paradigms, not on machine architecture or programming languages: The programming methods we discuss are useful on many kinds of parallel machine, and each can be expressed in several different parallel programming languages. Our programming discussion and the examples use the parallel language C-Linda for several reasons: The main paradigms are all simple to express in Linda; efficient Linda implementations exist on a wide variety of parallel machines; and a wide variety of parallel programs have been written in Linda.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.2 [**Programming Languages**]: Language Classifications—*parallel languages*; D.3.3 [**Programming Languages**]: Concurrent Programming Structures; E.1.m [**Data Structures**]: Miscellaneous—*distributed data structures; live data structures*

General Terms: Algorithms, Program Design, Languages

Additional Key Words and Phrases: Linda, parallel programming methodology, parallelism

INTRODUCTION

How do we build programs using parallel algorithms? On a spectrum of basic approaches, three primary points deserve special mention: We can use result parallelism, agenda parallelism, or specialist parallelism, terms we define. Corresponding to these basic approaches are three parallel *programming methods*—practical techniques for translating concepts into working programs; we can use message passing,

This working was supported by National Science Foundation SBIR Grant ISI-8704025 and by National Science Foundation Grants CCR-8601920, CCR-8657615, and ONR N00014-86-K-0310.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 0360-0300/89/0900-0323 $00.75

CONTENTS

distributed data structures, or live data structures.[1] Each programming method involves a different view of the role of processes and the distribution of data in a parallel program. The basic conceptual approaches and programming methods we have mentioned are not provably the only ones possible. But empirically they cover all examples we have encountered in the research literature and in our own programming experience.

Our goal here is to explain the conceptual classes, the programming methods, and the mapping between them. Section 1 explains the basic classes and methods, and sketches an example program under each of the three methods. Section 2 bridges the gap between the abstract and the practical by giving an overview of how these methods are implemented. Section 3 develops a simple example in greater detail, presenting and explaining code.

In presenting and explaining programming methods, we rely on the high-level parallel language C-Linda. Linda[2] is a language-independent set of operations that, when integrated into some base language, yields a high-level parallel dialect. C-Linda uses C; Fortran-Linda exists as well. Other groups are working on other languages[3] as Linda hosts. Our main topic is not Linda, any more than Pascal is the main topic in "Introductory Programming with Pascal" books. But we do need to present the basics of Linda programming. Linda is a good choice in this context for three reasons. (1) *Linda is flexible*: It supports all three programming methods in a straightforward fashion. This is important precisely because programming paradigms, *not* programming languages, are the topic here. The only way to factor language issues out of the discussion (at least partially) is to choose one language that will allow us to investigate all approaches. (2) *Efficient Linda implementations are available on commercial parallel machines.* We are discussing real (not theoretical) techniques, and for readers who want to investigate them, efficient Linda systems exist.[4] (3) *Linda has been used in a wide variety of programming experiments*—which give us a

[1] Although these methods are well-known, the latter two terms are not. Discussions of parallel programming methodology to date have been largely ad hoc, and as a result, the latter two categories have no generally accepted names. In fact, they are rarely recognized as categories at all.

[2] Linda is a trademark of Scientific Computing Associates, New Haven.

[3] Among them, Scheme, PostScript (see Leler [1989]), and C++; Borrman et al. [1988] describe a Modula-2 Linda, and Matsuoka and Kawai [1988] describe an object-oriented Linda variant.

[4] Linda has been implemented on shared-memory parallel computers like the Encore Multimax, Sequent Balance and Symmetry, and Alliant FX/8; on distributed memory computers like the Intel iPSC-2 hypercube; and on a VAX/VMS local-area network. Several independent commercial implementations now in progress—for example, at Cogent Research, Human Devices, and Topologix—will expand the range of supported architectures. Other groups have ports underway or planned to the Trollius operating system, to the BBN Butterfly running Mach, and to the NCUBE; Xu [1988] describes the design of a reliable Linda system based on Argus. A simulator that runs on Sun workstations also exists. The range of machines on which Linda is supported will be expanding significantly in coming months. Linda systems are distributed commercially by Scientific Computing Associates, New Haven.

basis in experience for discussing the strengths and weaknesses of various approaches. There are Linda applications for numerical problems like matrix multiplication, LU decomposition, sparse factorization [Ashcraft et al. 1989] and linear programming, and for parallel string comparison, database search, circuit simulation, ray tracing [Musgrave and Mandelbrot 1989], expert systems [Gelernter 1989], parameter sensitivity analysis, charged particle transport [Whiteside and Leichter 1988], traveling salesman, and others. We will refer to several of these programs in the following discussion.

1. CONCEPTS AND METHODS

How do we write parallel programs? For each conceptual class, there is a natural programming method; each method relates to the others in well-defined ways (i.e., programs using method x can be transformed into programs using method y by following well-defined steps). We will therefore develop the following approach to parallel programming:

To write a parallel program, (1) choose the concept class that is most natural for the problem; (2) write a program using the method that is most natural for that conceptual class; and (3) if the resulting program is not acceptably efficient, transform it methodically into a more efficient version by switching from a more-natural method to a more-efficient one.

First we explain the concepts—result, agenda, and specialist parallelism. Then we explain the methods: live data structures, distributed structures, and message passing. Finally, we discuss the relationship between concepts and methods, and give an example.

1.1 Conceptual Classes

We can envision parallelism in terms of a program's *result*, a program's *agenda of activities*, or an *ensemble of specialists* that collectively constitute the program. We begin with an analogy.

Suppose you want to build a house. Parallelism—using many people on the job—is the obvious approach. But there are several different ways in which parallelism might enter.

First, we might envision parallelism by starting with the finished product, the *result*. The result can be divided into many separate components: front, rear and side walls, interior walls, foundation, roof, and so on. After breaking the result into components, we might proceed to build all components simultaneously, assembling them as they are completed; we assign one worker to the foundation, one to the front exterior wall, one to each side wall and so on. All workers start simultaneously. Separate workers set to work laying the foundation, framing each exterior wall and building a roof assembly. They all proceed in parallel, up to the point where work on one component can't proceed until another is finished. In sum, each worker is assigned to *produce one piece of the result*, and they all work in parallel up to the natural restrictions imposed by the problem. This is the *result-parallel* approach.

At the other end of the spectrum, we might envision parallelism by starting with the crew of workers who will do the building. We note that house building requires a collection of separate skills: We need surveyors, excavators, foundation builders, carpenters, roofers and so on. We assemble a construction crew in which each skill is represented by a separate specialist worker. They all start simultaneously, but initially most workers will have to wait around. Once the project is well underway, however, many skills (hence many workers) will be called into play simultaneously: The carpenter (building forms) and the foundation builders work together, and concurrently, the roofer can be shingling while the plumber is installing fixtures and the electrician is wiring, and so on. Although a single carpenter does all the woodwork, many other tasks will overlap and proceed simultaneously with the carpenter's. This approach is particularly suited to *pipelined* jobs—jobs that require the production or transformation of a series of identical objects. If we are building a group of houses, carpenters can work on one house while foundation builders work on a second and surveyors on a third. But this strategy will often yield parallelism even when the job is defined in terms of a single object, as it does in the case of the construction of a single house. In sum, each worker is assigned to *perform one specified kind of work*, and they all work in parallel up to the natural restrictions imposed by the problem. This is the *specialist-parallel* approach.

Finally, we might envision parallelism in terms of an agenda of activities that must be completed in order to build a house. We write out a sequential agenda and carry it out in order, but at each stage we assign many workers to the current activity. We need a foundation, then we need a frame, then we need a roof, then we need wallboard and perhaps plastering, and so on. We assemble a work team of generalists, each member capable of performing any construction step. First, everyone pitches in and builds the foundation; then, the same group sets to work on the framing; then they build the roof; then some of them work on plumbing while others (randomly chosen) do the wiring; and so on. In sum, each worker is assigned to *help out with the current item on the agenda*, and they all work in parallel up to the natural restrictions imposed by the problem. This is the *agenda-parallel* approach.

The boundaries between the three classes can sometimes be fuzzy, and we will often mix elements of several approaches in getting a particular job done. A specialist approach might make secondary use of agenda parallelism, for example, by assigning a team of workers to some specialty—the team of carpenters, for example, might execute the "carpentry agenda" in agenda-parallel style. It is nonetheless a subtle but essential point that *these three approaches represent three clearly separate ways of thinking about the problem*:

Result parallelism focuses on the shape of the finished product; specialist parallelism focuses on the makeup of the work crew; and agenda parallelism focuses on the list of tasks to be performed.

These three conceptual classes apply to software as well. In particular,

(1) we can plan a parallel application around the data structure yielded as the ultimate result, and we get parallelism by computing all elements of the result simultaneously;
(2) we can plan an application around a particular agenda of activities and then assign many workers to each step; or
(3) we can plan an application around an ensemble of specialists connected into a logical network of some kind; parallelism results from all nodes of the logical network (all specialists) being active simultaneously.

How do we know what kind of parallelism, what conceptual class, to use? Consider the house-building analogy again. In effect, all three classes are (or have been) used in building houses. Factory-built housing is assembled at the site using prebuilt modules—walls, a roof assembly, staircases, and so on; all these components were assembled separately and (in theory) simultaneously back at the factory. This is a form of result parallelism in action. "Barn raisings" evidently consisted of a group of workers turning its attention to each of a list of tasks in turn, a form of agenda parallelism. But some form of specialist parallelism, usually with secondary agenda parallelism, seems like the most natural choice for house building: Each worker (or team) has a specialty, and parallelism arises in the first instance when many separate specialities operate simultaneously, secondarily when the many (in effect) identical workers on one team cooperate on the agenda.

In software as well, certain approaches tend to be more natural for certain problems. The choice depends on the problem to be solved. In some cases, one choice is immediate. In others, two or all three approaches might be equally natural. This multiplicity of choices might be regarded as confusing or off-putting; we would rather see it as symptomatic of the fact that parallelism is in many cases so abundant that the programmer can take his choice about how to harvest it.

In many cases, the easiest way to design a parallel program is to think of the resulting data structure—*result parallelism*. The programmer asks himself (1) is my program intended to produce some multiple-element data structure as its result (or can it be conceived in these terms)? If so, (2) can I specify exactly how each element of the resulting structure depends on the rest and on the input? If so, it's easy (given knowledge of the appropriate programming methods) to write a result-parallel program. Broadly speaking, such a program reads as follows: "Build a data structure in such-and-such a shape; attempt to determine the value of all elements of this structure simultaneously, where the value of each element is determined by such-and-such a computation. Terminate when all values are known." It may be that the elements of the result structure are completely independent—no element depends on any other. If so, all computations start simultaneously and proceed in parallel. It may also be that some elements can't be computed until certain other values are known. In this case, all element computations *start* simultaneously, but some immediately get stuck. They remain stuck until the values they rely on have been computed, and then proceed.

Consider a simple example: We have two n-element vectors, A and B, and need to compute their sum S. A result-parallel program reads as follows: "Construct an n-element vector S; to determine the ith element of S, add the ith element of A to the ith element of B." The elements of S are completely independent. No addition depends on any other addition. All additions accordingly start simultaneously and go forward in parallel.

More interesting cases involve computations in which there are dependencies among elements of the result data structure. We discuss an example in the next section.

Result parallelism is a good starting point for any problem whose goal is to produce a series of values with predictable organization and interdependencies, but not every problem meets this criterion. Consider a program that produces output whose shape and format depend on the input: a program to format text or translate code in parallel, for example, whose output may be a string of bytes and (perhaps) a set of tables, of unpredictable size and shape. Consider a program in which (conceptually) a *single* object is transformed repeatedly: an LU decomposition or linear programming problem, for example, in which a given matrix is repeatedly transformed in place. Consider a program that is executed not for value, but for effect: a real-time monitor-and-control program or an operating system, for example.

Agenda parallelism involves a transformation or series of transformations to be applied to all elements of some set in parallel. The most flexible embodiment of this type of parallelism is the master–worker paradigm. In a master–worker program, a master process initializes the computation and creates a collection of identical worker processes. Each worker process is capable of performing any step in the computation. Workers repeatedly seek a task to perform, perform the selected task, and repeat; when no tasks remain, the program (or this step) is finished. The program executes in the same way no matter how many workers there are, so long as there is at least one. The same program might be executed with 1, 10, and 1000 workers in three consecutive runs. If tasks are distributed on the fly, this structure is naturally load-balancing: While one worker is tied up with a time-consuming task, another might execute a dozen shorter task assignments.

For example, suppose we have a database of employee records and need to identify the employee with, say, the lowest ratio of salary to dependents. Given a record Q, the function $r(Q)$ computes this ratio. The agenda is simple: "Apply function r to all records in the database; return the identity of the record for which r is minimum." We can structure this application as a master–worker program in a natural way: The master fills a bag with data objects, each representing one employee record. Each worker repeatedly withdraws a record from the bag, computes r, and sends the result back to the master. The master keeps track of the minimum-so-far and, when all tasks are complete, reports the answer.

Specialist parallelism involves programs that are conceived in terms of a logical network. They arise when an algorithm or a system to be modeled is best understood as a network in which each node executes a relatively autonomous computation and internode communication follows predictable paths. The network may reflect a physical model or the logical structure of an algorithm (e.g., as in a pipelined or systolic computation). Network-style solutions are particularly transparent and natural when there is a physical system to be modeled. Consider a circuit simulator, for example, modeled by a parallel program in which each circuit element is realized by a separate process. There are also problems that partition naturally into separate realms of responsibility, with clearly defined intercommunication channels; further on we discuss a "cooperating experts" type of heuristic monitor that uses this kind of organization. In the last section, we discuss a pipeline type of algorithm, an algorithm understood as a sequence of steps applied to a stream of input values, with each stage of the pipe transforming a datum and handing it forward.

For example, suppose a nationwide trucking company needs to produce a large number of estimates for travel time between two points, given current estimates for road conditions, weather, and traffic. We might design a specialist-parallel program as follows: We embody a map of the continental United States in a logical network; each state is represented by its own node in the network. The Wyoming node is responsible for staying up-to-date on travel conditions in and expected transit time through Wyoming, and so forth. To estimate travel time from New Hampshire to Arkansas, we plan out a route and include a representation of this route within a data object representing a truck. We hand the "truck" to New Hampshire, which estimates its travel time through New Hampshire and then hands the truck to the next state along its route. Eventually the "truck" reaches Arkansas, which prints out the final estimate for its transit time. Note that large numbers of trucks may be moving through the logical network at any one time.

We conclude this survey of conceptual classes by mentioning two special classes that we will not deal with further, *data parallelism* and *speculative parallelism* (sometimes called *or-parallelism*). Data parallelism is a restricted kind of agenda parallelism: It involves a series of transformations each applied to all elements of a data structure simultaneously. If we start with an agenda of activities in which each item requires that a transformation be applied to a data structure, the agenda-parallel program we would derive would in effect be an example of data parallelism. Empirically, data parallelism is usually associated with synchronous machines (e.g., MPP [Goodyear Aerospace Co. 1979] and the Connection Machine [Hillis and Steele 1986]) and is accordingly tied to an implementation in which transformations are applied to all elements of some data structure not merely concurrently but *synchronously*: At each instant, each active worker is applying the same step of the same transformation to its own assigned piece of the structure. In this paper our focus is restricted to techniques that are used on general-purpose *asynchronous* parallel machines.[5] In "speculative parallelism," often associated with logic programming, but also significant in, for example, parallel algorithms for heuristic search (e.g., parallel alpha-beta search on game trees [Marsland and Campbell 1982]), a collection of parallel activities is undertaken with the understanding that some may ultimately prove to be unnecessary to the final result. Whenever a program's structure includes clauses like "try x, and if x fails, try y" (and so on through a list of other alternatives), we can get parallelism by working on x, y, and any other alternatives simultaneously. If and when x fails, y is already underway. We understand this under our schematization as another special form of agenda parallelism: Many workers are thrown simultaneously into the completion of a list of tasks, with the understanding that, ultimately, only one of the results produced will be incorporated in the finished product.

[5] This focus can be taken as arbitrary, but there is a reason for it. At present, synchronous or SIMD machines are rare and expensive; asynchronous machines can be built cheaply and are increasingly widespread. The imminent arrival of parallel workstations will add to the flood.

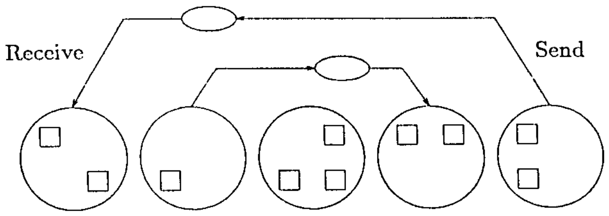

Figure 1. Message passing: The process structure—the number of processes and their relationships—determines the program structure. A collection of concurrent processes communicate by exchanging messages; every data object is locked inside some process. (Processes are round, data objects square, and messages oval.)

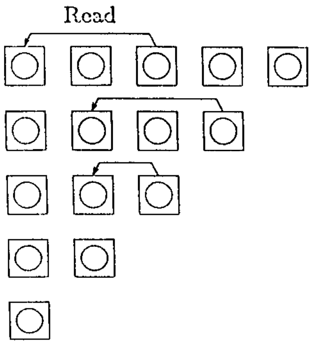

Figure 2. Live data structures: The result data structure—the number of its elements and their relationship—determines the program structure. Every concurrent process is locked inside a data object; it is responsible, in other words, for computing that element and only that element. Communication is no longer a matter of explicit "send message" and "receive message" operations; when a process needs to consult the value produced by some other process, it simply reads the data object within which the process is trapped.

1.2 The Programming Methods

In message passing, we create many concurrent processes and enclose every data structure within some process; processes communicate by exchanging messages. In message-passing methods, no data objects are shared among processes. Each process may access its own local set of private data objects only. In order to communicate, processes must send data objects from one local space to another; to accomplish this, the programmer must explicitly include send-data and receive-data operations in his code (Figure 1).

At the other extreme, we dispense with processes as conceptually independent entities and build a program in the shape of the data structure that will ultimately be yielded as the result. Each element of this data structure is implicitly a separate process, which will turn into a data object upon termination. To communicate, these implicit processes don't exchange messages; they simply "refer" to each other as elements of some data structure. Thus, if process P has data for Q, it doesn't send a message to Q; it terminates, yielding a value, and Q reads this value directly. These are "live-data-structure" programs (Figure 2).

The message-passing and live-data-structure approaches are similar in the sense that, in each, all data objects are distributed among the concurrent processes; there are no global, shared structures. In message passing, though, processes are created by the programmer *explicitly*; they communicate *explicitly* and may send values *repeatedly* to other processes. In a live-data-structure program, processes are created *implicitly* in the course of building a data structure; they communicate *implicitly* by referring to the elements of a data structure, and each process produces only a *single* datum for use by the rest of the program. Details will become clear as we discuss examples.

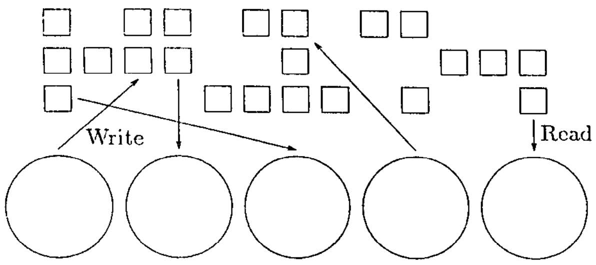

Figure 3. Distributed data structures: Concurrent processes *and* data objects figure as autonomous parts of the program structure. Processes communicate by reading and writing shared data objects.

Between the extremes of allowing all data to be absorbed into the process structure (message passing) or all processes to melt into data structures (live data structures), there is an intermediate strategy that maintains the distinction between a group of data objects and a group of processes. Because shared data objects exist, processes may communicate and coordinate by leaving data in shared objects. These are "distributed-data-structure" programs (Figure 3).

1.3 Where to Use Each

It's clear that result parallelism is naturally expressed in a live-data-structure program. For example, returning to the vector-sum program, the core of such an application is a live data structure. The live structure is an n-element vector called S; trapped inside each element of S is a process that computes $A[i] + B[i]$ for the appropriate i. When a process is complete, it vanishes, leaving behind only the value it was charged to compute.

Specialist parallelism is a good match to message passing: We can build such a program under message passing by creating one process for each network node and using messages to implement communication over edges. For example, returning to the travel-time program, we implement each node of the logical network by a process; trucks are represented by messages. To introduce a truck into the network at New Hampshire, we send New Hampshire a "new truck" message; the message includes a representation of the truck's route. New Hampshire computes an estimated transit time and sends another message, including both the route and the time-en-route-so-far to the next process along the route. Note that, with lots of trucks in the network, many messages may converge on a process simultaneously. Clearly, then, we need some method for queuing or buffering messages until a process can get around to dealing with them. Most message-passing systems have some kind of buffering mechanism built in.

Even when such a network model exists, though, message passing will sometimes be inconvenient in the absence of backup support from distributed data structures. If every node in the network needs to refer to a collection of global status variables, those globals can only be stored (absent distributed data structures) as some node's local variables, forcing all access to be channeled through a custodian process. Such an arrangement can be conceptually inept and can lead to bottlenecks.

Agenda parallelism maps naturally onto distributed-data-structure methods. Agenda parallelism requires that many workers set to work on what is, in effect, a single job. In general, any worker will be willing to pick up any subtask. Results developed by one worker will often be needed by others, but one worker usually won't know (and won't care) what the others are doing. Under the circumstances, it's far more convenient to leave results in a distributed data structure, where any worker who wants them can take them, than to worry about sending messages to particular recipients. Consider also the dynamics of a master–worker program, the kind of program that represents the most flexible embodiment of agenda parallelism. We have a collection of workers and need to distribute tasks, generally on the fly. Where do we keep the tasks? Again, a distributed data structure is the most natural solution. If the subtasks that make up an agenda item are strictly parallel, with no necessary ordering among them, the master process can store task descriptors in a distributed *bag* structure; workers repeatedly reach into the bag and grab a task. In some cases, tasks should be started in a certain order (even if many can be processed simultaneously); in this case, tasks will be stored in some form of distributed queue structure.

For example, we discussed a parallel database search carried out in terms of the master–worker model. The bag into which the master process drops employee records is naturally implemented as a distributed data structure—as a structure, in other words, that is directly accessible to the worker processes and the master.

1.4 An Example

Consider a naive n-body simulator: On each iteration of the simulation, we calculate the prevailing forces between each body and all the rest, and update each body's position accordingly.[6] We will consider this problem in the same way we considered house building. Once again, we can conceive of result-based, agenda-based, and specialist-based approaches to a parallel solution.

We can start with a result-based approach. It's easy to restate the problem description as follows: Suppose we are given n bodies and want to run q iterations of our simulation; compute a matrix M such that $M[i, j]$ is the position of the ith body after the jth iteration. The zeroth column of the matrix gives the starting position, and the last column, the final position, of each body. We have now carried out step 1 in the design of a live data structure. The second step is to define each entry in terms of other entries. We can write a function $position(i, j)$ that computes the position of body i on iteration j; clearly $position(i, j)$ will depend on the positions of each body at the previous iteration—will depend, that is, on the entries in column $j - 1$ of the matrix. Given a suitable programming language, we're finished: We build a program in which $M[i, j]$ is defined to be the value yielded by $position(i, j)$. Each invocation of *position* constitutes an implicit process, and all such invocations are activated and begin execution simultaneously. Of course, computation of the second column can't proceed until values are available for the first column: We must assume that, if some invocation of *position* refers to $M[x, y]$ and $M[x, y]$ is still unknown, we wait for a value and then proceed. Thus, the zeroth column's values are given at initialization time, whereupon all values in the first column can be computed in parallel, then the second column, and so forth (Figure 4).

Note that, if the forces are symmetric, this program does more work than necessary, because the force between A and B is the same as the force between B and A. This is a minor problem that we could correct, but our goal here is to outline the simplest possible approach.

We can also approach this problem in terms of agenda parallelism. The task agenda states "repeatedly apply the transformation *compute next position* to all bodies in the set." To write the program, we might create a master process and have it generate n initial task descriptors, one for each body. On the first iteration, each worker in a group of identical worker processes repeatedly grabs a task descriptor and computes the next position of the corresponding body, until the pile of task descriptors is used up (and all bodies have advanced to their new positions); likewise for each subsequent iteration. A single worker will require time proportional to n^2 to complete each iteration; two workers together will finish each iteration in time proportional to $n^2/2$, and so on. We

[6] There is a better ($O(n)$) approach to solving the n-body problem, developed by Greengard and Rokhlin [1987] of Yale; the new algorithm can be parallelized, but to keep things simple, we use the old approach as a basis for this discussion.

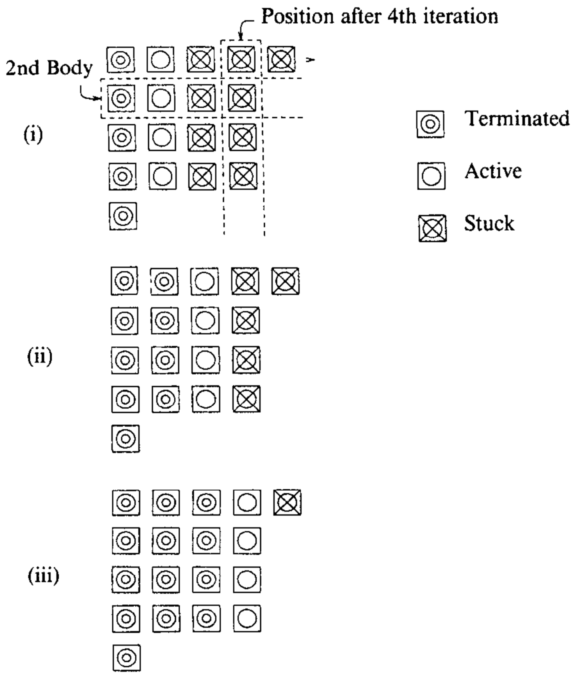

Figure 4. A live-data-structure approach to the n-body problem. To begin, we build an $n \times q$ matrix and install a process inside each element. The process trapped in element $M[i, j]$ will compute the position of the ith body after the jth iteration, by referring to the previous column, in which each body's last-known position will appear. The processes in column j are stuck until the processes in column $j - 1$ terminate, at which point all of column j can be computed in parallel. Thus, each column computes in parallel until values are known for the entire matrix.

can store information about each body's position at the last iteration in a distributed table structure, where each worker can refer to it directly (Figure 5).

Finally, we might use a specialist-parallel approach: We create a series of processes, each one specializing in a single body—that is, each responsible for computing a single body's current position throughout the simulation. At the start of each iteration, each process informs each other process by message of the current position of its body. All processes are behaving in the same way; it follows that, at the start of each iteration, each process *sends data to* but also *receives data from* each other process. The data included in the incoming crop of messages are sufficient to allow each process to compute a new position for its body. It does so, and the cycle repeats (Figure 6). (A similar but slightly cleaned up version of such a program is described by Seitz [1985].)

1.5 How Do the Three Techniques Relate?

The methodology we are developing requires (1) starting with a conceptual class that is natural to the problem, (2) writing a program using the programming method that is natural to the class, and then, (3) if necessary, transforming the initial program into a more efficient variant that uses some other method. If a natural approach also turns out to be an efficient approach, then obviously no transformation is necessary. If not, it's essential to understand the relationships between the techniques and

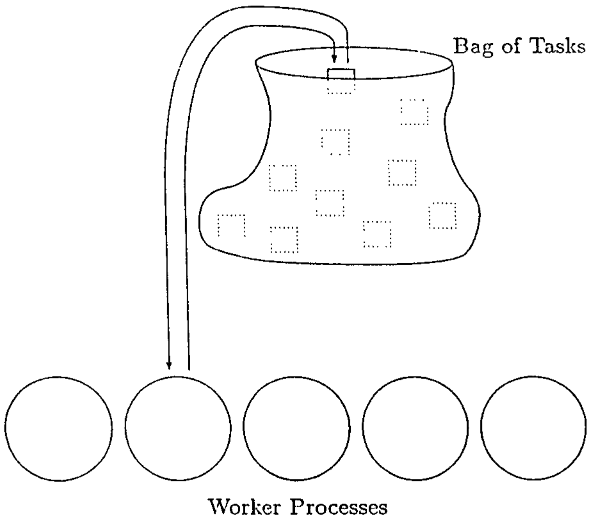

Figure 5. A distributed-data-structure version. At each iteration, workers repeatedly pull a task out of a distributed bag and compute the corresponding body's new position, referring to a distributed table for information on the previous position of each body. After each computation, a worker might update the table (without erasing information on previous positions, which may still be needed) or might send newly computed data to a master process, which updates the table in a single sweep at the end of each iteration.

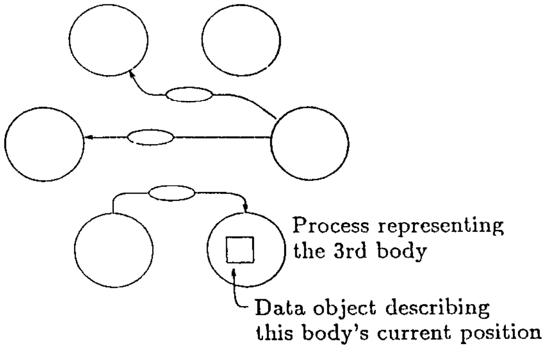
Process representing the 3rd body
Data object describing this body's current position

Figure 6. The message-passing version. Whereas the live-data-structure program creates nq processes (q was the number of iterations, and there are n bodies) and the distributed-data-structure program creates any number of workers it chooses, this message-passing program creates exactly n processes, one for each body. In each of the other two versions, processes refer to *global data structures* when they need information on the previous positions of each body. (In the live-data-structure version, this global data structure was the "live" structure in which the processes themselves were embedded.) But in the message-passing version, no process has access to any data object external to itself. Processes keep each other informed by sending messages back and forth.

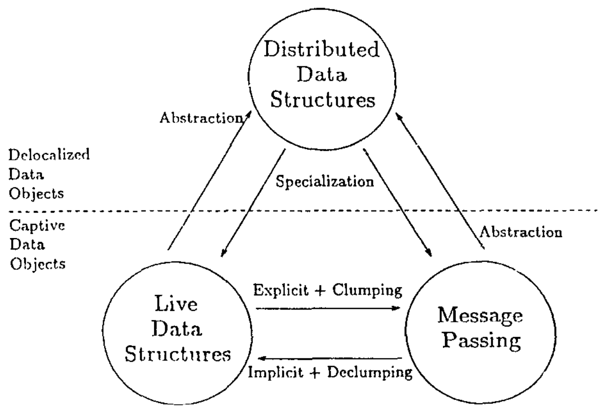

Figure 7. The game of parallelism.

the performance implications of each. After describing the relationships in general, we discuss one case of this transformation-for-efficiency in some detail.

1.5.1 The Relationships

The main relationships are shown in Figure 7. Both live data structures and message passing center on *captive data objects*: Every data object is permanently associated with some process. Distributed-data-structure techniques center on *delocalized* data objects, objects not associated with any one process, freely floating about on their own. We can transform a live-data-structure or a message-passing program into a distributed structure program by using *abstraction*: We cut the data objects free of their associated processes and put them in a distributed data structure instead. Processes are no longer required to fix their attention on a single object or group of objects; they can range freely. To move from a distributed structure to a live-data-structure or a message-passing program, we use *specialization*: We take each object and bind it to some process.

It is clear from the foregoing that live data structures and message passing are strongly related, but there are also some important differences. To move from the former to the latter, we need to make communication *explicit*, and we may optionally use *clumping*. A process in a live-data-structure program has no need to communicate information explicitly to any other process. It merely terminates, yielding a value. In a message-passing program, a process with data to convey must execute an explicit "send-message" operation. When a live-data-structure process requires a data value from some other process, it references a data structure; a message-passing process will be required to execute an explicit "receive-message" operation.

Why contemplate a move from live data structures to message passing, if the latter technique is merely a verbose version of the former? It isn't; message-passing techniques offer an added degree of freedom, which is available via "clumping." A process in a live-data-structure program develops a value and then dies. It can't live on to develop and publish another value. In message passing, a process can develop as many values as it chooses and disseminate them in messages whenever it likes. It can develop a whole series of values during a program's lifetime. Hence, clumping: We may be able to let a single message-passing process do the work of a whole collection of live-data-structure processes.

Table 1 summarizes the relationships in a different way. It presents an approximate and general characterization of the three classes. There are counterexamples in every category, but it's useful to summarize the spectrum of process and program types, from the large number of simple, tightly coordinated processes that usually occur in result parallelism, through

Table 1. Rough Characterization of the Three Classes

Complexity of processes			
	Skills	Tasks	
Result	One	One	Simpler
Specialist	One	Many	
Agenda	Many	Many	More complex

Program structure		
	Number of processes	Coordination
Result	High	Tight
Specialist	Moderate	Moderate
Agenda	Adjustable	Loose

the typically smaller collection of more complex, more loosely coupled processes in agenda parallelism.

1.5.2 *Using Abstraction and Then Specialization to Transform a Live-Data-Structure Program*

Having described some transformations in the abstract, what good are they? We can walk many paths through the simple network in Figure 7, and we can't describe them all in detail. We take up one significant case, describing the procedure in general and presenting an example; we close the section with a brief examination of another interesting case.

Suppose we have a problem that seems most naturally handled using result parallelism. We write the appropriate live-data-structure program, but it performs poorly, so we need to apply some transformations.

First, why discuss this particular case? When the problem is suitable, a live-data-structure program is likely to be rather easy to design and concise to express. It's likely to have a great deal of parallelism (with the precise degree depending, obviously, on the size of the result structure and the dependencies among elements). But it may also run poorly on most current-generation parallel machines, because the live-data-structure approach tends to produce *fine-grained* programs—programs that create a large number of processes each one of which does relatively little computing. Concretely, if our resulting data structure is, say, a ten-thousand-element matrix, this approach will implicitly create ten thousand processes. There is no reason in theory why this kind of program cannot be supported efficiently, but on most current parallel computers, there are substantial overheads associated with creating and coordinating large numbers of processes. This is particularly true on distributed-memory machines, but even on shared-memory machines that support lightweight processes, the potential gain from parallelism can be overwhelmed by huge numbers of processes, each performing a trivial computation.

If a live-data-structure program performs well, we're finished; if it does not, a more efficient program is easily produced by *abstracting* to a distributed data-structure version of the same algorithm. We replace the live data structure with a passive one and raise the processes one level in the conceptual scheme: Each process *fills in* many elements, rather than *becoming* a single element. We might create one hundred processes and have each process compute one hundred elements of the result. The resulting program is coarser grained then the original—the programmer decides how many processes to create and can choose a reasonable number. We avoid the overhead associated with huge numbers of processes.

This second version of the program may still not be efficient enough, however. It requires that each process read and write a single data structure, which must be stored in some form of logically shared memory. Accesses to a shared memory will be more expensive than access to local structures.

Ordinarily this isn't a problem; distributed data-structure programs can be supported efficiently even on distributed-memory (e.g., hypercube) machines. But, for some communication-intensive applications, and particularly on distributed-memory machines, we may need to go further in order to produce an efficient program. We might produce a maximally efficient third version of the program by using *specialization* to move from distributed data structures to message passing. We break the distributed data structure into chunks and hand each chunk to the process with greatest interest in that chunk. Instead of a shared distributed data structure, we now have a collection of local data structures, each encapsulated within and only accessible to a single process. When some process needs access to a "foreign chunk," a part of the data structure that it doesn't hold locally, it must send a message to the process that does hold the interesting chunk, asking that an update be performed or a data value returned. This is a nuisance and usually results in an ugly program, but it eliminates direct references to any shared data structures.

Under this scheme of things, we can see a neat and well-defined relationship among our three programming methods. We start with an elegant and easily discovered but potentially inefficient solution using live data structures, move on via abstraction to a more efficient distributed-data-structure solution, and finally end up via specialization at a low-overhead message-passing program. (We might alternatively have gone directly from live data structures to message passing via clumping.)

There is nothing inevitable about this procedure. In many cases, it's either inappropriate or unnecessary. It is inappropriate if live data structures are *not* a natural starting point. It is unnecessary if a live-data-structure program runs well from the start. It is partially unnecessary if abstraction leads to a distributed-data-structure program that runs well; in this case, there's nothing to be gained by performing the final transformation, and something to be lost (because the message-passing program will probably be substantially more complicated than the distributed-data-structure version). It's also true that message-passing programs are not always more efficient than distributed-data-structure versions; often they are, but there are cases in which distributed data structures are the optimal approach.

1.5.3 An Example

For example, returning to the n-body simulator, we discussed a live-data-structure version; we also developed distributed-data-structure and message-passing versions, independently. We could have used the live-data-structure version as a basis for abstraction and specialization as well.

Our live-data-structure program created $n \times q$ processes, each of which computed a single invocation of *position* and then terminated. We can create a distributed-data-structure program by *abstraction*. M is now a distributed data structure—a passive structure, directly accessible to all processes in the program. Its zeroth column holds the initial position of each body; the rest of the matrix is blank. We create k processes and put each in charge of filling in one band of the matrix. Each band is filled in column-by-column. In filling in the jth column, processes refer to the position values recorded in the j − 1st column. We now have a program in which number of processes is under direct programmer control; we can run the program with two or three processes if this seems reasonable (as it might if we have only two or three processors available). We have achieved lower process-management overheads, but the new program was easy to develop from the original and will probably be only slightly less concise and comprehensible.

Finally, we can use *specialization* to produce a minimal-overhead message-passing program. Each process is given one band of M to store in its own local variable space; M no longer exists as a single structure. Since processes can no longer refer directly to the position values computed on the last iteration, these values must be disseminated in messages. At the end of each iteration, processes exchange messages; messages hold the positions computed by each process on the last iteration. We have now achieved low process-management overhead and also eliminated the overhead of referring to a shared distributed data structure. But the cost is considerable: The code for this last version will be substantially more complicated and messier than the previous one, because each process will need to conclude each iteration with a message-exchange operation in which messages are sent, other messages are received, and local tables are updated. We have also crossed an important conceptual threshold: Communication in the first two solutions was conceived in terms of *references to data structures*, a technique that is basic to all programming. But the last version relies on message passing for communication, thus substituting a new kind of operation that is conceptually in a different class from standard programming techniques.

1.5.4 When to Abstract and Specialize

How do we know whether we need to use abstraction or to move onward to a message-passing program? The decision is strictly pragmatic; it depends on the application, the programming system, and the parallel machine. Consider one concrete datum: Using C-Linda on current parallel machines, specialization leading to a message-passing program is rarely necessary. Most problems have distributed-data-structure solutions that perform well. In this context, though, abstraction to a distributed-data-structure program usually *is* necessary to get an efficient program.

1.5.5 Another Path through the Network: Abstraction from Message Passing

When live-data-structure solutions are natural, they may involve too many processes and too much overhead, so we use abstraction to get a distributed-data-structure program. It's also possible for a message-passing, network-style program to be natural, but to involve too many processes and too much interprocess communication, in which case we can use abstraction, again, to move from *message passing* to distributed data structures. Suppose, for example, that we want to simulate a ten-thousand-element circuit. It is natural to envision one process for each circuit element, with processes exchanging messages to simulate the propagation of signals between circuit elements. But this might lead to a high-overhead program that runs poorly. Abstraction, again, allows us to create fewer processes and put each in charge of one segment of a distributed data structure that represents the network state as a whole.

In sum, there are many paths that a programmer might choose to walk though the state diagram shown in Figure 7. But the game itself is simple: Start at whatever point is most natural, write a program, understand its performance, and then, if necessary, follow the "efficiency" edges until you reach an acceptable stopping place.

1.6 Where Are the Basic Techniques Supported?

Although it is our intention in this article to survey programming techniques, not programming systems, a brief guide to the languages and systems in which the basic techniques occur may be helpful.

Message passing is by far the most widespread of the basic models; it occurs in many different guises and linguistic contexts. The best-known of message-passing languages is Hoare's influential fragment CSP [Hoare 1978], which inspired a complete language called Occam [May 1983]. CSP and Occam are based on a radically tight-knit kind of message passing: Both the sending and the receiving of a message are *synchronous* operations. In both languages, a process with a message to send blocks until the designated receiver has taken delivery. CSP and Occam are *static* languages as well: They do not allow new processes to be created dynamically as a program executes. CSP and Occam are for these reasons not expressive enough to support the full range of message-passing-type programs we discuss here.

Monitor and remote-procedure-call languages and systems are another subtype within the message-passing category (with a qualification we note below). In these systems, communication is modeled on procedure call: One process communicates with another by invoking a procedure defined within some other process or within a passive, globally accessible module. This kind of quasi-procedure call amounts to a specialized form of message passing: Arguments to the procedure are shipped out in one message, results duly returned in another. The qualification mentioned above is that, in certain cases, systems of this sort are used for quasi-distributed-data-structure programs. A global data object can be encapsulated in a module and then manipulated by remotely invoked procedures. (The same kind of thing is possible in any message-passing system, but is more convenient given a procedure-style communication interface.) Why *quasi*-distributed data structures? As we understand the term, a distributed data structure is directly accessible to many parallel processes *simultaneously*. (Clearly we may sometimes need to enforce sequential access to avoid corrupting data, but in general, many read operations may go forward simultaneously—and many write operations that affect separate and independent parts of the same structure may also proceed simultaneously, for example, many independent writes to separate elements of a single matrix.) Languages in this class support data objects that are global to many processes, but in general, they allow processes one-at-a-time access only. Nor do they support plain distributed data objects; a global object must be packaged with a set of access procedures.

Monitors were first described by Hoare [1974] and have been used as a basis for many concurrent programming languages, for example Concurrent Pascal [Brinch Hansen 1975], Mesa [Lampson and Redell 1980] and Modula [Wirth 1977]. (A *concurrent* language, unlike a parallel language, assumes that multiple processes inhabit the same address space.) Fairly recently they have been revived for use in parallel programming, in the form of parallel object-oriented programming languages (e.g., Emerald [Jul et al. 88]). A form of remote procedure call underlies Ada [U.S. Department of Defense 1982]; Birrell and Nelson's RPC kernel [Birrell and Nelson 1984] is an efficient systems-level implementation.

Another variant of message passing centers on the use of *streams*: Senders (in effect) append messages to the end of a message stream, and receivers inspect the stream's head. This form of communication was first proposed by Kahn [1974] and forms the basis for communication in most concurrent logic languages (e.g., Concurrent Prolog [Shapiro 1987] and Parlog [Ringwood 1988]) and in functional language extended with constructs for explicit communication (e.g., [Henderson 1982]).

Message passing of one form or another appears as a communication method in many operating systems—for example, the V kernel [Cheriton and Zwaenpoel 1985], Mach [Young et al. 1987] and Amoeba [Mullender and Tanenbaum 1986].

Distributed data structures are less frequently encountered. The term was introduced in the context of Linda [Carriero et al. 1986]. Distributed data structures form the de facto basis of a number of specialized FORTRAN that revolve around parallel do-loops, for example, Jordan's Force system [Jordan 1986]. In this kind of system, parallelism is created mainly by specifying parallel loops—loops in which iterations are executed simultaneously instead of sequentially. Separate loop iterations communicate through distributed structures that are adaptations of standard FORTRAN structures. Distributed data structures are central in Dally's CST [Dally 1988] and Bal and Tanenbaum's Orca [Bal and Tanenbaum 1987], and are supported in MultiLISP [Halstead 1985] as well.

Live data structures are a central technique in several languages that support so-called nonstrict data structures—data structures that can be accessed before they are fully defined. Id Nouveau [Nikhil et al. 1986], MultiLISP [Halstead 1985], and Symmetric LISP [Gelernter et al. 1987] are examples. This same idea forms the implicit conceptual basis for the large class of functional languages intended for parallel programming (e.g., ParAlfl [Hudak 1986], Sisal [Lee et al. 1988] and Crystal [Chen 1986]). Programs in these languages consist of a series of equations specifying values to be bound to a series of names. One equation may depend on the values returned by other equations in the set; we can solve all equations simultaneously, subject to the operational restriction that an equation referring to a not-yet-computed value cannot proceed until this value is available. The equivalent program in live-data-structure terms would use each equation to specify the value of one element in a live data structure.

2. PROGRAMMING TECHNIQUES FOR PARALLELISM

We have discussed conceptual classes and general methods. We turn now to the practical question: How do we build working parallel programs? In this section we sketch implementations of the pieces out of which parallel programs are constructed.

We start with a systematic investigation of distributed data structures. We give an overview of the most important kinds of distributed structures, when each is used, and how each is implemented. This first part of the discussion should equip readers with a reasonable tool kit for building distributed-data-structure programs. Of course we intend to discuss all three programming methods, but the other two are easily derived from a knowledge of distributed data structures, as we discuss in the following sections. We arrive at message passing by restricting ourselves to a small and specialized class of distributed structures. We arrive at live data structures by building distributed structures out of processes instead of passive data objects.

2.1 Linda

Linda consists of a few simple operations that embody the "tuple-space" model of parallel programming. A base language with the addition of these tuple-space operations yields a parallel-programming dialect. To write parallel programs, programmers must be able to create and coordinate multiple execution threads. Linda is a model of process creation and coordination that is *orthogonal* to the base language in which it is embedded. The Linda model doesn't care *how* the multiple execution threads in a Linda program compute what they compute; it deals only with how these execution threads (which it sees as so many black boxes) are created and how they can be organized into a coherent program. The following paragraphs give a basic introduction. Linda is discussed in greater detail and contrasted with a series of other approaches in Carriero and Gelernter [1989].

The Linda model is a *memory* model. Linda memory (called *tuple space* or *TS*) consists of a collection of logical tuples. There are two kinds of tuples: Process tuples are under active evaluation; data tuples are passive. The process tuples (which are all executing simultaneously) exchange data by generating, reading, and consuming data tuples. A process tuple that is finished executing turns into a data tuple, indistinguishable from other data tuples.

There are four basic TS operations, **out**, **in**, **rd**, and **eval**, and two variant forms, **inp** and **rdp**. **out(*t*)** causes tuple ***t*** to be added to TS; the executing process continues immediately. **in(*s*)** causes some tuple ***t*** that matches template ***s*** to be withdrawn from TS; the values of the actuals in ***t*** are assigned to the formals in ***s***, and the executing process continues. If no matching ***t*** is available when **in(*s*)** executes, the executing process suspends until one is, then proceeds as before. If many matching ***t***'s are available, one is chosen arbitrarily. **rd(*s*)** is the same as **in(*s*)**, with actuals assigned to formals as before, except that the matched tuple remains in TS. Predicate versions of **in** and **rd**, **inp** and **rdp**, attempt to locate a matching tuple and return 0 if they fail; otherwise, they return 1 and perform actual-to-formal assignment as described above. (If and only if it can be shown that, irrespective of relative process speeds, a matching tuple must have been added to TS before the execution of **inp** or **rdp** and cannot have been withdrawn by any other process until the **inp** or **rdp** is complete, the predicate operations are *guaranteed* to find a matching tuple.) **eval(*t*)** is the same as **out(*t*)**, except that ***t*** is evaluated after rather than before it enters TS; **eval** implicitly forks a new process to perform the evaluation. When computation of ***t*** is complete, ***t*** becomes an ordinary passive tuple, which may be **in**ed or read like any other tuple.

A tuple exists independently of the process that created it, and in fact many tuples may exist independently of many creators and may collectively form a data structure in TS. It's convenient to build data structures out of tuples because tuples are referenced associatively, somewhat like the tuples in a relational database. A tuple is a series of typed fields, for example, **("a string", 15.01, 17, "another string")** or **(0, 1)**. Executing the **out** statements

out("a string", 15.01, 17, "another string")

and

out(0, 1)

causes these tuples to be generated and added to TS. (The process executing **out** continues immediately.) An **in** or **rd** statement specifies a template for matching: Any values included in the **in** or **rd** must be matched identically; formal parameters must be matched by values of the same type. (It is also possible for formals to appear in tuples, in which case a matching **in** or **rd** must have a type-consonant value in the corresponding position.) Consider the following statement:

in("a string", ? f, ? i, "another string")

Executing this statement causes a search of TS for tuples of four elements, first element **"a string"** and last element **"another string"**, middle two elements of the same types as variables **f** and **i** respectively. When a matching tuple is found, it is removed, the value of its second field is assigned to **f** and its third field to **i**. The read statement, for example,

rd("a string", ? f, ? i, "another string")

works in the same way, except that the matched tuple is not removed. The values of its middle two fields are assigned to **f** and **i** as before, but the tuple remains in TS.

A tuple created using **eval** resolves into an ordinary data tuple. Consider the following statement:

eval("e", 7, exp(7)).

It creates a three-element "live tuple" and continues immediately; the live tuple sets to work computing the values of the string **"e"**, the integer **7**, and the function call **exp(7)**. The first two computations are trivial (they yield **"e"** and **7**); the third ultimately yields the value of **e** to the seventh power. Expressions that appear as arguments to **eval** inherit bindings from the environment of the **eval**-executing process for whatever names they cite explicitly and for read-only globals initialized at compile time. Thus, executing **eval("Q", f(x, y))** implicitly creates a new process and evaluates **"Q"** and **f(x, y)** in a context in which the names **f**, **y**, and **x** have the same values they had in the environment of the process that executed **eval**. The names of any variables that happen to be free in **f**, on the other hand, were *not* cited explicitly by the **eval** statement, and no bindings are inherited for them.[7] The statement

rd("e", 7, ? value))

might be used to read the tuple generated by this **eval**, once the live tuple has resolved to a passive data tuple—that is, once the necessary computing has been accomplished. (Executed before this point, it blocks until the active computation has resolved into a passive tuple.)

2.2 The Basic Distributed Data Structures

We can divide conventional "undistributed" data structures into three categories: (1) structures whose elements are identical or indistinguishable, (2) structures whose elements are distinguished by name, and (3) structures whose elements are distinguished by position. It's useful to subdivide the last category: (3a) structures whose elements are "random accessed" by position

[7] Future versions of the system may disallow the inheritance by eval-created processes of read-only globals. There are simple transformations from programs that rely on this feature to ones that do not.

and (3b) structures whose elements are accessed under some ordering.

In the world of sequential programming, category (1) is unimportant. A *set* of identical or indistinguishable elements qualifies for inclusion, but such objects are rare in sequential programming. Category (2) includes records, objects instantiated from class definitions, sets and multisets with distinguishable elements, associative memories, Prolog-style assertion collections, and other related objects. Category (3a) consists mainly of arrays and other structures stored in arrays, and category (3b) includes lists, trees, graphs, and so on. Obviously the groupings are not disjoint, and there are structures that can claim membership in several.

The distributed versions of these structures don't always play the same roles as their sequential analogs. Factors with no conventional analogs can furthermore play a major role in building distributed structures. *Synchronization* concerns arising from the fact that a distributed structure is accessible to many asynchronous processes simultaneously form the most important example. Notwithstanding, every conventional category has a distributed analog.

2.2.1 Structures with Identical or Indistinguishable Elements

The most basic of distributed data structures is a lock or semaphore. In Linda, a counting semaphore is precisely a collection of identical elements. To execute a *V* on a semaphore **"sem"**,

out("sem");

to execute a *P*,

in("sem").

To initialize the semaphore's value to *n*, execute **out("sem")** *n* times. Semaphores aren't used heavily in most parallel applications (as opposed to most concurrent systems), but they do arise occasionally; we elaborate in the next section.

A *bag* is a data structure that defines two operations: "add an element" and "withdraw an element." The elements in this case need not be identical, but are treated in a way that makes them indistinguishable. Bags are unimportant in sequential programming, but extremely important to parallel programming. The simplest kind of replicated-worker program depends on a bag of tasks. Tasks are added to the bag using

out("task", TaskDescription)

and withdrawn using

in("task", ? NewTask)

A simple example: consider a program that needs to apply some test to every element of a large file. (In one experiment we've done, the large file holds DNA sequences, and the test is "compare each sequence to a designated target sequence"; we need to discover which sequences in a database are "closest" under a string-matching-like algorithm to some designated sequence.) The program consists of a master and workers; the task is "compare the target sequence to sequence *s*." To withdraw a sequence from the bag, workers execute

in("sequence", ? seq)

The master reads sequences from the file and adds them to the bag (using a low-watermark algorithm to ensure that the bag doesn't overfill [Carriero and Gelernter 1988]); to add a sequence, it executes

out("sequence", seq)

Note that we can regard the set of all **("sequence", value)** tuples as a bag of indistinguishable 2-tuples; alternatively, we can say that "sequence" is the name of a bag of **values** and that **out("sequence", seq)** means "add **seq** to the bag called **"sequence"**."

Consider an example with some of the attributes of each of the two previous cases. Suppose we want to turn a conventional loop, for example

```
for (⟨loop control⟩)
  ⟨something⟩
```

into a parallel loop—all instances of *something* execute simultaneously. This construct is popular in parallel FORTRAN variants. One simple way to do the transformation has two steps: First we define a function **something( )** that executes one instance of the loop body and returns, say, 1. Then we rewrite as follows:

```
for (⟨loop control⟩)
  eval("this loop", something( ));
for (⟨loop control⟩)
  in("this loop", 1);
```

We have, first, created *n* processes; each is an active tuple that will resolve, when the function call **something( )** terminates, to a passive tuple of the form **("this loop", 1)**. Second, we collect the *n* passive result tuples. These *n* may be regarded as a bag or, equivalently, as a single counting semaphore that is V'ed implicitly by each process as it terminates. A trivial modification to this example would permit each iteration to "return" a result.

2.2.2 Name-Accessed Structures

Parallel applications often require access to a collection of related elements distinguished by name. Such a collection resembles a Pascal record or a C "struct." We can store each element in a tuple of the form

```
(name, value)
```

To read such a "record field," processes use **rd(name, ? val)**; to update it,

```
in(name, ? old);
out(name, new).
```

Consider, for example, a program that acts as a real-time expert monitor: Given a mass of incoming data, the system will post notices when significant state changes occur; it must also respond to user-initiated queries respecting any aspect of the current state. One software architecture for such systems is the so-called "process lattice": a hierarchical ensemble of concurrent processes, with processes at the bottom level wired directly to external sensors, and processes at higher levels responsible for increasingly more complex or abstract states. The model defines a simple internal information-flow protocol, with data flowing up and queries downward, and each node directly accessible at the user interface. Our prototype deals with hemodynamic monitoring in intensive care units [Carriero and Gelernter 1988]. Each process records its current state in a named tuple, whence it can be consulted directly by any interested party. The process that implements the "hypovolemia" decision procedure, for example, can update its state as follows:

```
in("hypovolemia", ? old)
out("hypovolemia", new).
```

Any process interested in hypovolemia can read the state directly. These state-describing tuples can thus be collectively regarded as a kind of distributed record, whose elements can be separately updated and consulted.

As always, the synchronization characteristics of distributed structures distinguish them from conventional counterparts. Any process attempting to read the hypovolemia field while this field is being updated will block until the update is complete and the tuple is reinstated. Processes occasionally need to wait until some event occurs; Linda's associative matching makes this convenient to program. For example, some parallel applications rely on "barrier synchronization": Each process within some group must wait at a barrier until all processes in the group have reached the barrier; then all can proceed. If the group contains *n* processes, we set up a barrier called **barrier-37** by executing

```
out("barrier-37", n)
```

Upon reaching the barrier point, each process in the group executes (under one simple implementation)

```
in("barrier-37", ? val);
out("barrier-37", val-1);
rd("barrier-37", 0).
```

That is, each process decrements the value of the field called **barrier-37** and then waits until its value becomes 0.

2.2.3 Position-Accessed Structures

Distributed arrays are central to parallel applications in many contexts. They can be programmed as tuples of the form (*Array name, index fields, value*). Thus, **("V", 14, 123.5)** holds the 14th element of vector *V*, **("A", 12, 18, 5, 123.5)** holds one element of the three-dimensional array *A*, and so forth. For example, one way to multiply matrices *A* and *B*, yielding *C*, is to store *A* and *B* as a collection of rectangular blocks, one block per tuple, and to define a task as the computation of one block of the product matrix. Thus, *A* is stored in TS as a series of tuples of the form

```
("A", 1, 1, ⟨first block of A⟩)
("A", 1, 2, ⟨second block of A⟩)
```

and *B* likewise. Worker processes repeatedly consult and update a *next-task* tuple, which steps though the product array pointing to the next block to be computed. If some worker's task at some point is to compute the *i*, *j*th block of the product, it reads all the blocks in *A*'s *i*th row band and *B*'s *j*th column band, using a statement like

```
for (next=0; next < ColBlocks;
    next++)
  rd("A", i, next, ? RowBand[next])
```

for *A* and similarly for *B*; then, using **RowBand** and **ColBand**, it computes the elements of *C*'s *i*, *j*th block and concludes the task step by executing the

```
out("C", i, j, Product)
```

Thus, **"C"** is a distributed array as well, constructed in parallel by the worker processes and stored as a series of tuples of the form

```
("C", 1, 1, ⟨first block of C⟩)
("C", 1, 2, ⟨second block of C⟩).
```

It's worth commenting at this point on the obvious fact that a programmer who builds this kind of matrix-multiplication program is dealing with two separate schemes for representing data: the standard array structures of the base language and a tuple-based array representation. It would be simple in theory to demote the tuple-based representation to the level of assembler language generated by the compiler: Let the compiler decide which arrays are accessed by concurrent processes and must therefore be stored in TS; then have the compiler generate the appropriate Linda statements. Not hard to do—but would this be desirable?

We tend to think not. First, there are distributed data structures with no conventional analogs, as we have noted; a semaphore is the simplest example. It follows that parallel programmers will not be able to rely exclusively on conventional forms and will need to master some new structures regardless of the compiler. But it's also the case that the dichotomy between *local memory* and *all other memory* is emerging as a fundamental attribute (arguably *the* fundamental attribute) of parallel computers. Evidence suggests that programmers cannot hope to get good performance on parallel machines without grasping this dichotomy and allowing their programs to reflect it. This is an obvious point when applied to parallel architectures without physically shared memory. Processors in such a machine have much faster access to data in their local memories than to data in another processor's local memory—nonlocal data are accessible only via the network and the communication software. But hierarchical memory is also a feature of shared-memory architectures. Thus, we note an observation like the following, which deals with the BBN Butterfly shared-memory multiprocessor:

> Although the Uniform System [a BBN-supplied parallel programming environment] provides the illusion of shared memory, attempts to use it as such do not work well. Uniform System programs that have been optimized invariably block-copy their operands into local memory, do their computation locally, and block-copy out their results. . . . This being the case, it might be wise to optimize later-generation machines for very high bandwidth transfers of large blocks of data rather than single-word reads and writes as in the current Butterfly. We might end up with a computational model similar to that of LINDA . . . , with

naming and locking subsumed by the operating system and the LINDA **in**, **read** and **out** primitives implemented by very high speed block transfer hardware. [Olson 1986]

Because the dichotomy between local and nonlocal storage appears to be fundamental to parallel programming, programmers should (we believe) have a high-level, language-based model for dealing with nonlocal memory. TS provides such a model.

Returning to position-accessed distributed data structures, synchronization properties can again be significant. Consider a program to compute all primes between 1 and n (we examine several versions of this program in detail in the last section). One approach requires the construction of a distributed table containing all primes known so far. The table can be stored in tuples of the following form:

("primes", 1, 2)
("primes", 2, 3)
("primes", 3, 5)
...

A worker process may need the values of all primes up to some maximum; it reads upward through the table, using **rd** statements, until it has the values it needs. It may be the case, though, that certain values are still missing. If all table entries through the kth are needed, but currently the table stops at j for $j < k$, the statement

rd("primes", j + 1, ? val)

blocks—there is still no j + 1st element in the table. Eventually the j + 1st element will be computed, the called-for tuple will be generated, and the blocked **rd** statement will be unblocked. Processes that read past the end of the table will simply pause, in other words, until the table is extended.

Ordered or linked structures make up the second class of position-accessed data structures. It's possible to build arbitrary structures of this sort in TS; instead of linking components by address, we link by logical name. If C, for example, is a *cons* cell linking A and B, we can represent it as the tuple

("C", "cons", cell),

where **cell** is the two-element array **["A", "B"]**. If **"A"** is an atom, we might have

("A", "atom", value).

For example, consider a program that processes queries based on Boolean combinations of keywords over a large database. One way to process a complex query is to build a parse tree representing the keyword expression to be applied to the database; each node applies a subtransformation to a stream of database records produced by its inferiors—a node might and together two sorted streams, for example. All nodes run concurrently. A Linda program to accomplish this might involve workers executing a series of tasks that are in effect linked into a tree; the tuple that records each task includes "left," "right," and "parent" fields that act as pointers to other tasks [Narem 1988]. Graph structures in TS arise as well; for example, a simple shortest-path program [Gelernter et al. 1985] stores the graph to be examined one node per tuple. Each node-tuple has three fields: the name of the node, an array of neighbor nodes (Linda supports variable-sized arrays in tuples), and an array of neighbor edge-lengths.

These linked structures have been fairly peripheral in our programming experiments to date. But there *is* one class of ordered structure that is central to many of the methods we have explored, namely streams of various kinds. There are two major varieties, which we call in-streams and read-streams. In both cases, the stream is an ordered sequence of elements to which arbitrarily many processes may append. In the in-stream case, each one of arbitrarily many processes may, at any time, remove the stream's head element. If many processes try to remove an element simultaneously, access to the stream is serialized arbitrarily at run time. A process that tries to remove from an empty stream blocks until the stream becomes nonempty. In the read-stream case, arbitrarily many processes read the stream simultaneously: Each reading process reads the stream's first element, then its second element, and so on. Reading processes block, again, if the stream is empty.

In- and read-streams are easy to build in Linda. In both cases, the stream itself consists of a numbered series of tuples:

```
("strm", 1, val1)
("strm", 2, val2)
...
```

The index of the last element is kept in a tail tuple:

```
("strm", "tail", 14)
```

To append **NewElt** to **"strm"**, processes use the following:

```
in("strm", "tail", ? index);
  /* consult tail pointer */
out("strm", "tail", index+1);
out("strm", index, NewElt);
  /* add element */
```

An in-stream needs a head tuple also, to store the index of the head value (i.e., the next value to be removed); to remove from the in-stream **"strm"**, processes use

```
in("strm", "head", ? index);
  /* consult head pointer */
out("strm", "head", index+1);
in("strm", index, ? Elt);
  /* remove element */.
```

Note that, when the stream is empty, blocked processes will continue in the order in which they blocked. If the first process to block awaits the jth tuple, the next blocked process will be waiting for the $j + 1$st, and so on.

A read-stream dispenses with the head tuple. Each process reading a read-stream maintains its own local index; to read each element of the stream, we use

```
index = 1;
⟨loop⟩ {
    rd("strm", index++, ? Elt);
    ...
  }
```

As a specialization, when an in-stream is consumed by only a single process, we can again dispense with the head tuple and allow the consumer to maintain a local index. Similarly, when a stream is appended-to by only a single process, we can dispense with the tail tuple, and the producer can maintain a local index.

In practice, various specializations of in- and read-streams seem to appear more often than the fully general versions.

Consider, for example, an in-stream with a single consumer and many producers. Such a stream occurs in one version of the prime-finding program we discuss: Worker processes generate a stream each of whose elements is a block of primes; a master process removes each element of the stream, filling in a primes table as it goes.

Consider an in-stream with one producer and many consumers. In a traveling-salesman program,[8] worker processes expand subtrees within the general search tree, but these tasks are to be performed not in random order but in a particular optimized sequence. A master process writes an in-stream of tasks; worker processes repeatedly remove and perform the head task. (This structure functions, in other words, as a distributed queue.)

Consider a read-stream with one producer and many consumers. In an LU-decomposition program [Bjornson et al. 1988], each worker on each iteration reduces some collection of columns against a pivot value. A master process writes a stream of pivot values; each worker reads the stream.

2.3 Message Passing and Live Data Structures

We can write a message-passing program by sharply restricting the distributed data structures we use: In general, a message-passing program makes use only of streams. The tightly synchronized message-passing protocols in CSP, Occam and related languages represent an even more drastic restriction: Programs in these languages use

[8] Written by Henri Bal of the Vrije Universiteit in Amsterdam.

no distributed structures; they rely only (in effect) on isolated tuples.

It's simple, then, to write a message-passing program. First, we use **eval** to create one process for each node in the logical network we intend to model. Often we know the structure of the network beforehand; the first thing the program does, then, is to create all the processes it requires. (Concretely, C-Linda programs have an **lmain** function that corresponds to **main** in a *C* program. When the program starts, **lmain** is invoked automatically. If we need to use **eval** to create *n* processes immediately, the **eval** statements appear in **lmain**.) In some cases the shape of a logical network changes while a program executes; we can use **eval** to create new processes as the program runs. Having created the processes we need, we allow processes to communicate by writing and reading message streams.

Live-data-structure programs are also easy to write given the passive distributed structures we've discussed. Any distributed data structure has a live as well as a passive version. To get the live version, we simply use **eval** instead of **out** in creating tuples. For example, we've discussed streams of various kinds. Suppose we need a stream of processes instead of passive data objects. If we execute a series of statement of the form

```
eval("live stream", i, f(i)),
```

we create a group of processes in TS:

```
("live stream", 1,
                  ⟨computation of f(1)⟩)
("live stream", 2,
                  ⟨computation of f(2)⟩)
("live stream", 3,
                  ⟨computation of f(3)⟩)
...
```

If **f** is, say, the function "factorial," then this group of processes resolves into the following stream of passive tuples:

```
("live stream", 1, 1)
("live stream", 2, 2)
("live stream", 3, 6)
...
```

To write a live-data-structure program, then, we use **eval** to create one process for each element in our live structure. (Again, **lmain** will execute the appropriate **evals**.) Each process executes a function whose value may be defined in terms of other elements in the live structure. We can use ordinary **rd** or **in** statements to refer to the elements of such a data structure. If **rd** or **in** tries to find a tuple that is still under active computation, it blocks until computation is complete. Thus, a process that executes

```
rd("live stream", 1, ? x)
```

blocks until computation of **f(1)** is complete, whereupon it finds the tuple it is looking for and continues.

3. PUTTING THE DETAILS TOGETHER

Finding all primes between 1 and *n* is a good example problem for two reasons: (1) It's not significant in itself, but there are significant problems that are similar; at the same time, primes finding is simple enough to allow us to investigate the entire program in a series of cases. (2) The problem can be approached naturally under several of our conceptual classes. This gives us an opportunity to consider what is natural and what isn't natural, and how different sorts of solution can be expressed.

3.1 Result Parallelism and Live Data Structures

One way to approach the problem is by using result parallelism. We can define the result as an *n*-element vector; *j*'s entry is 1 if *j* is prime, and otherwise 0. It's easy to see how we can define entry *j* in terms of previous entries: *j* is prime if and only if there is no previous prime less than or equal to the square root of *j* that divides it.

To write a C-Linda program using this approach, we need to build a vector in TS; each element of the vector will be defined by the invocation of an **is_prime** function. The loop

```
for(i = 2; i < LIMIT; ++i) {
  eval("primes", i, is_prime(i));
}
```

creates such a vector. As discussed in Section 2.2.3, each tuple-element of the vector is labeled with its index. We can now read the *j*th element of the vector by using

```
rd("primes", j, ? ok).
```

The program is almost complete. The **is_prime(SomeIndex)** function will involve reading each element of the distributed vector through the square root of *i* and, if the corresponding element is prime and divides *i*, returning zero;[9] thus,

```
limit = sqrt((double) SomeIndex) + 1;

for (i = 2; i < limit; ++i) {
  rd("primes", i, ? ok);
  if (ok && (SomeIndex%i == 0))
    return 0;
}
return 1;
```

The only remaining problem is producing output. Suppose that the program is intended to print all primes **1** through **LIMIT**. Easily done: We simply read the distributed vector and print *i* if *i*'s entry is **1**:

```
for(i = 2; i <= LIMIT; ++i) {
  rd("primes", i, ? ok);
  if (ok) printf("%d\n", i);
}
```

The complete program[10] is shown in Figure 8.

```
lmain()
{
  int   i, ok;

  for(i = 2; i < LIMIT; ++i) {
    eval("primes", i, is_prime(i));
  }

  for(i = 2; i <= LIMIT; ++i) {
    rd("primes", i, ? ok);
    if (ok) printf("%d\n", i);
  }
}

is_prime(me)
     int        me;
{
  int           i, limit, ok;
  double        sqrt();

  limit = sqrt((double) me) + 1;

  for (i = 2; i < limit; ++i) {
    rd("primes", i, ? ok);
    if (ok && (me%i == 0)) return 0;
  }
  return 1;
}
```

Figure 8. Prime finder: Result parallelism.

3.2 Using Abstraction to Get an Efficient Version

This program is concise and elegant, and was easy to develop. It derives parallelism from the fact that, once we know whether k is prime, we can determine the primality of all numbers from $k + 1$ through k^2. But it is potentially highly inefficient: It creates large numbers of processes and requires relatively little work of each. We can use abstraction to produce a more efficient, agenda-parallel version. We reason as follows:

(1) Instead of building a live vector in TS, we can use a passive vector and create worker processes. Each worker chooses some block of vector elements and fills in the entire block. "Determine all primes from 2001 through 4000" is a typical task.

Tasks should be assigned in order: The lowest block is assigned first, then the next-lowest block, and so forth. If we have filled in the bottom block and the highest prime it contains is k, we can compute in parallel all blocks up to the block containing k^2.

How do we assign tasks in order? We could build a distributed queue of task assignments, but there is an easier way. All tasks are identical in kind; they differ only in starting point. So we can use a single tuple as a next-task pointer, as we discuss

[9] In practice, it might be cheaper for the *i*th process to compute all primes less than root of *i* itself, instead of reading them via **rd**. But we are not interested in efficiency at this stage.

[10] Users of earlier versions of C-Linda will note that, although formals used to be addresses, for example, "**? &ok**," C-Linda 2.0 assumes that formals will be variables, on analogy with the left side of assignment statements. The code examples use the new version.

in the matrix-multiplication example in Section 2.2.3. Idle workers withdraw the next-task tuple, increment it, and then reinsert it, so the next idle worker will be assigned the next block of integers to examine. In outline, each worker will execute the following:

```
while(1) {
  in("next task", ? start);
  out("next task", start + GRAIN);

  ⟨find all primes from start to start +
      GRAIN⟩
}
```

GRAIN is the size of each block. The value of **GRAIN**, which is a programmer-defined constant over each run, determines the granularity or task size of the computation. The actual code is more involved than this: Workers check for the termination condition and leave a marker value in the next-task tuple when they find it. (See the code in Figures 9 and 10 for details.)

(2) We have accomplished "abstraction," and we could stop here. But, since the goal is to produce an efficient program, there is another obvious optimization. Instead of storing a distributed bit vector with one entry for each number within the range to be searched, we could store a distributed *table* in which all primes are recorded. The ith entry of the table records the ith prime number. The table has many fewer entries than the bit vector and is therefore cheaper both in space and in access time. (To read all primes up to the square root of j will require a number of accesses proportional not to $\sqrt{j}$, but to the number of primes through $\sqrt{j}$.)

A worker examining some arbitrary block of integers doesn't know *a priori* how many primes have been found so far and therefore cannot construct table entries for new primes without additional information. We could keep a primes count in TS, but it's also reasonable to allow a master process to construct the table.

We will therefore have workers send their newly discovered batches of primes to the master process; the master process builds the table. Workers attach batches of primes to the end of an in-stream, which in turn is scanned by the master. Instead of numbering the stream using a sequence of integers, they can number stream elements using the starting integer of the interval they have just examined. Thus, the stream takes the following form:

```
("result", start, FirstBatch);
("result", start+GRAIN,
     SecondBatch);
("result", start+(2*GRAIN)
     ThirdBatch);
...
```

The master scans the stream by executing the following loop:

```
for (num = first_num; num < LIMIT;
      num += GRAIN) {
  in("result", num, ? new_primes);

  ⟨record the new batch for eventual
      output⟩;

  ⟨construct the distributed primes
      table⟩;

}.
```

This loop dismantles the stream in order, **in**ing the first element and assigning it to the variable **new_primes**, then the second element, and so on.

The master's job is now to record the results and to build the distributed primes table. The workers send prime numbers in batches; the master disassembles the batches and inserts each prime number into the distributed table. The table itself is a standard distributed array of the kind discussed previously. Each entry takes the form:

```
("primes", i, ⟨ith prime⟩,
              ⟨ith prime squared⟩).
```

We store the square of the ith prime along with the prime itself so that workers can simply read, rather than having to compute, each entry's square as they scan upward through the table. For details, see Figure 10.

(3) Again, we could have stopped at this point, but a final optimization suggests it-

```
#include "linda.h"

#define GRAIN 2000
#define LIMIT 1000000
#define NUM_INIT_PRIME  15

long primes[LIMIT/10+1] =
  {2,3, 5, 7, 11, 13, 17, 19, 23, 29, 31,  37,  41, 43,  47};
long p2[LIMIT/10+1] =
  {4,9,25,49,121,169,289,361,529,841,961,1369, 1681,1849,2209};

lmain(argc, argv)
     int argc;
     char *argv[];
{
  int eot, first_num, i, num, num_primes, num_workers;
  long new_primes[GRAIN], np2;

  num_workers = atoi(argv[1]);
  for (i = 0; i < num_workers; ++i)
    eval("worker", worker());

  num_primes = NUM_INIT_PRIME;
  first_num = primes[num_primes-1] + 2;

  out("next task", first_num);

  eot = 0;  /* becomes 1 at "end of table" -- i.e., table complete */
  for (num = first_num; num < LIMIT; num += GRAIN) {
    in("result", num, ? new_primes: size);

    for (i = 0; i < size; ++i, ++num_primes) {
      primes[num_primes] = new_primes[i];

      if (!eot) {
                  np2 = new_primes[i]*new_primes[i];
                  if (np2 > LIMIT) {
                       eot = 1;
                       np2 = -1;
                  }
      out("primes", num_primes, new_primes[i], np2);
      }
    }
  }
  /*  " ? int" means  "match any int; throw out the value" */
  for (i = 0; i < num_workers; ++i) in("worker", ? int);

  printf("%d: %d\n", num_primes, primes[num_primes-1]);
 }
```

Figure 9. Prime finder (master): Agenda parallelism.

```
worker()
{
  long count, eot, i, limit, num, num_primes, ok, start;
  long my_primes[GRAIN];

  num_primes = NUM_INIT_PRIME;

  eot = 0;
  while(1) {
    in("next task", ? num);
    if (num == -1) {
      out("next task", -1);
      return;
    }
    limit = num + GRAIN;
    out("next task", (limit > LIMIT) ? -1 : limit);
    if (limit > LIMIT) limit = LIMIT;

    start = num;
    for (count = 0; num < limit; num += 2) {
      while (!eot && num > p2[num_primes-1]) {
        rd("primes", num_primes, ? primes[num_primes], ? p2[num_primes]);
        if (p2[num_primes] < 0)
          eot = 1;
        else
          ++num_primes;
      }
      for (i = 1, ok = 1; i < num_primes; ++i) {
        if (!(num%primes[i])) {
          ok = 0;
          break;
        }
        if (num < p2[i]) break;
      }
      if (ok) {
        my_primes[count] = num;
        ++count;
      }
    }
    /* Send the control process any primes found. */
    out("result", start, my_primes: count);
  }
}
```

Figure 10. Prime finder (worker): Agenda parallelism.

self. Workers repeatedly grab task assignments and then set off to find all primes within their assigned interval. To test for the primality of k, they divide k by all primes through the square root of k; to find these primes, they refer to the distributed *primes* table. But they could save repeated references to the distributed global table by building local copies. Global references (references to objects in TS) are more expensive than local references.

Whenever a worker reads the global *primes* table, it will accordingly copy the data it finds into a local version of the table.

It now refers to the global table only when its local copy needs extending. This is an optimization similar in character to the *specialization* we described in Section 1: It saves global references by creating multiple local structures. It isn't "full specialization," though, because it doesn't eliminate the global data structure, merely economizes global references.

Workers store their local tables in two arrays of longs called **primes** and **p2** (the latter holds the square of each prime). Newly created workers inherit copies of these global arrays (declared above the master's code in Figure 9) when they are created. The notation **object: count** in a Linda operation means "the first **count** elements of the aggregate named **object**"; in an **in** or a **rd** statement, **? object: count** means that the size of the aggregate assigned to **object** should be returned in **count**.

3.3 Comments on the Agenda Version

This version of the program is substantially longer and more complicated than the original result-parallel version. On the other hand, it performs well in several widely different environments. On one processor of the shared-memory Sequent Symmetry, a sequential C program requires about 459 seconds to find all primes in the range of one to three million. Running with 12 workers and the master on 13 Symmetry processors, the C-Linda program in Figures 9 and 10 does the same job in about 43 seconds, for a speedup of about $10\frac{1}{2}$ relative to the sequential version, giving an efficiency of about 82 percent. One processor of an Intel iPSC-2 hypercube requires about 421 seconds to run the sequential C program; 1 master and 63 workers running on all 64 nodes of our machine require just under 8 seconds, for a speedup of about $52\frac{1}{2}$ and an efficiency of, again, roughly 82 percent.

If we take the same program and increase the interval to be searched in a task step by a factor of 10 (this requires a change to one line of code: We define **GRAIN** to be 20,000), the same code becomes a very coarse-grained program that can perform well on a local-area network. Running on eight Ethernet-connected IBM RTs under Unix,[11] we get roughly a 5.6-times speedup over sequential running time, for an efficiency of about 70 percent. Somewhat lower efficiencies on coarser-grained problems are still very satisfactory on local-area nets. Communication is far more expensive on a local-area net than in a parallel computer, and for this reason networks are problematic hosts for parallel programs. They are promising nonetheless because, under some circumstances, they can give us something for nothing: Many computing sites have compute-intensive problems, lack parallel computers, but have networks of occasionally underused or (on some shifts) idle workstations. Converting wasted workstation cycles into better performance on parallel programs is an attractive possibility.[12]

In comparing the agenda- to the result-parallel version, it's important to keep in mind that the more complicated and efficient program was produced by applying a series of simple transformations to the elegant original. So long as a programmer understands the basic facts in this domain—how to build live and passive distributed data structures, which operations are relatively expensive and which are cheap—the transformation process is conceptually uncomplicated and can stop at any point. In other words, programmers with the urge to polish and optimize (i.e., virtually all expert programmers) have the same kind of opportunities in parallel as in conventional programming.

Note that, for this problem, agenda parallelism is probably less natural than result parallelism. The point here is subtle, but is nonetheless worth making. The most natural agenda-parallel program for primes finding would probably have been conceived as follows: Apply T in parallel to all integers from 1 to *limit*, where T is simply "determine whether n is prime." If we understand these applications of T as completely independent, we have a program that will work and is highly parallel. It is

[11] Unix is a trademark of AT&T Bell Laboratories.

[12] The iPSC-2 system is largely the work of Robert Bjornson and the Unix LAN kernel of Mauricio Arango and Donald Berndt.

not an attractive solution, though, because it's blatantly wasteful: In determining whether j is prime, we can obviously make use of the fact that we know all previous primes through the square root of j.

The master–worker program we developed *on the basis of the result-parallel version* is more economical in approach, and we regard this version as a "made" rather than a "born" distributed-data-structure program.

3.4 Specialist Parallelism

Primes finding had a natural result-parallel solution, and we derived an agenda-parallel solution. There is a natural specialist-parallel solution as well.

The sieve of Eratosthenes is a simple prime-finding algorithm in which we imagine passing a stream of integers through a series of sieves: A 2-sieve removes multiples of 2, a 3-sieve likewise, then a 5-sieve, and so forth. An integer that has emerged successfully from the last sieve in the series is a new prime. It can be ensconced in its own sieve at the end of the line.

We can design a specialist-parallel program based on this algorithm. We imagine the program as a pipeline that lengthens as it executes. Each pipe segment implements one sieve (i.e., specializes in a single prime). The first pipe segment inputs a stream of integers and passes the residue (a stream of integers not divisible by 2) onto the next segment, which checks for multiples of 3 and so on. When the segment at the end of the pipeline finds a new prime, it extends the sieve by attaching a new segment to the end of the program.

One way to write this program is to start with a two-segment pipe. The first pipe segment generates a stream of integers; the last segment removes multiples of the last-known prime. When the last segment (the "sink") discovers a new greatest prime, it inserts a new pipe segment directly before itself in line. The newly inserted segment is given responsibility for sieving what had formerly been the greatest prime. The sink takes over responsibility for sieving the *new* greatest prime. Whenever a new prime is discovered, the process repeats.

First, how will integers be communicated between pipe segments? We can use a single-producer, single-consumer in-stream. Stream elements look like

("seg", ⟨destination⟩, ⟨stream index⟩, ⟨integer⟩).

Here, **destination** means "next pipe segment"; we can identify a pipe segment by the prime it is responsible for. Thus, a pipe segment that removes multiples of 3 expects a stream of the form

("seg", 3, ⟨stream index⟩, ⟨integer⟩).

How will we create new pipe segments? Clearly, the "sink" will use **eval**; when it creates a new segment, the sink detaches its own input stream and plugs this stream into the newly created segment. Output from the newly created segment becomes the sink's new input stream. The details are shown in Figure 11.

The code in Figure 11 produces as output merely a count of the primes discovered. It could easily have developed a table of primes and printed the table. There is a more interesting possibility as well. Each segment of the pipe is created using **eval**; hence, each segment turns into a passive tuple upon termination. Upon termination (which is signaled by sending a 0 through the pipe), we could have had each segment yield its prime. In other words, we could have had the program collapse upon termination into a data structure of the form:

("source", 1, 2)
("pipe seg", 2, 3)
("pipe seg", 3, 5)
("pipe seg", 4, 7)
. . .
("sink", MaxIndex, MaxPrime).

We could then have walked over and printed out this table.

This solution allows less parallelism than the previous one. To see why, consider the result-parallel algorithm: It allowed simultaneous checking of all primes between $k + 1$ and k^2 for each new prime k. Suppose there are p primes in this interval for some k. The previous algorithm allowed us to

```
lmain()
{
  eval("source", source());
  eval("sink", sink());
}
source()
{
  int   i, out_index = 0;

  for (i = 5; i < LIMIT; i += 2) out("seg", 3, out_index++, i);
  out("seg", 3, out_index, 0);
}
sink()
{
  int   in_index = 0, num, prime = 3, prime_count = 2;

  while(1) {
    in("seg", prime, in_index++, ? num);
    if (!num) break;
    if (num % prime) {
      ++prime_count;
      if (num*num < LIMIT) {
        eval("pipe seg", pipe_seg(prime, num, in_index));
        prime = num;
        in_index = 0;
      }
    }
  }
  printf("count: %d.\n", prime_count);
}
pipe_seg(prime, next, in_index)
     int        prime, next, in_index;
{
  int   num, out_index = 0;

  while(1) {
    in("seg", prime, in_index++, ? num);
    if (!num) {
      out("seg", next, out_index, num);
      return;
    }
    if (num % prime) out("seg", next, out_index++, num);
  }
}
```

Figure 11. Prime finder: Specialist parallelism.

discover all *p* simultaneously, but in this version they are discovered one at a time, the first prime after *k* causing the pipe to be extended by one stage, then the next prime, and so on. Because of the pipeline, "one at a time" means a rapid succession of discoveries; but the discoveries still occur sequentially.

The specialist-parallel solution is not quite as impractical as the result-parallel version, but it is impressively impractical nonetheless. Consider one data point: In searching the range from one to one thousand, the structure-parallel version is 30 times *slower* on an 18-processor Multimax than the sequential C program on a single processor. These results make an instructive demonstration of an important if largely *sub rosa* phenomenon in parallel programming. A parallel program is always costlier than a conventional, sequential version of the same algorithm: Creating and coordinating processes take time. Running an *efficient* parallel program on many processors allows us to recoup the overhead and come out ahead in absolute terms; thus, the master–worker primes-finding experiment demonstrates absolute speedup over a comparable sequential program. An *in*efficient parallel program may demonstrate impressive *relative* speedup—it may run faster on many processors than on one, which is true of the specialist-parallel program under discussion—without ever amortizing the "overhead of parallelization" and achieving *absolute* speedup. Readers should be alert to this point in assessing data on parallel programming experiments.

For this problem, our specialist-parallel approach is clearly impractical. Those are the breaks. But readers should keep in mind that exactly the same program structure *could* be practical if each process had more computing to do. In some related problem areas, this would be the case. Furthermore, the dynamic, fine-grained character of this program makes it an interesting but not altogether typical example of the message-passing *genre*. A static, coarse-grained message-passing program (e.g., of the sort we described in the context of the n-body problem) would be programmed using many of the same techniques, but would be far more efficient.

3.5 Simplicity

The prime-finding example raises a final interesting question. Our efficient parallel version is significantly more complicated than a conventional, sequential prime finder. Does parallelism mean that programming necessarily becomes a more complicated activity than it used to be?

It's clear that a "simple problem" in the sequential world is not necessarily still simple in the parallel world. But, to grasp the implications of this fact, we need to consider two others: Many problems that are "sequentially simple" are also simple in parallel, and some problems that are complex under sequential assumptions are *simpler* in parallel. Computing prime numbers efficiently is the kind of problem that, because of substantial interdependence among subtasks and the "sequential" nature of the underlying algorithm (larger primes are determined on the basis of smaller primes), is substantially trickier in parallel than it is sequentially.[13] Many of the most successful and widely used applications of parallelism, on the other hand, involve problems that are much simpler than this to parallelize, and in these cases the parallel codes are much closer to the sequential originals. These problems generally parallelize at a fairly coarse grain and require only limited intertask communication. They use exactly the same techniques we have developed here. They are less interesting as case studies, but often of great practical significance. A final category of application is simpler in parallel than it would be as a sequential program. The ICU monitor program discussed in Section 2.2.2 is a good example: It's most naturally expressed as an ensemble of concurrently active experts. This kind of application may sound esoteric, but it's our view that programs of this sort, programs involving large collections of heterogeneous experts communicating via a simple global protocol, will become increasingly widespread and significant.

What's the bottom line? It would be foolish to deny, when all is said and done, that parallelism *does* make programming a more complex skill to master. Expanding the range of choices makes any job harder; expanding the capabilities of a machine (whether a hardware or a software ma-

[13] As we discuss in [Carriero and Gelernter 1988], the problem was brought to our attention by a researcher who found it anything but simple to write an efficient parallel solution to a related primes-finding problem.

chine) often results in a more complicated design. (Compare a color to a black-and-white television, by way of analogy, or a modern workstation to a PDP-8.)

The parallel primes finder is a more complicated (software) machine than the sequential version, but it has acquired the new and valuable capacity to spread itself over a large collection of processors, where the sequential version can't cope with more than one. Good programmers will have no difficulty learning the new techniques involved, and once they do, they will have access to a new and powerful class of software machinery.

4. CONCLUSIONS

In the primes example, one approach is the obvious practical choice. But it is certainly *not* true that, having canvassed the field, we have picked the winner and identified the losers; that's not the point at all. The performance figures quoted above depend on the Linda system and the parallel machine we used. Most important, they depend on the character of the primes problem. We lack space to analyze more than one problem in this way. The fact is, though, that in almost every case that we have considered, an efficient parallel solution exists. Agenda-parallel algorithms programmed under the master–worker model are often but not always the best stopping point; all three methods can be important in developing a good program. Discovering a workable solution may require some work and diligence on the programmer's part, but no magic and nothing different in kind from the sketch-and-refine effort that is typical of all serious programming. All that is required is that the programmer understand the basic methods at his disposal and have a programming language that allows him to say what he wants.

We expect technology to move in a direction that makes finer-grained programming styles more efficient. This is a welcome direction for several reasons: Fine-grained solutions are often simpler and more elegant than coarser-grained approaches, as we've discussed; larger parallel machines, with thousands of nodes and more, will in some cases require finer-grained programs if they are to keep all their processors busy. But the coarser-grained techniques are virtually guaranteed to remain significant as well. For one thing, they will be important when parallel applications run on loosely coupled nodes over local- or wide-area networks. (Whiteside and Leichter have recently shown that a Linda system running on 14 VAXes over a local-area network can, in one significant case at least, beat a Cray [Whiteside and Leichter 1988]. This Cray-beating Linda application is in production use at SANDIA.) Coarser-grained techniques will continue to be important on "conventional" parallel computers as well, so long as programmers are required or inclined to find maximally efficient versions of their programs.

Attempting an initial survey of a new, rapidly changing field is a difficult proposition. We don't claim that our categorization is definitive. We've left some issues out and swept others under the rug. We do think that this survey is a reasonable starting point, both for researchers intent on a better understanding of the field as a whole and for programmers with a parallel machine and some compute-intensive applications at hand. The evidence is clear: Parallelism can lead to major gains in performance; parallel programming is a technique that any good programmer can master. In short, as Portnoy's analyst so aptly put it [Roth 1985], *Now vee may perhaps to begin. Yes?*

ACKNOWLEDGMENTS

Thanks to the Linda group at Yale and particularly to its senior members—Robert Bjornson, Venkatesh Krishnaswamy, and Jerrold Leichter—for vital contributions to the research on which this paper is based. Thanks to the referees and to Professor Peter Wegner in particular for useful guidance. Thanks to Professor Martin Schultz for his indispensable support for our work and for the entire systems research effort at Yale. Thanks finally to the National Science Foundation and to the Office of Naval Research, who made the work possible, and above all to Dr. Richard Lau of ONR, a good example of the rarest and most

valuable species in computer science, the funding visionary.

REFERENCES

ASHCRAFT, C., CARRIERO, N., AND GELERNTER, D. 1989. Is explicit parallelism natural? Hybrid DB search and sparse LDL^T factorization using Linda. Res. Rep. 744, Dept. of Computer Science, Yale Univ., New Haven, Conn., Jan.

BAL, H. E., AND TANENBAUM, A. S. 1987. Orca: A language for distributed object-based programming. Internal Rep. 140, Dept. Wiskunde en Informatica, Vrije Universiteit, Amsterdam, Dec.

BIRRELL, A. D., AND NELSON, B. J. 1984. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb.), 39–59.

BJORNSON, R., CARRIERO, N., AND GELERNTER, D. 1989. The implementation and performance of hypercube Linda. Res. Rep. 690, Dept. of Computer Science, Yale Univ., New Haven, Conn., Mar.

BJORNSON, R., CARRIERO, N., GELERNTER, D., AND LEICHTER, J. 1988. Linda, the portable parallel. Res. Rep. 520, Dept. of Computer Science, Yale Univ., New Haven, Conn., Jan.

BORRMAN, L., HERDIECKERHOFF, M., AND KLEIN, A. 1988. Tuple space integrated into Modula-2, Implementation of the Linda concept on a hierarchical multiprocessor. In *Proceedings of CONPAR '88*, Jesshope and Reinartz, Eds. Cambridge Univ. Press, New York.

BRINCH HANSEN, P. 1975. The programming language Concurrent Pascal. *IEEE Trans. Softw. Eng. SE-1*, 2, 199–206.

CARRIERO, N., AND GELERNTER, D. 1988. Applications experience with Linda. In *Proceedings of the ACM Symposium on Parallel Programming* (New Haven, July). ACM, New York, pp. 173–187.

CARRIERO, N., AND GELERNTER, D. 1989. Linda in context. *Commun. ACM 32*, 4 (Apr.), 444–458.

CARRIERO, N., GELERNTER, D., AND LEICHTER, J. 1986. Distributed data structures in Linda. In *Proceedings of the ACM Symposium on Principles of Programming Languages* (St. Petersburg, Jan.). ACM, New York.

CHEN, M. C. 1986. A parallel language and its compilation to multiprocessor architectures or VLSI. In *Proceedings of the ACM Symposium on Principles of Programming Languages* (St. Petersburg, Jan.). ACM, New York.

CHERITON, D. R., AND ZWAENPOEL, W. 1985. Distributed process groups in the V Kernel. *ACM Trans. Comput. Syst. 3*, 2 (May), 77–107.

DALLY, W. J. 1988. Object-oriented concurrent programming in CST. In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications.* (Pasadena, Jan, 1988) JPL/Caltech, p. 33.

DEPARTMENT OF DEFENSE, U.S. 1982. *Reference Manual for the Ada Programming Language.* ACM AdaTEC, July.

GELERNTER, D. 1989. Information management in Linda. In M. Reeve and S. E. Zenith, eds., *Parallel processing and artificial intelligence*, J. Wiley (1989):/23–34, *Proceedings of AI and Communicating Process Architectures* (London, July).

GELERNTER, D., JAGGANATHAN, S., AND LONDON, T. 1987. Environments as first class objects. In *Proceedings of the ACM Symposium on Principles of Programming Languages* (Munich, Jan.). ACM, New York.

GELERNTER, D., CARRIERO, N., CHANDRAN, S., AND CHANG, S. 1985. Parallel programming in Linda. In *Proceedings of the International Conference on Parallel Processing* (St. Charles, Ill., Aug.). IEEE, 255–263.

GILMORE, P. 1979. Massive Parallel Processor (MPP):/Phase One Final Report. Tech. Rep. GER-16684, Goodyear Aerospace Co., Akron.

GREENGARD, L., AND ROKHLIN, V. 1987. A fast algorithm for particle simulations. *J. Comput. Phys. 73*, 2 (Dec.), 325–348.

HALSTEAD, R. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (Oct.), 501–538.

HENDERSON, P. 1982. Purely functional operating systems. In *Functional Programming and Its Applications*, J. Darlington, P. Henderson, and D. A. Turner, Eds. Cambridge Univ. Press, New York, pp. 177–192.

HILLIS, W. D., AND STEELE, G. L. 1986. Data parallel algorithms. *Commun. ACM 29*, 12 (Dec.), 1170–1183.

HOARE, C. A. R. 1974. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct.), 549–557.

HOARE, C. A. R. 1978. Communicating sequential processes. *Commun. ACM 21*, 11 (Aug.), 666–677.

HUDAK, P. 1986. Parafunctional programming. *Computer 19*, 8 (Aug.), 60–70

JORDAN, H. F. 1986. Structuring parallel algorithms in an MIMD, shared memory environment. *Parallel Comput. 3*, 93–110.

JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. 1988. Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst. 6*, 1 (Feb.), 109–133.

KAHN, G. 1974. The semantics of a simple language for parallel processing. In *Proceedings of the IFIP Congress 74*. North Holland, 471.

LAMPSON, B. W., AND REDELL, D. D. 1980. Experience with processes and monitors in Mesa. *Commun. ACM 23*, 2 (Feb.), 105–117.

LEE, C. C., SKEDZIELEWSKI, S., AND FEO, J. 1988. On the implementation of applicative languages on shared-memory, MIMD multiprocessors. In *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming* (New Haven, Aug.). ACM, New York.

LELER, W. 1989. PIX, the latest NEWS. In *Proceedings of COMPCON Spring '89* (San Francisco, Feb.). IEEE.

MARSLAND, T. A., AND CAMPBELL, M. 1982. Parallel search of strongly ordered game trees. *ACM Comput. Surv. 14*, 4 (Dec.), 533–552.

MATSUOKA, S., AND KAWAI, S. 1988. Using tuple space communication in distributed object-oriented languages. In *Proceedings of OOPSLA '88* (San Diego, Sept. 25–30), 276–284.

MAY, M. D. 1983. Occam. *SIGPLAN Not.* (ACM) *18*, 4 (April), 69–79.

MULLENDER, S. J., AND TANENBAUM, A. S. 1986. The design of a capability-based distributed operating system. *Comput. J. 29*, 4 (Mar.), 289–300.

MUSGRAVE, F. K., AND MANDELBROT, B. B. 1989. Natura ex machina. *IEEE Comput. Graph. Appl. 9*, 1 (Jan.), 4–7.

NAREM, J. E. 1988. DB: A parallel news database in Linda. Tech. memo, Dept. of Computer Science, Yale Univ., New Haven, Conn., Aug.

NIKHIL, R., PINGALI, K., AND ARVIND. 1986. Id Nouveau. Memo 265, Computation Structures Group, MIT, Cambridge, Mass.

OLSON, T. J. 1986. Finding lines with the Hough Transform on the BBN Butterfly parallel processor. Butterfly Proj. Rep. 10, Dept. of Comput. Science, Univ. of Rochester, New York, Sept.

RINGWOOD, G. A. 1988. Parlog86 and the dining logicians. *Commun. ACM 31*, 1 (Jan.), 10–25.

ROTH, P. 1985. *Portnoy's Complaint.* Fawcett Crest, p. 309 (first published by Random House, New York 1967).

SEITZ, C. 1985. The cosmic cube. *Commun. ACM 28*, 1 (1985), 22–33.

SHAPIRO, E., ED. 1987. *Concurrent Prolog Collected Papers.* Vols. 1 and 2. MIT Press, New York.

WHITESIDE, R. A., AND LEICHTER, J. S. 1988. Using Linda for supercomputing on a local area network. In *Proceedings of Supercomputing*, (Orlando, Fla., Nov.), 192–199.

WIRTH, N. 1977. Modula: A language for modular multiprogramming. *Softw. Pract. Exp. 7*, 3–35.

XU, A. S. 1988. A fault-tolerant network kernel for Linda. Tech. Rep. MIT/LCS/TR-424, Laboratory for Computer Science, MIT, Cambridge, Mass., Aug.

YOUNG, M., ET AL. 1987. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, Tex., Nov.). ACM, New York, pp. 63–76.

Received May 1988; final revision accepted April 1989.