Parallel computers are becoming increasingly widespread. The question now is what to do with them. Can programming-language researchers make the power of parallelism accessible to programmers?

If many computers work simultaneously on one problem, they should be capable of solving it faster than a single computer working alone. It should therefore be possible to build a powerful machine by connecting many subcomputers together into a single parallel computer. This much has been common wisdom since the sixties. It is less well-known that parallel computing, long a notorious hangout for utopians, theorists, and backyard tinkerers, has almost arrived and is definitely for sale. Alliant, ELXSI, Encore, BBN, Intel, NCUBE, Sequent, and Thinking Machines among others market parallel computers. There will be more soon. A large number of experimental projects continue to thrive as well, among them Caltech’s Cosmic Cube, IBM’s RP3, and AT&T Bell Labs’ S/Net. It seems clear that, within the foreseeable future, parallel machines will be available to most serious computer users.

Once you have unpacked your parallel computer and plugged it in, what do you do with it? Parallelism is only useful, of course, in solving problems that may be attacked by many cooperating agents working simultaneously. There seem to be many such problems; the dominant group comes from numerical analysis and computational physics, and others involve graphics, image processing, systems, and artificial intelligence. It is therefore fairly simple to find a good problem. Having done so, step two is to write a program that solves the problem and runs on your parallel machine. Here is where the difficulties start.

How is a parallel machine to be programmed? There are two main possibilities. We can provide either a parallelizing compiler or a parallel programming language.

In the first case, programmers write conventional, non-parallel programs and leave it up to a smart compiler and runtime system to parallelize them. This solution is attractive because it absolves users from thinking about parallelism and, of course, it allows them to run their old programs on new parallel machines without rewriting. A good deal of progress has been made in this direction, and it is and will continue to be an important one. Here, however, we are exclusively concerned with the other possibility. Equipped with a parallel programming language, programmers formulate algorithms that are explicitly parallel—imagined in terms of many cooperating agents instead of a single one. The articles in this issue all deal with the design and implementation of languages of this sort.

Parallel programming languages versus parallelizing compilers

Why should we force programmers to worry about parallelism if they don’t really have to? If they might plausibly con-
continue to program in the usual way and leave parallelism to the compiler? Researchers often argue that, while a parallelizing compiler can transform an existing algorithm, it can’t rewrite the algorithm. Programmers freed to think in parallel sometimes invent entirely new ways of solving problems. Consider, as a simple example, the parallel alpha-beta algorithm described by Finkel and Fishburn. The original alpha-beta algorithm is strongly sequential: it searches a tree of game positions, using information developed earlier in the search to cut off later subsearches that can’t possibly improve on already-discovered sequences of moves. The parallel version is related but different. Starting with a given game state, it explores all next-states in parallel; any subcomputation may be cut off as a result of any other. The original was asymmetric, but the parallel version is symmetric. Rethinking of this sort is an example of algorithm design, not of simple parallelizing transformation.

Parallel languages have other points in their favor: Significant program classes exist whose members seem too complex or irregular ever to be optimally parallelized by compilers. But parallelizing compilers have further advantages as well. Both alternatives will be actively pursued for some time to come.

Parallel programming sounds like a good idea, then—except that it’s impossible, or at least too hard to be worth the effort; so says a large contingent of worried skeptics. They argue that the complexity of explicitly-parallel programming—all those processes active at once, all those bits zinging around in every direction—is simply too great for the average programmer to bear. Here again there are serious arguments pro and con, discussed in several of the articles following. It is worth noting, though, that researchers with parallel programming experience are in many cases the least fretful. Chuck Seitz of Caltech leads the group that built the Cosmic Cube multiprocessor. According to Seitz,

Programming experimental concurrent computers like the Cosmic Cube is not much harder than programming sequential computers, if the problem lends itself to a concurrent solution and the system software and programming tools are no worse than what we are used to on sequential computers. Those of us who have worked on the Linda project (discussed in ‘‘Linda and Friends’’) have noticed the same thing: Parallel programming (in our limited and preliminary experience) doesn’t seem to be harder in any fundamental way than conventional programming. Parallel programming may be too hot to handle—its proponents may be just about to fall off the edge of earth, even if they haven’t yet. But for now, most researchers seem willing to soldier on.

Considerations like these have given rise to a research effort whose goal is to domesticate parallelism by providing languages in which it is convenient to write explicitly-parallel programs. Designers of parallel languages hope to turn parallelism into an ordinary household programming technique, ready and waiting whenever compute-intensive applications call for it.

What kind of parallel language?

The articles that appear in this issue are by no means an exhaustive survey, but they do span the spectrum. It’s useful to classify the systems they discuss into parallel Algol-based languages (Ada and Linda), parallel Lisps and logic languages (Multilisp and Concurrent Prolog), and parallel functional languages (Parlil). Note that, in the schismatic, ideology-ridden world of programming language research, these alternatives are rarely considered side-by-side. The underlying thesis of this special issue is that they must be. No fair-minded observer can ignore any of them.

Consider the parallel Algol-based languages first. One of the first entrants was a fascinating and influential fragment that has become known as CSP; Occam is a complete language that derives from Hoare’s proposal. CSP’s simplicity and elegance have been widely admired (though seldom imitated, unfortunately). It has been argued on the other hand that the language is low-level in character (all message buffering, for example, must be implemented by the user) yet not particularly flexible. It is virtually impossible, for example, to implement a fully-asynchronous send in CSP, in other words a send operation that returns to the sending process immediately without awaiting reception or acknowledgement by a receiving process. CSP nonetheless continues to be studied widely and to be highly influential. It has been taken up not only as a programming language but as a clean notion for parallel algorithms and a good vehicle for analyzing parallel processing.

Ada and Linda, the two Algol-based languages discussed in the first and second articles in this issue, are each very different from CSP. Ada was designed originally not for parallel applications but for embedded systems. Such systems, however, are often made up of many concurrent, communicating processes multiplexed on a single processor. Ada provides the concurrent processes and the communication tools needed to support such structures and is therefore a natural candidate for parallel programming as well. Since Ada is the official programming language of the U.S. Department of Defense, its place in history is secure. It’s interesting to note, nonetheless, that the language was originally greeted with considerable enthusiasm in academic circles and has since been almost completely abandoned, to the point where it is occasionally denounced by critics who don’t seem to know anything about it. This widely-noticed phenomenon is disturbing and worth pondering. The wheel will no doubt turn again. Linda for its part is not even the official language of the Liechtenstein Department of Defense, but it has attracted its share of attention nonetheless. Its philosophy of providing a small number of simple but very-high-level communication operators, and then daring the implementors to support them efficiently, sets it sharply apart both from CSP and from Ada.
Note that it’s sometimes argued that Linda-style communication can be simulated in CSP, or CSP-style communication in Ada, or Ada-style communication in Linda and so forth, the intention being to demonstrate that CSP (or Ada or Linda) is somehow better than the others. Such contentions are largely meaningless. The goal of programming language research is to devise good abstractions. Once a programmer has decided that he prefers CSP’s abstractions to Ada’s, what he needs is an efficient CSP implementation. The fact that he can simulate CSP in Ada is of little importance to him. All it proves is that one way of implementing CSP is to use Ada—and if you want CSP, there are likely to be much more efficient ways to get it.

Parallel Lisps and Concurrent Prolog

Most artificial intelligence programs are written in Lisp, and the Lisp family continues to grow in popularity for applications of all sorts. AI in particular seems like a logical domain for parallel programming. Many AI programs rely either on heuristic search through huge solution spaces, or on the repeated examination of large numbers of knowledge-base records. Both sorts of activity often seem to be parallelizable. It’s interesting to note that, in parallel AI research to date, there’s been more interest in special-purpose parallel architectures than in parallel programming techniques. But as general-purpose parallel machines become available, it seems more likely that AI researchers will want to run programs on them, for the same reasons they seek out fast unprocessors today.

In developing parallel AI applications (and arguably within many other domains as well), some form of parallel Lisp seems like a natural choice. Bert Halstead’s Multilisp is one such language; others have been proposed as well. Multilisp, however, was one of the first parallel Lisps and is now one of the best developed. Halstead’s article describes the language and serves as a good general introduction to parallel programming besides. His “Parallel programming in the large” section raises issues that are among the most important and least sufficiently-discussed in parallel programming.

As languages go, Prolog and Lisp don’t have very much to do with each other. Prolog researchers and Lisp researchers often address the same audience, though, and the natural application domains of these two languages overlap to some extent. Prolog is one possible source language for a parallelizing interpreter or compiler, but it can also serve as a basis for a language that allows explicitly-parallel programming. Shapiro describes one such language. His work on Concurrent Prolog is noteworthy as well in ways that go beyond the logic programming and Prolog context. Most parallel languages support parallelism with pieces bolted onto a conventional framework. It’s arguably more satisfying to design a parallel language by revealing the latent parallelism in a language model’s foundations; this is what Shapiro has done, and his work is remarkable for its elegance and its conceptual economy.

Functional languages

Functional programming tops the charts as the uncontested hottest topic in programming languages today. Functional languages are probably less widely understood than they are admired, though; papers on the topic are often hard for nonspecialists to read, given the customary mathematical formalism. Paul Hudak’s discussion of a functional language for explicitly parallel programming is a lucid and readable introduction to the topic.

Functional languages are simply one kind of programming language. They ought to be (and are entitled to be) judged by the same standards as all the rest. They emerge when we eliminate the idea of a program state that may be modified and manipulated by assignment statements, and redefine programming as exclusively the definition and evaluation of expressions. Proponents believe that the simple, elegant languages that result conduce to more orderly, more rigorous, more verifiable, and ultimately more efficient programming. Opponents worry about losing expressivity as a result of the expression-evaluation-only model, and are troubled on sleepless nights by a variety of heretical thoughts. “You can’t write an AI program in a functional language,” says Drew McDermott of Yale, coauthor of the well-known AI programming text. Systems programmers are easily upset by a model that seems to collide head-on with basics like locks, semaphores, and physical devices under program control. Proponents describe functional languages as natural tools for scientific programming. But Keshav Pingali of MIT remarks that Professor Arvind’s group conducted an experiment in which they translated a two-thousand line numerical Fortran code into a functional language. The result was three thousand lines long, and the experiment convinced them that the complexities of dealing with large data structures made functional languages unworkable for this kind of application. They are now programming in a new form of language proposed by Pingali in which functional characteristics are augmented with logic-language features.

Many other groups, of course, continue to experiment with functional programming. It’s to be hoped that an ever-increasing proportion of the computing community will read their papers carefully and benefit from their new ideas. After all, the designers of functional languages are estimable researchers. (“Their proponents are often brilliant intellectuals” notes James Morris, who is one of them.) But there is nothing magic, sinister, or comprehensibly deep in these languages, and readers are urged to judge them in the same way they judge other language proposals.

What next?

All these various languages are fine for the present, observers have been known to
remark, but what will happen when we have really big machines, parallel machines with tens of millions of nodes and beyond? How can we possibly learn anything about these hyperparallel computers from the present generation of puny specimens? Won't computing have to change in thoroughgoing, fundamental ways if we are ever to manage these huge machines? For that matter, won't programming as we know it inevitably disappear in the process?

It's impossible to doubt that computing will change radically. But it's almost as difficult to believe that the new forms that emerge will be completely unrelated to our current practices. Some hyperparallel machines will be assembled out of simple, tightly-coupled elements. Others, however, will no doubt be generalizations of current designs in which each computing element is a powerful, self-sufficient unit. (Such a machine's computing nodes should differ from a current machine's mainly in providing strong support for some kind of higher-level parallel programming model.) Users of this latter kind of machine will presumably be able to count on help from sophisticated program-generating tools. Source programs may ultimately look more like concise, formal descriptions than most programs do now; such formal descriptions might conceivably result from informal exchanges in English between the user and a description-synthesizer program. But however rarefied our source programs become, some entity will have to understand how to transform them into assemblages of real, simultaneously-executing processes.

Parallel language research is developing this understanding right now. And those who wait expectantly for the demise of programming as we know it are advised, finally, to consider the goal of the 1957 designers of Fortran: the elimination of programming. Fortran was an automatic coding system, designed to allow programs to be replaced by quasi-mathematical formulas. As the designers understood it, their goal was largely achieved. Of course, the same problems of designing, coding, debugging, and testing familiar in assembly programming simply reemerged at a new and higher level. Advocates of ultra-high-level "non-programming" are advised to expect the same.

For now, parallel-language researchers are using their systems as laboratories, and the results are changing our understanding of programming languages and of the process of computation. They are seeking, at the same time, to bring about a qualitative change in the role of computers by enormously increasing the speed at which programs execute. These projects should keep them off the streets for some time to come.

References


David Gelernter is an associate professor of computer science at Yale University. He received his BA from Yale in 1976 and his PhD in computer science from SUNY at Stony Brook in 1983. He is interested in programming languages, parallelism, and artificial intelligence.

Readers may write to Gelernter at the Dept. of Computer Science, PO Box 2158, Yale Station, New Haven, CT 06520-2158.