Parallel Programming: Issues and Questions

Shrekanzt S. Thakkar, Sequent Computer Corp.

The parallel processing issue must be addressed on three levels: (1) architectural issues include how to organize and implement computers that have multiple processing elements. (2) Software issues include applications and system software. (3) Algorithm issues involve how to design parallel algorithms.

Architects of parallel-processing systems struggle with important trade-offs. Their choices often lead to controversy over which architecture is best. In shared-memory machines, these trade-offs include bus bandwidth, memory contention and coherence, and locking. In distributed systems, these trade-offs include interconnection topology and communications overhead.

Systems software — operating systems, compilers, and debuggers — is dramatically affected by the architectural choices made. For example, an optimizing parallel compiler for a shared-memory machine will optimize code differently than for a distributed machine. Similarly, debugging a shared-memory program is quite different from the debugging a message-passing, distributed machine.

Parallel programming challenges software professionals to rethink old approaches and make difficult — often controversial — choices.

Applications software — fluid flow, simulation, and transaction processing — is also dramatically affected by architectural choices. However, some architectural decisions can be hidden from the applications programmer. For example, if the application developer uses a sequential language such as Fortran and if the Fortran compiler automatically parallelizes the application, the developer does not
need to know the details of parallel programming.

A convincing argument can be made for abandoning the tradition of serial programming that has evolved over the last 40 years. New languages are evolving that have implicit parallelism and that advocate a new way of looking at programming. But abandoning serial programming also means abandoning the rich tradition of algorithm design. New algorithms require rethinking — this time, in parallel. Parallel algorithms offer perhaps the greatest hope for major performance improvements.

Controversy surrounds both parallel languages and tools: Which approach is best? How can a certain approach be implemented? Should languages be explicitly parallel or should tools make sequential programs parallel? Should functional languages replace imperative languages altogether? What tools are needed for parallel programming? How do they differ from existing tools for serial programming?

The articles in this issue address all three issue areas. While we have purposely avoided focusing on the architectural trade-offs, our authors have given them important consideration.

The first article addresses what type of programming languages and environments are available today on multiprocessor systems. It describes programming on the Sequent Balance, Intel iPSC, and T-Rack and discloses that programming on some of these systems is very difficult because the user must understand the architecture's operational characteristics.

The second article introduces a new way to optimize parallel programs called grain packing. Grain packing reduces total execution time by balancing execution time and communication time and is applicable to extended serial and concurrent programming languages. The authors also expose misconceptions about the efficiency of load balancing and explicit parallel constructs.

Because so many large programs are written in Fortran, much attention has been focused on ways to detect parallelism in these programs so that they do not have to be rewritten. The third article surveys the recent progress in compiler and translator technology to do this. It describes techniques that detect which iterations of a Fortran Do loop are independent and the synchronization schemes that ensure that separate processors get the correct data.

Functional programming languages are well-suited for parallel computation because they do not place artificial constraints on the evaluation order. The Church-Rosser property guarantees determinacy regardless of the execution order. Two articles in this issue describe the implementation of functional languages on parallel architectures.

The first is a description of a programming methodology called parafunctional programming and an implementation of a parafunctional language on a distributed and a shared-memory architecture. The second is an implementation of SISAL on a shared-memory architecture. SISAL addresses an outstanding problem associated with functional language design and implementation: copy optimization. Strictly speaking, SISAL is a single-assignment language, so it requires frequent data structure copying in some programs. The authors stress the importance of doing this optimization well.

The final article describes the design and implementation of a distributed version of a Prolog interpreter running a network of workstations. This system offers easy exploitation of And parallelism using fork and join primitives implemented by message passing. The authors show that their system can achieve practical speedups on large, existing Prolog programs.

I thank all the reviewers and authors who contributed to this issue of IEEE Software and the parallel programming track at the 1988 Hawaii International Conference on System Sciences.

Shreekant Thakkar is a coauthor of an article in this issue. His biography appears on p. 22.