We chose computation graphs as the basis for representing programs for abstract machines. A computation graph is a program for some conceptual computation expressed in terms of the operations of an abstract machine. It is a directed graph where the nodes represent schedulable units of computation and arcs represent dependency relationships between source and sink pairs.

The computation is executed by transversal of the directed graph along the paths defined by the dependency relations associated with the arcs. Only the binding of operators to type instances is always associated with a node. Either the program defining the schedulable unit of computation or the type instances or both may arrive as values on arcs or be permanently bound to a node.

A node is therefore an abstraction for execution on a processor and memory. An arc is an abstract representation of a communication channel or a control channel that may include memory or execution of a synchronization protocol. The initial abstract machine will normally have a processor for every schedulable unit of computation and sufficient channels and control lines coupling the processors so that each dependency relation can be assigned its own set of resources.

The formulation proceeds through specification of successively more realizable abstract machines and mapping of program graphs to these abstract machines until a hardware-realizable machine is reached.

Performance is evaluated by establishing realizations, usually in software, for each abstract machine. The framework given by graphical representation of programs allows a generic simulator, in which representations of schedulable units of computation and dependency relations can be implemented by parameterized templates.

Acknowledgment

The conceptual foundation for this research was established in the course of research on parallel computing sponsored by the Department of Energy, the National Science Foundation, and the Air Force Office of Scientific Research. Establishment of a research environment and experimental research is to be sponsored by DARPA as a part of the Strategic Computing Initiative.

References

1. J.C. Browne, "Framework for Formulation and Analysis of Parallel Computation Structures," *Proc. 18th Hawaii Int'l Conf. System Science*, Jan. 1985, pp. 2-7.

J.C. Browne,
Department of Computer Sciences, University of Texas, Austin, TX 78712

Algorithm Design for Different Computer Architectures, Argonne National Laboratory

Within the last 10 years numerical analysts have realized the need to involve themselves not only in algorithm development, but also in the design of the software that embodies the numerical algorithms. Issues such as robustness, ease of use, and portability are now standard fare in any discussion of numerical algorithm design and implementation. The portability issue, in particular, becomes formidable as the evolution of new and exotic architectures makes a reality of the concepts of concurrent processing, shared memory, and pipelining, all of which are intended to increase performance. Ironically, it is tempting to assume that portability must always carry with it an unacceptable degradation in efficiency for any machine architecture. We contend that this assumption is erroneous and that its widespread adoption could seriously hamper the effective use of future machines.

Future architectures promise a profusion of computing environments. The existing forerunners have already given many software developers cause to reexamine the underlying algorithms for efficiency. However, it seems an unnecessary effort to recast these algorithms with only one computer in mind, regardless of its speed. The efficiency of an algorithm should not be discussed in terms of its realization as a computer program. Even within a single architectural class, the features of one system may improve the performance of a given program, while features of another system may have the opposite effect.

Software developers should begin to identify classes of problems suitable for parallel implementation and to develop efficient algorithms for each area. With such a wide variety of computer systems and architectures in use or proposed, the challenge for algorithm designers is to develop algorithms—and ultimately software—that are both efficient and portable.

Research approaches. There appear to be three approaches to improving algorithm efficiency and portability. They are not mutually exclusive, and each can contribute to an effective solution.

The first approach is to express algorithms in terms of modules at a high level of granularity. When software is moved from one architecture

to another, the basic algorithms remain the same, but the modules are changed to suit the new architectures.[1]

A second approach is to create a model of computation that represents the computing environment and characterizes the salient features of an architectural category. Software is written for the model and then transformed to suit a particular realization of an architecture that fits the model.[2] The general categories of MIMD and SIMD are, of course, too crude, but additional details can be specified. For example, an MIMD model might be characterized by the number of processors, communication vehicle, access to shared memory, and synchronization primitives. Software written for such a model could be transformed to software for a specific machine by a macroprocessor or specially designed preprocessor.

A third approach, not discussed in detail here, is to write the software in high-level language constructs such as array processing statements. Again, a preprocessor could generate suitable object code for a particular architecture.

Algorithms as modules. Of the three approaches, expressing algorithms in terms of modules with a high level of granularity seems preferable where it is applicable. In particular, it seems applicable to certain basic software library subroutines, which usually shoulder the bulk of the work in a wide variety of numerical calculations. Where successful, expressing algorithms as modules will enhance both software maintenance and use.

Software maintenance would be enhanced because more of the basic mathematical structure would be retained within the formulation of the algorithm. The fine computational detail required for efficiency would be isolated within the high-level modules.[3] Software users would benefit through the ability to move existing codes to new environments and could experience a reasonable level of efficiency with minimal effort.

A key issue in the success of this approach is to identify a level of granularity that will permit efficient implementations across a wide variety of architectural settings. Individual modules can then be dealt with separately and retargeted for efficiency on quite different architectures. This will have the effect of concealing a particular machine's peculiarities from potential software users and allowing them to concentrate on the application, not the computing environment.

Application. Of course, the algorithmic approach described above has limited application. It works well for linear algebra and could be effective in any application dominated by these calculations.

Other areas, where the approach will not work, need algorithms that focus on architectural features at a deeper level. The goal in these efforts must be to exploit key architectural features, not the particular realization. Here, the approach based on a computation model can be useful. As multiprocessor designs proliferate, research efforts should focus on generic algorithms that can be easily transported across various implementations. If a code has been written in terms of high-level synchronization and data management primitives that are expected to be supported by every member of the computation model, then these primitives need only be customized to a particular realization. A very high level of transportability can be achieved by automating the transformation of these primitives. The benefit to software maintenance, particularly for large codes, is in the isolation of synchronization and data management peculiarities.

The desire for portability is often at odds with the need to efficiently exploit the capabilities of a particular architecture. Nevertheless, algorithms should not be intrinsically designed for a particular machine. One should be prepared to give up a marginal amount of efficiency to reduce manpower requirements for using and maintaining software.

There are many ways to implement the general ideas we have described. Although we are not prepared to recommend a particular implementation with any degree of finality, our experience indicates the feasibility of the two approaches discussed here. We believe that a high level of transportability can be achieved without seriously degrading potential performance, and we encourage others to consider the challenge of producing efficient, transportable software for these new machines.

Acknowledgment

This work was supported by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the US Department of Energy under Contract W-31-109-Eng-38.

References

1. J.J. Dongarra and S.C. Eisenstat, "Squeezing the Most out of an Algorithm in Cray Fortran," *ACM Trans. Math. Software*, Vol. 10, No. 3, Mar. 1984, pp.221-230.
2. J.J. Dongarra and D.C. Sorensen, "A Parallel Linear Algebra Library for the Denelcor HEP," in *Parallel MIMD Computation: The HEP Supercomputer and its Applications*, J.S. Kowalik, ed. (to be published).
3. E.L. Lusk and R.A. Overbeek, *Implementation of Monitors with Macros: A Programming Aid for the HEP and Other Parallel Processors*, technical report ANL-83-97, Argonne National Laboratory, Argonne, Ill. Dec. 1983.

J.J. Dongarra, Brian T. Smith, and Danny C. Sorensen,
Argonne National Laboratory, 9700 Cass Ave., Argonne, IL 60439