Paradigms and Compilers for Parallel Processing: Guest Editor's Introduction

BOLESLAW K. SZYMANSKI

Department of Computer Science and Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY 12180; e-mail: szymansk@cs.rpi.edu

This is an introduction to a special issue devoted to the current research in the area of foundations of parallel scientific computing. In recent years, high-performance computing underwent a deep transformation. The declining share of the parallel processing market held by traditional supercomputers and the waning popularity of the single-instruction multiple-data (SIMD) machines, together with the increasing role of clusters of workstations, created the conditions for the rapid spread of parallel computing in government and industry. The emphasis shifted from record-breaking performance for any price to price-performance optimization. The successful companies, such as Silicon Graphics or IBM, use the performance gains driven by the general market to improve the performance of their parallel machines. In contrast, companies that relied on processors designed specifically for their architectures, like Kendall Square Research or the Thinking Machine Corporation, were less successful in staying in the computer design market. Parallel processing has been becoming ubiquitous at all levels of computing technology. In microprocessor design, superscalar techniques (executing multiple instructions at the same time) are now a standard.

It is important to realize that despite these trends, parallel computing captures a small fraction of the overall information technology industry. The parallel computer industry constitutes about 0.5% of the U.S. information technology market. Such a small percentage of the overall market indicates a narrow user base that can be easily saturated with new products. In addition, parallel computing has been highly dependent on government policies; institutions and government-supported universities traditionally constituted more than one half of all users.

The weaknesses of parallel machines stem from the following two factors:

1. Narrow application base: Parallel architectures are best suited to solve large and highly tuned, course-grained, and/or data-parallel problems.
2. Rapid change of hardware: Every new generation of parallel architectures differs from the previous one, forcing the users to redevelop their applications. Often, porting and tuning an application to a new architecture can take as long as the time needed to introduce a new architecture, making the ported code obsolete at the moment it is ready for use.

Part of the difficulty in making parallel computing widespread and popular has been the lack of standards in parallel programming interfaces. As discussed below, such standards are emerging and gaining widespread acceptance.

Several different architectural approaches to parallel processing are slowly converging to a similar solution. The workstations interconnected through a fast network, when dedicated to a single application, behave like a multiprocessor. The modern shared memory multiprocessor relies on an interconnection network between the global memory and local processor caches, and therefore behaves similarly to the distributed memory multiprocessor. Finally, distributed memory machines through extensive use of message caching, faster interconnection networks, and smaller communication latencies approach in their behavior shared memory machines with local caches. The overall trend is to use powerful computing nodes interconnected through a high-speed network of large capacity. The associated trend is to rely on standard, commodity

Received March 1996
Revised March 1997

Scientific Programming, Vol. 6, pp. 159–162 (1997)
CCC 1058-9244/97/020159-04

off-the-shelf components to improve the price-performance ratio of such architectures.

Parallel programmers face a daunting challenge, especially with increasing large and complex applications. They must identify parallelism in an application, extract, and translate that parallelism into their codes, design and implement communication and synchronization that preserve the program semantics, and foster the efficiency of parallel execution. All these steps must be guided by the currently available architectures which may change tomorrow, making some of the designs suboptimal or inefficient. Not surprisingly, in such an environment parallel programming has experienced a long and difficult maturation process. Yet, two basic paradigms emerged: data parallelism and message passing. The first one is popular because of its simplicity. In this paradigm there is a single program (and therefore a single thread of execution) which is replicated on many processors and each copy operates on a separate part of the data. A SIMD version of this approach requires hardware support and is considered useful only for a limited range of applications. Its loosely synchronized counterpart, often referred to as a single-program multiple-data (SPMD) paradigm, is more universal. SPMD parallel computation execution consists of two stages:

1. The computational stage, when copies of the same program are executed in parallel on each processor locally. The execution can differ in the conditional branches taken, the number of loop iterations executed, etc.
2. The data exchange stage, when all processors concurrently engage in exchanging nonlocal data.

It should be noted that the data exchange stage is very simple in the case of shared memory machines (when it can be enforced by use of locks or barriers). By reordering the computation and properly selecting the frequency of synchronization, partial interleaving of computation and communication stages can be achieved. The SPMD model matches well the needs of scientific computing which often requires applying basically the same algorithm at many points of a computational domain. SPMD parallel programs are conceptually simple because of a single program executing on all processors, but they are more complex then SIMD programs.

For complicated applications, running a single program across all parallel processors may be unnecessarily restrictive. In particular, dynamically changing programs with unpredictable execution times result in poorly balanced parallel computations when implemented in SPMD paradigm. This is because SPMD processes synchronize at the data exchange stage, and none of the processes can proceed to the next computational stage until all others reach the data exchange stage.

The memory distributed machines use message passing for exchanging data between different processors. The SPMD model may shield the user from specifying the detailed data movements thanks to data distribution directives from which a compiler generates the message-passing statements. However, the users which decide to write the message-passing statements themselves have full control over the program execution. In particular, the user may define how many and which processors synchronize at the given instance of parallel execution. This approach gives the user a lot of flexibility at the cost of requiring the user to make a very intricate and detailed description of the program. The programs tend to be longer and more complex than their SPMD counterparts, and therefore more error prone. However, once debugged and tuned up, the programs are more efficient. The flexibility of the messaging-passing model makes it applicable for a variety of problems. As discussed below, the standard library of functions for message passing, MPI, is becoming a universal tool for parallel software development.

There is a lot of research parallel programming languages with different flavors to choose from, starting from functional, dataflow to object oriented, logical, etc. However, the majority of running scientific parallel programs were written in Fortran. Since the 1950s, this language was a favorite choice for writers of scientific programs and particularly for generations of graduate students in applied sciences. Over the years, Fortran underwent a remarkable transformation, from one of the first languages at all to the first language with a well-defined standard (Fortran 66) to the structured programming of Fortran 77, to data parallel and object-oriented Fortran 90, and finally to the newest standard of high-performance Fortran (HPF). Each generation brought with it new features and set a new standard for the manufacturers of hardware and compilers. Critics of HPF argue that the HPF language is not general enough. In particular, HPF does not allow for dynamically defined alignments and distribution that are permitted in some other languages (HPF+, Vienna Fortran). However, standardization of the language features is extremely important for users, compilers, and tool writers because it protects their software investments against changes in the architecture. In that respect, the introduction of Fortran 90 and then HPF is an important step forward toward more stable parallel software.

HPF can be seen as the flagship of the data-parallelism camp. On the other hand, the supporters of message passing-based parallel programming achieved standardization of their approach in the message-passing interface (MPI).

Parallel processing is at a critical point of its evolution. After a long period of intense support by government and academia, it slowly moves to derived the bulk of its support from the commercial world. Such a move brings with it a change of emphasis from record-breaking performance to price performance and sustained speed of program execution. The winning architectures are not only fast, but also economically sound. As a result, there is a clear trend towards widening the base of parallel processing both in hardware and software. On the hardware side, that means using off-the-shelf commercially available components (processors, interconnection switches) which benefit from a rapid pace of the technological advancement fueled by the large customer base. The other effect is the convergence of different architectures thanks to spreading the successful solutions among all of them. Workstations interconnected by a fast network approach the performance of parallel machines. Shared memory machines with multilevel caches and sophisticated prefetching strategies execute programs with efficiency similar to that of distributed memory machines.

On the software side, the widening base of the users relies on standardization of parallel programming tools. By protecting the programmer's investment in software, standardization promotes development of libraries, tools, and application kits that in turn attract more end-users to parallel processing. It appears that parallel programming is ending a long period of craft design and is entering a stage of industrial development of parallel software.

The articles included in this issue were selected from 15 submissions. Although the selected articles cover a range of topics, they share a common theme which is widening the applicability of parallel processing. The first article, "Towards Architecture-Adaptable Parallel Programming," by Santhosh Kumaran and Michael J. Quinn from Oregon State University, focuses on the divide-and-conquer approach to designing parallel software portable across a range of parallel architectures. For the price of limited generality at the level of the program design methodology, the authors simplify porting an application to a new platform. Their system provides the user with divide-and-conquer templates for expressing parallelism. It also automatically selects the efficient parallel algorithm for the specified architecture.

The newly developed multithreaded architectures provide a means for escaping the limitation of the data-parallelism paradigm. The idea of mixing data and task parallelism is not new. However, without multithreading support, it is very difficult to implement. In "Data-Parallel Programming in a Multithreaded Environment," Matthew Haines (the University of Wyoming) and Piyush Mehrotra and David Cronk (ICASE) discuss a run-time based package that supports parallel program execution on a multithreaded architecture. Their package handles the problem of relative indexing and collective communication among the thread groups, thereby enabling different groups of threads representing different tasks of a parallel application to run concurrently.

As discussed above, global address space, supported by shared memory machines, is an attractive abstraction for parallel programming. However, straightforward shared memory implementation is not scalable. So-called distributed shared memory (DSM) is an important alternative that combines a shared memory programming model with the scalability of distributed memory machines. In "Implementation and Performance of DSMPI," Luis M. Silva and João G. Silva (the University of Coimbra) and Simon Chapple (Quadstone Ltd.) report on their experience with the parallel library called DSMPI. The authors implemented DSMPI on top of MPI to provide DSM abstraction to the library users. The article compares efficiency of programs using DSMPI with those using MPI directly. The test programs were run on a network of workstations and a Cray T3D. The results indicate that for several applications DSMPI library overhead is negligible.

Adaptive parallelism in which the number of processors running an application changes at execution time is explored in "Run-Time and Compiler Support for Programming in Adaptive Parallel Environments" by Guy Edjlali, Gagan Agrawal, Alan Sussman, Jim Humphries, and Joel Saltz (the University of Maryland). The authors developed a run-time library that enables the user to redistribute data and computation when the number of participating processors changes. It also supports recomputation of loop boundaries and communication patterns for redistributed computation. The authors discuss how their library can be integrated with an HPF compiler. The reported results indicate that the approach is effective if the changes in the number of executing processors are infrequent. The library is targeted for parallel computation on a network of nondedicated workstations which is quickly becoming an important parallel computing engine.

The last article focuses on fundamental issues in any transforming parallel compiler design, which is

the representation of complex range sets arising when the arrays are traversed through complicated subscript expressions. The article entitled "Precise Analysis of Array Usage in Scientific Programs" was written by M. Manjunathaiah and Denis A. Nicole (University of Southampton). It introduces a new technique for an exact representation of the results of binary operations on array sections. Array sections define which array elements are written or read by the program statements. Sections that are regular, simple, or convex can be compactly represented through shape descriptors. However, some operations on sections, most notably union, may create the sections that cannot be represented by shape descriptors. The technique presented in this article enables a compact and exact representation of the union result.

In preparation of this special issue, article review was an important stage, both for the selection of the best submissions as well as for improving the published articles. Many thanks are due to the following volunteers for their timely and thoughtful reviews: Peter Berezany (European Center of Excellence for Parallel Computation), Zbigniew Chamski and Michael F. O'Boyle (University of Manchester), Raja Das (Georgia Institute of Technology), Susan F. Hummel (Polytechnic University), Wesley Kaplow (AT&T Bell Laboratories), Charles H. Koelbel (Rice University), David R. Kohr (Argonne National Laboratory), Sandeep Kumar (North Carolina State University), Lenore M. R. Mullin (University at Albany), David O'Hallaron (Carnegie Mellon University), Patricia Pinea (Allegheny College), Yuan Shi (Temple University), David Skillicorn (Oxford University), Mark A. Sweany (Michigan Technological University), Parimala Thulasiraman and Xinan Tang (McGill University), David Wonnacott (Haverford University), Ewa Deelman, Mukkai Krishnamoorthy, David Musser, Mohan Nibhanapudi, Charles Norton, Peter Tannenbaum, Wesley Tuner, and Louis Ziantz (Rensselaer Polytechnic Institute).