Parallel Architectures and How to Program Them

Rudolf Eigenmann
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign

Abstract

This article discusses issues in designing and exploiting parallel architectures as seen from a research group that has been focusing on the development of compilers for such machines. For scientific and engineering applications, Fortran languages have been the primary means of expressing algorithms. Current programmers of high-performance parallel machines have to make a choice between sequential, portable programs and efficient, parallel codes that come at a significantly higher development cost. The much needed compiler technology for parallelizing sequential programs automatically is currently being developed, but has not yet delivered tools that consistently yield efficient code. Programmers have to specify parallelism explicitly to make up for this shortcoming. Currently, users are faced with two rather different parallel programming models, both reflecting the underlying machine organization: shared-address space machines provide a name space for data that needs to be seen by multiple parallel tasks; message-passing machines provide separate address spaces and data needs to be exchanged by explicitly sending and receiving messages. The question of which model will dominate in the long term is one of the issues currently being debated.

1 Introduction

This article tries to shed some light on the huge field of parallel architectures and their programming technology. This will be done from the angle of a research group at the University of Illinois that has been involved in this field for more than two decades. During this time, researchers at our University have pioneered new parallel machine architectures, programming methodologies and tools, compilers, algorithms and applications, as well as benchmarks for evaluating these technologies. A small selection of this work is described in [BENP93, KDL+93, GNP+90, CKPK90][1].

This position statement is, without doubt, somewhat biased by our main field of research: the development of compiler technology for parallel computers. However, compiler research has put us in the middle of some practical questions: what applications do we need to consider; what language and tools do we need to provide to the user; and, how can we design better machines that allow us to implement these user services in the best way possible.

It is not evident why we want parallel computers. When it comes to discussing user interfaces, this issue needs careful consideration. For example, one should be aware that buying a fast parallel machine is not the proper response to the need for increased computing power, if a number of programs could actually be run on a set of less expensive, independent computers. Frequently, system administrators of expensive parallel machines find out that it is more cost-effective to assign parallel CPUs to different programs, instead of making them available to individual applications. As a result, the communication hardware – the part that makes the difference between a loose bunch of computers and a parallel architecture – ends up totally underutilized. If we invest in learning about parallel programming, then we have to target a machine that is actually operated as a parallel computer. Because this is expensive, we have to be certain that the need for high performance in a single application is a crucial one.

[1] These and many other reports can be retrieved by anonymous ftp from ftp.csrd.uiuc.edu:CSRD_Info/reports

Scientific and engineering applications are the playing ground for our research. "Supercomputer" technology is the most mature in these fields. Non-numerical computing has been mentioned as a potential application area of high-performance parallel machines for many years. But, to date, we have seen only a few industrially significant applications of that sort. This will change as languages used in non-numerical areas receive increased support on parallel machines, but perhaps more-so because the boundaries between supercomputers, high-performance workstations, and most recently even personal computers are becoming increasingly blurred.

In order to understand the changes that the field of high-performance computing is going through, we need to consider some political forces as well. The Advanced Research Projects Agency (ARPA) has been directing the research and development in U.S. industry and academia quite significantly. Massively parallel computers have been strongly promoted by this funding agency over the past decade. With the current changes this agency is undergoing – among others as a result of the 1993 change in the U.S. presidency – this trend will clearly have to prove its merits before it will be continued.

2 Parallel architectures for high-performance computing

This article focuses on the issues of how to compose multiple processors into a moderately or highly parallel computer and how to take advantage of this computing power in application programs. It does not deal with the inner structure of individual processors. However, it shall be noted that advances in instruction-level parallelism have been significant over the past few years. The interested reader will find thorough introductions in many books; [Hwa93] is a recommended example.

Interconnection technology is what makes a parallel computer out of a number of independent processors. Several interconnection structures have become known over the years and shall only briefly be mentioned here. Bus-based shared-memory systems are perhaps the oldest ones. Additionally there are mesh, torus, cube, ring, omega, crossbar, and tree connections that have been explored in several experimental and commercial architectures. Although these structures are well-known, there is continuous pressure on their designers to increase speed and keep up with the increasing computational power of individual processors (i.e., they have to maintain the "computation to communication ratio").

Perhaps the easiest way to architect processor and network components into a parallel computer is to build a network port for each processor to exchange messages over the network. This message passing (MP) scheme makes maximum use of off-the-shelf components and, hence, appeals as a cost-effective way to build parallel systems. Examples of such systems are the Connection Machine CM5, Intel paragon, and the IBM SP1.

Sending and receiving messages is a costly operation. Some application programs can run efficiently in parallel only if the communication latencies are very low. Latencies can be reduced by building the network interface into the processor-to-memory connection. Accessing memories directly from the network bypasses the processor and, ideally, can be as simple and fast as an address decoding operation. In addition, the requesting processor can access remote memory through normal read and write instructions, rather than through time-consuming input/output port operations. Obviously such techniques require more than off-the-shelf components and they operate at high speed. Consequently they are more expensive. The advantage is not only reduced latency, but also an important impact on the programming model: each processor can now access the other processor's memory as just another address range. We will use the term shared (or global) address space (SA) for these machines. Examples are the Cray T3D and the Convex Exemplar. Figure 1 displays simple models of the message passing and shared address architecture. The distinguishing feature of the two is the position of the network interface.

Both SA and MP machines usually provide additional processor-private memory for fast access to local data. Even in SA machines there can be a significant difference between *local* and *remote* accesses. All of these machines are *distributed memory machines*. Shared-memory machines provide uniform memory access latency, which can be achieved by placing all memories close to all processors. Such systems are believed to be not scalable. However, it should be noted that such systems are still the top

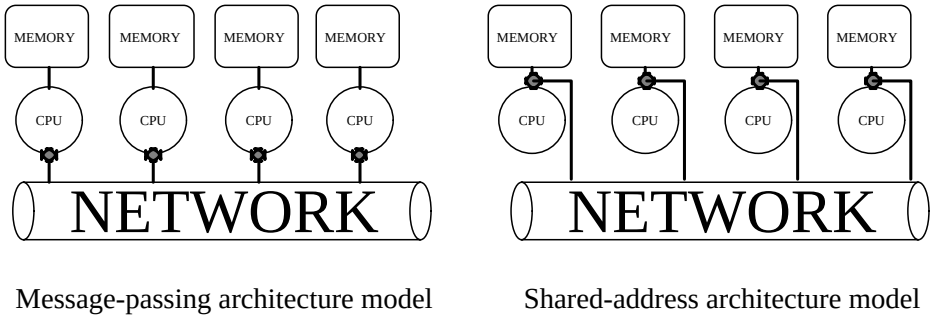

Figure 1: One important difference between message-passing (MP) and shared-address (SA) architectures is that MP implementation is possible with low-cost I/O ports to network. Efficient SA implementations need more expensive address path switches.

performers in some important benchmarks. For example, the Cray YMP/C90 ranks number one in the Perfect Benchmarks and, except for the Cray T3D in some cases, is reported to perform best on the NAS Benchmarks as well.

To further reduce the latency of remote data accesses in SA machines, the design of cache systems is currently given much attention. Although cache technology is quite advanced in uniprocessor environments, the design of a highly parallel cache is still an unsolved problem. The field is currently being explored and we can find all variants between no caching of remote data (as in the Cray T3D) and "All cache systems" (as in the Kendall Square machine).

The natural question is, "What is the best system"? This question is being discussed with sometimes religious vigor. It seems we can do better. Obviously MP systems have proven to be applicable to a set of programs and they have demonstrated high, cost-effective performance. This has promoted the interest in this machine class. It is also a fact that SA and even shared memory systems hold the record on some benchmarks, as noted above, which may be part of the reason why they have gotten renewed attention. Clearly there is a need for a more comprehensive evaluation of current parallel machine architectures. Many machines have been built and, so far, it remains the user's responsibility to deal with this variety. The next two sections will deal with the user's view of these machines in more detail.

3 Languages for scientific and engineering applications

Sequential languages

Fortran The reader who is starting to familiarize him or herself with the area of program development for scientific and engineering applications is well advised to learn the standard Fortran77 language. Not only is the majority of important programs written in Fortran77, but the compiler optimization technology for this language is also the most sophisticated. For example, the automatic detection of parallelism is a discipline that has been dealing almost exclusively with Fortran77. Hence, new languages would not only have to take the hurdle of proving their superiority to the user, but they would also have to wait for the adaptation of a substantial body of compilation and optimization techniques.

C and C^{++} Only recently C and, most recently C^{++}, have become more popular is this field. Today, however, C programs that rely on compiler technology for automatic parallelization are practically nonexistent. One big issue in C is pointer data structures, whose access patterns are very difficult for compilers to investigate. We are only at the very beginning of resolving this problem.

Fortran extensions Most computer vendors provide Fortran77 extensions. Although potentially very useful (e.g., dynamic allocate/deallocate functions or variable-sized data declarations), these extensions make programs machine-dependent. The programmer is usually better off not taking advantage of such features, unless programs are short-lived. Implicitly, most Fortran compilers also "support" extensions by failing to enforce the rules of the language standard. For example, passing real-type variables to subroutine parameters that are declared complex violates the standard; but it is usually not enforced by the compiler, and therefore used by programmers. Traditional Fortran compilers translate on a subroutine by subroutine basis, and "interprocedural operation" is only a relatively young capability. Other language rules cannot be enforced by the compiler; the programmer has to be well aware of this fact, since compiler optimization may count on the rules being observed. Such an example is an array that is passed to two different subroutine parameters, each with a different index (e.g., `a(i1)` is passed to array `x()` and `a(i2)` is passed to `y()`). If the variables `i1` and `i2` are derived from program input data, the aliased (i.e., overlapping) parts of x() and y() cannot be determined at compile time. The Fortran77 standard prohibits the modification of aliased array parts, which allows the compiler to do certain optimizations. It is the user's full responsibility to enforce this rule.

Fortran90 Some Fortran77 extensions include array notation, (e.g., a(1:n) referring to the array section from 1 to n). This is also one of the best-known features of the Fortran90 standard language. Beyond this construct, many language implementations that "support Fortran90 features" offer little new. Although there are now a growing number of true Fortran90 compiler announcements, these tools are still unable to optimize program sections that take advantage of the Fortran90 pointer, aggregate structures, or module features. Fortran90 has the reputation of being one of the most complex languages. Users may see this as more of an advantage than a problem; however, it is clear that compiler technology will have to take many years to deal with this complexity. This complexity may have been the reason that, despite the fact that the Fortran90 standard is now 4 years old and has been discussed as Fortran8X for many more years, there is only a small number of programs written that use its features.

Parallel languages

Compiler technology that is able to automatically parallelize standard Fortran programs works well for some programs, but it has severe limitations in others. Programmers that wish to exploit parallel machines to a high degree usually must learn an extensive amount about compiler optimizations and rearrange their sequential code so that compilers can recognize the parallelism. In many cases, the user may wish to specify parallelism explicitly in the program. Here, the user leaves the realm of standard portable programs. Parallel Fortran dialects are much less uniform since they are in flux and have to adapt to changing machine architectures in order to provide an effective interface between user and machine. Many computer vendors add directives to the standard Fortran language which, for example, define that a loop shall run in parallel and that data shall be private to the loop. Such directives vary widely from vendor to vendor.

Whether the user specifies parallelism in the form of directives or through new parallel language constructs, is equivalent in many respects. One frequently named advantage of directive-based parallel extensions is that a sequential program can be derived easily by ignoring the directives. In the case of explicitly parallel languages, more carefully defined constructs serve as the advantage.

X3H5 parallel Fortran and C This ANSI standard defines extensions to the two programming languages Fortran90 and C. The committee that defined these language extensions has recently completed its work, and the language definitions are in the process of being formally approved as ANSI standards[Ame94]. The X3H5 programming model addresses shared-memory machines. Currently, there seems to be significantly less interest in X3H5 than in the language described in the next paragraph; consequently, very few programs are written using this standard. Whether this changes as a result of the renewed interest in SA machines, remains to be seen.

High Performance Fortran Another notable effort to standardize a parallel Fortran language is the High-Performance Fortran Forum, which is currently in the process of redefining the version 1 HPF proposal [For93]. The HPF effort started from the perception that programming in message passing constructs for MP machines is tedious, particularly the process of distributing data and generating the appropriate messages. HPF lets the user specify the data distribution directly and leaves the generation of messages up to the compiler. In addition to data distribution constructs, HPF includes directives such as "independent" (parallel loops) and "private" (local data) that are applicable to machines with a shared address space as well. Information about HPF can be retrieved via the world-wide web from http://www.erc.msstate/hpff/home.html.

Message passing languages Because HPF is young, most available programs on message-passing machines are currently still written in message passing languages (i.e., in Fortran77, Fortran90, or C plus calls to send and receive subroutines). Numerous message-passing libraries have led to the message passing interface forum, whose goal is to create of a message passing standard (MPI). The current document [For94] can be retrieved via the world wide web from http://www.mcs.anl.gov/mpi/index.html. MPI defines, among others, constructs for point-to-point communication as well as for collective operations.

4 Programming models and methods

Sequential versus parallel model Today's programming tools are unable to map ordinary programs written in standard languages to parallel machines in such a way that they run consistently at high speed. The user who is willing to invest time to learn a more machine-specific programming model may be rewarded with a significant performance gain. Hence, there is a tradeoff between achievable performance and software development costs. Although, recently there have been substantial improvements to compiler technology, this tradeoff can be expected to continue into the foreseeable future. A survey of parallelizing compiler technology can be found in [BENP93].

Another important consideration when selecting a programming model is the tradeoff between portability and performance. Programs written in standard languages are obviously more portable than those written in machine-specific dialects. Because standard languages such as Fortran77/90 or ANSI C are sequential languages, the performance of such programs is limited, as explained above. Two attempts to standardize parallel languages were mentioned in the previous section: ANSI X3H5 and HPF. However, even if these efforts succeed, one cannot expect that they will yield efficient portable programs. Whoever has attended one of these standardization meetings knows that many constructs that made it into the final language could only be agreed upon after sometimes painful performance compromises.

Figure 2 displays the framework of programming models and languages used in this paper.

Message-passing vs. shared-address model Even the user who is willing to learn a parallel programming model will not find a single school of thought. One of the most significant current debates is the question of message-passing (MP) versus shared-address (SA) space model. Both models support the notion of (processor-)private and global shared data. In the SA model, global data can be read and written in the usual way (although it may be important for the user to be aware of the fact that data residing on a remote memory has a longer access latency than data residing on the local memory). In the MP model, remote data cannot be accessed directly; instead, a message must be sent requesting the remote processor to read or write the item on its memory module. The debate is over the question of whether it's reasonable to construct MP machines if SA machines can be built to provide the user with the much wanted SA programming model. It includes the following questions: how much more expensive is it to build SA machines; how much larger is the range of applications that can be run efficiently on SA machines; in reality, how difficult is the MP programming model; and, to what degree can future compilers translate SA programs onto MP machines.

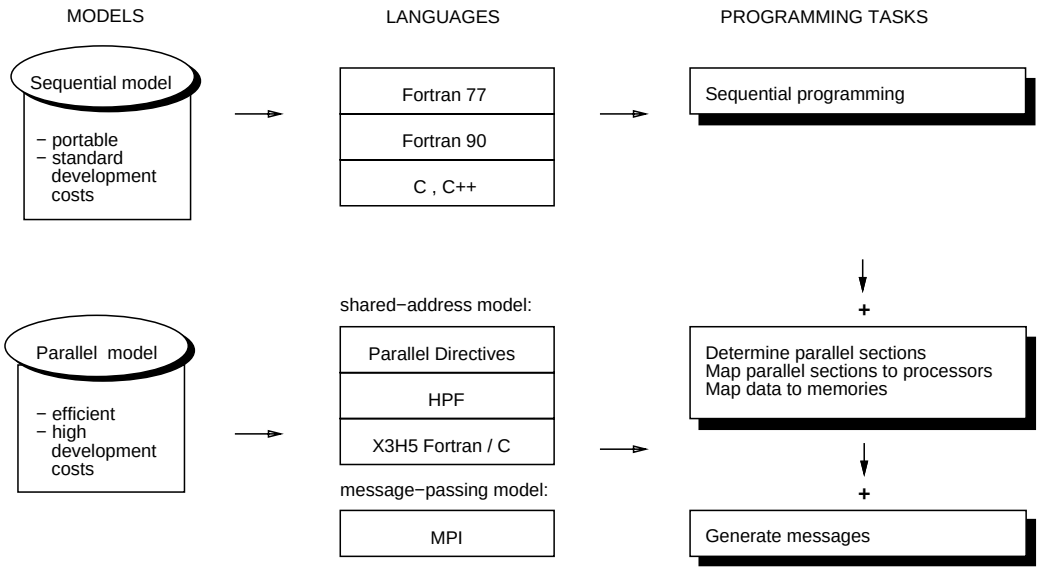

Figure 2: Programming models and languages for scientific and engineering applications

Finding parallelism How much and what form of parallelism does the user need to deal with? In both models it is the user's responsibility to identify parallelism – mostly in the form of parallel loops. Parallelizing compilers can often help find an initial set of parallel loops so that the programmer's task is simplified to identifying parallelism in the time-consuming loops left serial by the compiler. Such a programming methodology is described in [Eig93]. Compiler support for message-passing models is very limited. Usually it is the user's full responsibility to find parallel program sections.

Mapping parallelism to processors Once parallel sections are identified, there is an important additional programming step to map them to the actual machine. In the SA model, this involves transformations such as: interchanging loops to optimize memory reference patterns; splitting loops into two nested loops (loop blocking) – also to optimize memory reference patterns; serializing loops that have insufficient parallelism; and, inserting data prefetch operations to hide memory latencies. These techniques are of equal importance in the MP model as well. Additional steps include generating messages and partitioning parallel loops into parts executed by each processing node. These latter steps are perhaps the most tedious in message-passing models and have led to the development of the High-Performance Fortran language. HPF provides constructs for data distribution, thus allowing the compiler to generate messages automatically. HPF is an SA model and, although initially developed with a focus on MP machines, is now being implemented on some of the newer SA machines.

Mapping data to processors How much data distribution does the user need to deal with? This is the question which HPF focuses on. HPF provides constructs for mapping data onto multiple processing nodes in various ways. Although it is commonly agreed that successful data distribution is crucial to good performance, it is unclear what can be gained by what effort of data layout optimization. The user will have to experiment with data distribution schemes on current parallel machines and find out how much there is to gain him- or herself. The question of how well this process can be automated in a compilation tool is very open as well.

5 Conclusions

In the past five years, machine peak performances have greatly increased. The gap between peak and actual performance has increased as well. We have seen architectural trends going from shared-memory machines (with a modest number of processors) to massively parallel systems (with relatively high communication latencies). Currently we see the trend going back to systems with shared address spaces. Given this variety of architectures, it seems time to step back and look at what we have done; this is perhaps the most important conclusion today. Given that the problem of exploiting the variety of machines has been pushed onto the user, the issue of programming them has become crucial.

This has brought us to the dilemma at hand: to achieve good performance, programmers should use a parallel language that is close to the machine; on the other hand, among application scientists and engineers, there is a growing dissatisfaction with the necessity to invest time in computer science issues. The demand for standard language support is increasing as well.

Consequently, future trends will be greatly influenced by the degree to which automatic tools can transform standard languages into efficient parallel code. Presently, to what extent this will be possible is unforeseeable. The research area dealing with this question is very active and has made significant advances over the past few years. However, we are still far from the ultimate goal of automatically compiling a wide spectrum of standard language programs for a wide spectrum of machines.

References

[Ame94] American National Standard Institute. *X3H5 Parallel Extensions to Fortran, X3H5/94-SD1*, 1994.

[BENP93] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2), February 1993.

[CKPK90] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer Performance Evaluation and the Perfect Benchmarks^{TM}. *Proceedings of ICS, Amsterdam, Netherlands*, March 1990.

[Eig93] Rudolf Eigenmann. Toward a Methodology of Optimizing Programs for High-Performance Computers. *Conference Proceedings, ICS'93, Tokyo, Japan*, pages 27–36, July 20-22, 1993.

[For93] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical report, Rice University, Houston Texas, May 1993.

[For94] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee, Knoxville, Tennessee, May 1994.

[GNP+90] K. Gallivan, E. Ng, B. Peyton, R. Plemmons, J. Ortega, C. Romine, A. Sameh, and R. Voigt. Parallel Algorithms for Matrix Computations. *SIAM Publications, Philadelphia, PA*, 1990.

[Hwa93] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, Inc., 1993.

[KDL+93] D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C.-Q Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U.M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, J. Andrews, and S. Turner. The Cedar System and an Initial Performance Study. *Proceedings of the 20th Int'l. Symposium on Computer Architecture, San Diego, CA*, May 16-19, 1993.