Portable Parallel Programming Languages

Introductory paper to the session of this title at the Workshop on Challenges for Parallel Processing, International Conference on Parallel Processing, 1996

Rudolf Eigenmann

School of Electrical and Computer Engineering
Purdue University
1285 EE Bldg.
West Lafayette, Indiana, 47907
eigenman@ecn.purdue.edu

Abstract

In this workshop session, three speakers present their viewpoints and contributions to the topic of portable parallel programming languages. They are Dennis Gannon from Indiana University, David Loveman from Digital Equipment Corporation, and David Padua from the University of Illinois. Their presentations discuss extensions to C^{++}, Fortran, and compiler support for standard languages on parallel architectures, respectively. This paper introduces the topic *performance portability* on parallel machines. We will then try to draw a picture of where we are today in providing portable parallel languages to a growing user community, given that the three contributions represent major milestones along this path.

1 Introduction

What does "portable parallel programming languages" mean? Where do we stand today in providing such languages? How much more is there to do before we can claim that portable parallel programming is a reality?

These are questions that this article, with the help of the accompanying papers, is trying to answer. This section will provide motivation for the topic. We will summarize the three accompanying papers in the context of other approaches in Section 2. These papers represent major development efforts in portable parallel programming languages, discussing the HPC^{++} language design effort [2], the High-Performance Fortran effort [6], and an effort to support standard, sequential languages through automatic parallelization [3], respectively. They supply important evidence for characterizing the current state of the art in portable parallel software engineering. In Section 3 we will discuss this state and try to understand the significance of the three contributions for the long-term goal of providing highly portable parallel programming systems.

1.1 Is Portability Important?

There are many reasons to talk about portability: Software developed on a small desktop computer may eventually have to run on a large mainframe, desktop programs may need to run on a variety of platforms, computational problems keep changing so that existing software now has to run on machines with more resources (faster, more memory, better network connection), problems that are irregular in their computational characteristics may best run on heterogeneous machine suites, or our software simply outlives that ma-

chine, on which it was developed. This makes portability an essential, necessary attribute of all software!

In some cases porting programs is easy for the user. Machines can be "upward compatible" so that, for example, object code can be ported from an older to a newer machine. For the implementors, this compatibility is usually very costly.

In many other cases achieving portability is up to the software developers. Doing so is difficult even on sequential machines. It is not unusual that a piece of software, written in a standard programming language, tests correctly on the development machine, but then fails on a new platform. Possible reasons are that language standards cannot safeguard against certain programming mistakes, the compiler is not sophisticated enough to enforce language standards, and some language rules are not even enforcible by compilers.

An even bigger portability issue arises when we go from uniprocessors to parallel computers. Creating a work plan for a task to be performed by a group of workers simultaneously needs intrinsically more attention to details than creating a sequential work plan. One way of looking at this is that when doing work sequentially, we are used to processing objects step by step. While we operate on these objects, they can be temporarily in an inconsistent state. This is not a problem as long as we are the only ones working. We are confident that our steps lead to a consistent object. However, if there are other workers trying to help, they may look at the object at a bad time or modify it so we get confused. Therefore, coordination is needed. Coordination usually results in one worker having to wait until the object is in a reasonably well-defined state. This introduces delays, which we want to avoid if they are not absolutely necessary. Hence we want different work plans for sequential and parallel task forces. Consequently, the work plan is no longer portable.

The coordination necessary in a parallel program and its sources of inefficiencies are one reason for the lack of software portability. Additional reasons come from the fact that high-speed computers support parallelism in many different forms, such as instruction-level parallelism, vector parallelism, loop-level parallelism, and task parallelism. Furthermore, data can be placed in many different types of storage, such as registers, caches, local memories, or shared memories. Today's computer architectures are very diverse in the levels of parallelism and data storage they provide. If the programmer has to find the best mix of these resources for every machine, the resulting lack of program portability is unavoidable. However, even if the management of these resources is provided by sophisticated hardware mechanisms, compilers, or operating systems, the programmer often has to take special care in writing the programs such that these mechanisms can be effective. This means, for example, that the programmer must make sure that data placed in longer-latency storage by the compiler get accessed infrequently. Or, that the levels of parallelism supported "automatically" by the given machine will be apparent to the optimization tool.

1.2 Isn't there an easy solution to the portability problem?

Many products announced on todays market claim that they achieve portable software. They are not really wrong. But they are using the term portability differently from how it is understood here. In our context we are concerned with *performance portability*. A program that runs at high speed on one computer system should run with reasonable efficiency on another system. In many current computer applications, performance portability may not be an issue. If the machine speeds are satisfactory, correctness may be the primary concern when retargeting a program at this new computer. For those situations our discussion is irrelevant. Instead we address the reader who deals with the other important class of applications, where high computer performance is an essential and costly resource.

Even for performance portability one could argue that we could just "buy portability with *some* performance." After all, computers keep getting increasingly powerful. Can't we use a

fraction of this power to make up for the inefficiencies introduced by tools that map standard, portable languages onto high-speed machines? After all, this very same argument proved to be correct in the past, when we went from architecture-specific, un-portable assembly languages to portable, standardized high-level languages (HLLs)! Why is it not true any more?

The reality is that the inefficiency introduced by HLLs on sequential machines could be kept at a percentage of the overall performance. However, the inefficiency introduced through parallel processing increases with the number of processors. This means that the higher the "peak performance" – implemented through amassing parallel processors – the less efficiently a program runs. After a certain point, the application performance may even deteriorate with the increasing peak performance of a machine. Today, we are at a point where this reality is present in many programs.

2 Approaches to Portable Languages and Three Milestones

Portable programming languages can be realized in several ways:

2.1 Supporting sequential languages

Referred to above as the "easy solution" for the user, pursuing this approach is very demanding on the implementor's side. The paper by Padua and his research group at the University of Illinois shows what we have achieved so far and where we might be heading [3]. Entitled "Restructuring Programs for High-Speed Computers with Polaris", the paper describes new powerful techniques to translate programs written in standard sequential Fortran into parallel programs. The Polaris compiler is able to generate output for several parallel machines, hence they port sequential programs to these parallel computers automatically. The machine class includes parallel architectures that provide a shared address space. The papers shows, that on a SGI Challenge machine, Polaris is able to take advantage of the parallel processing capability across a range of programs.

2.2 Converging on the "best" machine architecture

Portability becomes a non-issue if the machines, to which it is difficult to port programs, are not being used. Although this is an important point to keep in mind, we will not discuss it further. It leads to questions of whether the variety of parallel machines offered by today's vendors will continue to be there, whether companies that have over-promised their machine performance will cease to exist, and – of course – what systems will prevail. In part, discussions of such issues can be found in other workshop sessions, where we ask for the right interconnection network [1] for creating the right architectures, and where we ask for evaluating their performance [8].

2.3 Creating a new portable language that will be used by everyone

Attempts to provide new programming languages for exploiting parallel machines abound. At least implicitly, most of them claim to be portable, although sometimes it is not taken to mean performance portability. Among the many interesting approaches are those who allow the programmer to specify parallelism implicitly, hence providing portability across a wide spectrum, from sequential to parallel machines. Dataflow and functional languages (e.g., SISAL [4], Id [7]) are just two of many paradigms that have intrigued us by the idea of being easy for the compiler to derive parallelism and easy for the architecture to exploit this parallelism. Another example is APL [5], which has pioneered concepts of data parallel programming. Some of its features can now be found in Fortran 90.

Although the many proposed languages have contributed to the discussion of new language standards, very few of them have become used more widely than in a laboratory environment or in a specific area. It almost seems to be true

that "the good languages are never the used languages". Because of this, in this workshop session we are also asking for extensions to commonly-used languages with features that enhance their performance portability.

2.4 Enhance current languages with performance-portability features

This topic is the focus of two accompanying papers:

Loveman at Digital Equipment Corporation [6] describes High-Performance Fortran and other enhancements to the Fortran language. His paper is entitled "Fortran: A Modern Standard Programming Language for Parallel Scalable High-Performance Computing". It describes how Fortran has evolved from its first definition to the youngest draft standard of Fortran 95.

In 1966 Fortran was designed as an "easy-to-learn language for which a compiler could generate very efficient code for a variety of computer architectures"; hence, a portable language. Among the revisions, FORTRAN 77 and Fortran 90 have become ANSI standards whereas the High-Performance Fortran Forum is developing a de-facto standard. Most important for our question of portability across parallel machines is the youngest of these efforts: HPF. Loveman's paper briefly reviews the features added by FORTRAN77 and Fortran 90 and then introduces HPF with its directives for data layout and placement (align and distribute) and parallel language elements (`forall` construct and `independent` directive). Examples illustrate the usefulness of the HPF features for porting programs between different architectures and maintaining them over time.

The paper "Portable Parallel Programming in HPC^{++}" by Gannon and his research group at Indiana University [2] describes a similar effort involving industry and academia nationally and internationally. The team is exploring versions of C^{++}, enhanced with features to express parallel programs in a portable manner.

HPC^{++} addresses both computer architectures that provide a single, shared address space (*context* in HPC^{++} terminology) and distributed address spaces (multiple *contexts*). Parallelism can be expressed explicitly in the form of parallel loops or by spawning parallel threads. Of further importance are the standard library functions (STL), parallel versions of which are being designed in the HPC^{++} effort. They support an implicit parallel programming style that composes programs from library functions that are already optimized for the machine at hand. Multiple contexts are supported through constructs such as distributing data structures (*objects* in C^{++}) and identifying the subrange of an object that belongs to the context of the current processor. For multiple-context situations, HPC^{++} supports an SPMD programming style where all participating processors execute the same program, each operating on the assigned data subrange. Data mapped to other contexts can be accessed via *global pointers*. Such accesses may be implemented in the underlying runtime system by communicating messages.

3 Discussion: How Portable can Parallel Programs be Today?

Perhaps the most challenging task of our portability discussion is to analyze the evidence that we have today and to compose a picture of the current state of the art in parallel programming languages. The reader shall be warned that this is by nature somewhat biased. However, the arguments presented here are not intended as a criticism of the presented projects. In the contrary, the accompanying papers are among the prime efforts underway to provide portable parallel languages. From them we can learn how far we have come, and how far we still need to go in order to achieve our ultimate goal of highly-portable software.

We will try to find answers to the following questions:

- How can portable parallel programming be accomplished for platforms with great archi-

tectural diversity in a way that still exploits machine-dependent features?

- What is the minimum set of new language features that the user who is knowledgeable of well-established languages will have to learn?

- Can we quantify the tradeoff between ease-of-use and achievable performance?

3.1 Ways of accomplishing performance portability

The three contributions give us insight into two different ways of exploiting machine-dependent features while retaining a certain level of program portability. The approach of automatically transforming standard sequential languages puts most emphasis on the portable user interface. Put somewhat extremely, we could argue that retaining the fully portable standard language is of such paramount importance to the large, non-computer-knowledgeable user community, that we cannot afford to change this interface. Machine-specific features have to be exploited by intelligent translators and we may have to accept lower performance where these translators fail to perform their optimizations. Padua's group is helping us to explore this dimension of the portability space. They have shown that significant progress has been made over the past few years, but also that much space remains to be explored in the future.

The alternative approach is being studied by the HPF and HPC^{++} communities. These two efforts chose to extend the existing standards of Fortran and C^{++}, respectively. An extreme view of this approach would be that taking advantage of machine-specific features is so important that we must retrain our programmers to learn new essential language features that enable portable performance. The big question is: What are these features? HPF explores the answer of providing programmers with a shared-address-space view of a parallel machine in which, however, one has to be aware of the fact that data is distributed across many processors' memories. The programmer defines in what way this distribution will happen, but the system (i.e., the compiler) does the actual tedious bookkeeping of array and loop indices and the orchestration of send/receive-type communication between processors. HPC^{++} explores the space even a step further. It allows the programmer to define multiple contexts, clearly reflecting the specific feature of the distributed-address-space machine class. Library functions are provided that facilitate the distribution and bookkeeping of objects across such multiple contexts. However, the programmer is aware of the SPMD execution of the program, which has multiple threads of control, each operating in its own context.

3.2 What is the minimum set of new language features that the user will have to learn?

The two approaches directly reflect the amount of language features that the user has to learn. If we manage to provide tools that optimize standard languages for parallel computers effectively, we relieve the user from having to learn any new language elements. In contrast, HPF requires the user to learn about parallel activities and data distribution; in addition, HPC^{++} requires the user to learn about multiple contexts and how to communicate between them.

Loveman has argued that, as languages and standards evolve, HPF-like features may be found in future standard Fortran. Perhaps, such a next generation language will unify the two seemingly different approaches. Perhaps users of some future Fortran or C dialect will write their first program version using a sequential language subset and will rely on automatic optimization technology. As the user community becomes increasingly aware and knowledgeable of parallel programming constructs, second generations of programs may be written or rewritten, expressing parallelism where sequential algorithms are not inherent to the problem. In all cases, optimization technology as described by Padua is an essential and probably increasingly important ba-

sic capability of all compilers. Both Gannon and Loveman have made important contributions to this field as well. Hence, expertise on how far compilers can go and where the user has to become involved, can be found in all teams that participate in the search for the most portable parallel language.

3.3 Can we quantify the tradeoff between ease of use and achievable performance?

One of the most important questions when comparing different approaches is how they perform. We would like to have evidence of how much performance and software development cost we gain or lose if we follow the one or the other approach.

The success rate with which automatic parallelization can be achieved is approximately one in two programs. This was measured with the Polaris compiler, given test suites such as the Perfect and the SPECfp95 benchmarks. Hence we have a few initial points on our tradeoff curve. However, more data on automatic parallelization will be needed to complement these points for new application areas (e.g., irregular and non-numerical programs), machines that do not support a shared address space, and non-Fortran languages.

For HPF and HPC^{++} we don't have evidence yet that would let us fill in the tradeoff diagram. However, several HPF compilers have recently become available and we can expect that performance numbers, across a wide range of applications written in this language, will become available soon. HPC^{++} is an even younger effort and we cannot yet expect hard proof of its concept. Thus, we will have to ask the same question in the future again.

Assessing ease of use is even more difficult than quantifying performance in terms of program execution speed. Many questions are important and most of them are difficult to answer: What languages and machine concepts do we teach our young software engineers? What are the supporting tools that help us learn new languages on-demand and help us find programming errors quickly? How willing are companies to invest in retraining their programmers if there is indication of long-term benefits? Therefore, assessing the ease of use ultimately remains with the reader and he or she will have to complete the tradeoff curve subjectively.

4 Conclusions

Performance portability is a very important property of a "good" parallel computer language. Providing such a language takes efforts at two fronts: Defining the language elements that are adequate for expressing a parallel computation and implementing the necessary compiler technology that can do the "uninteresting" programming tasks automatically.

Significant progress has been made at both fronts, as evidenced by the three accompanying papers in this workshop session. However, much more work is needed before the user community of parallel machines can see programs written in truly portable languages. Most importantly, we will need to provide more data that lets us compare different approaches for portable parallel languages. New language extensions for Fortran and C^{++} are being defined with broad support of industrial and academic groups. Compilers that support these extensions are now becoming available. We are ready to use these tools and to evaluate the obtainable performance of real application programs and how it compares to those obtained through compiler-optimized standard sequential languages.

References

[1] Seth Abraham. Interconnection networks: Dimensions in design. In *1996 ICPP Workshop on Challenges for Parallel Processing*, pages 45–51, August 1996.

[2] Peter Beckman, Dennis Gannon, and Elizabeth Johnson. Portable parallel programming in C^{++}. In *1996 ICPP Workshop on Challenges for Parallel Processing*, pages 132–139, August 1996.

[3] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Lee, T. Lawrence, J. Hoeflinger, D. Padua, Y. Paek, P. Petersen, B. Pottenger L. Rauchwerger, P. Tu, and S. Weatherford. Restructuring programs for high-speed computers with Polaris. In *1996 ICPP Workshop on Challenges for Parallel Processing*, pages 149–162, August 1996.

[4] J.T. Feo, D. C. Cann, and R. R. Oldenhoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, December 1990.

[5] Kenneth E. Iverson. *A Programming Language*. Wiley, New York, 1962.

[6] David Loveman. Fortran: A modern standard programming language for parallel scalable high performance technical computing. In *1996 ICPP Workshop on Challenges for Parallel Processing*, pages 140–148, August 1996.

[7] R.S. Nikhil. Id-Nouveau (version 88.0) reference manual. Technical report, MIT Laboratory for Computer Science, Cambridge, Mass., 1988.

[8] John Rice. Measuring the performance of parallel computations. In *1996 ICPP Workshop on Challenges for Parallel Processing*, pages 8–9, August 1996.