published in the Proceedings of
Supercomputing Symposium 92 (SS'92), Montreal, June 7-10, 1992

Towards a Portable Parallel Programming Environment

G. Gao, L. Hendren, P. Panangaden
School of Computer Science, McGill University

M. Feeley
Département d'IRO, Université de Montréal

L. Tao
Faculty of Engineering and Computer Science, Concordia University

M. Hancu, H. Hum, J. Lebensold, V. Van Dongen
Centre de Recherche Informatique de Montréal

Abstract

This paper presents an important project whose objective is to develop a *Portable Parallel Programming Environment*, called EPPP, for current and future generation parallel computers. Our environment will be *portable* in the sense that the user will be able to rapidly port his/her application on a variety of parallel architectures. That is, our goal is to allow the programmer to develop, debug and tune his/her application on a workstation; a simulator will provide the user's execution platform at low cost. To produce an executable program for a particular architecture, the high-level program will be combined with domain-specific and architecture-specific information to produce a compiled program that will run efficiently on the specified architecture. This information will be provided by the environment, the user, or a combination of both.

The proposed environment integrates a parallel programming paradigm, a compiling methodology, a simulator, and a complete set of analysis tools. Benchmark programs will include the traditional scientific array-based applications and more general-purpose applications that use dynamic data structures.

1 Introduction

This paper introduces the scientific framework for a proposed three-year project whose aim is to develop a Portable Parallel Programming Environment called EPPP. This integrated environment will be used for the development of programs that can be targeted to a wide variety of parallel computers. The environment itself will run on a UNIX-based workstation, and will include both compiler and performance analysis tools. The EPPP environment, once completed, will mark an important technological advance in parallel programming methodology and program portability. The motivation for building a Portable Parallel Programming Environment is well summarized by the following quotation taken from Supercomputing Review, February 1991:

> *The lack of application software is harming the parallel computing industry. To correct this deficiency, portable software tools must gain popular support.*

The inability to program parallel machines has hitherto been a severe impediment to the use of parallel machines. As a result the spectacular technological advances in high performance computing have had much diminished payoff at the level of industrial and scientific users. In order to develop parallel programs that harness the computational power that is available it is imperative that there be interactive, partially-automated assistance in the parallel program development process. EPPP is thus an urgently needed software development environment and methodology. Once made available to the industry, it will increase their competitive edge in advanced parallel software development and enable them to fully exploit the benefits offered by parallel computers for their particular applications.

After a two-year feasibility study that has been undertaken at CRIM in collaboration with the universities of McGill, Concordia, and Montreal, we have begun a project, pre-competitive in nature, which uses an innovative global approach. It relies on the combination of a parallel programming paradigm, a compiler and a development environment to assist the user in the preparation of efficient and structured parallel programs. It is clear that the complexity of parallel programs is such that adding a few new linguistic features comes nowhere near offering a solution to the problem. Nor

can pure compiler technology applied to traditional programs automatically parallelize the program to the extent that the underlying algorithm really permits. Hence the inevitable move towards interactive, semi-automated program development as proposed in EPPP.

2 Background

The contemporary computing landscape includes a rapidly broadening spectrum of parallel computer architectures: ranging from powerful uniprocessor architectures with aggressive instruction-level parallelism to what are commonly referred to as massively parallel computers for high performance scientific computation.

At the heart of these developments lies the single chip microprocessor based on the RISC (for Reduced Instruction Set Computer) architecture — the so called "killer micros" foreseen by Eugene Brooks [3]. The economics of computer technology ensures that the processor on a chip is, and will continue to be, the most cost-effective basis for high performance computer design. There is a strong indication that the rapid VLSI technology advance, which we have seen in the 1980's, will continue through the coming decade. Contemplating such an evolution of the microelectronic technology leads to the following conception of future trends:

- The performance of a conventional (RISC) processor will be dramatically increased by enhanced instruction-level (so called *fine-grained* parallelism. There are a number of architectural proposals which follow this choice: e.g. superscalar, superpipeline, and VLIW — all of which can issue multiple operations simultaneously. These processors are widely used as the CPUs for high-performance workstations. Thus we see that parallelism is making its way into the hitherto mundane world of desktop computing.

- Multiprocessor machines based on the connection of many nodes with microprocessor CPUs will continue to progress rapidly. As research into parallel architecture continues to make progress, more powerful, and scalable, parallel machines will become readily available with competitive cost/performance level. We have already seen that massively

parallel computers are making strong inroads into the territory formerly dominated by conventional (vector) supercomputers. It appears that parallel computers will be destined to take over much of general purpose computing from conventional "main frame" computers.

In contrast to these rapid advances in architecture and hardware, software development methodology for parallel machines has been slow in progressing. For instance, a cursory examination of the software tools provided today by some parallel computer vendors makes manifest the primitive state of the field. Parallel programming paradigms accessible to most users and portable to different parallel architectures are long overdue. Parallelizing compilers, based on analysis and transformation of existing programs, are still in their infancy. Finally, there is a serious lack of commercially available tools that integrate the programming paradigm and compiling technology and which allow users to develop efficient and correct parallel programs for various parallel machines.

As a result, the programmer often cannot develop efficient parallel programs without understanding details of the hardware architecture. One can gauge the difficulty of programming parallel machines by the establishment of an annual award just to acknowledge a "successful" effort in porting an application program to a parallel machine. The drawbacks of the lack of parallel programming support are:

- **Programmability:** It is neither practical nor desirable to ask scientists or engineers to become experts in parallel architectures in order to be able to write efficient parallel code. Nor should they be encumbered with programming details related to low-level synchronization and communication primitives of the underlying machines.
- **Portability:** An application developed for one parallel machine may have to be completely rewritten for another platform.

In the past decade we have witnessed a substantial effort, in both academia and industry, on the part of researchers learning to program parallel computer systems. Their experiences have often been painful, but from them we have learned many lessons. The task of programming parallel computers to utilize effectively various forms of potential parallelism in different

parallel machines is much more complex than programming sequential computer systems. We need a solution framework, an approach which should not merely consist of separate or isolated tricks in language design, compiling, performance analysis tools, or machine architecture improvement. We need a parallel programming system which integrates innovative ideas at all levels (programming paradigms, language support, compilers, and tools) in a unified and consistent environment. Furthermore, we must not alienate scientists who program in sequential languages such as C or Fortran and who have already spent considerable effort developing software in these languages.

Finally, we are also encouraged by some emerging interesting parallel programming paradigms and related compiling effort in the past few years. Among them, the one most relevant to the proposed research is the *Single Program Multiple Data* model (SPMD) which seems to be a promising approach for a wide class of distributed memory MIMD machines [8].

3 Our approach

The portable parallel programming environment we are developing will support the parallel software development process by combining three ingredients: a parallel programming paradigm, a compiler, and a development environment. We have chosen to develop a parallel programming environment evolving from C-like language with possible extensions. There are several reasons for this decision.

- There is a large amount of software that has already been developed in C. Many more C programs are being written everyday for a wide range of different hardware platforms, and there is no sign that this trend will slow down soon. Therefore, we feel that one challenge for researchers in the areas of parallel programming is to work towards solutions that: are easily assimilated and adapted by the community programming in C-like languages.
- C is known for its flexibility, in particular for its ability to handle both static and dynamic structures. Although Fortran-like languages still dominate the world of scientific numerical computation, more and

more scientific C programs are developed. Furthermore more scientists are becoming aware of the power of C for expressing dynamic data structures thus opening up possibilities that were simply not thought of, till recently, in the context of scientific numerical calculations.

- While there has been considerable progress in the parallelization of programs written in Fortran-like languages, little progress has been reported for parallelizing C. There are challenging research problems to be solved for parallelizing C-like programs, particularly those that use dynamic data structures.

- Also, many of the analysis and transformation techniques developed for C-like languages will be immediately applicable to other languages such as Fortran and its variants.

Our system is an integrated environment capable of performing program analysis and transformation to exploit parallelism at various levels suitable for different target architectures. The objectives of the EPPP system are to support the following aspects.

- **Program Analysis.** Before performing program transformations, the system should analyze and structure the input program. For C-like languages, the fundamental analyses are to detect dependency and interference. A clean subset of C is the starting point for such analyses. A key feature of the EPPP system is that such analyses are general, accurate and pervasive. In this regard, *analyzability* is a key factor in the system described by this proposal. For example, pointer analysis techniques are presented in [7, 9] while array analysis is considered in [2, 12].

- **Program transformation.** After the program analysis phase, the system will perform program transformations to expose parallelism at various levels. The focus is on loop transformation techniques. In EPPP, a collection of program transformation techniques will be developed, using the dependence information gathered during the program analysis phase. These techniques will be organized into an interactive system which will provide transformations of three main sorts: transformations for exposing fine-grained parallelism (see for example [1, 5]), transformations for exposing coarse-grain parallelism and

global transformations (as explained in [12, 11]). In all three instances, novel strategies and methods (based on our current research results) will be developed depending on the program characteristics, the architectural model, and machine specific information.

- **Data Parallel Paradigms for C** In the EPPP system we intend to support data-parallel paradigms for C. Our SPMD model for C will support Fortran-style array-based data-parallel paradigms, as well as more general types of programs that may use dynamic data structures. We also go beyond the traditional SPMD model presented in [8], and study some alternative ways to support data parallel paradigm based on the single-program multithreaded control model.

  The key features of EPPP include (1) exploiting both user-directed and automatic data partitioning and distribution for array-based SPMD model, (2) exploiting data parallelism for dynamic data structures in non-scientific programs.

- **Mapping techniques**

  The mapping takes place at two-levels. During the compilation, mapping techniques are used for generating the code for the given target architecture. In particular, mapping tools that calculate the amount of computation involved for a particular data distribution of an array can be the basis for automatic data partitioning [6]. After the compilation, re-mapping techniques are used when the number of processors is different from the one at compile time, such as in fault-tolerant and dynamically partitionable architectures. See for example [10].

- **Simulation environment**

  The EPPP system is organized in an integrated environment with a set of algorithms and tools packaged in a user-friendly fashion. Driven by the objectives outlined above, the tool sets should guide the users in developing a correct and efficient parallel program, and tune their programs to achieve the best results. A simulator will enable the user to analyze his/her application on a Unix-based workstation. See also [4]. It will include:

  - a source-level interpreter,
  - a machine assembly-level simulator,
  - a performance visualization tools.

– and some debugging facilities.

Our approach will also concentrate on providing structured methods for integrating both domain-specific and architecture-specific information into the parallel program. These methods will allow the separation of the parallel program from architecture-specific details, and will also provide a basis for our program mapping and program transformation techniques.

Although the principles of EPPP are applicable to a broad range of high-performance parallel machines, it is unrealistic to expect that we will be able to define an implementation strategy in a *completely* machine independent fashion. Thus, we choose to focus on some representative architectures to design our prototype. Five different types of architectures will be considered.

- Superscalar/VLIW based computers, such as the RS/6000.
  In superscalar/VLIW processor-based computers, instruction pipelining must be exploited to efficiently use its potential parallelism.

- SIMD parallel computers, such as the DECmpp 12000.
  SIMD (*single instruction stream, multiple data stream*) multiprocessors represent a spectrum of parallel machines that have been successfully built with a large number of processors and applied to scientific computations where the problem domain has a regular structure.

- MIMD shared-memory parallel computers.
  Among the MIMD (*multiple instruction stream, multiple data stream*) multiprocessors we would like to investigate are the class of machines with a non-uniform memory access architecture. This represents a class of parallel machines in which the (physically) distributed, (logically) shared memory of the machine can be referenced by any processor but the cost of accessing a particular physical location varies with the distance between the processor and the memory module being accessed.

- MIMD distributed-memory parallel computers.
  We also would like to investigate multiprocessor systems with (logically) distributed memories, such as the Volvox Transputer network. Each processor in this type of system contains its own memory, and

there is no shared memory in the system (either logically or physically).

Frequently, these systems have a switching network which provides a rich interconnection mapping between processor nodes. Therefore, communication costs between any pair of nodes varies according to their distance in the particular topology.

- multicomputers.
  For example, a network of RS/6000 interconnected via fiber-optic links is a multicomputer since each processor is an autonomous computer.

4 Conclusion

In contrast to the fast evolution of parallel architectures, the software development necessary to exploit the full capabilities of these platforms is slowly progressing. This paper introduces the global framework of a project that addresses this problem. In this project, we propose an environment that integrates all the necessary components that will enable the user to develop his/her application on a Unix-based workstation, in a portable fashion. Its main components are:

- a clean language based on C and a parallel programming paradigm based on the SPMD model.
- a compiler technology that includes analysis, program transformation and mapping techniques.
- a simulator environment that will enable the user to analyze the performance of his/her application program.

After having studied the feasibility of this approach, we are currently working on the design of the various parts mentioned above.

References

[1] A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm. In *Proceedings of the Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[2] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, MA, 1988.

[3] E. Brooks. The attack of the killer micros, 1989. presentation in the teraflop computing panel discussion. *Supercomputing 1989, Reno, Nevada*, 1989.

[4] J. Bruner et al. Chief: A parallel simulation environment for parallel systems. Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, November 1990.

[5] G. R. Gao, Y. B. Wong, and Qi Ning. A Petri-Net model for fine-grain loop scheduling. In *Proceedings of the '91 ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 204–218, Toronto, Canada, June 1991.

[6] Manish Gupta and Prithviraj Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compiler on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3(No. 2):179–193, March 1992.

[7] Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, January 1990.

[8] A. H. Karp. Programming for parallelism. *IEEE Comput. Mag.*, 20(5):43–57, May 1987.

[9] William A. Landi. *Interprocedural aliasing in the presence of pointers.* PhD thesis, Rutgers University, 1992.

[10] L. Tao. Mapping parallel programs onto parallel systems with torus and mesh based communication structures. Ph.d. thesis, technical report ms-cis-88-59, University of Pennsylvania, 1988.

[11] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London and MIT Press, Cambridge, MA, 1989.

[12] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.