Programming Languages and Models for Parallel Multi-level Architectures

Claudio Schepke, Nicolas Maillard
Grupo de Processamento Paralelo e Distribuído
Instituto de Informática – Universidade Federal do Rio Grande do Sul
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil
{cschepke,nicolas}@inf.ufrgs.br

Abstract

The demands of large computational resources has stimulated the search for new alternatives to add performance to computer architectures. Parallel approaches are used to increase the power of processing of computers. These approaches include the use of multi-core, multiprocessors and clusters and grids architectures. The use of multi-level parallelism is a solution to increase the power of computers processing in the actual stage of computers development. However, parallel programming this levels is not a trivial work. On the other hand, there are many solutions to program efficiently a specific parallel level. In addition, there are some works that show solutions to program more than one parallel level. In this way, this work presents the state of the art in terms of parallel programming techniques, including parallel programming languages and models.

1. Introduction

In the recent years a large number of computers systems has been available to the market, with a considerable range of resources that meets the needs of developers of software. This fact can be clearly seen when holding the composition of the 500 machines with greater capacity for processing of the world, which are composed of several processing units (processors) interconnected, either through a common bus, or through special networks, which are aimed for many applications [12]. Usually, these solutions are based on the development of parallel architectures [14]. The use of vector machines, multiprocessors, and, currently *multi-core* systems has been some of the alternatives. These technologies can also be combined through the formation of *clusters* and *grids*. Thus we have the composition of platforms with multiple levels of parallelism.

At the same time that there is progress in the development of hardware, especially on parallel architectures, there is a need to provide resources for programming compatible with the different computing environments available. Moreover, is also important that there are mechanisms for programming able to integrate the various existing parallel architectures, simplifying the programming process. Thus, there is a layer of abstraction between the application itself and its platform for implementation, which may be made through specific programming libraries.

Though already exist apparently satisfactory solutions, specially for homogeneous systems, the way to explore in practice the most of computer resources existing in each of the different parallel machines that compose the system is not easy. Mechanisms for efficient programming for a specific form of parallelism cannot be apply to the others because they are generally quite specific. Integrate them into a single application, that is, use them combined in a single code, seems to be the most appropriate solution as a way to potentially what each mechanism offers individually. The challenge is to determine how best to achieve this integration dynamically in an environment with multi-level parallelism, scheduling the processes to maximize the use of resources.

In this context, this article presents as theme of research the parallel programming of environments with multiple levels of parallelism, which involves the convergence of many forms of parallelism and high performance solution for applications development. This proposal includes the study of tools and resources of parallel programming that can hold an abstraction of different levels of parallelism.

2. Multi-level Parallelism

Besides the application itself and the adopted strategy of parallelization, environment, planning and implementation must also be considered for the development of a concurrently application. The parallel computing environments, especially for *cluster* and *grid*, are increasingly heterogeneous of its composition. On the other hand, architectures *multi-core* with different amounts of processing cores be-

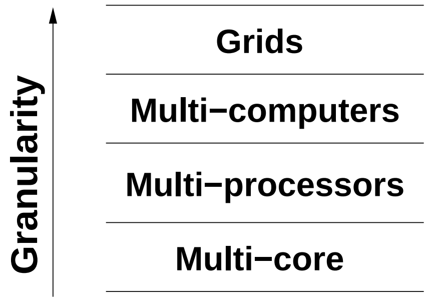

Figure 1. Multi-level Parallelism

gins to emerge. Consequently, these environments also providing a multi-level parallelism.

Multi-level parallelism has several levels of parallel abstractions The different levels of abstraction of parallel processors may be in themselves (*multi-core*), an internal computer (*multiprocessor*) or between multiple computers (*clusters* and *grids*), creating a hierarchy as shown in Figure 1. This figure shows that the granularity of the process or task increases as higher levels of parallelism are adopted.

The management of each level of parallel abstraction is done through specific mechanisms:

- **In processor level** - The flow of instructions is defined by the core or by the implementation of register in hardware. Thus, the control is done by instructions in *assembler*.
- **In level of operating system core** - The flow of instructions is defined by processes or *threads*. The control of the flow of instructions is done through calls to the operating system.
- **In level of *middleware management*** - The flow of instructions is a communicant process. The control is done through libraries for inter-processes communications.

It is therefore up to the programmer to use different tools for the implementation of a program to explore many levels of parallelism.

You can imagine, for example, a program implemented with the technique of divide and conquer, which create heavy processes in the first division and then pass to shoot *threads* as a way to use processors efficiently *multi-core* in an environment of *cluster*.

Ensuring the portability of applications and efficient use of resources is the great difficulty of existing parallel implementing environments, because the programming tools currently available are designed specifically to only one level of parallelism, which limits the potential of its use on another level. Moreover, it is difficult to control the way a parallel application will be enforced, since, regardless of the level of abstraction, different ways of mapping the flow of instructions may occur. Who decides this is a library of communication or a compiler.

3. Classical Parallel Programming Interfaces

Classical *Application Programming Interfaces* (APIs) for parallel programming are quite specific, focusing only one level of parallelism. In this sense, there are some patterns of programming for each level of parallelism, as follows:

- *Multi-core*: we utilize the standard **Posix Threads** as a way to create light processes (*threads*) [1].
- Multiprocessors: we use the **OpenMP** library, which allows call parallel operations through simple commands (reserved words) during the writing of code [5].
- Multicomputers: for inter-processors communication it are adopted the standard **Message Passing Interface** (MPI), especially in languages Fortran and C, where it is possible to communicate synchronous and asynchronous, collective and mapping de communications using Cartesian coordinates [9], and **Java Remote Method Invocation** (Java RMI), which allows the call of remote methods, ensuring the implementation of a distributed applications [7].

Another classic pattern of programming that can be mentioned is ***High Performance Fortran*** (HPF). HPF is an extension and modification of the standard language Fortran. It was developed to high performance for multicomputers based on parallel architectures and has portability for different architectures.

HPF support the use of data parallel programming technique and has open interfaces and interoperability with other languages as C and paradigms of programming as MPI. Another characteristic of HPF is the provision for future improvements in the language, and on implementations of standards Fortran and C. Because of these characteristics many applications of High Performance Computing (HPC) were implemented using HPF, especially for weather and climatology.

Although the resources previously presented have become a pattern for each type of environment, they hardly offer the same performance when ported to other environments. The same is true for other similar and less popular programming parallel APIs. Thus we can conclude that you can not program effectively, with a single interface, different levels of parallel abstraction.

4. Programming Languages and Models for Multi-level Architectures

Some attempts to overcome the limitations imposed by traditional parallel programming interfaces have already been proposed [2]. These solutions are based on *Single Program Multiple Data* (SPMD) or *Partitioned Global Address Space* (PGAS) models of programming.

The SPMD model offers mechanisms to specify the parallel computing and the distributed data structures. The distributed data and their allocation in each processor must be manually set. During the execution of the program occur also stages of communication and synchronization. Thus, we can say that there is a multi-cooperation of the application because the developer is aware of all the interactions of the program.

The main advantage of SPMD model is the simplicity. With this type of programming language is clearer to understand the functioning of the parallel implementation. With that, it is possible get a high level of transparency in execution, allowing also a high level of portability. Furthermore, the explicit management shadows to understand the algorithm, contributing to the introduction of programming errors. It is also required to the user to manually control the data fragmentation, communication and synchronization among the instances of the program.

The languages developed in the PGAS model have a model of memory in which a global address area space is logically partitioned so that each part is from a local processor. This kind of language is typically implemented in distributed memory systems and create a virtual address space using libraries of communication.

PGAS languages offers abstractions for the construction of distributed data structures and communication among cooperatives instances of the code. Although the purpose of these languages is to increase the capacity of writing the code, they are still limited in terms of providing an overview of parallel computing.

Some programming languages that allow the programmer considers a large-scale computing environment as a unified system, similar a shared memory environment are presented as follow.

- ***Unified Parallel C* (UPC)** - Is an extension of the C programming language developed at Berkeley University for high performance computing in large-scale parallel machines [16]. The language provides an uniform model of programming for both shared memory and distributed systems. UPC abstract the SPMD model of programming where the parallelism is set before the execution, and each flow of execution is destined for a processor. So the environment can be consider as a single system of shared memory in which processors can read and write the variables, though they are physically associated with a single processor.

- **Co-Array Fortran (CAF)** - It has similar properties to UPC implementation, being implemented in Fortran [11]. CAF is an elegant extension of Fortran, supporting the SPMD model of programming. Moreover, the language includes features of the next standard version of Fortran. The name of the language comes from the implementation of a new type of array called *co-array*. This feature is used to refer *images* (multiple cooperatives instances) of a program in a SPMD model. Each image can access remote instances of a variable by the indexing of a dimension of *co-array*. A variable declared in a dimension *co-array* allocates each image into a copy of the variable. The way to create a *co-array* is similar to create arrays in Fortran. The language provides also synchronization cooperatives routines to coordinate the images.

- **Titanium** - is a language developed at Berkeley as a SPMD paradigm for Java [15]. Titanium increases various features of Java, including support for iterations with multi-dimensional vectors, sub-vectors and copy operations, classes with unchanged values and *regions* that support memory management oriented to performance as an alternative garbage collector. The language support among instances of the program developed in SPMD through primitives of synchronization and communication, methods and variables that enable the synchronization in isolation, and a concept of private and shared reference.

- **Chapel** - is a programming language developed by Cray, being part of a larger project known as *Cascade* [4]. Chapel provides a higher level of abstraction for the expression of parallel programs than other programming languages. Moreover, the language offers a separation between the development of the algorithm and the details of the implementation of data structures. Chapel supports a *multithreaded* model of programming, providing abstractions for data and tasks parallelism.

- **Fortress** - is a tool for efficient and secure high-performance programming designed by SUN [13]. The language is based on Fortran. While its syntax is innovative, Fortress was developed enabling programming similar to a mathematical notation. With this, it is believed that the development of the code is easier for scientists. The fundamental components of a code Fortress are *objects* that define the variables and methods, and *traits*, which declare a set of methods, both abstract and concrete. Fortress is an interpreted language, and the interpreter runs on the Java Virtual Machine, implementing a small part of the specification of language.

- **X10** - X10 is an experimental programming language developed by IBM in partnership with academic institutions [6]. The purpose of the language is to offer new techniques for implementation that provide a scalable and optimized parallel environment managed in run-time. X10 offers all traditional features of Java programming language, for both *Symmetric Multi-Processors* (SMP) and *clusters* environments.

All these tools provide a layer of abstraction, making homogeneous the mechanism of implementation. At the same time, it is not possible to extract parallelism in all levels that the architectures offer, since these were not always adequately abstracted in the tools, as for the case of *multi-core* architectures.

However, solutions to this problem can be made using, for example, the inclusion of *threads* in the creation of new processes. An example is the combination of MPI and *threads* for the development of applications, both in user level as in MPI implementation level [10]. If this is done in a transparent form, there is another level of parallelism included in the abstraction tools.

Another aspect that can be mentioned is the possibility of creating dynamic processes, namely the creation and launch of new processes in run-time. Thus, it is possible to load balancing in run-time, or even better distribute the granularity of the processes. In according, several improvements and implementations have already been developed in the resources existing in the interprocessors communication library MPI2 [8], especially for launching the mechanism of dynamic processes offered by the *spawn* function [3]. MPI2 standard offers other important resources to HPC, such as parallel reading of files, which is especially useful for pre and post data processing.

5. Conclusion and Future Works

This article contributed especially, listing a wide range of techniques and resources for parallel programming, as for one level of parallelism, as for more than one level, through abstractions of the real parallel model used. These resources are fundamental to understanding how can be made a program exploring effectively three levels of parallelism for example, defining properly the parallelism of the application in each layer.

The use of multi-level parallel architectures is a solution to increase the computing capacities, making possible the efficient computing of large applications. Among these software applications we have a special interest for weather and dynamics of fluids simulations. The idea is continue this work determining forms to distribute data for a large application among different parallel levels.

References

[1] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, Reading, 2001.

[2] D. E. Bernholdt. Component architectures in the next generation of ultrascale scientific computing: challenges and opportunities. In *CompFrame '07: Proceedings of the 2007 Symposium on Component and Framework Technology in High-Performance and Scientific Computing*, pages 1–10, New York, NY, USA, 2007. ACM.

[3] M. C. Cera, G. P. Pezzi, M. Pilla, N. Maillard, and P. Navaux. Improving the Dynamic Creation of Processes in MPI-2. In *Proceedings of Euro-PVM/MPI*, volume 4192, Bonn, Germany, 2007. Lecture Notes in Computer Science.

[4] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, August 2007.

[5] R. Chandra. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, USA, 2001.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, 2005.

[7] J. Farley. *Java Distributed Computing*. OŔEILLY, Cambridge, MA, USA, 1998.

[8] W. Gropp, L. Ewing, and R. Thakur. Using MPI-2 - Advanced Features of the Message-Passing Interface. *The Mit Press*, 1999.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[10] J. V. Lima and N. Maillard. Aplicações Dinâmicas MPI-2 com Threads. In *ANAIS, Oitava Escola Regional de Alto Desempenho*, Porto Alegre / RS / Brasil, 2008. Sociedade Brasileira de Computação - UFPEL / UNISC / UCS.

[11] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[12] Top 500. Top 500 Supercomputing Site, Jun. 2008. Disponível em <http://www.top500.org>. Acesso em jun. de 2008.

[13] M. Weiland. Chapel, Fortress and X10: novel languages for HPC. Technical report, University of Edinburgh, Edinburgh-UK, October 2007.

[14] B. Wilkinson and M. Allen. *Parallel Programming: Using Networked Workstations and Parallel Computers*. Prentice Hall, New Jersey, 1998.

[15] K. Yelick, L. Semenzato, G. Pike, C. Miyamato, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.

[16] K. Yellick, D. Bonachea, and C. Wallace. A Proposal for a UPC Memory Consistency Model, v1.0. Technical report LBNL-54983, Lawrence Berkeley National Lab, May 2004.