Environment for the Automatic Development and Tuning of Parallel Genetic Algorithms

Paola Caymes-Scutari and Germán Bianchini

Laboratorio de Investigación en Cómputo Paralelo/Distribuido (LICPaD)
Departamento de Ingeniería en Sistemas de Información, Facultad Regional Mendoza
Universidad Tecnológica Nacional. (M5502AJE) Mendoza, Argentina.

Abstract. The use of high performance computing has been gaining more and more followers in the different branches of Science and Engineering given the potential offered to deal with complex and big problems, and the increasing economic facilities to configure some kind of parallel machine. However, the parallel programming paradigm involves additional aspects to the merely functional which could provoke different kinds of bottlenecks in the performance of the applications. Such difficulties may represent critical obstacles specially for the non-expert users. In this paper we present an environment to provide general support for the automatic development and tuning of parallel applications. The environment provides an interface to guide the user in the specification of the problem and the solution, which makes transparent the process of code generation and instrumentation. Because the parallelization of any problem/solution is hard to generalize, the environment tackles different classes of problems. In this article, we introduce the class of Parallel Genetic Algorithms.

1 Introduction

The efficient use of parallel systems presents different limitations in particular for general and non-expert users, given the amount of aspects involved by the paradigm, such as concurrency, parallel programming models, the synchronization, the communications and the different kinds of middleware, among others. In addition, there is a set of additional concepts which are crucial to the suitable behaviour of the parallel application: the decomposition and mapping techniques, the granularity, the concurrency degree, the load balancing, the scalability, etc. [5]. Through the years, different approaches and/or tools appeared to make easier the task of the user. The more significant approaches to our field of study are on the one hand the communication libraries such as PVM [2] or MPI [6] which encapsulate the communication primitives, and on the other hand the skeletons to implement certain parallel behaviour patterns [1, 7, 8, 10]. However, even though these approaches offer facilities at certain levels, they also present certain restrictions and require a certain degree of knowledge about the concepts related to concurrency and parallelism in order to carry out an adequate implementation of the program. The main reason why these limitations are present is related to the diverse nature of each problem and of each execution environment, which makes more difficult the automation of the development and performance improvement processes. Such limitations have motivated the development of an environment (presented in Section 2) to provide support and automation in the generation and tuning of classes of parallel applications, by taking profit from common structures and techniques to decompose, assign, compute and communicate.

In particular, the environment includes the class of Parallel Genetic Algorithms (PGAs) [9]. Genetic algorithms (GAs) use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover [3]. A typical GA is initialized with a population of random guesses and directs the population (along a series of time steps or generations) toward convergence at the global optimum. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. The fitness function is defined over the genetic representation, it measures the quality of the represented solution, and it is always problem dependent. The GA stops after a predetermined number of iterations or when a termination condition is satisfied. In order to exploit this search method and explore the search space in parallel, we consider in particular the class of PGAs which involves *multiple populations* (or islands) and incorporates the *migration* operator [9]. In this approach, the general idea is to execute several single GAs in parallel and to apply an additional genetic operator, the migration, to interchange the characteristics of the populations at certain regular periods of time. The migration can be applied in different manners and following diverse criteria to determine which individual to migrate and which one to replace. In this way, new search spaces could be explored at each population. This approach is frequently called *island model*.

2 Automatic Development and Tuning Environment

A completely automatic parallelization of any application is a hard (if not impossible) task, as mentioned before. This is why the presented environment for the automatic development and tuning of parallel applications deals with classes of problems, and obeys to a cooperative approach. Thus, on the one hand the intervention of the user is required for the characterization of the pair problem-solution. And on the other hand, the environment is prepared to: i) provide the means to guide and assist the users in depicting the pair problem-solution, ii) formalize in a specification the information provided by the user, iii) support the automatic generation of the parallel application by combining the information provided by the user and the corresponding skeleton of the class of problem, and iv) provide the support to automatically tune the behaviour of the application. From the architectural point of view, the environment is mainly divided into two modules: an **interface** to guide the user along the problem specification process, and a **translator** that transforms the specification into the tunable parallel application. The specification language has been defined by abstracting the elements and the parameters of the class of problem under consideration.

The interface (or first module) takes as input the entities described by the user. The first objective of this module is to provide a clear and intuitive interface to the non expert user, thereby facilitating the characterization of the problem and its solution so that the subsequent generation of the source code of the application is transparently carried out. The interface assists the user in the process of entering the parameters and operators of the PGA without need for the user to have specific knowledge relating to the specification language, the programming one, or even without knowing the appearance of the code that will constitute the automatically generated parallel application. For the majority of the parameters and operators, the interface provides default values and/or definitions that facilitate and guide the user's choices along the initial steps. In the particular case of the PGAs, some of the parameters to specify are for example: the fitness function, the termination condition, the type of selection, the size of the population, the amount of available computational nodes, among others. The second objective of this interface is to formalize a specification with the aim of condensing all the obtained information, according to the predefined syntax for the specification. Such a specification is the output of the first module. At the same time, it constitutes the input of the second module.

The existence of the specification as an independent unit within the process of automatic generation of the application is important for two reasons: on the one hand, it allows completing the specification of the problem in several sessions before building the application. On the other hand, the already existing specifications can easily being retrieved for making modifications to them or to generate new specifications from them, allowing for the reuse of some parts of the specification, such as those relating to the execution environment, population sizes and individuals, etc.

The translator (or second module) manages the information provided by the specification for interpreting the skeleton of the PGA. Thus, the fitness function is included in the corresponding method of the skeleton, the corresponding type of selection is set, and so on. Once the skeleton is completely filled, the application is built. In consequence, the output of the second module is in this case the PGA application's source code. Note that the generated application is already prepared to be dynamically tuned according to the current conditions of the execution environment, given that the application inherits the instrumentation inserted into the skeleton for that purpose. The advantage is that the whole translation, instrumentation and tuning processes are completely transparent to the user.

The instrumentation inserted into the application for tuning purposes, is related to the improvement of diverse performance bottlenecks, inherent to the parallel paradigm, and to the particular class of problem. In this case, we selected the class of GAs given that the method has been applied to an extremely wide range of problems [4]. Despite the potential of this search method, it has a series of parameters whose behaviour depends on the problem, the input or the environment. Some examples are the rate of migration, the selection method, the rate of mutation, etc. Fortunately, by means of the tuning process, they may be overcome according to the particular conditions of each execution.

3 Conclusions

The use of parallel/distributed systems is increasing given the computational power provided to solve big and complex problems. However, the parallelization of any application is not a trivial task, and the effort of parallelizing an application not always ensures an acceptable performance (expected or required). In this article, we present an environment which provides automatic generation and tuning of classes of parallel applications. The user cooperates with the environment by classifying the problem-solution to solve, and by characterizing the elements necessaries to depict that pair problem-solution. The environment automatically generates a formal specification encapsulating all the information, which is used in the following steps for generating the instrumented source code of the application. In consequence, the user is able to generate a parallel and tunable application in a transparent manner, even with no knowledge aboutthe specification and/or programming languages, the tuning technologies, or the parallel paradigm issues.

References

1. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P3L: A structured high level programming language and its structured support. Concurrency: Practice and Experience, 7(3), 225–255 (1995)
2. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.: PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Network Parallel Computing. MIT Press. Cambridge, MA (1994)
3. Goldberg, D.E.: Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, Reading, MA (1989)
4. Goldberg, D.E.: Genetic and evolutionary algorithms come of age. Communications of the ACM, 37(3), 113–119 (1994)
5. Gramma, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing (2nd Ed.). Pearson Addison Wesley (2003)
6. Groop, W., Lusk, E., Skjellum, A.: Using MPI - Portable Parallel Programming with the Message-Passing Interface (2nd Ed.). The MIT Press (1999)
7. Kulkarni, S.: An intelligent framework for Master-Worker applications in a dynamic metacomputing environment. Computer Science Department. University of Wisconsin - Madison (2001)
8. MacDonald, S., Anvik, J., Bromling, S., Schaeffer, J., Szafron, D., Tan, K.: From patterns to frameworks to parallel programs. Parallel Computing 28(12), 1663–1684 (2002)
9. Nowostawski, M., Poli, R.: Parallel genetic algorithm taxonomy. KES 1999: 88–92 (1999)
10. Sérot, J., Ginhac, D.: Skeletons for parallel image processing: an overview of the SKIPPER project. Parallel Computing 28(12), 1685–1708 (2002)