

EDITORIAL

High-level parallel programming in a heterogeneous world

1 | INTRODUCTION

During the last decade, parallel programming has evolved in an unprecedented way. Fifteen years ago, the future of parallel computing seemed to consist on the advent of multicore processors composed by an ever-increasing number in the core count per CPU, and their interconnection to form larger clusters. Programming models, such as OpenMP¹ that allows to transform sequential C and Fortran codes into parallel versions with low effort, without requiring to explicitly handle threads or to share memory among them, seemed to be the winner choice in a world where computers would include more and more cores inside a single chip. Message-passing paradigms, such as MPI,² provided a solution to interconnect these computers in larger facilities.

Suddenly, the use of dedicated hardware, originally designed to render graphics, to perform general computations, changed the rules of the game. The advent of GPUs as general computing device and their associated programming model, CUDA,³ steadily colonized the TOP500 supercomputing list.⁴ The success of this computing model, composed of very simple computational units that are orchestrated to perform the same operations on an increasing number of data elements, made the landscape of supercomputing much more interesting.

Since GPU programming is not precisely an easy task, the last ten years saw the rise of new programming models for heterogeneous systems, which aim to offer a unified view of the different computational units. These proposals intend to reduce the complexity of parallel programming, while keeping the performance obtained by their manually programmed counterparts. To succeed, these proposals should pass the test of time, being extensible enough to include new computing devices that still do not exist. New computing architectures, such as the family of XeonPhi coprocessors,⁵ pose new challenges to the designers of these high-level parallel programming models.

To raise the level of abstraction, parallel programming patterns have emerged as a way of expressing parallelism in existing sequential applications. Many algorithms match with parallel patterns, being easily exploitable by heterogeneous parallel architectures.⁶ Algorithmic skeletons are commonly used to express parallel patterns since the 90s.⁷ They abstract the application functionalities from the implementation details, allowing a common representation of algorithms in multiple and diverging parallel platforms.

In this context is where the “High-Level Parallel Programming” conference takes place. Starting in 2001, HLPP has gathered specialists from all around the globe to discuss recent advances in the topic of how to develop better software frameworks that cope with this increasing level of complexity. The particular format of the HLPP conference, with informal proceedings distributed among its attendants, allows the presentation of cutting-edge, ongoing work that receive direct feedback from the audience, thus enriching the research carried out. In an era of big conferences that attract several hundreds of researchers that should compete for the attention of the audience, scattered among different tracks that take place in parallel, the HLPP series of workshop/symposia keeps the spirit of specialist meetings, where all the speakers are equally accessible and where interesting discussions are carried out after each presentation.

This special issue of the International Journal of Parallel Programming contains revised papers, selected from those presented at The 10th International Symposium on High-Level Parallel Programming and Applications (HLPP 2017), held in Valladolid, Spain, July 10-11th, 2017. The program committee of HLPP 2017 accepted 14 papers out of 20 full paper submissions, covering both foundational and practical issues in high-level parallel programming and applications. Authors of several selected papers were invited to submit extended versions to this special issue. Nine papers went through the peer review process of *Concurrency and Computation: Practice and Experience*. Finally, seven papers were selected to be published in this special issue.

2 | IN THIS ISSUE

The problem of parallelizing legacy code starts with the selection of what parts of the code can be run in parallel. In their contribution, Atre et al⁸ propose a solution to this problem based on the concept of Computational Unit, helping to identify both task- and loop-parallelization opportunities. The experimental results, including several benchmark suites, demonstrate the usefulness of their method.

A way to ease parallel computing is to leave to the runtime the calculation of different parameters needed to distribute a task among processors and to keep execution control. The present issue includes several contributions to this problem. Moreton-Fernandez and Gonzalez-Escribano⁹ present an interesting technique that calculates the specific communications derived from data-parallel codes with or without periodic boundary

conditions on affine access expressions. This solution moves to runtime part of the compiler-time analysis needed to generate these expressions, also reducing the development effort.

Rasch and Gorlatch¹⁰ propose a simple-to-use framework for automatic program optimization by choosing the most suitable values of program parameters, allowing to tune a broad class of applications written in different programming languages.

Ernstsson and Kessler¹¹ propose a solution based on skeletons to the problem of manage data locality on large clusters. This solution is based on the use of lazy evaluation to record invocations and dependences of sequences of transformations, using tiling to keep chunks of container data in the same working set, thus improving cache usage.

Torquati et al¹² describe the design of an automatic concurrency control algorithm, allowing to implement power-efficient communications on shared-memory multicore systems and to automatically switch between nonblocking and blocking concurrency protocols.

Other contributions focus on different aspects of parallel programming performance. Anand et al¹³ propose an algorithm that employs a partial rollback mechanism for conflict resolution in the context of Software Transactional Memory, demonstrating its usefulness with respect to pure abort mechanisms for applications with large transaction delays and making the case for using both approaches simultaneously.

Areias and Rocha¹⁴ propose a new implementation for the Tabling technique used in Prolog to handle recursion and redundant computation. Their work proposes a new table design that improves the representation of multi-dimensional arrays, introducing less overheads and improving the execution time of the benchmarks considered.

In summary, we believe that the interesting contributions included in this issue are good examples of how the high-performance-computing community works day after day to improve the usefulness of current and future parallel systems, easing the exploitation of their resources and reducing the complexity that is inherent to the development of concurrent code.

ACKNOWLEDGMENTS


We would like to thank all the authors, reviewers, and editors involved in the elaboration of this special issue, including also the reviewers who were involved in the HLPP'2017 symposium, where preliminary versions of the papers were previously selected. We are especially grateful to Prof. Geoffrey Fox, Editor-In-Chief of Concurrency and Computation: Practice and Experience, for approving this special issue and for his help along the process of its preparation.

ORCID

J. Daniel Garcia  <http://orcid.org/0000-0002-1873-9706>

Diego R. Llanos  <http://orcid.org/0000-0001-6240-9109>

J. Daniel Garcia¹ 

Diego R. Llanos² 

¹Computer Science and Engineering Department, Universidad Carlos III de Madrid, Madrid, Spain

²Computer Science Department, University of Valladolid, Valladolid, Spain

Correspondence

J. Daniel Garcia, Computer Science and Engineering Department, Universidad Carlos III de Madrid, 28911 Madrid, Spain.

Email: josedaniel.garcia@uc3m.es

Present Address

J. Daniel Garcia, Av. Universidad 30, 28911 Leganes, Madrid, Spain.

REFERENCES

- Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming. *IEEE Comput Sci Eng*. 1998;5(1):46-55.
- Gropp WD, Lusk E, Skjellum A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Vol 1. Cambridge, MA: MIT Press; 1999.
- Sanders J, Kandrot E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Boston, MA: Addison-Wesley Professional; 2010.
- TOP500 supercomputer sites. www.top500.org. Accessed 2018.
- Jeffers J, Reinders J. *Intel Xeon Phi Coprocessor High Performance Programming*. Boston: Newnes; 2013.
- McCool M, Robinson A, Reinders J. *Structured Parallel Programming: Patterns for Efficient Computation*. Waltham, MA: Morgan-Kaufmann; 2012.
- Gorlatch S, Cole M. Parallel skeletons. In: Padua D, ed. *Encyclopedia of Parallel Computing*. Boston, MA: Springer; 2011:1417-1422.
- Atre R, Ul-Huda Z, Wolf F, Jannesari A. Dissection sequential programs for parallelization—an approach based on computational units. *Concurrency Computat Pract Exper*. 2019;31:e4770. <https://doi.org/10.1002/cpe.4770>
- Moreton-Fernandez A, Gonzalez-Escribano A. Automatic runtime calculation of communications for data-parallel expressions with periodic conditions. *Concurrency Computat Pract Exper*. 2019;31:e4430. <https://doi.org/10.1002/cpe.4430>
- Rasch A, Gorlatch S. ATF: a generic, directive-based auto-tuning framework. *Concurrency Computat Pract Exper*. 2019;31:e4423. <https://doi.org/10.1002/cpe.4423>
- Ernstsson A, Kessler C. Extending smart containers for data locality-aware skeleton programming. *Concurrency Computat Pract Exper*. 2019;31:e5003. <https://doi.org/10.1002/cpe.5003>

12. Torquati M, De Sensi D, Mencagli G, Aldinucci M, Danelutto M. Power-aware pipelining with automatic concurrency control. *Concurrency Computat Pract Exper*. 2019;31:e4652. <https://doi.org/10.1002/cpe.4652>
13. Anand AS, Shyamasundar RK, Peri S. STMs in practice: partial rollback vs pure abort mechanisms. *Concurrency Computat Pract Exper*. 2019;31:e4465. <https://doi.org/10.1002/cpe.4465>
14. Areias M, Rocha R. Multi-dimensional lock-free arrays for multithreaded mode-directed tabling in Prolog. *Concurrency Computat Pract Exper*. 2019;31:e4491. <https://doi.org/10.1002/cpe.4491>