Teaching Parallel and Distributed Computing topics for the Undergraduate Computer Science Student

Marcelo Arroyo
Departamento de Computación, Universidad Nacional de Río Cuarto
Río Cuarto, Argentina

***Abstract*—Parallel and distributed systems programming skills has become a common requirement in the development of modern applications. It is imperative that an updated curriculum in computer science include these topics not only as advanced (often elective) programming courses. There is a general consensus that parallel programming topics should be spread in contents of several core courses and these topics should be included in any undergraduate computer science or software engineering curriculum. In this paper we describe how parallel and distributed computing and, specifically concurrent and parallel programming topics, are being included in the updated computer science curriculum of the degree in computer science at the Río Cuarto National University, Argentina. Also, we cover some suggested approaches for teaching parallel programming topics in a set of core courses to achieve a consistent, increasing and complete training in high performance computing.**

To achieve these goals, we propose a set of modules which includes basic and advanced high performance computing, parallel and distributed systems programming topics, to be included in core courses.

Also, we describe the use of existing tools and the development of new high level tools suitable for teaching parallel programming which can be used in different courses. The aim of using these tools and techniques is to reduce the gap between sequential and parallel programming teaching methods.

***Keywords*-Parallel programming, Education, Syllabus, Undergraduate Curriculum, Program skeletons.**

I. INTRODUCTION

The advances in hardware in the last years have allowed the development of very complex applications with high demands of computational resources. Also, people wants results in shortest times as possible.

Areas as CAD systems, scientific computing, simulation, computer animation, games and others, have grown thanks to the remarkable evolution of the computing hardware and the development of new programming techniques and algorithms which exploit efficiently all resources of modern computing systems.

Today, we have affordable computer systems with many CPU cores, complex multilevel memory systems, hyperthreading, SIMD co-processors and other features comparable to supercomputers of just a few years ago.

Recent developments in hardware are going in the direction of increasing the multiplicity of components executing in parallel more than the increasing of speed of some specific components.

A study group at the University of California at Berkeley, in 2006, showed that power management, memory access speed and instruction-level parallelism, constitute a barrier to continuous performance improvement of individual (sequential) computing *cores*[1]. This barrier was called *the power wall.*

These limits indicates that increasing of performance should be achieved by increasing parallel components (*cores*). They reports that it have many benefits as efficient power consumption and support for modern applications (multimedia, web browsers, etc).

In the same project, the team states: *Writing programs that scale with increasing numbers of cores should be as easy as writing programs for sequential computers*[2].

This statement raises a challenge for researchers and teachers in computer science and software engineering. This requires the development of new programming techniques, methods and tools.

Another dimension contributing to the complexity in the field of parallel and distributed computing (PDC) is the heterogeneity of current hardware platforms.

Currently, is common to find applications as modern games, scientific applications, simulation and others which are using GP-GPUs to take advantage of its high performance on SIMD operations. Software development tools for CPUs offer new programming models and techniques like NVIDIA CUDA^{TM}[11] or OpenCL[12].

Besides some of these topics has been included in some postgraduate or advanced courses, we think we are at a state of the art that these topics should be part of undergraduate education.

This requires that we need to make an immediate update of the undergraduate curricula in computer science.

The main contributions of this paper are the selected topics on high performance computing and how to include them as small modules in the selected core courses. The dependencies between these courses imposes an incremental approach. We show a set of suggested pedagogical methods and tools to teach parallel and distributed systems programming.

Also, we discuss the useful necessary background to learn parallelism concepts in a more natural way. In particular, we show how domain specific languages (DSLs) could be used for teaching PDC topics for both novice or advanced students. We have developed a C++ library (embedded DSL) with a set of parallel patterns (skeletons) suitable for easy development of parallel programs. This library is based on the meta-programming technique known as *expression templates*. This library enable us to write programs with implicit parallelism on different (programmer transparent) parallel platforms.

Currently, in Argentina, we are discussing the contents of minimal core topics for undergraduate computer science curriculum, which all Argentinian universities should fulfill. Hopefully this paper will help that too.

This paper is organized as follows.

The next section describes the high performance computing topics already included in some advanced courses, their dependencies and their relations with traditional topics.

In the following sections we propose how to include new small modules in basic core courses and the teaching levels. The relation between courses and topics was defined taking into account the *NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates* proposal.

Then, we describe the set of methods and tools which could be used to teach the modules described, including our pedagogical and practical points of view.

In particular we describe the background concepts previously taught required to introduce parallel programming in an almost natural and implicit way. We show how using high level programming constructions and libraries to be used for teaching parallel programming for both freshmen and advanced computer science students.

Finally there are our conclusions and future work.

II. PDC topics in the traditional curriculum

The aim of providing relevant skills to software developers to write high quality, efficient, portable and scalable applications, requires identifying the set of concepts needed to develop a proposal for changes in the contents of traditional courses of computer science.

Fortunately, we can to refer to existing proposals like the *NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates*[3]. We were selected as early adopters in spring 2011 and provide some feedback to the initial version of the proposal. We are in the process of development of educational resources to provide them to the *Distributed Computing Curriculum Development and Educational Resources* (CDER).

The NFS/IEEE-TCPP proposal provides us with a broad set of topics and how deep should be spread in different courses in a undergraduate curriculum in computer science and software engineering.

Fortunately, in our curriculum we already have a classification of courses in areas with a close match with the proposed areas in [3].

Below is a list of courses of our degree in computer science curriculum that we aim to include topics of parallelism and distributed computing.

- **Architecture and Computer Systems**
  - Computer organization (2nd year)
  - Simulation (4th year)
  - Operating Systems (5th year)
  - Telecommunications and Distributed Systems (5th year)
- **Programming**
  - Introduction to Algorithms and Programming (1st year)
  - Advanced Programming (2nd year)
  - Data Structures and Algorithms I (2nd year)
  - Data Structures and Algorithms II (3rd year)
- **Programming Languages**
  - Comparative Analysis of Programming Languages (3th year)
  - Compilers (4th year)
- **Software engineering**
  - Systems Analysis and Design (3th year)
  - Software Engineering (3th year)
  - Databases (3th year)
- **Theory of Computation**
  - Automata's and Languages Theory (4th year)
  - Computability and Complexity (5th year)
- **Elective (advanced) courses**
  - Concurrency and Parallelism (4th/5th year)
  - Software Validation and Verification (4th/5th year)
  - Databases II (4th/5th year)

Some topics on concurrency, parallelism and distributed computing already are covered in courses such as *Comparative Analysis of Programming Languages*, *Operating Systems* and *Telecommunications and Distributed Systems*.

We have some elective (more advanced) courses which also cover topics of parallel and distributed computing as *Concurrency and Parallelism* and *Software Verification*.

A. PDC topics already covered in core courses

Some of courses listed above cover specific topics of concurrency, parallel and distributed computing systems and programming. Below we describe how and were the current curriculum covers different topics of PDC and the learning level using Bloom's classification[4][5]:

- K=know the term (basic literacy)
- C=comprehend so as to paraphrase/illustrate
- A=Apply in some way (requires operational command)

Also we describe the methods, techniques and tools used in each core course.

- *Comparative Analysis of Programming Languages*
  - Concurrency, non determinism, race conditions and deadlock (A).
  - Shared Memory model (A).
  - Message passing model (A).
  - Synchronization: dataflow variables, monitors, locks, semaphores and others (A).
  - Programming languages support for concurrency (C).

  Concurrency topics is introduced. Students have to solve practical exercises on interleaving, races and deadlock detection and correction, programming simple algorithms (sorting, matrix operations, ...). Students implements some synchronization primitives as semaphores and use high level mechanisms as monitors. In the lab, students use the Oz programming language and the Mozart environment[6]. They also use other programming languages as C++ and Java.
- *Operating Systems*
  - Concurrency (processes, threads), non determinism, race conditions and deadlock revisited (A).
  - Hardware architectures: Memory hierarchies, SMP, context switching (A).
  - Synchronization: locks, condition variables and semaphores (A).
  - Interprocess communication mechanisms (A).

  Lab projects related with concurrency and parallelism: Improvements to the SMP scheduler. Low level implementation (in C) of condition variables, and semaphores in *xv6*[7].

  Xv6 is a simple educational operating system running on x86 architecture. Students develop programming projects including virtual memory improvements (COW, file permissions, ...) which require taking into account concurrency problems (race conditions and deadlocks) mainly in the shared memory programming model.
- *Telecommunications and Distributed Systems*
  - The message passing model and network protocols (A).
  - Multiprocessor and interconnected networks architectures. Clusters (C).
  - Frameworks: J2EE, .NET, CORBA, web services (C).
  - Distributed systems and algorithms: distributed memory, transactions, global state, election algorithms and others (A).

    In this course students use different programming languages (C, C++, Java) and some libraries (as MPI[15]) to implement algorithms and simple distributed systems.
- *Concurrency and Parallelism* (elective)
  - Introduction to parallel architectures (C)
  - Models for concurrency revisited (A).
  - Formal verification of concurrent programs using Owiki-Gries techniques (A).
  - Parallel programming with threads and OpenMP (A).
  - Parallel programming with MPI (A).
  - Introduction to GP-GPU architectures and programming model (C).
  - Performance analysis (A).

  In this course students use different programming languages (C++, Fortran) to implement several practical projects. The course focus in parallel algorithms and correctness of implementations rather than obtaining very good performance.

The four courses described above (one of them is elective) includes many topics of PDC but other important concepts are missing yet.

For example, in the computer architecture field, a lot of topics in core courses are not fully covered. The advanced topics of parallel programming algorithms and tools are covered in an elective course. This means that not all students will acquire parallel programming skills.

In the next section we describe the proposal how PDC topics are being included in another core courses. In this way, in the courses described above with some PDC topics (Operating Systems, Telecommunications and Distributed Systems, Concurrency and Parallelism), we could eliminate some hours of teaching the needed basic contents and put the focus in more specific and advanced topics.

III. Spreading PDC topics in others core courses

In this section we show the proposal of spreading PDC topics in several core courses. The table I contains a list of main PDC topics and its relationship with each course were these topics should be taught. For each topic the number of hours of teaching and the required learning level is shown. All courses in table I are in core (none is elective).

With this proposed curriculum, we are anticipating the teaching of many parallel and distributed computing topics. Our proposal takes into account the teaching order of topics from basic ones to the most advanced because students must take courses in the sequence imposed by curriculum.

Moving some background topics to courses in the early years of the curriculum, gives more room to teach advanced and specific topics in the courses at advanced years. For example, in the *Operating Systems* course would be taught some topics as advanced power management. Similarly, in the *Telecommunications and Distributed Systems* course would be possible to spend more hours for teaching distributed algorithms and web services and in the *Compilers course* could be possible to include a module on automatic parallelization.

Course: Computer Organization		
Topic	Learning level	Hours
Taxonomy (Flynn's classification)	C	0.5
Numerical representations	A	2
Performance metrics	K	1.5
SIMD instructions	C	1
Pipelines	C	1.5
Co-processors and GPUs	C	1.5
Multicore	K	0.5
Buses	K	0.5
NUMA and Cache	K	1.5
Power consumption	K	0.25
Course: Advanced Programming		
Topic	Learning level	Hours
Recursive decomposition	A	2
High order programming	A	4
Course: Data Structures & Algorithms II		
Topic	Learning level	Hours
Shared memory (threads)	A	1
Synchronization	A	2
Data races and deadlock	A	2
Data parallel algorithms	A	1
Parallel loops and recursion	A	0.5
Parallel divide/conquer	A	0.5
Par. dynamic programming	A	0.5
Parallel search	A	1
Course: Comparative Analysis of Languages		
Topic	Learning level	Hours
Atomicity	A	1
Shared memory concurrency	A	1.5
Message passing model	A	1.5
Parallel language constructs	A	1
Implicit parallelism	A	1
Course: Systems Analysis and Design		
Topic	Learning level	Hours
Parallel design patterns	A	2
Course: Software Engineering		
Topic	Learning level	Hours
Frameworks for parallel programing	C	1.5
Course: Simulation		
Topic	Learning level	Hours
Parallel numerical methods	A	2
Course: Compilers		
Topic	Learning level	Hours
Parallel optimizations	C	1
Directives based parallelism	C	1
Course: Software verification		
Topic	Learning level	Hours
Reasoning on concurrency	A	2.5
Modeling concurrent programs	A	2
Race conditions and deadlock detection	C	2
Course: Computability and Complexity		
Topic	Learning level	Hours
Parallel computer models (PRAM,...)	A	2
Complexity measures for parallel models	A	2

Table I
PDC TOPICS SPREAD IN CORE COURSES.

IV. TECHNIQUES AND TOOLS

From our own experience, after teaching several courses and topics of concurrency and parallelism, we found that some PDC topics are descriptive and do not have too much learning difficulty for students. Some examples could be parallel architectures and some communication topologies, the description and applicability of performance measures and learning some basic tools, like MPI, OpenMP or threads libraries.

The main challenge is to give to students real skills for parallel and distributed systems programming and provide solid foundations on non-determinism, race conditions, deadlock avoidance, and parallel algorithms design. You also need to provide them with skills on getting high performance and scalability.

After teaching concurrency and parallelism to students with different background we can see the importance of solid foundations mainly in programming languages concepts and paradigms, data structures and algorithmic problem solving.

For complex problems, abstractions are a fundamental conceptual tool to get good solutions.

We saw that students have a better understanding and they can solve complex problems if they have solid foundations on:

- Programming language paradigms (functional, relational and statefull programming models)
- Programming language semantics (informally, at least)
- Run-time environments
- Memory management
- Abstraction: abstract data types and functional abstraction
- High order programming
- Generics

- Algorithmic problem solving
- Program transformation
- Solid background in formal methods:
  - Program verification and derivation
  - Model checking

For the reasons mentioned above, we think that teaching parallel programming should be a natural extension of sequential programming. It is not new, many authors proposed ideas and tools to get the shift to *thinking in parallel*[24].

Currently, almost all specific courses on high performance computing and parallel and distributed systems programming offered in Argentina, focus in teaching some traditional parallel algorithms in the shared memory and message passing models using threads, OpenMP and MPI.

A few courses are specific to GP-GPU programming (almost all using NVIDIA-CUDA^{TM}). Currently, almost there is no offer of heterogeneous parallel computing courses.

These approaches are not the most suitable for students and researches of other disciplines.

In 2012, we taught two courses on PDC (the first at our university and the other at Rosario National University). In both courses we had researchers and PhD students coming from other disciplines without much programming experience.

These researchers and professionals needs to develop solutions on their own fields, therefore it is necessary to provide them with high-level tools or language abstractions to focus on the applicability of parallel and distributed design patterns to a given problem.

These patterns should abstract the underlying hardware architecture details and focus on algorithmic.

The same approach could be provided to the freshmen in computer science, to get a natural shift to *thinking in parallel* from the very beginning.

For all the above, teaching parallelism for the freshmen based on parallel patterns and frameworks seems to be the most promising approach. This idea is not new. Many researchers have proposed this approach in the past[8] and many tools of this kind have been developed[9].

This libraries define abstractions (parallel skeletons) which are similar to higher-order functions found in modern functional programming languages and should be familiar to students with some experience with Haskell or ML.

In our curriculum, in the first programming course, students learn imperative programming and basics on abstract data types implementation and basic functional abstraction.

The second programming course (*advanced programming*) covers modern functional programming and formal program verification and program derivations using a *calculus of programs*[10].

Students who passed these courses should have no trouble understanding parallel skeletons because they are familiar with higher-order functional programming and generic polymorphism.

A. Using parallel skeletons

The suggested approach for teaching parallelism for the freshmen is based on the construction of *Domain Specific Languages (DSLs)*. Parallel abstractions can be seen as a DSL. In our experience teaching parallelism and distributed systems programming has shown that it is better to think a parallel algorithm as a pattern which includes sequential components (or other parallel patterns).

Thus, an application can be developed in a compositional way.

Many libraries of this kind has been developed[9]. Some of these are for specific problem domains o specific data structures, as *Parallel Expression Templates for Large Vectors* (`PRiSM`)[16] and *CUDA Expression Templates*[14], suitable for linear algebra. *Parallel Object-Oriented Methods and Applications* (`POOMA`)[18] is a library (also based on C++ expression templates[22]) for writing parallel PDE solvers using finite-difference and particle methods.

`VecCL`[19] is also a expression templates library which targets OpenCL. It define vector templates with a broad set of operators.

These libraries often implement parallel communicational patterns and targets some specific platform. For example, `PRiSM` use threads (Intel's Threading Building Blocks[17]) and OpenMP, while *CUDA Expression Templates* libraries targets to NVIDIA GPUs.

To abstract from specific platform targets and to hide some implementation details we develop a small C++ library, called *Parallel and Distributed Templates* (`pdt`) also based on *expression templates*. This library focus on teaching parallel programming by means of parallel patterns or *skeletons*. The goal is to develop a full multi-target implementation (threads, OpenMP, MPI and OpenCL) with a common interface and (semi) automatic device target selection.

The `pdt` library has two set of templates. The first templates group, suitable for data parallelism, is based on the definition of *parallel skeletons* for homomorphic computation structures proposed in [20], which are suitable for implicit parallelism and resemble some patterns of the

Data parallel skeletons
$map_A(f,[x_1,\ldots,x_n]) = [f(x_1),\ldots,f(x_n)]$
$reduce_A(\oplus,[x_1,\ldots,x_n]) = x_1 \oplus \ldots \oplus x_n$
$scan_A(\oplus,[x_1,\ldots,x_n]) = [x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \ldots \oplus x_n]$
$\ldots$
$map_T(f, x_1(x_2,x_3)) = f(x_1)(f(x_2),f(x_3))$
$reduce_T(f,\oplus,\otimes,,x_1(x_2,x_3)) = f(x_1) \oplus (f(x_2) \otimes f(x_3))$ $\ldots$
$skel_A$ means a skeleton on arrays $skel_T$ means a skeleton on trees $\oplus$ and $\otimes$ are associative binary operators
Architectural skeletons
$pipeline(is, se_1, se_2, \ldots, se_n)$
$workers(is, se, N)$ ...   Where is is an input stream se is a *stream expression* (using streams input values)

Table II
SOME PATTERNS OF PDT LIBRARY.

SkeTo library[21] to implement most of data parallel patterns.

Data parallel patterns works on distributed data structures as vectors, matrices and trees.

The second group of templates define some architectural patterns as pipeline, workers and others. They are based on distributed streams.

The library is extensible, so programmers can define their own patterns. Higher level patterns can be defined by composition of simpler patterns. Users can define additional distributed data structures taking into account the *homomorphism constraints*.

Table II describe some patterns defined in the pdt library.

Data parallel patters are based on distributed iterators which are abstractions of accessors to distributed data elements. Distributed iterators contain references to memory devices. In this way the expressions are executed on corresponding devices.

For example, below is an excerpt of program which apply a translation on an image:

```
...
pdt::array< pdt::pair<byte> > img(N,M);
pdt::scatter(img,n,m,cluster_nodes);
pdt::copy_to(gpu_device,img);
map(img,(_x * 2 , _y * 3));
...
```

The above code show how to define a distributed array, distribute parts to other cluster nodes and then each node compute an expression on each element on its GPU.

The expression $(\_x + 2, \_y + 3)$ means that on each element (pair (x, y)) of array img will be added 2 to the first component and 3 to the second.

Currently, the library targets OpenMP, MPI and (partially) OpenCL.

Some patterns as architectural patterns use expressions on input streams values to describe values passed to tasks from a coordinator component.

For instance, the pipeline template coordinates the synchronization between the pipeline stages. The stage s_i behavior is given by the expression se_i.

Architectural skeletons coordinates communication between different tasks (stages) using *distributed streams*[1].

Synchronization in data parallel skeletons and architectural skeletons is hidden to user.

The library overload some operators to allow write patterns with a more natural syntax. For instance, a pipeline pipeline(s,e_1,...,e_k) can be written as s » e_1 » ...» e_k.

We plan to use this kind of libraries to teach parallelism for freshmen and researchers of other disciplines. In courses for advanced students in computer science we plan they will get involved in the analysis, development and improvements of library components on some targets. Students will be encouraged to define higher level patterns and some domain specific skeletons an languages.

Generic programs (or skeletons) provide the basis for structured parallelism and they allow to get *performance predictability* and *portability*.

Each skeleton/target pair is associated with a performance model which can be used to predict the performance of a program written using the skeleton on the target.

Another advantage of using skeletons is *correctness*. Parallel skeletons can be formally or semi-formally verified (as in[23]).

V. CONCLUSION AND FUTURE WORK

We have described how we are teaching parallelism and distributed computing topics are in the current undergraduate curriculum at Río Cuarto University, Argentina.

Traditionally these topics were covered in a few courses which can be taken only by advanced students.

In this paper we showed a proposal to integrate PDC topics in basic core courses of current curriculum. The integration of topics throughout the curriculum will allow early training for computer science students. Thus, we hope

[1]In shared memory model, they are common streams with synchronization primitives (condition variables). In message passing model streams represent synchronous communication channels

to get the aim *parallel thinking*.

We also propose to teachers the teaching levels and the approximate number of hours they should spend for each module in each course.

We have been careful in the selection of topics in each course to achieve a progressive and continuous training throughout the curriculum.

The selected topics have been contrasted with the *NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing*.

We plan to continue working on `pdt` library targeting patterns to different platforms and implementing automatic device targeting based on dynamic device detection.

We are developing teaching materials (slides and tutorials) for each module for introducing parallel topics in core courses. We hope to contribute to the CEDR with this resources.

Using parallel skeletons in courses with students from other disciplines has been successful so it is interesting to keep working on that line. We are planning the development of extensions to templates to generate graphical views of parallel programs. Almost all students think would help a lot when designing parallel programs.

Also, we plan to do an pedagogical study on the application of the proposed teaching approach to the freshman and other disciplines students and then collect and publish the results.

We will monitor the teaching/learning process and we'll collect qualitative and quantitative information in the coming years. To do that we plan to design a survey to apply to teachers and students to collect data on some metrics and to collect opinions and suggestions.

References

[1] Asanovic, K. et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. UCB/EECS-2006-183, University of California, Berkeley, Dec. 18, 2006.

[2] Kriste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick *A View of the Parallel Computing Landscape*, Communications of the ACM, October 2009, Vol. 52, No 10.

[3] *NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates*, Version I1, Dec 2012. Website: http://www.cs.gsu.edu/ tcpp/curriculum/index.php Curriculum Working Group

[4] Anderson, L.W. and Krathwohl (Eds.). *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. New York: Longman. 2001.

[5] Huitt, W., Bloom et al.'s *Taxonomy of the Cognitive Domain*. Educational Psychology Interactive. Valdosta, GA: Valdosta State University. 2009. http://www.edpsycinteractive.org/topics/cogsys/bloom.html.

[6] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press. ISBN 0-262-22069-5, March 2004.

[7] Russ Cox, Frans Kaashoek and Robert Morris. *Xv6, a simple Unix-like teaching operating system*. http://pdos.csail.mit.edu/6.828/2011/xv6.html

[8] Murray Cole. *Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming*. Parallel Computing 30, 389-406. Elsevier. 2004.

[9] Horacio González-Vélez and Mario Leyton. *A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers*. SOFTWARE PRACTICE AND EXPERIENCE, 1-26. John Wiley & Sons. 2010.

[10] Javier Blanco. Silvina Smith, Damián Barsotti. *Cálculo de Programas*. FAMAF - UNC. ISBN: 978-950-33-0642-0. 2008.

[11] NVIDIA. *CUDA. NVIDIA parallel platform and programming model*.

[12] Khronos OpenCL Working Group (Editor: Aaftab Munshi). *The OpenCL Specification*. Version 1.2. 2012. http://www.khronos.org/registry/cl/

[13] Rob Farber. *CUDA Application and Development* ISBN-10: 0123884268 ISBN-13: 978-0123884268. 2011.

[14] Jie Chen. *Automatic Offloading C++ Expression Templates to CUDA Enabled GPUs*. Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International.

[15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard. Version 3.0*. September 21, 2012. http://www.mpi-forum.org/docs/docs.html

[16] Laurent Plagne, Frank Hülsemann. *Parallel Expression Template for Large Vectors*. 2009.

[17] Reinders, J. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc. 2007.

[18] *Parallel Object-Oriented Methods and Applications Library*. http://acts.nersc.gov/formertools/pooma/index.html

[19] Denis Demidov. *VexCL: Vector expression template library for OpenCL* http://www.codeproject.com/Articles/415058/VexCL-Vector-expression-template-library-for-OpenC

[20] K. Matsuzaki, K. Kakehi, H. Iwasaki, Z. Hu, and Y. Akashi. *A fusion-embedded skeleton library*. In 10th International Euro-Par Conference (EuroPar 2004), volume 3149 of Lecture Notes in Computer Science, pages 644–653. Springer, August 2004.

[21] Kiminori Matsuzaki, Kento Emoto. *Implementing Fusion-Equipped Parallel Skeletons by Expression Templates*. Implementation and Application of Functional Languages: 21st International Symposium, IFL 2009, Revised Selected Papers, Lecture Notes in Computer Science 6041, Springer, pp.72-89, 2010.

[22] T. Veldhuizen. *Expression Templates*. In C++ Gems, pages 475–487. SIGS Publications, Inc., New York, NY, USA, 1996.

[23] Wadoud Bousdira, Frédéric Loulergue and Julien Tesson. *A verified library of algorithmic skeletons on evenly distributed arrays*. ICA3PP'12 Proceedings of the 12th international conference on Algorithms and Architectures for Parallel Processing - Volume Part I Pages 218-232. Springer-Verlag Berlin, Heidelberg. ISBN: 978-3-642-33077-3. 2012.

[24] Guy Blelloch. *Parallel Thinking*. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2009.

[25] Noman Javed and Frédéric Loulergue*Parallel Programming with Orleans Skeleton Library*. Technical report RR-2011-05. Universite D'Orleans. 2011.

[26] *The C++ Boost Libraries*. http://www.boost.org/