Teaching Parallel and Distributed Computing to Undergraduate Computer Science Students

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 any updated curriculum in computer science must include these topics not only as advanced (often elective) programming courses. There is a general consensus that parallel programming topics should be spread throughout the undergraduate 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 and some parallel and distributed systems programming topics, to be included in core courses.

Finally, we describe the use of existing tools and the development of new high level tools, as parallel patterns, useful 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.

Index Terms—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 require results in shortest times as possible.

Some 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 rather than increasing speed of individual devices.

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[2]. 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 research project, the team states: Writing programs that scale with increasing numbers of cores should be as easy as writing programs for sequential computers[3].

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 components in current hardware.

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 GPUs offer new programming models and techniques like NVIDIA CUDA™[12] or OpenCL[13].

Some of these topics has been included in some postgraduate or advanced courses, we think that the state of the art has evolved enough so these topics should be part of undergraduate education.

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

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 and 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 parallel platforms hiding the implementation details.

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 points of view.

In particular, we describe the basic topics previously taught. These topics are required to introduce parallel programming topics in an almost natural and implicit way. We show how to use high level programming constructions and libraries for teaching parallel programming for both: freshmen and more 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 the identification of a minimal 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[4]. 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 contribute to the Distributed Computing Curriculum Development and Educational Resources (CDER).

The NFS/IEEE-TCPP proposal provides us with a broad set of topics, teaching levels and how this topics 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 by topic areas and we have a close match with the proposed areas in [4].

Below is a list of courses of our degree in computer science curriculum. This curriculum is the base 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 (3rd 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 systems already are covered in core courses such as Comparative Analysis of Programming Languages, Operating Systems and Telecommunications and Distributed Systems.

We have some elective (advanced) courses which also cover topics of parallel and distributed computing as in 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 where the current curriculum covers different topics of PDC and the learning level (using Bloom’s classification[5][6]):

- **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 and message-passing models (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[7]. Also, they 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[8].

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[16]) 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 Owick-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 a set of programming projects. This course focus in parallel algorithms and correctness of implementations rather than getting high performance.

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

For example, in the computer architecture area, several topics are not fully covered in core courses. Advanced topics of parallel programming algorithms and tools are covered in an elective course. This means that not all students will acquire the same parallel programming skills.

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

### III. Spreading PDC topics in core courses

In this section we show the proposal of spreading PDC topics in some core courses. The table I contains a list of main PDC topics and its relationship with courses where these topics should be taught. We show the number of hours of teaching and the required learning level for each topic. All courses in table I are in core (neither 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 basic PDC 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 reserve more hours for teaching distributed algorithms, grid computing, and other advanced topics. In the Compilers course could be possible to include a module on optimization of code generation to support automatic parallelization or better usage of hyper-threading facilities like instruction’s reorder.
TABLE I
PDC TOPICS SPREAD IN CORE COURSES.

<table>
<thead>
<tr>
<th>Course: Computer Organization</th>
<th>Learning level</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taxonomy (Flynn’s classification)</td>
<td>C</td>
<td>0.5</td>
</tr>
<tr>
<td>Numerical representations</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>Performance metrics</td>
<td>K</td>
<td>1.5</td>
</tr>
<tr>
<td>SIMD instructions</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>Pipelines</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Co-processors and GPUs</td>
<td>C</td>
<td>1.5</td>
</tr>
<tr>
<td>Multicore</td>
<td>K</td>
<td>0.5</td>
</tr>
<tr>
<td>Buses</td>
<td>K</td>
<td>0.5</td>
</tr>
<tr>
<td>NUMA and Cache</td>
<td>K</td>
<td>1.5</td>
</tr>
<tr>
<td>Power consumption</td>
<td>K</td>
<td>0.25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course: Advanced Programming</th>
<th>Learning level</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursive decomposition</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>High order programming</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>Implicit parallelism</td>
<td>C</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course: Data Structures &amp; Algorithms II</th>
<th>Learning level</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared memory (threads)</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>Synchronization</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>Data races and deadlock</td>
<td>A</td>
<td>1.5</td>
</tr>
<tr>
<td>Parallel loops and recursion</td>
<td>A</td>
<td>0.5</td>
</tr>
<tr>
<td>Parallel divide/conquer</td>
<td>A</td>
<td>0.5</td>
</tr>
<tr>
<td>Par. dynamic programming</td>
<td>A</td>
<td>0.5</td>
</tr>
<tr>
<td>Parallel search</td>
<td>A</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course: Comparative Analysis of Languages</th>
<th>Learning level</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomicity</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>Shared memory concurrency</td>
<td>A</td>
<td>1.5</td>
</tr>
<tr>
<td>Message passing model</td>
<td>A</td>
<td>1.5</td>
</tr>
<tr>
<td>Parallel language constructs</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>Implicit parallelism</td>
<td>A</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course: Systems Analysis and Design</th>
<th>Learning level</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel design patterns</td>
<td>A</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course: Simulation</th>
<th>Learning level</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel algorithms for numerical methods</td>
<td>A</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course: Compilers</th>
<th>Learning level</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel optimizations</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>Directives based parallelism</td>
<td>C</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course: Software verification</th>
<th>Learning level</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reasoning on concurrency</td>
<td>A</td>
<td>2.5</td>
</tr>
<tr>
<td>Modeling concurrent programs</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>Race conditions and deadlock detection</td>
<td>C</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course: Computability and Complexity</th>
<th>Learning level</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel computer models (FRAM,…)</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>Complexity measures for parallel models</td>
<td>A</td>
<td>2</td>
</tr>
</tbody>
</table>

From our own experience, after teaching several courses and topics of concurrency and parallelism, we found that some PDC topics are more descriptive than others and they are not so complex for students. Some examples of these topics are parallel architectures and some communication topologies, the description and applicability of performance measures and learning some basic tools, like MPI, OpenMP or threads libraries.

However, other topics need to be studied in more depth. We think the main challenge is to give to students real skills for parallel and distributed systems programming and provide them solid foundations on non-determinism, race conditions, deadlock avoidance, and parallel algorithms design. We 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 correct 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. This idea is not new, many authors have proposed ideas and tools to get the shift to thinking in parallel[25].

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

We have few courses on GP-GPU programming (almost all using NVIDIA-CUDA™). Currently, almost there is no offer of courses on heterogeneous parallel programming.

These approaches are not the most suitable for students and researches coming from 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 with little programming experience.

These people need to develop computer programs to get solutions on their own fields. Therefore it is necessary to provide them with high-level tools and language abstractions to focus on the applicability of parallel and distributed design patterns to a set of given problems.

These patterns should abstract the underlying hardware architecture details and focus on PDC algorithms. We think 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, to teach parallelism for the freshmen based on parallel patterns and frameworks seems to be the most promising approach[1]. This idea is not new. Many researchers have proposed this approach in the past[9]. Many tools of this kind have been developed[10].

This libraries define abstractions (parallel skeletons) with an interface 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 focus on abstract data types implementation and basic functional abstraction.

The second programming course (advanced programming) covers modern functional programming, formal program verification and program derivations using a calculus of programs[11].

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 have been developed[10]. Some of these are for specific problem domains or specific kinds of data structures, as *Parallel Expression Templates for Large Vectors (PRiSM)*[17] and *CUDA Expression Templates*[15], suitable for linear algebra. *Parallel Object-Oriented Methods and Applications (POOMA)*[19] is a library (also based on C++ expression templates[23]) for writing parallel PDE solvers using finite-difference and particle methods.

*VecCL*[20] is also a expression templates library which targets OpenCL. This library declare 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[18]) 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 group of templates, suitable for data parallelism, is based on the definition of *parallel skeletons* for homomorphic computation structures proposed in [21]. Our library is suitable for teaching implicit parallelism and resemble some patterns of the SkeTo library[22] 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 (producers and consumers).

The library is extensible, so students can define their own patterns. Higher level patterns can be defined by composition of simple 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 patterns are based on distributed iterators which are abstractions of accessors to distributed data elements. Distributed iterators contain references to memory devices. In this way, index expressions access data at the corresponding device.

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

\[
\ldots
\text{pdt::array< pdt::pair<byte> > img(N,M);}
\text{pdt::scatter(img, n, m, cluster_nodes);}
\text{pdt::copy_to(gpu_device, img);}
\text{map(img, (_x * 2, _y * 3));}
\ldots
\]

The above code show how to define a distributed array, distribute parts to other cluster nodes. Then each node compute...
patterns with a more natural syntax. For instance, a pipeline
skeletons is hidden to user.

different tasks (stages) using
distributed streams

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 syn-
chronization 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\((is, os, se_1, se_2, \ldots, se_n)\) can be written as \(is \rightarrow se_1 \rightarrow \ldots \rightarrow se_k \rightarrow os\).

We think using this kind of libraries to teach parallelism for
freshmen and people coming from other disciplines is the best
approach.

In advanced computer science courses we plan that
students 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.

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

**TABLE II**

<table>
<thead>
<tr>
<th>Data parallel skeletons</th>
</tr>
</thead>
<tbody>
<tr>
<td>(map_A(f, [x_1, \ldots, x_n]) = [f(x_1), \ldots, f(x_n)])</td>
</tr>
<tr>
<td>(reduce_A(\oplus, [x_1, \ldots, x_n]) = x_1 \oplus \ldots \oplus x_n)</td>
</tr>
<tr>
<td>(scan_A(\oplus, [x_1, \ldots, x_n]) = [x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \ldots \oplus x_n])</td>
</tr>
<tr>
<td>(\ldots)</td>
</tr>
<tr>
<td>(map_P(f, [x_1, x_2, x_3]) = [f(x_1), f(x_2), f(x_3)])</td>
</tr>
<tr>
<td>(reduce_P(f, \oplus, [x_1, x_2, x_3]) = f(x_1) \oplus f(x_2) \oplus f(x_3))</td>
</tr>
<tr>
<td>(\ldots)</td>
</tr>
<tr>
<td>(skel_A) means a skeleton on arrays</td>
</tr>
<tr>
<td>(skel_P) means a skeleton on trees</td>
</tr>
<tr>
<td>(\oplus) and (\otimes) are associative binary operators</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Architectural skeletons</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipeline((is, os, se_1, se_2, \ldots, se_n)) workers((is, os, se, N)) \ldots</td>
</tr>
<tr>
<td>Where</td>
</tr>
<tr>
<td>(is) is an input stream (producer)</td>
</tr>
<tr>
<td>(os) is an output stream (consumer)</td>
</tr>
<tr>
<td>(se) is a stream expression (using streams input values)</td>
</tr>
</tbody>
</table>

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 shown in[24]).

We could see that those students having some skills on
modern functional programming (with some experience on
Haskell or ML) get the parallel thinking naturally.

These students are more trained on recursive decomposition
and high order programming. Parallel patterns become familiar
to them. They quickly relate map and reduce parallel
patterns with its corresponding sequential and widely used,
high order functions. They immediately see that parallel scan
patterns correspond with sequential folds, and so on.

**V. CONCLUSION AND FUTURE WORK**

We have described how we are teaching some parallelism
and distributed computing topics 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 shown a proposal to integrate PDC topics
in basic core courses of current curriculum. The integration
of topics throughout the curriculum will allow easy PDC
training for computer science students. Thus, we hope to get
the aim of 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 these
resources.

Using parallel skeletons in courses with students from
other disciplines has been successful. We think that this
approach is very promising. We are planning the development
of extensions to templates to generate graphical views of
parallel programs. Almost all students think that 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’ll monitor the teaching/learning process and we’ll collect qualitative and quantitative information in the next years. To do that, we plan to design a survey to apply to teachers and students and collect and analyze data based on some metrics to produce some results.

REFERENCES

[12] NVIDIA. CUDA. NVIDIA parallel platform and programming model.