Teaching PDC Programming for 4th year Computer Science Students and Professionals Coming from Other Disciplines

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

Abstract—**Teaching high performance computing for the undergraduate computer science student and other disciplines professionals, teachers and researchers is a big challenge. Experiences on teaching parallel and distributed programming led us to find some approaches to reduce the gap with sequential programming.**

In this paper we expose some experiences teaching parallel programming and we show some learned lessons. In particular we show some results using parallel patterns in a top-down approach for teaching parallel programming.

I. Introduction

Since 2009, we are offering the elective course Parallel Programming for 4th or 5th year computer science undergraduate students. This course is often taken by advanced and PhD students, professionals, teachers and researchers coming from other disciplines (math, engineering, physics, biology and others). Many of these students do not have a solid background in computer programming. Obviously they have needs in their own disciplines to solve problems requiring high performance computing. It's common in this group of people who have basic programming skills and sometimes they knows a single programming language (like FORTRAN, for example).

In 2012, we teach a similar course at Rosario National University, Argentina. Here also we had the same kind of students as described above. Obviously, there is a high demand for courses on high performance computing from people coming from other disciplines.

The undergraduate computer science students had similar background in both universities. We found the same problems to teach parallel programming topics due to this heterogeneous audience.

The computer science students have solid skills on programming, algorithmic problem solving, programming languages and paradigms (functional, logic and object-oriented programming), data structures and algorithms. They have notions at *can apply* level on concurrency topics: non-determinism, race conditions, synchronization, deadlock, and shared-memory and message-passing programming models. These topics were covered in the 3rd year (core) course Comparative Analysis of Languages which focus on programming paradigms and computation models. At the end of this course students develop programs using threads and message-passing primitives using the Oz programming language and the Mozart environment.

By using the Oz programming language enable us to introduce data-flow variables as the first synchronization primitive.

Computer science students also have solid background on algorithm analysis (time and space complexity measures). These concepts are covered on three core courses:

- Advanced Programming: 2nd year course.
- Data Structures and Algorithms: 2nd year course.
- Algorithms Design: 3rd year course.

The main problem is how to introduce some of this concepts to students coming from other disciplines. For those students, we try to achieve knowledge level of algorithms complexity and concurrency topics by means of an introductory module, mainly by showing non-determinism, races and deadlock problems and the needs for synchronization.

The other problem is how to teach programming tools to people with no solid foundations on programming techniques and programming languages.

In the next sections we show an approach to solve these issues.

II. Reducing the gap with sequential programming

The Parallel Programming course begins with introductory topics on parallel architectures, parallel programming models and the basics of performance analysis. For the computer science students, some of these topics are just a review of known concepts. Most of these topics are new for non computer science students.

To reducing the gap with sequential programming we introduce high level parallel and distributed programming constructs by means of high-level patterns.

Each pattern is presented first in graphical form and we show the corresponding high level contracts. These contracts help to students to see how to apply the patterns.

The pattern are implemented in C++ as a library developed by the teaching team. The library is suitable to implement simple architectural and communication patterns. The library does not focus on performance but in its applicability.

By using patterns, students can quickly develop and observe the behavior of programs running on clusters, multicore systems and GP-GPU equipped systems. They build parallel solutions by composition of simple components.

At this point students focus on algorithmic problem solving and the design and analysis of parallel/distributed algorithms rather on implementation details.

The details of implementation of these patterns are introduced later, when the low-level libraries (threads, OpenMP, MPI, OpenCl, ...) and other tools are described.

III. Differentiation on teaching between two groups of students

After the introduction and application of patterns, computer science students are encouraged to develop new parallel abstractions and to modify, extend or improve the implementation of some patterns.

We think the approach used is suitable to develop a course with a such heterogeneous audience. The main topics are introduced for everyone equally but differs in the practical part.

For example: Decomposition techniques are introduced. As we said, we use patterns for domain and task partitioning.

As a first step all students are encouraged to apply and compose some patterns to solve some classical problem and to do performance analysis. Patterns helps a lot with dealing with performance analysis. In this way all students were users of provided tools. This is very useful to training students coming from another disciplines.

In the second step, computer science students are encouraged to develop new abstractions and to implement new patterns. Also they have to improve, extend or re-implement (to other target) the patterns used.

Finally, all students are encouraged to get a better performance with respect to their initial solutions.

IV. Tools used and exercises

As we said above, we use some simple high-level libraries for parallel programming (based on well-known parallel patterns) developed by the teaching team. Some of this tools are implementations of parallel skeletons, implemented as C++ templates (inspired on existing libraries)[1].

These parallel patterns are implemented on shared-memory (using pthreads) and message-passing (MPI) models. Some patterns (e.g. `map`) targets to SIMD devices (OpenCL).

Other tools used were SkeTo, vecCL and others.

We propose exercises as matrix multiplication, sorting and searching. Other exercises are more discipline specifics as Fast Fourier Transform (for engineers) or string processing (useful for biology students).

Below, we show some patterns used:

`task-pool<expr,p,c>`	master-slave
`pipeline<p,c,`$e_1,\ldots,e_k$`>`	pipeline
`map<data,expr>`	apply `expr` to `data`
`reduce<data,op>`	reduce `data` using `op`
...	...

Where `data` is some distributed data structure and `p/c`: are producer/consumer tasks.

V. Top-down teaching approach

The teaching approach is top-down. At the start, students become users of high-level abstractions. They have to apply some patterns to solve the given problems. As the course progress we study the patterns design and implementation details.

Using high-level patterns and skeletons at first stages of the course, allows students to see results on the very beginning. This encourage them to deal with complex problems and focus on problem solving and not in low-level programming details.

Then, computer science students will deal with the details (threads, OpenMP, MPI, CUDA/OpenCL, ...). They have to develop new patterns and to improve (or re-implement) some existing ones.

VI. Results

We applied this approach in the last two courses taught. Results were promising. Fourth-year computer science and other disciplines students were able to develop complex parallel applications.

All students were able to apply the techniques taught as decomposition and performance analysis and they understood concepts on parallel architectures and topologies.

Students were able to compare quickly different implementations with same patterns (on different targets).

We have identified some core topics which are fundamental to learning (and to make teaching easier) parallel programming. Some of these concepts are:

- Data structures and algorithm design: for understanding distributed data structures, communication patterns and performance measures.
- Functional and high-order programming: for understanding parallel skeletons and to detect implicit parallelism.
- Object-oriented programming: for understanding parallel patterns, contracts and high- level abstractions.
- Programming language concepts and paradigms: maybe one of the main problems with students coming from other disciplines.
- Meta-programming: for patterns development.

[1]The *pdt* library contains a set of C++ templates (using meta-programming techniques) which implements some parallel skeletons developed by author with focus on teaching parallel programming.