Integrating Parallel Computing in the Curriculum of the University Politehnica of Bucharest

Mihai Carabaş, Adriana Drăghici, Grigore Lupescu, Cosmin-Gabriel Samoilă, and Emil-Ioan Sluşanschi(✉)

University Politehnica of Bucharest, 313 Splaiul Independenţei, Bucharest, Romania
{mihai.carabas,adriana.draghici,cosmin.samoila,emil.slusanschi}@cs.pub.ro, grigore.lupescu@gmail.com

Abstract. The continuous shift of hardware computing architectures, from single to many-core processors, as well as the blurring of the hardware - software interface, has made the introduction of parallel and distributed computing topics in the undergraduate curriculum an essential requirement for any quality computer science program. The University Politehnica of Bucharest offers a unique approach, employing a heterogeneous hardware and software teaching and computing infrastructure, to its over 450 students enrolled in undergraduate studies of Computer Science and Electrical Engineering. In this study we present two of the most important lectures covering PDC topics at the UPB.

Keywords: Parallel programming · Python
Performance optimization · GPU computing · PDC
Undergraduate education

1 Introduction

Given the current evolution of the IT industry, Parallel and Distributed Computing is seen as an essential topic for any IT professional. University "Politehnica" of Bucharest is one of the oldest and most prestigious engineering school in Romania. Over the last 20 years, the Computer Science and Engineering Department has conferred a special importance to the PDC curricula. The importance of parallel and distributed systems as well as a distinction between parallel versus distributed systems is discussed in [20,23], and the approach offered by the UPB is consistent with the view presented therein, since our curriculum already contains different courses for distributed and parallel systems. Most courses containing PDC issues are taught in the first three years of CS and touch a wide audience of around 400–500 students per year. Similar lectures are being offered around the world by various CS groups in Tennessee [19], Cadiz [24], or Cluj-Napoca [22].

This paper is structured as follows. In Sect. 2 we outline the PDC curriculum in the UPB undergraduate CS and EE programs. In turn, Sect. 3 presents the practical activities in two of the lectures concerned with PDC. Section 4 presents the student progress evaluation process, whereas Sect. 5 outlines the interest and involvement of the IT industry in Romania and abroad towards the PDC issues being taught to our undergraduate students. In Sect. 6 we enumerate the lessons learned through the years while offering the PDC curriculum, and we conclude in 7 with some conclusions and an outline of possible improvements to our present approach.

2 Parallel and Distributed Computing Curriculum

In the bachelor program of the UPB, we can find three main lectures where Parallel and Distributed Computing issues are presented to the students, namely the Parallel and Distributed Algorithms (PDA), Computer Systems Architecture (CSA), and Parallel Processing Architectures (PPA). In this paper we will focus on the Computer Systems Architecture and Parallel Processing Architectures lectures as introduced in Sects. 2.1 and 2.2 respectively. Section 2.3 briefly goes through other PDC graduate courses, not covered thoroughly in this paper. The number of students taking the PDA and CSA lectures ranges from 350 to 450 each year – these two lectures being compulsory for all students enrolled at the Computer Science and Engineering Department. The PPA lecture gathers from 130 to 150 students, in the Advanced Computer Architectures specialization of our bachelor Computer Science Program.

2.1 Computer Systems Architecture

The Computer Systems Architecture lecture is presented in the sixth semester of bachelor study. This lecture presents the fundamentals of design and structure of numerical computing systems. The main topics covered include:

– Processor Memory Switches descriptions of computing systems.
– Various taxonomies of computing systems.
– Fundamentals of SIMD and MIMD design, architectures, and applications.
– Hierarchical and non-hierarchical switches.
– Switches for inter-processor and processor-memory communication.
– Inter-cluster and intra-cluster communication protocols.
– The roof-line model.
– Advanced CPU and GP-GPU computing architectures.
– Debugging and performance evaluation and analysis of computer programs.
– Profiling and tracing computer codes on modern processing platforms.
– Parallel correctness challenges.
– Benchmarking computing systems.
– Analysis of top 500 systems architectures over the years.

The practical activities over the course of the entire semester deal with three different topics, namely concurrent programming in Python, serial code optimization, profiling and OpenCL programming in C. The lecture as well as the practical activities are being update continuously. More details are given in Sect. 3.

2.2 Parallel Processing Architectures

The PPA lecture is given in the ninth semester of bachelor study. The main objective of this lecture is the assimilation of fundamental concepts concerning parallel processing architectures design, programming and configuration. During this lecture students learn to analyze parallel processing models, as well as synchronization issues in complex parallel and distributed systems. During the lecture, the following topics are discussed:

- The evolution of parallel processing systems.
- The concepts of concurrency and parallelism.
- Indicators for evaluating parallel structures.
- Parallel systems classifications.
- General characteristics of parallel processing systems.
- Mathematical models of parallel computation.
- Relationships between parallel architectures and parallel algorithms.
- Parallel computation limits and levels of parallelism.
- Synchronization in parallel and distributed systems.
- Parallel system architectures with practical examples.

In this lecture, the practical activities are split between two phases: the first six weeks of the semester in which students learn and practice advanced issues on OpenMP, MPI, and PThreads programming, while the remaining eight weeks of the semester are spent working in teams of two or three on software projects in which they are attempting to parallelize given serial computer programs.

2.3 Graduate Lectures on PDC

The graduate lectures gather from 25 to 40 students, in the Advanced Computer Architectures [1] and Parallel and Distributed Processing Systems specializations [10] of our bachelor Computer Science Program.

Parallel Programming is a lecture outlining a series of programming paradigms in the context of modern parallel computer architectures. It offers an overview of parallel programming models considering issues such as productivity, performance, and portability and presenting a number of models for communication, synchronization, memory consistency and runtime systems. Various parallel programming paradigms with shared- and distributed-memory, parallel global address shared space, and other atypical paradigms are presented.

High Performance Scientific Computing presents state-of-the-art parallel computing architectures in the context of modern parallel programming paradigms. Topics include mathematical modeling, numerical methods and data structures employed in HPC, from systems of differential equations, automatic differentiation, optimization problems, solving systems of nonlinear equations, to basic linear algebra and chaotic systems. The lecture also tackles scientific applications requiring HPC systems with examples from research and industry.

3 Practical Activities

3.1 Computer Systems Architecture

The CSA practical activities follow the structure of most of the courses in our faculty's curriculum for the first six semesters, consisting of weekly two-hour labs and three or four homework assignments every two or three weeks. Through the years such a structure received positive feedback from students and proved very efficient in the development of their technical skills and on their understanding and application of the subjects presented during lectures. In terms of organization, our activities bring something new to the students: they are split into three, formerly four, distinct topics and technologies, the homeworks require not just coding but also analysis and performance evaluation and they offer the students a chance to enrich their presentation skills.

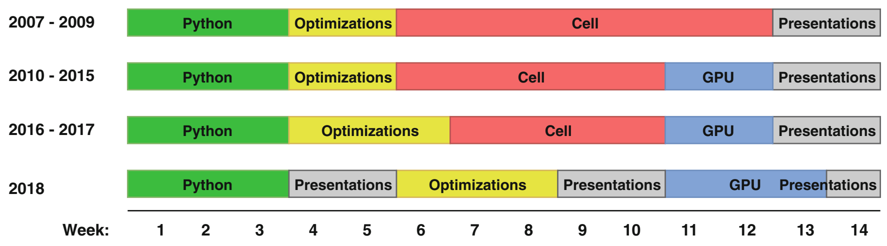

Fig. 1. CSA Practical activities through the years.

The topics taught during CSA's lab activities cover concurrency and multi-threaded programming (in Python), optimizations and profiling (in C), parallelization of computationally intensive programs, using Cell, OpenCL [21], or CUDA [16]. We adapted this structure based on technological evolution, as shown in Fig. 1. For conciseness, throughout this article we will refer to these parts of the CSA's activities using the technology/language used for them.

The last two weeks of the semester were dedicated to presentation sessions, in which students chose a topic related to the course or the lab and presented it in 10 min. The presentations were extremely varied and up to date to the newest trends in high performance computing, parallelism and concurrency and even embedded systems (e.g. a cluster build out of Raspberry Pi boards). This year, we replaced these presentations with ones in which they present their homeworks, a decision we discuss in Sect. 6.

The main reason we chose Python was the desire to present *concurrency* concepts in multi-thread programming in a widely used language. Due to its quick learning curve, a simple threading API and the fact that we are focusing on correctness of concurrent programs and not on parallelization performance, Python is the right choice for our needs. The concepts learned during the lab exercises and from the homework assignment can be easily applied to other languages too.

During the weeks dedicated to concurrent programming students learn Python's basic syntax and data structures, how to create and manage threads and how to protect the access to shared resources using locks and semaphores, events, conditions and synchronized queues.

The first of the three labs dedicated to *program optimization* offers an introduction into Intel and AMD general purpose CPU architectures, with exercises meant to detect the actual size of the line of cache and of the LLC – Last Level Cache – of the processors in the lab. The second lab is treating a number of serial optimizations of the well-known matrix multiply kernel, on CPUs, from improving access to vectors, to loop order optimization and block-matrix optimizations. This section concludes with a lab on performance optimization using dedicated software tools such as Valgrind, perf and Intel Parallel Studio. Therefore, students learn about the specifics of CPU architectures, serial optimization techniques, and how to identify performance bottlenecks using specialized tools.

We have chosen the OpenCL programming paradigm as it is a natural transition from the previous *IBM Cell architectures* we used to teach. Similar to the IBM CELL framework there is a clear distinction between *device* GPU (as the SPE for IBM Cell) and the *host* CPU (as the PPE for IBM Cell). From the execution point of view the *host* is responsible for managing the *device* hardware similar to the PPE that managed the SPEs. Likewise the OpenCL kernels need to be compiled for the specific target *device* and sent out by the HOST which highlights how a true heterogeneous system works in the back-end. Although OpenCL programming is a difficult topic, it is worth learning since it offers students a deeper understanding of the advantages and limitations of most co-processing architectures, like GPUs, FPGAs, ASICs, etc.

While CUDA is the de-facto standard when it comes to the HPC industry, *OpenCL* provides a better understanding of the underlying interactions in components of a heterogeneous system. Since we also touch upon the systems programming side, we consider OpenCL is better suited in Academia than CUDA. OpenCL was designed to support any number of devices (e.g. CPU, GPU, FGPA, ASIC) from any vendor, while CUDA is a closed ecosystem targeting only NVIDIA GPU hardware. Thus, our students learn how to query for platforms and devices of different vendors, how to allocate and manage buffers as well as how to perform cross compilation of kernels. They also understand that the software stack induces latency and can significantly impact performance. The transition from OpenCL to CUDA is easy, since for beginners CUDA represents a simplification of the OpenCL API. We offer three OpenCL labs: the first focuses on the host side interactions with the device (queries, buffer allocation, kernel enqueue), and the next two focus more on the underlying architecture of a GPU and how to design an efficient kernel program.

3.2 Parallel Processing Architectures

The PPA labs consist of two different parts, namely a hands-on section of labs and a team project. The hands-on section tackles advanced issues concerning PThreads, OpenMP [15], MPI programming [18], and concludes with a profiling and parallel debugging lab. The team project is focused on deploying, under our team's supervision, shared as well as distributed memory programming techniques on serial applications chosen by the students. The projects conclude with presentations in front of the class outlining the benefits and drawbacks of each particular programming approach, as well as the influence of the underlying machine and system architecture on the performance of the chosen application.

4 Student Assessment and Evaluation

4.1 Lab Activity and Homeworks

During the two-hour lab activities of the **Computer Systems Architecture** lab students get the chance to practice their coding and apply the concepts learned during the lectures or from the lab's wiki page [3]. The exercises also challenge them to look for performance issues, optimizations and also understand the architectures their code runs on. During each lab, the teaching assistants present and explain the main concepts in the first 15–30 min and then help the students with their tasks (individual explanations, debugging, discussions about their results).

We use the wiki as support for labs and homeworks. On each lab's page we offer an overview of the topic, examples, links to additional resources and tasks. Most of the labs also provide a code skeleton the students can build upon. Only the lab about the profiling tools has less coding and its flow is tutorial-like, with students having precise instructions on what to create, click, and run.

The first concurrency lab focuses on exercising Python syntax and its challenge is teaching the fundamentals of a new language in just two hours. Therefore, we varied the difficulty and the amount of tasks through the years. We first offered many short tasks that covered a lot of concepts but the students' feedback showed us that it is more important to provide the tasks a story and not require the use of that many language features, so the current exercises simulate a coffee machine. The second concurrency lab starts with a simple exercise that requires the creation of threads that concurrently modify a list, and then requires the implementation of well-known concurrency problems, like producer-consumer or dining philosophers. In the third lab students work with events, conditions and barrier objects to implement a gossiping algorithm and a master-slave scenario.

The Cell labs provided code skeleton that the students adapted and the tasks covered all the topics presented on the wiki: creation and management of SPE threads, vectorization, data transfers using DMA, double buffering, mailboxes and caches. The exercises were compiled and run on our cluster. Students understood the concepts but their performance during the labs was hindered by C programming aspects such as data alignment. Therefore, we have included in the optimization labs some tasks for allocations and pointer casts.

The purpose of the OpenCL GPU/CPU labs is to understand the main differences between programming on a CPU and on a GPU. For a typical lab, students have a skeleton code on which they will have to fill in the gaps for the proper execution to take place. The labs gradually go from a high level view of

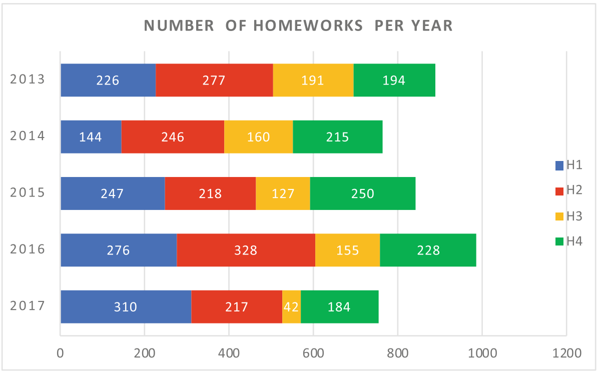

Fig. 2. Number of submitted CSA homeworks (H1, H2, H3 and H4) in the last five years. In 2016 and 2017 the students had to choose between H3 (Cell) and H4 (GPU), they were not required to do both.

the OpenCL stack to the low level details of kernel programming on a certain architecture.

The common themes of the **Computer Systems Architecture** homework assignments are the following: threads that concurrently access each other's data in order to apply an algorithm for the Concurrency track, implementing a BLAS [2] operation in several ways (basic, optimized, basic compiled with flags) and comparing the performance against the library's implementation for the Optimization track, parallelization of a serial algorithm in OpenCL. The Cell homeworks revolved around image and video processing and required the use of DMA transfers. The students also had to perform optimizations using vector operations and double buffering. With the exception of the concurrency homework, the students have to provide relevant graphs and explanations about their solution's performance. For the Python track, we encourage students not only to write correct concurrent code, but also respect a coding style and document it. To ease the evaluation and to help them, their homeworks are also tested with Pylint [11], a Python code analysis tool. As an incentive, we offer bonus points to homeworks exhibiting high Pylint scores.

We are addressing a large number of students each semester, which makes it overwhelming to evaluate more than 500 homeworks per semester only for one course, as presented in Fig. 2. Moreover, the fact that the CSA homeworks require running the solutions on various architectures and also measuring their performance, makes it more difficult to integrate with VMChecker, an automatic grading system. We use VMChecker only for OpenCL GPU assignments, while for the rest we provide public tests and scripts that automate the runs, so that students can test their solutions before submitting them on the course's platform. Over-subscription of cluster queues by students is one disadvantage to using VMChecker for the GPU assignments, since it requires constant monitoring so that the system remained responsive (i.e. queues not full). The end result is faster grading but the trade-off comes from deploying and managing the system.

For the **Parallel Processing Architectures** projects, students are divided in groups of two or three and decide on the project's topic, usually the parallelization of CPU-intensive applications written mostly in C or C++. Then each week we decide together with the students on the tasks they have to do until the subsequent week. The parallelization paradigms they use include PThreads, OpenMP and MPI, and usually, each member of the team is in charge of one parallelization strategy. At the end of the semester, each team presents before the entire class the results, outlining the lessons learned, the benefits as well as the drawbacks of each programming paradigm in the context of their particular software application. The projects are done in teams, however grading is individual, to ensure fairness and accountability of our student's effort.

4.2 Computing Infrastructure

Computing infrastructure is one of the key elements in applying theoretical aspects shown in lectures, especially in computer architecture and parallel programming. For our courses we need a variety of platforms (e.g. x86 CPUs, embedded ARM CPUs, specialized PowerPC CPUs, or GPUs) for students to be able to compare them and a high number of units (CPUs/GPUs) in order to assess the performance of parallel implementations. Therefore, we rely on the Computing Cluster of our department, where all the HPC resources are aggregated, as summarized in Table 1.

Table 1. The CS computing cluster.

Nodes	Node type	CPU	GPU	RAM
32	IBM HS21	Intel Xeon E5405	–	16 GB
28	IBM HS22	Intel Xeon E5630	–	32 GB
16	IBM LS22	AMD Opteron 2435	–	16 GB
4	IBM QS22	Cell BE Broadband	–	8 GB
8	IBM PS703	IBM Power7	-	32 GB
4	IBM iDataPlex dx360M3	Intel Xeon X5650	8 NVidia Tesla M2070	32 GB
3	HPE ProLiant BL460c	Intel Xeon E5-2670	7 NVidia Tesla K40m	128 GB

The storage infrastructure of the CS Cluster is currently composed of multiple systems with different capacities, such as: an IBM Storage Fibre Channel DS3950 with 30 TB, a Dell PowerVault with 120 TB, and a HPE MSA P2000 with 6 TB of storage space. On these storage servers we installed, over time, multiple file system solutions for distributed and parallel computing systems. Among others, we explored NFS, Lustre FS, and GlusterFS. Lustre FS did not scale because of the significant configuration and restart times. GlusterFS was a good solution, however once we reached 30 million files, the system slowed down significantly.

The current NFS solution employs 10 Gbps links, with fast disks which offer good scaling for about 20 TB of data and multiple millions of user files.

Network connectivity within the cluster is ensured by 56 Gbps Infiniband links connecting computational nodes to the centralized storage; normal Gigabit Ethernet links for network and storage connectivity; and 10 Gigabit Ethernet links for network, storage and Internet connectivity. Currently the uplink uses 2x10 Gbps Ethernet links.

The hardware infrastructure described previously is complemented by the use of the Moodle [7] open-source learning system. Our Moodle implementation integrates students database information with automatic accounts creation, course creation based on the CS curricular structure, and course enrollment for students based on their contracts. Moodle fulfills most of our needs with: storage for resources (documents, slides), interactivity with students via forums and feedbacks and assignments upload and grading. For the collaborative design and deployment of lab materials we use a Dokuwiki [4] instance on the same server. During the Dokuwiki integration with our Moodle system and our student database, our team also contributed back to upstream with different features that would help others implement a similar system. In the near future we plan to integrate automatic programming assignment verification using the VMchecker [14] tool.

Cluster management is currently achieved using Open Grid Scheduler [9], and will shortly be migrated to Torque [13]. We offer our students and users interactive, as well as non-interactive (i.e. batch-mode) use of our systems. Having a significant number of compute nodes as well as a big number of end-users places a high demand on our software stack (e.g. compilers, libraries, applications, tools). Thus, the management of different software versions is done by employing the `module` feature. It basically sets/unsets environment variables depending on the desired version of software being selected. Best performance for all our packages is obtained by compiling most of the stack directly from sources and creating our own RPMs for each cluster node architecture.

4.3 Final Examination and Feedback

The final examinations for both CSA and PPA typically consist of two different parts: a theoretical section of 50 min, where students are required to answer to 10 questions from the entire lecture; and a practical section of 45 min in which students have to solve a practical assignment linked to the lab activity. The total scores of the lab activity, homeworks, theoretical and practical examination are then added together to give the final grade for each lecture participant. During the last two weeks of the semester, students offer their feedback to our team - of course the information is available for us only after the end of the examination period. Student feedback is a constant source of improvement of our activity, and a good indication of the interest towards PDC subjects in our Department. Over the last years, we have thus constantly striven to offer our students access to the most advanced processor and parallel systems architectures.

5 Industry Involvement

Between the eighth and ninth semesters of their bachelor program, students are required to spend at least twelve weeks in internships or summer-schools on topics related to their chosen field of study. Over the years, a number of summer-schools have been organized in our Department, on topics ranging from High Performance Computing, Embedded Systems, Security, Mobile Development, Artificial Intelligence, GPU programming, Machine learning to Computer Vision and 3D-Graphics technologies. The industry has also diversified its internship offer to students, with topics on Business software development, Cloud programming, Artificial Intelligence, Embedded Systems, IoT, Mobile, Gaming, Networking, Telecommunications. To this end, the "Stagii pe Bune" [12] and "Junio.ro" [6] platforms were jointly developed by people from our Department and from the IT industry. More recently, students apply to internships abroad. Participation in Google, Facebook, and Microsoft internship programs is constantly growing. Some examples of companies, typical programming requirements, and representative technologies covered by their internships are given in Table 2.

Table 2. Internship listings.

Company	Requirements	Technologies
NXP	C/C++, Python, OpenCL, knowledge of microprocessors architecture	Automotive, IoT
Intel	C/C++, Python, Bash, Profiling skills	Microarchitecture Design and Optimizations
BitDefender	Algorithms, C/C++	Big Data Analysis, System Programming
Adobe	Algorithms, C/C++	Big Data Tehnologies, Application Design

Each year our team is considering the requirements and feedback received from the industry when redesigning or adapting our curricula for the next year. This process is smoothed by our integration of a significant number of teaching assistants (TAs) directly from industry professionals. Thus our students can learn where different types of problems presented at our laboratories occur in the daily life of an IT engineer. Another advantage of having input directly from an industry that is evolving so fast is having an objective view of how our teaching materials helps our students fulfill their job requirements.

6 Lessons Learned

Through the years our team has adapted to the feedback received from students in previous generations. For example, we introduced specific sections in the practical exercises of the labs based on common mistakes or challenging parts of the student assignments. To illustrate this point: we observed misunderstandings on how the threads run, a tendency for busy waiting and a wrong usage of events. We therefore provided more explanations and examples in the lab's wiki and offered exercises based on code skeletons, that showed incorrect approaches and asked students to improve them.

To encourage students to submit more homeworks, we developed a system of soft deadlines for two or three weeks after they are published, and only then impose a final hard deadline. Nonetheless, we observed that most students start working on homeworks in the last few days before the soft deadline, which had a significant impact on our hardware resources. Therefore, a few years ago we introduced a further incentive for submitting homeworks early – in the form of bonus points – an approach which proved quite successful.

To better assess the students' understanding of their homework and also tackle the plagiarism problem, we introduced this year the requirement for homework presentations in front of the class. To improve the uniformity of the TA's evaluation, we created homework evaluation guidelines, as well as typical errors and questions which should be posed to students during their evaluation. Over the years we have used automatic grading systems along with MOSS [8] and Etector [5] code plagiarism detection systems.

At the end of each semester, our entire team takes part in a debrief where we discuss all the problems we encountered during the lecture, practical activities and homework assignments. Possible improvements, owners and solutions are offered, and each point is then taken under consideration at the setup meeting of our group in the next academic year.

7 Conclusions and Outlook

7.1 Conclusions

The team at the Computer Science and Engineering Department of the UPB is striving to improve the presentation of PDC concepts in its undergraduate curricula. In the lectures, students are taught general architecture and design aspects of PDC, while in the practical activities they explore various software approaches best suited to illustrate those general concepts. Assignments and homeworks are then meant to check that the relevant desired skills have been learned by our students. In this article, we outline the content of the lectures, the student evaluation process, as well as the lessons learned over time, and the improvements we introduced in our content and approach. The IT industry is exhibiting a particular interest in our graduates, and their PDC skills are highly appreciated. This is also due to the fact that we have continued to evolve our Materials and Methods constantly, as new technologies emerge. At the same time however, we aim for our students to have a fundamental understanding of how parallel and distributed processing architectures work, from both the hardware and software perspective. As new architectures emerge continuously, driven now by emerging domains such as AI or IoT, the essential building blocks remain the same – and PDC is one of those blocks.

7.2 Outlook

We are constantly adapting our curriculum as the industry evolves. One interesting direction are cross-API intermediate languages such as SPIR which provide the underlying runtime for several APIs, such as OpenCL, Vulkan, SyCL, OpenMP, or OpenACC. Moreover, we are considering the addition of a section exemplifying the interaction of OpenCL with popular data analytics and machine learning frameworks such as Anaconda, by using the PyOpenCL [17] wrapper, thus linking together the CSA labs on Python and OpenCL.

Acknowledgements. The authors would like to thank Professor Nicolae Tăpuş, Alexandru Herişanu, Răzvan Dobre, Vlad Spoială, Dan Dragomir, Alexandru Olteanu, and Voichiţa Iancu for their valuable contributions to the CSA and PPA curriculum. This work is partially supported by project Sovarex, ID: 10PS/2017.

References

1. ACA master program. https://cs.pub.ro/index.php/education/courses/68-mas/aca?layout=. Accessed 14 May 2018
2. BLAS Basic Linear Algebra Subprograms. http://www.netlib.org/blas/. Accessed 14 May 2018
3. CSA wiki. http://cs.curs.pub.ro/wiki/asc/. Accessed 8 May 2018
4. Dokuwiki homepage. https://www.dokuwiki.org/dokuwiki. Accessed 14 May 2018
5. ETector homepage. http://www.etector.org/show.cgi. Accessed 14 May 2018
6. Junio homepage. https://junio.ro. Accessed 27 Apr 2018
7. Moodle homepage. https://moodle.org. Accessed 14 May 2018
8. Moss - for a Measure Of Software Similarity. https://theory.stanford.edu/~aiken/moss/. Accessed 14 May 2018
9. Open Grid Scheduler. http://gridscheduler.sourceforge.net/. Accessed 14 May 2018
10. PDPS master program. https://cs.pub.ro/index.php/education/courses/70-mas/pdps?layout=. Accessed 14 May 2018
11. Pylint homepage. https://www.pylint.org/. Accessed 25 Apr 2018
12. Stagii pe bune homepage. https://stagiipebune.ro. Accessed 27 Apr 2018
13. Torque Resource Manager. http://www.adaptivecomputing.com/products/open-source/torque/. Accessed 14 May 2018
14. VMChecker. https://github.com/rosedu/vmchecker. Accessed 25 Apr 2018
15. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel Programming in OpenMP. MK Inc., San Francisco (2001)
16. Cook, S.: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, 1st edn. MK Inc., San Francisco (2013)
17. Pierro, M.D.: Portable parallel programs with Python and OpenCL. Comput. Sci. Eng. **16**, 34–40 (2014)
18. Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, Knoxville, TN, USA (1994)

19. Ghafoor, S., Brown, D.W., Rogers, M.: Integrating parallel computing in introductory programming classes: an experience and lessons learned. In: Heras, D.B., Bougé, L. (eds.) Euro-Par 2017. LNCS, vol. 10659, pp. 216–226. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75178-8_18
20. Paprzycki, M., Wasniowski, R., Zalewski, J.: Parallel and distributed computing education: a software engineering approach. In: Ibrahim, R.L. (ed.) CSEE 1995. LNCS, vol. 895, pp. 187–204. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-58951-1_104
21. Munshi, A., Gaster, B., Mattson, T.G., Fung, J., Ginsburg, D.: OpenCL Programming Guide, 1st edn. Addison-Wesley Professional, Upper Saddle River (2011)
22. Niculescu, V., Bufnea, D.: Experience with teaching PDC topics into Babeş-Bolyai University's CS courses. In: Heras, D.B., Bougé, L. (eds.) Euro-Par 2017. LNCS, vol. 10659, pp. 240–251. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75178-8_20
23. Raynal, M.: Parallel computing vs. distributed computing: a great confusion? (position paper). In: Hunold, S., et al. (eds.) Euro-Par 2015. LNCS, vol. 9523, pp. 41–53. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27308-2_4
24. Tomeu-Hardasmal, A.J., Salguero, A.G., Capel, M.I.: Integration of ICT in concurrent and parallel programming lectures. In: Hunold, S., et al. (eds.) Euro-Par 2015. LNCS, vol. 9523, pp. 114–124. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27308-2_10