International Journal of Engineering Education Vol. 31, No. 5, pp. 1343–1351, 2015
Printed in Great Britain

0949-149X/91 $3.00+0.00

Teaching Introductory Parallel Computing Course with Hands-On Experience*

NATALIJA STOJANOVIC and EMINA MILOVANOVIC
Faculty of Electronic Engineering, University of Nis, Aleksandra Medvedeva 14, 18000 Nis, Serbia.
E-mail: natalija.stojanovic@elfak.ni.ac.rs

This paper presents an innovative course designed to teach parallel computing to undergraduate students with significant hands-on experience. This course represents an introduction to the main topics of parallel, distributed and high-performance computing (HPC). The course introduces main concepts and architectures used in parallel computing today, and improve students' skills to develop parallel programs using major parallel programming paradigms: MPI (Message Passing Interface), OpenMP (Open-Multiprocessing). The main objective of the course is to teach practical parallel programming tools and techniques for MIMD with shared memory, MIMD with distributed memory and SIMD. Each of these software tools can be used to give students experience with parallelization strategies, and ability to rate the quality and effectiveness of parallel programs. We evaluate the success of our approach through the use of testing and survey and provide directions for further improvements in teaching parallel programming.

Keywords: parallel computing; parallel programming; high performance computing; education; MPI; OpenMP; CUDA

1. Introduction

The advances and proliferation of multiprocessor computing architectures during the last decade have given rise to advanced parallel and distributed computing research and development [1]. Recently, there has been an increase in the acceptance and implementation of parallel and distributed computing, both for high-performance scientific computing and for a wide spectrum of general-purpose computing applications [2]. Accordingly, parallel and distributed computing has moved from mostly elective graduate courses to become a core component of undergraduate computing curriculum. Therefore, it is not sufficient for current computer science graduates to master sequential software development and appropriate software tools and methods. With the expected rapid changes and advancements in high performance computing in the coming years, based on multi-core processors, many-core GPUs, cluster of workstations and distributed data centers in clouds, there is an increasing need for including parallel computing topics in computer science curriculum at an early stage. Such an introductory course will be the foundation for later undergraduate and graduate courses that should include broad- and deep-based topics in parallel, distributed and high-performance computing. The need for introducing parallel and distributing courses in undergraduate computer science curriculum is widely recognized in the computing community through various initiatives, such as ACM/IEEE Computer Science Curricula 2013 [3] and NSF/TCPP Curriculum Initiative on Parallel and Distributed Computing—Core Topics for Undergraduates [4].

Many researchers and educators have considered how parallel computing should be introduced in undergraduate courses [5, 6]. Since many universities lack the funds to purchase expensive parallel computers, cost effective alternatives are proposed to teach parallel computing and programming methods and techniques. The rapid change in computing platforms, programming languages and environments, and software development tools, make the challenge for educators to establish appropriate and effective curriculum related to parallel and distributed computing.

At the University of Nis, we regularly teach an introductory parallel computing course considering parallel computer architectures and parallel programming methods and techniques. As parallel computing platforms we use multi-core and many-core processor architectures, network of computers and parallel software simulators. We present our past and current experiences in teaching introductory parallel computing course that confirm that such a course can be taught successfully with considerable hands-on experience, with a limited budget.

The rest of this paper is organized as follows. Section 2 presents the related work in teaching parallel and distributed computing across undergraduate and graduate computing curricula. Section 3 describes important parallel computing models associated and the way they can be introduced in an undergraduate course. Section 4 presents a practical approach to introducing parallel

* Accepted 25 April 2015.

computing to undergraduate students with considerable hands-on experience. Section 5 evaluates our approach and confirms its benefits. Finally, Section 6 provides some conclusions.

2. Related work

With the increasing presence of multi-core processors, many-core graphics processing units (GPUs), network and clusters of computers/workstations, there is a need for including broad- and deep-based courses in parallel and distributed computing at various levels in the Computer Science and Engineering curriculum. Several approaches presented in the literature have been proposed in an attempt to integrate parallel computing and programming topics across the computer science curriculum.

Danner and Newhall in [7] discuss changes made to incorporate parallel and distributed topics into all levels of undergraduate liberal arts computer science curriculum. The center of their changes is a new intermediate-level course that introduces students to computer systems and to parallel computing that serves as a foundation to many upper-level courses that provide more breadth and depth of coverage of parallel and distributed computing topics.

Arroyo in [8] proposes a set of related parallel and distributed programming topics, to be included in the updated computer science curriculum core courses at the Río Cuarto National University, Argentina. They discuss several approaches for teaching parallel programming topics in a set of core courses to achieve a consistent, increasing and complete training in high performance computing. They describe the use of parallel and distributed computing tools suitable for teaching parallel programming in different courses.

Deo et al. in [9] present an approach to include an interdisciplinary course on computational modeling with a focus on parallel programming across the undergraduate curriculum. They argue that computational modeling is a fundamental process in all scientific disciplines, and that there is a necessity to effectively employ massively parallel high-performance computing machines in scientific computations.

Brown et al. in [10] present their experiences in teaching concepts of parallel computing in two undergraduate programming courses and an undergraduate hardware design course. They describe how parallel concepts have been integrated in the courses, the assessments, and the results.

Gross in [11] presents an approach for including parallel programming in the engineering-oriented undergraduate curriculum. They added an introduction to parallel programming to the list of mandatory courses in the 2nd semester, exposing students to three styles of parallel programming: threads with shared memory, CSP-style message passing, and OpenMP-based parallel programming. They discuss benefits and disadvantages of including an introductory class on parallel computing early in the undergraduate curriculum and point to several issues that will be considered in the future.

Rivoire in [12] presents an approach to teaching parallel programming models, by offering a breadth-first introduction to multi-core and many-core programming for upper-level undergraduates. The students gained programming experience with three different parallel programming models related to multicore and manycore computing. Their assessments showed that the course gave students a broad skill set in parallel programming and ongoing developments in the field.

Freitas in [13] describes an experiment on a traditional parallel programming Computer Science course trying to explore the best schedule for introducing parallel programming topics in order to improve the quality of learning. Their results show that the best results are achieved when the OpenMP model is introduced before the MPI model. They conclude that such a schedule properly emphasizes parallel programming concepts and improve student motivation and learning.

Fienup in [14] presents their experience with incorporating parallel computing topics into the Computer Science curriculum at the University of Northern Iowa via the Computer Architecture course. The paper discusses details related to course goals, parallel programming tools, and techniques for teaching parallel programming topics.

Praun in [15] presents an introductory parallel programming course at the bachelor level organized along the tiers of parallelism. These tiers are from higher to lower abstraction levels and are defined according abstractions and concepts needed when developing a parallel program. The goal of the class is to introduce fundamental principles of parallel systems and to serve as a platform for further exploration in specialized parallel computing courses. The course has a significant share of lab sessions and programming projects using X10 programming language. The first experience in teaching this course shows very positive student feedback.

Keller in [16] describes the teaching approach, the content and the lessons learned of a lecture on parallel programming for undergraduate students. The focus of the course was on to provide hands-on experience in developing parallel code for a HPC cluster. The aim of the lecture was to provide an in-depth understanding of several actual parallel programming paradigms: OpenMP, MPI and

OpenCL, as well as MapReduce based on Apache Hadoop platform. The strong point of this course lays in practical parallel programming and work on real life codes.

Pacheco in [17] presents an approach for teaching parallel computing to undergraduate computer science students early in the curriculum. The emphasis is on hands-on experience and early start of "thinking in parallel", while formalism and rigor are less important. The main topics of their course are based on using MPI, Pthreads, and OpenMP. The coursework includes weekly homework assignments, five programming assignments, two midterms and a final exam. They report satisfactory outcomes, student enthusiasm and motivation to further study graduate parallel computing topics.

Our approach to teaching introductory parallel computing course at the undergraduate level is similar to [16] and [17] regarding hands-on experience, but in our opinion, it is more balanced and appropriate as the introductory undergraduate course in parallel computing. We take care that the curriculum is not overwhelmed for a single semester course, as it seems in Keller's approach. Also, in our course we cover in-depth the currently most important parallel programming paradigms, OpenMP, MPI, and basics of CUDA, and demonstrate the effectiveness of applying parallel programming paradigms over them in solving real life computing and data-intensive problems. Also we use Parallaxis simulator [18] for demonstrating SIMD concepts that are not explicitly supported by mentioned programming models.

3. Models of parallel computing

To achieve high performance, application software needs to effectively use parallelism available in current computer architectures. There are several types of parallel computing architectures, based on parallel and distributed processing principles. Some of such architectures employ parallel processing at a single computing node while others are built from collections of networked, possibly heterogeneous computers/workstations. Because of its scalability and availability these networked environments represent a promising solution for low-cost, high performance computing. There are also different models of parallel programming based on different topologies that define connection of processors and memory modules. The three most commonly used models are: SIMD (Single Instruction Multiple Data), MIMD (Multiple Instructions Multiple Data) with shared memory and MIMD with distributed memory. Parallel software platforms that we use in our course to support these models are:

- OpenMP (Open-Multiprocesing) for MIMD with shared memory.
- MPI (Message Passing Interface) which enables that a network of workstations is treated as MIMD with distributed memory.
- Parallaxis software simulator of SIMD computer [18].

Nowadays, each desktop or laptop computer empowered with multi-core processor becomes a small parallel system. Consequently, OpenMP has been developed to enable writing of parallel programs for shared-memory multiprocessor platforms. MPI has become the major model of programming distributed-memory applications on a cluster of workstations. Parallaxis is a sophisticated SIMD simulator which runs on a variety of platforms. Parallaxis is a machine-independent language for data-parallel programming. Programming in Parallaxis is done on a level of abstraction with virtual processors and virtual connections, which may be defined by an application programmer.

3.1 OpenMP

Shared-memory systems include many CPUs that share the same physical memory. This kind of architecture is sometimes called MIMD (Multiple Instruction Multiple Data) with shared memory. Until recently, shared-memory systems cost hundreds of thousands of dollars and were affordable only by large companies and scientific/education institution. Nowadays, multi-core machines, in which two or more CPUs share a common memory, in the desktop, laptop computers and even in smartphones, are widely available.

OpenMP represents a set of compiler directives, library routines, and environment variables which enable programmer to specify to the compiler which instructions to execute in parallel. Also, OpenMP provides programmer to define how to distribute these instructions among the threads that will run the code. OpenMP is not a new programming language. Rather it is a notation that can be added to a sequential program in C, C++ and Fortan to describe how the work can be shared among the threads that execute on different processors or cores, as well as to order access to shared data [22].

OpenMP supports the so-called fork-join programming model (Fig. 1).This approach assumes that the program begins execution as a single thread. Every time this thread encounters OpenMP parallel construct during program execution, it creates a set of threads. It then becomes a parent thread and cooperates with other threads of the program execution. At the end of parallel construct only the initial

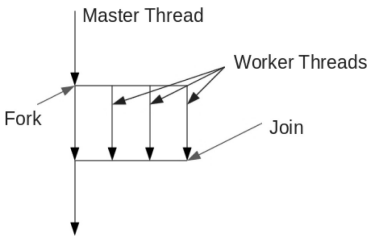

Fig. 1. OpenMP fork-join programming model.

thread continues execution, interrupting the execution of others.

To properly implement parallel OpenMP application, it is expected that a student specifies the parallel parts of the program and the method of parallelism applied. The responsibility of OpenMP framework is to classify the parts of the program and create appropriate threads, as well as to allocate a piece of code to be executed by each thread. The method of work division can have a significant impact on the program performance. Students usually use OpenMP to parallelize loops. They also have opportunity to use *schedule* clause to perform different scheduling policies which determine how loop iterations are mapped to the threads. In loop parallelization, the students also get confronted with problems caused by data races and dependencies.

3.2 MPI

A set of commodity PCs (nodes) networked together can be used as a parallel processing system. The PCs are individual machines, which can be uniprocessor or multiprocessor. Networking them together and using parallel-processing software environments, such as MPI, can form very powerful parallel systems (Fig. 2). MPI has become the major model of programming parallel distributed-memory applications. MPI is a specification, not an implementation of library routines, helpful for users that write portable message-passing programs in C/C++ and Fortran. There are different implementations of MPI: MPICH, OpenMPI, LAM, WMPI, Cray MPI, etc. The most popular MPI implementation is MPICH2 [19].

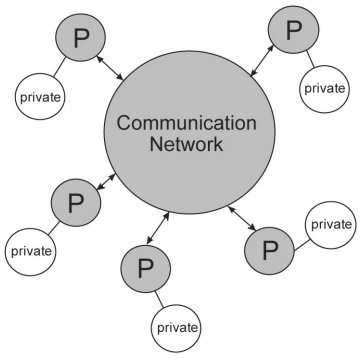

Fig. 2. The overall architecture of network of workstations-MPI.

The major goal of MPI, as with most standards, is a degree of portability across different machines. MPI has the ability to run on heterogeneous systems, groups of processors with distinct architectures [20, 21]. Therefore, MPI provides a computing model that hides many architectural differences. It is not necessarily to know whether the program code sends messages between processors of the same or different architectures. The MPI implementation automatically performs any necessary data conversion and utilizes the correct communications protocol. This means that the same source code can be executed on a variety of machines as long as the MPI library is available. Message passing works by creating processes which are distributed among a group of processors. The basic assumption behind MPI is that multiple processes work concurrently using messages to communicate and collaborate with each other. In this course, students learn about communication paradigms of MPI and consider the advantages and disadvantages of blocking and non-blocking communication. By applying the knowledge regarding collective communication, user-defined communicators, derived datatypes and virtual topologies the students are able to apply different parallelization strategies to distribute work among multiple processors.

MPI is not a simulator; it uses multiple processors to perform the work. This means that execution times can be captured and analyzed. It can be concluded from observing many parallel algorithms implemented using MPI that they run slower than their sequential counterparts. The overhead associated with forking and killing processes and the time it takes to send messages across the network are the main causes of the slowdown. Actual speedup mostly occurs in algorithms that perform large amounts of computation and require small amounts of data transfer. However, the slowdown opens the eyes to students as to the limitation of parallel computations, and a discussion of the causes is very educational. Even though slowdowns, the ability to capture timing data does allow the students to make relative comparisons between alternative parallel algorithms.

3.3 Parallaxis

In Parallaxis [18] each processing element has its own local memory, but all processing elements execute common instruction stream synchronously. Parallaxis allows the programmer to specify the number of (simulated) processing elements and the topology which connects the elements. Parallaxis supports two types of variables, scalar and vector. Scalar variables reside on the central control unit,

and vector variables reside on the processing elements. When writing programs for Parallaxis, the student needs to determine what parts of the algorithm can be executed in parallel and what parts have to be executed sequentially. Vector variables have to be specified for the data that will participate in the parallel portion of the algorithm. Corresponding scalar arrays need to be created to support I/O operations. Load and store statements copy data between a scalar array and a vector variable. Operations referred to vector variables are: propagate operation which copy data from one processing element to another and reduction operators, such as a global addition, summarize data across all processing elements. Parallaxis lets students experiment with different connection topologies and mapping parallel algorithms onto specific topologies, but they do not have chance to experiment with the speedup of a true parallel machine.

4. Programming assignments and projects

To efficiently teach and evaluate students in introductory parallel computing, we organize assignments, programming projects and a final exam. Since, the course emphasis is on parallel programming, the projects are an important part of the course. They give students hands-on experience on using MPI and OpenMP on commodity computers.

Assignment 1: The purpose of the first assignment is to implement sequential and OpenMP application for matrix multiplication. Students need to use an OpenMP directive for loop parallelization, which splits up loop iterations among the threads. The number of threads is adjusted to the number of cores. Both the execution time of sequential application and application implemented using OpenMP directives are measured and compared, for various matrix dimensions. Students learn to determine what variables are private/shared and also which loop in loop nest is better to parallelize to obtain better performance.

Assignment 2: The purpose of this assignmentis to implement sequential and OpenMP application for finding number of prime numbers between 2 and N. Students need to use an OpenMP directive for loop scheduling with and without reduction clause for combining partial results in threads. Again, the execution time of sequential application and application implemented using OpenMP directives are measured and compared, for different values of N. Students learn how to implement program with critical directive if it is necessary and how the use of reduce clause can affect performance gains.

Assignment 3: The purpose of this assignment is to transform a loop with loop-carried dependences into one which can be parallelized easily. The loop with loop-carried dependences is the loop where dependence exists across iterations. If loop-carried dependence exists, it can prevent safe loop parallelization. A student needs to unroll loop, discover dependences and transform code in a way that makes this loop parallelizable. Also, if there is doubly-nested loop with loop-carried dependence, a student must examine which index of possible two, guarantees parallelization. Also, the student examines how loops interchange affects performance.

After the first three assignments, students gain knowledge about the communication mechanisms, scalability and performance issues. Therefore, it is time to implement more sophisticated parallel code, execute it and get new conclusions.

Project 1: The purpose of this project is to implement sequential and OpenMP version of spatial join algorithm. Students need to implement the spatial join between a large dataset related to trajectories of moving objects and the dataset on the road network on which they move in order to perform map-matching. The result of the map-matching process is a dataset containing points at the road segments that are the closest to the appropriate trajectory points. This way moving points that represent trajectories are matched to corresponding road segments at which their movements occur. Students should execute implemented solutions on Intel Core 2 Duo T5870 2GHz CPU and Intel i7-2670QM 2,2GHz multicore architecture. Intel Core 2 Duo has two cores, while Intel Core i7 has 4 cores with implemented hyper-threading (HT) technology. After measuring the speedup and the execution time, students are required to show the results in the table for different number of threads. An example of obtained results is shown in Table 1.

Students can conclude that for Core 2 Duo the best performance results are when using two threads, while performance decreases for both algorithms when engaging more than two threads. For the Core i7 processor, hyper-threading technology enhances the speedup for applications with two to eight threads. The maximum speedup in the case of map-matching is obtained for eight threads.

Assignment 4: The purpose of this assignment is to get experience in MPI point-to-point commu-

Table 1. Performance of map-matching application obtained for different number of threads

Core2Duo			**i7**		
threads	T_P	S_P	**threads**	T_P	S_P
2	181,006	1,824	2	71,585	1,954
4	186,489	1,771	4	42,661	3,278
6	191,441	1,725	6	31,016	4,509
8	222,351	1,485	8	25,823	5,416

nication functions. Process with rank 0 (master process) need to divide the input array of p*(p+1) elements (p-number of worker processes) and send a part of array with 2*i elements to every worker process (i-process rank). Every worker process needs to find the sum of values in its part of array and print it together with its rank. Students learn how to implement *point*-to-*point* communication mechanisms using *point*-to-*point* communication routines.

Assignment 5: Process with rank 0 (master process) needs to send and receive the same message to/from all worker processes using *blocking* and *non-blocking point*-to-*point* communication mechanisms. The communication time in both cases is measured. Students need to consider solutions depending on message size that is sent/received. Students learn how to implement *blocking* and *non-blocking point*-to-*point* communication mechanisms and how using *non-blocking point*-to-*point* communication mechanism can affect performance.

Assignment 6: The purpose of this assignment is to get experience with collective communication functions and functions for creating derived types. The implemented solution needs to find minimum value in the part of square matrix A_{nxn}(n-even number) that consists of columns with even index (j=0,2,4,. . .). A matrix is initialized in a master process. Every process needs to get elements from previously mentioned columns but from corresponding n/p rows (p- number of processes, n is evenly divisible by p) and finds local minimum. Finally, master process finds and print global minimum in matrix. Sending data from master process is realized using derived data types. Computing global minimum is realized using collective communication function.

Assignment 7: The purpose of this assignment is to implement MPI program which finds and prints minimal odd number with a specified property and the rank (id) of process that contains it. Odd numbers are in the interval [*a,b*] (*a* and *b* are given constants). A number possesses the specified property if it is divisible by a given number *x*. During examination (whether the number possesses the specified property or not) each process generates and examines corresponding odd numbers as shown in Table 2. Final results need to be printed by a process that contains minimal count of numbers with specified property. The purpose of this assignment is to find a solution using only collective communication functions. To find the process that contains minimal odd number, as well as the process with minimal count of numbers with a specified property students have to use MPI_Reduce with MPI_MIN_LOC operator.

After these MPI assignments, students gain knowledge about the communication mechanisms, scalability and performance issues. Therefore, they need to implement more sophisticated code, execute it and get new conclusions.

Project 2: The purpose of this project is to implement matrix multiplication and to see how different methods of implementation can affect performance. Both matrices are square matrices of the same order *n*. Students have to implement three different methods for matrix multiplication, execute corresponding implementations for various matrix dimensions and various numbers of nodes. The execution times of these implementations are measured using MPI_Wtime function. After execution of their applications, they compare obtained results.

In the first method, matrices are divided in *k* x *k* blocks. The master process distributes blocks of *k* x *k* dimension extracted from matrices A and B to every worker process. The number of worker processes is $(n/k)^3$. Each worker process performs multiplication of corresponding blocks in parallel and sends result to the master process. The master process performs addition of received results.

In the second method, matrices are divided in *k* x *k* blocks and these blocks form the corresponding row vectors of matrix A and corresponding column vectors of matrix B. The master process distributes corresponding row vectors of matrix A and corresponding column vectors of matrix B to the worker processes. The number of worker processes is $(n/k)^2$. Every worker process performs multiplication of corresponding row vector and column vector and addition of partial results and sends obtained values to master process.

In the third method, the master process distributes the corresponding column vectors of matrix A and the corresponding row vectors of matrix B to the worker processes. The number of worker processes is *n*/*k*. Every worker process performs multiplication of corresponding row and column vectors in parallel and generates partial products. Afterwards, MPI_Reduce function performs addition of corresponding partial products in worker processes, in order to generate result elements of matrix C.

The implementation and running of proposed

Table 2. The example for assignment 7

P0	P1	P2	P3
3	5	7	9
11	13	15	17
19	21	23	25
27	29	31	

$number\_of\_processes = 4, a = 3, b = 31, x = 5 \Rightarrow \min = 5,$
$id = 1, number\_of\_numbers\_with\_specified\_property = 3,$
$id\_of\_process\_that\_pr \text{ int } s\_results = 0$

three methods is performed on the various numbers of nodes (2, 4, 6, 8) for the various matrix dimensions (from 64 to 1024, power of 2) and block size equals 64x64. After measuring the speedup and the execution time, students are required to show results in the tables for different number of nodes and matrix dimensions. The first method shows small performance gain for smaller matrix dimensions, as the number of nodes arises. But, the speedup is less than one. Also, for larger matrix dimensions, it shows performance degradation as the number of nodes arises, although speedup has values greater than 1. Students can conclude that it is consequence of a fact that there are a large number of processes that have a large communication overhead and there is not enough processing load in each process to overcome this overhead. Students can conclude that the speedup of the second and the third method, for a small matrix dimensions, although greater than one, decreases, as the number of nodes increases. For larger dimensions (>128) the second and the third method speedup increases and has values greater than one, as the number of nodes increases. The student can conclude that the third method speedup is always greater than the second method speedup, as the number of nodes increases. All of these facts exclude the first method for parallel execution of matrix multiplication on more than two nodes. At the same time, students can conclude that the second and the third implementations are useful for parallel implementation of matrix multiplication. An example of a table that is obtained for matrix dimension N=512 is shown in Table 3.

Assignment 7: The purpose of this assignment is to find a value that approximates the value of an integral using Parallaxis. The range of the integration has to be broken into a large number of strips. The number of processing elements is equal to the number of strips. At each strip, a rectangle whose height is the value of the integrand at the middle of the strip is placed. Each processing element calculates the area of corresponding rectangle as the product of function value at the middle of the strip and the width of the strip. The sum of the areas of each rectangle equals an approximation to the integral. The reduction function is used for this purpose. The students learn how to divide data among processing elements and execute the same function in order to get integrand in corresponding point. Also, they learn how to use the reduction function to sum the partal results.

Table 3. Performance of matrix-multiplication application obtained on different number of nodes

N=512						
Nodes	T_1	T_2	T_3	S_1	S_2	S_3
2	0.849	0.792	0.697	1.47	1.57	1.79
4	0.999	0.692	0.588	1.25	1.80	2.12
6	1.063	0.602	0.548	1.17	2.07	2.28
8	1.226	0.589	0.482	1.02	2.11	2.58

The last part of the course introduces GPU as massively parallel architecture consisting of many small, efficient cores designed for handling multiple threads simultaneously. The initial parts of lectures are dedicated to showing differences between CPU and GPU. CPU is latency optimized but is built with complex and power inefficient hardware. Compared with commodity CPU, GPU has an order of magnitude higher computation power, as well as memory bandwidth. It provides general parallel processing capabilities and general-purpose programming languages such as NVIDIA CUDA (Compute Unified Device Architecture) [23]. CUDA includes programming model along with hardware architecture that supports data-parallel type of implementation. CUDA C/C++ compiler, libraries, and runtime software enable programmers to access data-parallel computation model and develop and accelerate data-intensive applications. The key concepts of CUDA programming were introduced to students through very simple examples. This part of the course is not extensively covered, yet. In the future, we plan to provide more in-depth coverage by discussing concrete programming examples and compare performance of CUDA solutions with corresponding OpenMP solutions.

Parallel software design should also allow a hybrid approach that integrates different levels of parallelism, e.g. hybrid parallelization with MPI+OpenMP and MPI+CUDA. An introduction of combined programming models and integration of students' code into a more complicated problem solutions represent our plans for the future.

5. Evaluation

The main goal of this course is not only to teach students about theoretical concepts in parallel computing, but to give them significant hands-on experience through assignments and programming projects. For the theoretical part of the course, PowerPoint slides are used that are available to students at Faculty's Moodle platform. Parallel programming practices of the course include assignments and programming projects. For each of them students need to implement and demonstrate parallel program, analyze its behavior, evaluate performance and speedup and explain results. Also, students are required to answer the questions about assignments/projects. Such test is worth 20% of the assignment grade. After the group of assignments, a quiz is organized that estimates

students' progress. About 73% of all students have developed MPI programs and answer the questions successfully. About 80% of all students have developed OpenMP programs and answer the questions successfully. The final exam requires that students synthesize their knowledge about different parallel computing architectures and programming models learned.

The outcome of the course is very positive. Based on students' comments and questionnaires, most students are satisfied with the course. Our positive experience is also justified by:

- Increased interest in performing independent research work and final projects at the end of undergraduate and in the first year of graduate studies.
- Increased enrollments in elective high-performance computing courses in graduate computer science curriculum, such as High Performance Computing, Cloud Computing, BigData, etc.
- Increased interest of students to pursue bachelor and master thesis related to parallel, distributed and high/performance computing.

In the current Computer Science curriculum, Parallel computing course is included in the last semester (8th) of undergraduate studies. Today, it is difficult to ignore parallel computing in even the core of a CS undergraduate curriculum. So we believe students can consider parallel solution of problem they want to solve earlier in education, and have tried to introduce such course (or two courses) in the previous semesters (6th and 7th).

6. Conclusions

The increasing availability of parallel architectures, multi-core and many-core processors, and computer clusters demands the introduction of parallel computing/programming in the undergraduate computer science curriculum. Current students and IT professionals need to know how to develop parallel software applications on these architectures to achieve best possible efficiency, performance, and scalability. Therefore, traditional parallel computing courses require a shift toward more hands-on experience even with commodity parallel architectures.

Our current experience proves that parallel computing course can be taught successfully with a hands-on experience and a limited budget. In our course we introduced OpenMP (for programming shared memory MIMD computers). Also, we introduce concept of MIMD computers with distributed memory using MPI framework and software tools on interconnected computers of varied architecture. We introduce Parallaxis as simulator of SIMD computers. In this way, student gets hands-on experience with all models associated with parallel processing. The key concepts of CUDA programming were introduced to students through very simple examples. In future, we plan to provide more coverage of CUDA solutions, as well as introduction and coverage of hybrid approach as MPI+OpenMP and/or MPI+CUDA.

References

1. P. Pacheco, *An Introduction to Parallel Programming*, Addison Wesley, 2nd edition, 2003.
2. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams and K. A. Yelick, The landscape of parallel computing research: A view from Berkeley, Vol. 2., *Technical Report UCB/EECS-2006-183, EECS Department,* University of California at Berkeley, December 2006.
3. ACM/IEEE-CS Joint Task Force, Computer science curricula 2013, www.acm.org/education/CS2013-final-report.pdf, Dec 2013, Accessed 03 April 2015.
4. S. K. Prasad et al., NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing—Core Topics for Undergraduates, Version I, Dec 2012, http://www.cs.gsu.edu/~tcpp/curriculum/index.php, Accessed 03 April 2015.
5. A. Minaie, R. Sanati-Mehrizy, Incorporating Parallel Computing in the Undergraduate Computer Science Curriculum, *American Society for Engineering Education*, 2009
6. D. J. John and S. J. Thomas, Parallel and Distributed Computing across the Computer Science Curriculum, IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014, Phoenix, AZ, USA, 19–23 May 2014, pp. 1085–1090.
7. A. Danner and T. Newhall, Integrating Parallel and Distributed Computing Topics into an Undergraduate CS Curriculum, *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, Cambridge, MA, USA, 20–24 May 2013, pp. 1237–1243.
8. M. Arroyo, Teaching Parallel and Distributed Computing topics for the Undergraduate Computer Science Students, *IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW),* Cambridge, MA, USA, 20–24 May 2013, pp. 1297–1303.
9. N. Deo, F. Hussain, S. K. Jha and M. Vasudevan, Introducing parallel programming across the undergraduate curriculum through an interdisciplinary course on computational modeling, *IEEE Technical Committee on Parallel Programming,* Boston, MA, USA, July 2013.
10. C. M. Brown, L. Yung-Hsiang and S. Midkiff, Introducing Parallel Programming in Undergraduate Curriculum, *IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW),* Cambridge, MA, USA, 20–24 May 2013, pp.1269–1274.
11. T. R. Gross, Breadth in depth: a 1st year introduction to parallel programming, *In Proceedings of the 42nd ACM technical symposium on Computer science education*, March 2011, pp. 435–440.
12. S. Rivoire, A Breadth-First Course in Multicore and Manycore Programming, *Proceedings of the 41st ACM technical symposium on Computer science education, SIGCSE '10*, USA, March 10–13, 2010, pp. 214–218.
13. H. C de Freitas, Introducing parallel programming to traditional undergraduate courses, *Frontiers in Education Conference (FIE)*, Seattle, WA, USA, 3–6 Oct. 2012, pp. 1–6.
14. M. Fienup Parallel Computing in the Computer Science Curriculum via the Computer Architecture Course, *Midwest Instruction and Computing Symposium*, Verona, WI, USA, April 25–26, 2014.
15. C. von Praun, Parallel Programming: Design of an Overview Class, *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, San Jose, CA, USA, June 4, 2011.

16. R. Keller, Teaching parallel programming to undergrads with hands-on experience, *Workshop on Education for High-Performance Computing, EduPDHPC-13*, Nov 16, 2014.
17. P. Pacheco, Teaching Parallel Programming to Lower Division Undergraduates, *1st NSF/TCPP Workshop on Parallel and Distributed Computing Education, EduPar 2011*, Anchorage (Alaska), USA, May 16, 2011.
18. Parallaxis-III—A Structured Data-Parallel Programming Language, http://robotics.ee.uwa.edu.au/parallaxis/, Accessed 03 April 2015.
19. MPICH2—the implementation of the MPI standard, http://www.mcs.anl.gov/research/projects/mpich2/, Accessed 03 April 2015.
20. M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongara, *MPI: Complete Reference*, The MIT Press, 1998.
21. T. Rauber and G. Rünger, *Parallel Programming: For Multicore and Cluster Systems*, Springer, Berlin, Heidelberg, 2010.
22. B. Chapman, G. Jost and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*, MIT Press, 2008.
23. D. Kirk and W. M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Elsevier, 2010.

Natalija Stojanovic is an Assistant Professor at the Computer Science Department, Faculty of Electronic Engineering, University of Niš, Serbia. She received her PhD, MSc, and BSc degrees in Computer Science from the University of Niš, in 2009, 2003 and 1999, respectively. Her current research interests include high-performance parallel and distributed computing architectures and programming models, data-intensive applications and cloud computing. She successfully participates in several international and national projects in those and related domains.

Emina I. Milovanovic is a Full Professor at the Department of Computer Science, Faculty of Electronic Engineering, University of Nis, Serbia. Her research interests include parallel, systolic and distributed algorithms, FPGA designs, cluster computing, computer architectures, fault-tolerant systems and computer networks. She has published more than 120 research articles in leading international journals and/or conference proceedings, 6 textbooks, and 5 book chapters in international monographs.