# Teaching high–performance computing systems – a case study with parallel programming APIs: MPI, OpenMP and CUDA

Pawel Czarnul[0000−0002−4918−9196], Mariusz Matuszek[0000−0001−7551−256X] and Adam Krzywaniak[0000−0003−1904−2510]

Faculty of Electronics, Telecommunications and Informatics
Gdansk University of Technology, Narutowicza 11/12, 80-233 Poland
pczarnul@eti.pg.edu.pl

**Abstract.** High performance computing (HPC) education has become essential in recent years, especially that parallel computing on high performance computing systems enables modern machine learning models to grow in scale. This significant increase in the computational power of modern supercomputers relies on a large number of cores in modern CPUs and GPUs. As a consequence, parallel program development based on parallel thinking has become a necessity to fully utilize modern HPC systems' computational power. Therefore, teaching HPC has become essential in developing skills required by the industry. In this paper we share our experience of conducting a dedicated HPC course, provide a brief description of the course content, and propose a way to conduct HPC laboratory classes, in which a single task is implemented using several APIs, i.e., MPI, OpenMP, CUDA, hybrid MPI+Pthreads, and MPI+OpenMP. Based on the actual task of verifying Goldbach's conjecture for a given range of numbers, we present and analyze the performance evaluation of students' solutions and code speed-ups for MPI and OpenMP. Additionally, we evaluate students' subjective assessment of ease of use of particular APIs along with the lengths of codes, and students' performance over recent years.

**Keywords:** teaching HPC, parallel computing, HPC education, MPI, CUDA, OpenMP

## 1 Introduction

In recent years, a considerable increase in computational power has become possible primarily due to the incorporation of an increasingly larger number of cores into both CPUs and accelerators, such as GPUs, and improvements in memory sizes and bandwidth [11]. This applies to all the segments of computing devices, i.e., the data center, server, workstation, desktop, and mobile CPUs and GPUs. As a consequence, the development of parallel programs has become a necessity in order to make the most of the computational power of the computing devices within each computer/node as well as in clusters of machines. Teaching HPC has

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29

then become the issue of the utmost importance. A number of Application Programming Interfaces (APIs) exist for general purpose programming in HPC, i.e., OpenMP, Pthreads for shared memory systems and multithreaded programming for multi-core CPUs, as well as offloading to accelerators, NVIDIA CUDA and OpenCL for GPUs, OpenCL for programming shared memory CPU+GPU systems, and Message Passing Interface (MPI) for distributed memory systems [11]. Use of frameworks allows for easier programming at a higher level of abstraction but it comes at the cost of performance overheads. For example, KernelHive allows parallelization of computations among the nodes of a cluster using OpenCL as computational kernels and Java based management layer(s) supporting workflow model execution. The authors of [28] demonstrated overheads up to around 11% compared to the highly tuned MPI+OpenCL solution. Access to HPC resources can be granted either via ssh, remote shell [10], or through various middlewares, including Web interfaces [12], which are, however, often adapted over time [34]. Courses in HPC can also be delivered using container virtualization and open-source cloud technologies [2].

In this paper, we provide subjective assessment of the programming difficulty using a selected set of APIs that a population of programmers face. We also analyze the execution times and scalability of parallel codes written by them for various input data sizes and for the codes programmed with selected APIs. For this purpose, we propose a methodology that uses the same programming task, i.e., the implementation of Goldbach's conjecture for a range of input numbers, which is the material for a separate student laboratory, and focuses on various programming APIs such as: OpenMP, MPI, CUDA, and selected hybrid combinations. Our research results allowed us to conclude how the performance of codes, written by various programmers to solve a particular task using a given technology, differs. Additionally, we provide information on the length of codes for the analyzed APIs as well as track our students' performance over recent years.

The outline of the paper is as follows. In Section 2 we discuss the related works, while in Section 3 we present the motivation for our work. Section 4 contains a detailed description of the High Performance Computing Systems course from which we gathered the data analyzed in this paper. In Section 5 we provide details of our evaluation of the programmers and codes, including the proposed methodology, students' subjective evaluation data, practical evaluation of the students' codes, using objective metrics, and add subsequent discussion. Finally, Section 6 contains the study conclusions and the scope identified for further research.

## 2 Related work

*Importance of HPC education* HPC education has become essential in recent years, especially that parallel computing on HPC systems enables modern machine learning models to grow in scale [9]. Accelerating deep learning is the key for future large models development [4]. However, as the authors of [27] claim,

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29

computing educators are only beginning to recognize the need for HPC education. The industry demand for talented sofware developers with experience in the HPC area has raised a concern if HPC skills should not be introduced earlier in the university education, i.e., for undergraduates [17], or even for secondary school students [6]. In this context, teaching HPC has become essential in developing relevant skills required by the industry, and already over a decade ago HPC was recommended to be taught as part of Computer Science Engineering education [29] at all the university degree levels.

*Teaching HPC academic standards* The most common technologies used in teaching HPC are C and C++, with significantly less frequent practice of JAVA, Fortran or Python [7]. The most commonly used software APIs for HPC education is OpenMP, MPI, and CUDA. As confirmed by the most recent review [32], C++ along with MPI have become primary tools in teaching computational science, specifically when demonstrating parallel programming. In this context, we want to emphasize that our experience and conclusions from dealing with the method of teaching HPC presented in this paper are based on the technologies that are the industry and academic standard, i.e., C++, MPI, OpenMP, and CUDA.

As stated in [19], based on the review of 94 papers related to teaching HPC, the predominant method is still traditional learning, whereas projects (problem based learning), gamification, and collaborative learning are employed in far fewer cases. Also, the teaching objective is almost always programming, while the architecture or parallel thinking focus are rare. In this paper, we present the experience from conducting a course which combines project based teaching with more focus on architecture and parallel thinking, specifically emphasizing the differences between three parallel APIs and CPU vs GPU architecture.

*Evolution of HPC courses* In the past two decades, teaching HPC has evolved from Parallel Programming courses, where MPI and OpenMP, among others, were used as exemplary APIs for writing multithread and multi–process programs [25]. Also, Parallel Distributed Computing has gained more attention and various combinations of parallel APIs have been used to demonstrate the capabilities of diverse platforms and architectures, as reported by [30], i.e., using not only MPI and OpenMP for HPC on CPU clusters, but also employing GPGPU and APIs, such as CUDA, or presenting distributed processing of large data sets with Hadoop. Although the majority of courses use C and C++ as primary programming languages, natively supported by OpenMP, CUDA or MPI, some of them present a different approach with OpenMP-like directives used with Java and Pyjama compiler [20].

Early courses in HPC encountered different barriers such as lack of access to typical HPC clusters. The problems were addressed with using small educational clusters such as Mozart, which unblocked teaching HPC [5]. In developing countries, e.g. in Mexico, as reported by [33], even nowadays one of the barriers, besides deficiencies in HPC educational infrastructure, is still lack of interest from students in learning HPC.

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29

On the other hand, many HPC courses with the well established scope tend to explore innovative approaches to HPC education, such as focus on gaining cluster configuration skills instead of just introducing HPC APIs, proposed by [22], or emphasizing parallel distributed computing patterns, proposed by [1]. The authors of [14] explore how using a higher level of abstraction, i.e., the Thrust framework for parallel GPU programing, enhances programmer productivity. Our HPC course proposed for postgraduates [13] introduces such concepts as: blocking and non-blocking communication, overlapping of compute and communication, and dynamic load balancing. The method described in this paper is the extension of the previous studies, focusing on the practical incorporation of various parallel APIs (OpenMP, MPI, CUDA, hybrid MPI+OpenMP), into the solution for a well defined task and running its various implementations in several system types (multicore CPU, cluster of CPUs, and GPU).

*Methods of teaching HPC* Following the challenges arising with the evolution of HPC education, novel methods enhancing HPC courses are being proposed. At the early stages of education, teaching Computational Thinking was proposed by [21] for elementary and middle school students. For undergraduates, the authors of [24] propose teaching inexperienced programmers with a set of analogies helpful in understanding the basic concept of parallelism. On the other hand, the authors of [23] propose a challenge–based approach inspired by Parallel Programming Marathons in Brazil.

The authors of [8] identified three major issues in HPC education and addressed them by creating their own HPC curriculum system by empowering students' innovation capability with a project–based Parallel Programming course, involving real–life examples from the HPC field, such as Computational Fluid Dynamics, or 3-D simulations of human ventricular tissues. Lack of suitable HPC environments dedicated to teaching HPC skills is often addressed by proposing small–scale clusters. For instance, [26] proposed a 25–node cluster based on Raspberry Pi nodes, while [3] presented their small–scale cluster based on Odroid boards.

## 3 Motivations

Considering the evolution of HPC courses in the past two decades and the shift of focus in HPC education towards more parallel thinking and project–based classes, we were motivated to propose a new method for the laboratory part of our original HPC course [13]. It assumed the implementation of the same programming task using several representative parallel programming APIs. While we did not aim at performing an inter-API or inter-software stack performance comparison, the method allowed us to provide meaningful assessment of the programmers' relative performance within each API. Additionally, the obtained results provided information on the most important general purpose parallel programming APIs because these cover the most important types of computer systems and compute devices, i.e., shared-memory single node systems with

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29

multi-core CPUs (programmed with OpenMP) and GPUs (programmed with CUDA), and distributed memory multi-node systems, programmed with: MPI, MPI+OpenMP, and MPI+Pthreads. For this purpose, we incorporated both theoretical knowledge on parallelization concepts, paradigms and APIs as well as practical tasks using particular APIs into our High Performance Computing Systems (HPCS) course, offered to all the computer science first semester MSc level students. For the practical part, the same programming task – verification of Goldbach's conjecture – was given to students for implementation using the aforementioned APIs.

## 4 High Performance Computing Systems course

### 4.1 Aims, structure and scope

The goal of the course is to acquire the neccessary knowledge and skills that allow for designing, implementation, deployment as well as testing parallel implementations, and solving a given computational problem. The primary objective for the student is to be able to develop scalable parallel programs that would obtain high speed-ups, taking advantage of modern parallel environments. The latter include both shared and distributed memory systems (clusters) with multi-core CPUs and GPUs. Assessment of students' work involves taking and passing a theoretical exam (with a score of at least 50%) as well as obtaining at least 50% of the points on lab exercises. Both the theoretical and the practical parts contribute to the final grade equally.

The teaching approach involves getting to know parallel processing paradigms representative of computationally demanding applications and, subsequently, discussing specific techniques allowing for their efficient parallelization. The following paradigms are considered [11]:

1. Dynamic master-slave – the master process/thread initially partitions the input data into chunks which are distributed among the slaves dynamically.
2. Geometric Single Program (Stream) Multiple Data (Stream) – the computations are performed in iterations over a domain which is partitioned geometrically among the processes/threads for parallel updates of their subdomains, followed by the synchronization/communication (boundary data exchange). This paradigm typically refers to simulations of physical phenomena.
3. Pipelining – data chunks of the input stream are passed through a pipeline and parallel processing of particular chunks is performed.
4. Divide and conquer – the original problem is partitioned into subproblems recursively down to the leaves of either a balanced or an imbalanced tree. Subtrees are processed in parallel by processes/threads with synchronization corresponding to data merging and subsequent passing up towards the root of the tree.

After the students have learnt these concepts and techniques, mapping relevant solutions onto specific APIs and environments is discussed during this course.

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29

The mapping is done mainly with distributed memory systems in mind. For this purpose, MPI is used. Additionally, the HPCS course features the basics of shared memory programming, including multithreaded programming for multi-core CPUs – using OpenMP, and for GPUs – using NVIDIA CUDA.

### 4.2 Lecture

The lecture includes the following components:

1. Introduction to computing in parallel environments: model of a parallel application: basic parameters of an application and a parallel system, topologies; assignment criteria; latency and bandwidth; specifications of real systems.
2. Parallel architectures: Shared Memory, Distributed Shared Memory, Distributed Memory.
3. Parallel processing paradigms: master-slave, SPMD, pipelining, divide-and-conquer; examples.
4. MPI: model of an application; execution, various implementations, queueing systems; send communication modes; non-blocking communication; blocking communication; collective vs. point-to-point communication, creation of data types and packing; communicators; dynamic load balancing: repartitioning, "ghost nodes"; examples of applications in MPI – performance on real clusters; spawning processes dynamically, one-way communication; checkpointing of parallel aplications; MPI+multi-threading; Parallel I/O in MPI; mapping paradigms to MPI.
5. OpenMP – API: directives and functions; synchronization constructs; access to shared resources; minimization of synchronization and overheads; using OpenMP for accelerators; examples; mapping paradigms to OpenMP.
6. CUDA – model of an application, grid parameters, performance vs. usage of resources; optimized memory access patterns; using GPU shared memory; examples; mapping paradigms to CUDA.

### 4.3 Laboratory tasks

Typically, there are 6 laboratory classes (90-minute each) during which students are given tasks that usually require modifications of and building on the provided sample code and the corresponding manual. Each student is expected to complete the task either during a given lab session or during the following meetings (penalties apply in case of delay). We offer the following manuals along with the corresponding exemplary codes:

1. **MPI Embarrassingly parallel computations** with parallel reduction of results by the designated process. *Example:* parallel computations of a series such as for computing the pi number. Even though partitioning of the series among the processes for load balancing of computations is straightfoward, it could still be done in various ways, which might potentially affect the solution accuracy, depending on what elements are added in the process (for instance, adding numbers with similar absolute values in each process vs. adding both large and small numbers).

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29

2. **MPI dynamic master-slave example**. The master partitions the input data into chunks which are then distributed among slave processes. The latter return results to the master, waiting for subsequent data chunks. This scheme, assuming that the ratio of time spent on computations to communication is high, and the number of data chunks is considerably larger than the number of slave processes, allows for effective load balancing and high speed-ups. *Example:* parallel numerical integration of a function on a given range.
3. **CUDA basics** of parallel programming based on programming operations on vectors and matrices such as weighted addition of vectors, multiplication of a matrix and a vector, multiplication of a sparse matrix by a vector, dot product. Gains from using CUDA shared memory are demonstrated.
4. **OpenMP dynamic master-slave** code in which threads fetch the available packet identifiers in a critical section, fetch the packet afterwards, and then process these in parallel. The supplied manual further discusses and demonstrates that the critical section may become a bottleneck when the number of threads is very large, and proposes to use named critical sections to improve the performance for smaller thread groups. *Example:* parallel numerical integration – the initial range is partitioned into a number of data packets at least a few times (4 to 5+) larger than the number of threads.
5. **MPI dynamic master-slave** example with overlapping communication and computations using non-blocking MPI communication routines. This code extends the previous example with initial sending of a data packet per slave, followed by non-blocking sends of another data packet per slave. This facilitates overlapping computations in a slave by sending a new data packet from the master, and receiving the previous result packet from the slave. For applications using larger data packets this shall lead to the minimization of the total application execution time [11]. MPI_Isend, MPI_Irecv and MPI_Wait(all) routines are used.
6. **MPI+Pthreads**. Extension of Task 1 with multithreading. Multithreading modes in MPI are discussed and Pthreads are spawned in each process for multithreaded processing on a multi-core CPU or CPUs. Inter-thread communication using MPI_THREAD_MULTIPLE is discussed vs. optimized reduction within the processes and using MPI_Reduce. *Example:* parallel computations of a series, e.g., computing the pi number.

## 5 Evaluation

### 5.1 Methodology

Within this work, we aim at the following targets:

1. Individual assessment of technologies by students (difficulty, usefulness).
2. Assessment of performance in a programmers' population sample by measuring the differences in times as well as speed-ups. The differences can stem from either a better selection of the algorithm and/or its parallel design and/or implementation.

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29

3. Measuring the lengths of codes using respective APIs.

For these purposes, we collected both subjective opinions of our students (survey) and objective measures, i.e., execution times of the codes for various technologies and for various input data sizes. We shall note that for the latter we used an isolated environment, the same that was used by the students to develop and test the codes. What is important, the same application was implemented by the students using various parallel programming APIs.

### 5.2 Computational task

Goldbach's conjecture states that any positive even integer number can be written as a sum of two prime numbers [15]. Goldbach's conjecture can benefit significantly from the initial generation of the sieve of Eratosthenes, which is used in the parallel verification of the conjecture. The best strategy for verification of the conjecture for an $[a, b]$ range might depend on the $a$ and $b$ values, as well as $b - a$. Several strategies and parallel implementations were investigated by the authors of [31], including the ones using OpenMP, MPI hybrid MPI-OpenMP parallelism, as well as OpenACC GPU accelerated computing. The evaluation of CUDA's Unified Memory (UM) and dynamic parallelism (DP) for Goldbach's conjecture was verified in [18], assuming that the verification is performed for a vector of input numbers. The second mechanism, tested for numbers in the order of $10^{10}$, resulted in performance loss compared to the standard, non-DP version. On the other hand, verification of Goldbach's conjecture for 10,000 numbers from $10^8$ to $10^{12}$ using Unified Memory resulted in practically the same performance as in the standard non-UM version. In this case, this is a positive result since UM-based implementation is typically more straightforward and shorter.

In this paper, we use Goldbach's conjecture to evaluate students' approaches to parallel implementation and assess the group's relative performance. We asked students to carry out 5 laboratory tasks, each meant to implement the verification of Goldbach's conjecture, using the following 3 technologies: MPI, OpenMP, CUDA, and their 2 combinations: MPI+OpenMP, MPI+Pthreads. Each program was expected to verify the conjecture for all the numbers in the given input $[a, b]$ range.

### 5.3 Students' subjective evaluation of parallel programming APIs

Figure 1 presents students' subjective evaluation of how easy it was to program with a given parallel programming API (higher grade means easier). It can be seen that OpenMP, allowing for a relatively straightfoward extension of sequential codes with directives and library calls, is clearly perceived as the easiest to use, followed by CUDA and MPI with practically the same median values.

### 5.4 Practical evaluation of students' codes using objective metrics

Given the limited laboratory time, students devised various implementations that consequently resulted in various execution times, as presented next. Figures 2, 3 and 4 present boxplot type charts for execution times for the codes

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29

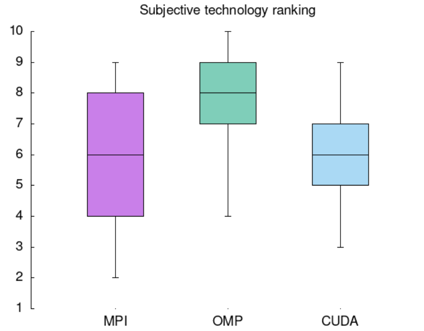


Fig. 1: Students' subjective evaluation of programming easiness using a particular programming API

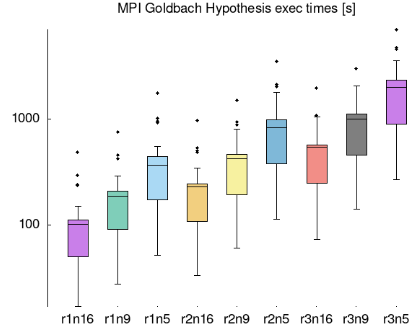


Fig. 2: Execution times for MPI codes

that were not based on the Eratosthenes sieve. This approach was adopted by the majority of students for the codes implemented using a single technology: MPI, CUDA, and OpenMP respectively.

The testbed environment consisted of 16 workstations, each equipped with an Intel Core i7-7700 CPU with four physical cores and Hyperthreading, 16GB of RAM and interconnected by a 1GBit Ethernet network. In the following charts, r$x$, i.e., r1, r2, and r3, denotes three different input data ranges of numbers to be verified against the Goldbach conjecture, respectively. They are: r1=[50.000.000:100.000.000], r2=[50.000.000:150.000.000], and r3=[50.000.000: 250.000.000].

Additionally, n$x$ in the charts indicates $x$ number of nodes used in the MPI environment (with a single MPI process slot on each node) while t$x$ indicates $x$ number of threads used in the OpenMP version of the code.

After running the students' codes in our environment, we present the execution times for the analyzed parallel programming APIs and the ranges in Figure 2 (MPI), Figure 3 (CUDA), and Figure 4 (OpenMP). Figures 5 and 6 present the results for hybrid technologies, i.e., MPI+Pthreads and MPI+OpenMP respectively, for configurations employing 8 threads per CPU and 16 MPI processes, across all the input data ranges.

It should be noted that the objective of these tasks was to develop parallelized and scalable (versus the number of processes or/and threads) code using a given technology, rather than performing a direct comparison of the technologies performance.

Apart from the aforementioned results without the sieve, Table 1 summarizes the results that utilized the sieve-based approach. It includes the proportions of students utilizing a given technology who developed a sieve-based solution, and the best times for the sieve-based approaches. Efficient sieve-based solutions were proposed by the students for two technologies: OpenMP and MPI.

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29

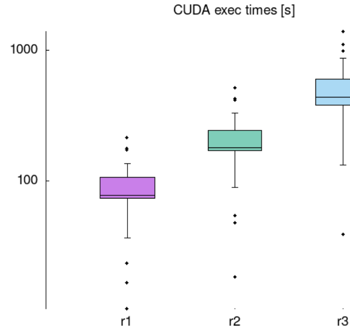


Fig. 3: Execution times for CUDA codes

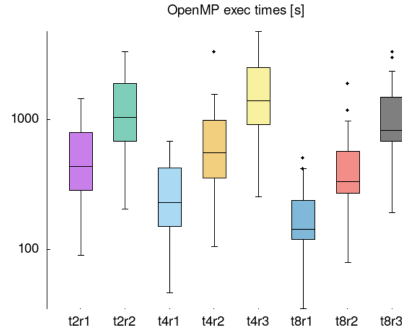


Fig. 4: Execution times for OpenMP codes

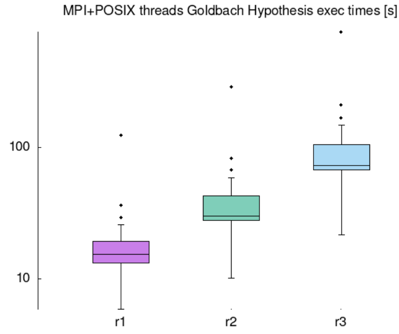


Fig. 5: Execution times for MPI+Pthreads codes

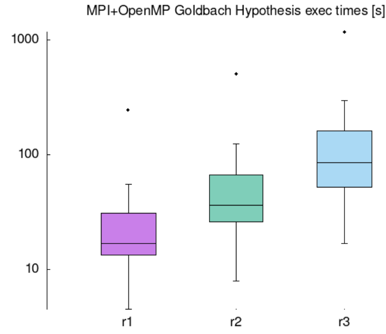


Fig. 6: Execution times for MPI+OpenMP codes

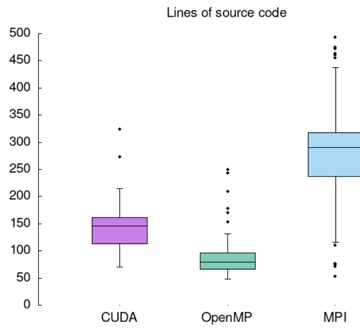


Fig. 7: Numbers of lines for the analyzed codes

| **Version** | **OpenMP** | **MPI** |
|---|---|---|
| **Percentage of sieve-based codes [%]** | 9.6% | 6.9% |
| **r1 best [s]** | 1.44 | 3.98 |
| **r2 best [s]** | 2.32 | 5.18 |
| **r3 best [s]** | 4.13 | 6.74 |

Table 1: Sieve based codes

Additionally, we assessed the speed-ups of the students' codes. Table 2 presents the average speed-ups of MPI codes between 4 and 8, as well as between 4 and 15

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29

workers/slaves. Table 3 shows the speed-ups between 2 and 4, as well as between 4 and 8 threads using OpenMP.

Table 2: Speed-ups of MPI codes

| | r1 | r2 | r3 | optimal |
|---|---|---|---|---|
| p4-p8 | 1.95 | 1.99 | 2.00 | 2.00 |
| p4-p15 | 3.47 | 3.61 | 3.72 | 3.75 |

Table 3: Speed-ups of OpenMP codes

| | r1 | r2 | r3 |
|---|---|---|---|
| t2-t4 | 1.90 | 1.88 | NA |
| t4-t8 | 1.41 | 1.43 | 1.43 |

Figure 7 depicts boxplot graphs of lines of codes for the implementations using 3 major technologies: CUDA, OpenMP, and MPI.

Furthermore, Figures 8 and 9 present the grading results from the theoretical and laboratory course components of those students who passed the course, over the period of the last 7 years (0.5 is the passing threshold, 1 is the maximum).

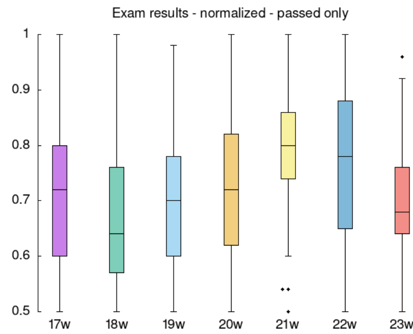


Fig. 8: Evaluation of theoretical component in successive years

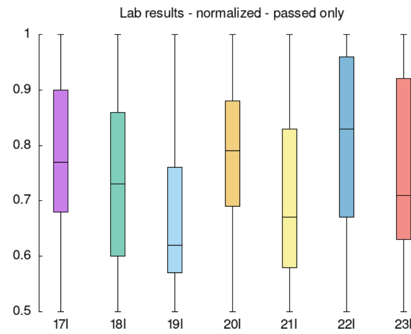


Fig. 9: Evaluation of laboratory component in successive years

### 5.5 Discussion

We observed a systematic decrease in the population sample size during the experiment progress, i.e., at the beginning of the course all the students attempted to solve the assignments (MPI and OpenMP). As soon as they secured the minimum required to pass the course, we observed a significant drop in the submitted assignments (CUDA, MPI+OpenMP, MPI+Pthreads). We shall note that during the initial verification of solutions handed in by the students during the laboratory classes, the test ranges were smaller than these studied in the paper. Consequently, some of the solutions resulted in timeouts in this analysis.

In terms of execution times for a particular API and the range size, we see observe that the performance of students' codes can differ by an order of magnitude practically for all the APIs.

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29

From Table 1 we can conclude that for MPI and OpenMP there is a small number of much faster codes that benefit from implementations using the Eratosthenes sieve and other optimizations. We shall note that time could play an important role in the submission of assignments, as students are generally expected to deliver their solutions by the end of a 90-minute laboratory session, on which the assignment was given. Later submissions resulted in penalties. Consequently, students apparently leaned to simpler solutions first, in order to secure passing a given assignment, and attempted the optimized versions when time permitted. We conclude that the proportions listed in Table 1 reflect the work of the most ambitious students. Additionally, we see that in the case of CUDA and hybrid codes students were not able to submit faster sieve-based codes which might indicate that these technologies appeared more challenging to develop optimized solutions, given the time limit.

In terms of speed-ups shown in Table 2, we can conclude that, since the presented speed-ups are the average values that are so close to the optimal ones (assuming the ideal scaling), using a provided well scaling template (for the master-slave MPI exercise) results in very good speed-ups, especially since the Goldbach conjecture code features a large compute/communication time ratio. Still, we can observe the anticipated growth in speed-ups with the increasing problem size. In the case of speed-ups obtained for the OpenMP codes shown in Table 3, we should realize that they were run on an Intel i7-7700 CPU with 4 physical cores and HyperThreading (HT) for a total of 8 logical processors. Thus, we can see that between 2 and 4 threads the speed-ups are very good – 1.88 and 1.9 (for r3 and 2 threads we did not run the code due to its long execution times). Using the HT technology resulted in approx. 41-43% gain over a physical core performance, which is in line with other reference results [11].

Boxplots presenting the line counts of programmers' codes using OpenMP, CUDA, and MPI, shown in Figure 7, reveal that, OpenMP codes were generally the shortest, followed by those in CUDA, and then in MPI, the latter being considerably longer. For the MPI codes, many students used the provided master-slave template that could be filled in with master and slave content specific to Goldbach's conjecture implementation.

From Figures 8 and 9 we can see that both the exam and laboratory results were relatively stable over the period of 7 years. However, some differences might be indicated. We can observe that for the 2021 COVID edition, the exam median was noticeably higher and the interquartile range was narrower. Additionally, only for that year the outliers were close to the 50% passing threshold, which denotes that the vast majority of the results was higher than 60%. We attribute this observation to the remote/online mode of exam taking at that time. Interestingly, students achieved relatively lower scores for the laboratory assignments then.

Furthermore, for each technology, we measured the correlation between students' subjective view of ease of use of a particular technology, and their code execution times for that technology. We found out that there is no meaningful

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29

correlation between the two, as the correlation coefficients obtained for MPI, OpenMP, and CUDA were: -0.055, 0.152, and 0.066 respectively.

## 6 Conclusion and future work

In the paper, we presented details of the High Performance Computing Systems course that has been conducted at Gdańsk University of Technology for several years already. Based on a specific laboratory exercise – verification of Goldbach's conjecture for a range of numbers, we analyzed the performance of students' parallel codes solving that problem, implemented using several parallel programming APIs, including: MPI, OpenMP, CUDA, MPI+Pthreads, and MPI+OpenMP. We not only analyzed the execution times of a group of programmers with medians, boxplots and outliers giving relative spread of students' performance, but also studied speed-ups for MPI and OpenMP for three input range sizes. We distinguished sieve and non-sieve based solutions. We found out that the students subjectively evaluate the ease of programming as greater for OpenMP, compared to MPI and CUDA. We also provided medians and boxplots of the codes' line counts, and of students' performance in the theoretical and laboratory components of the course over the past 7 years.

Based on our findings, we concluded that using the proposed course format with a single task problem allowed students to focus on the differences between APIs and technologies used, rather than spend a considerable amount of time getting to know the details of specific problems. In terms of evaluation, we confirmed that using the same algorithmic approach, the performance of students' codes can differ by an order of magnitude, which is consistent with other findings on programmers' performance [16].

In the future, we plan to repeat the experiments with other exercises as well as look into the possible optimizations such as overlapping communication and computations for codes with considerable communication and synchronization times.

## References

1. Adams, J., Brown, R., Shoop, E.: Patterns and exemplars: Compelling strategies for teaching parallel and distributed computing to cs undergraduates. In: 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum. pp. 1244–1251 (2013). https://doi.org/10.1109/IPDPSW.2013.275
2. Al-Jody, T., Aagela, H., Holmes, V.: Inspiring the next generation of hpc engineers with reconfigurable, multi-tenant resources for teaching and research. Sustainability **13**(21) (2021). https://doi.org/10.3390/su132111782
3. Alvarez, L., Ayguade, E., Mantovani, F.: Teaching hpc systems and parallel programming with small-scale clusters. In: 2018 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC). pp. 1–10 (2018). https://doi.org/10.1109/EduHPC.2018.00004
4. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. ACM Comput. Surv. **52**(4) (aug 2019). https://doi.org/10.1145/3320060

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29

5. Bernreuther, M., Brenk, M., Bungartz, H.J., Mundani, R.P., Muntean, I.L.: Teaching high-performance computing on a high-performance cluster. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) Computational Science – ICCS 2005. pp. 1–9. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
6. Brođanac, P., Novak, J., Boljat, I.: Has the time come to teach parallel programming to secondary school students? Heliyon **8**(1), e08662 (2022). https://doi.org/https://doi.org/10.1016/j.heliyon.2021.e08662
7. Carneiro Neto, J.A., Alves Neto, A.J., Moreno, E.D.: A systematic review on teaching parallel programming. In: Proceedings of the 11th Euro American Conference on Telematics and Information Systems. EATIS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3544538.3544659
8. Chen, J., Impagliazzo, J., Shen, L.: High-performance computing and engineering educational development and practice. In: 2020 IEEE Frontiers in Education Conference (FIE). pp. 1–8 (2020). https://doi.org/10.1109/FIE44824.2020.9274100
9. Coates, A., Huval, B., Wang, T., Wu, D.J., Catanzaro, B.C., Ng, A.Y.: Deep learning with COTS HPC systems. In: International Conference on Machine Learning (ICML) (2013)
10. Czarnul, P.: Integration of compute-intensive tasks into scientific workflows in beesycluster. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) Computational Science – ICCS 2006. pp. 944–947. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
11. Czarnul, P.: Parallel Programming for Modern High Performance Computing Systems. CRC Press, Taylor & Francis (2018), iSBN 9781138305953
12. Czarnul, P.: Teaching high performance computing using beesycluster and relevant usage statistics*. Procedia Computer Science **29**, 1458–1467 (2014). https://doi.org/https://doi.org/10.1016/j.procs.2014.05.132, iCCS
13. Czarnul, P., Matuszek, M.: Use of ict infrastructure for teaching hpc. In: 2019 IEEE 14th International Conference on Computer Sciences and Information Technologies (CSIT). vol. 1, pp. xvii–xxi (2019). https://doi.org/10.1109/STC-CSIT.2019.8929841
14. Daleiden, P., Stefik, A., Uesbeck, P.M.: Gpu programming productivity in different abstraction paradigms: A randomized controlled trial comparing cuda and thrust. ACM Trans. Comput. Educ. **20**(4) (oct 2020). https://doi.org/10.1145/3418301
15. Guy, R.: Unsolved problems in number theory. Springer Science & Business Media (2013)
16. Guzdial, M.: Is there a 10x gap between best and average programmers? and how did it get there? (November 2014), communications of the ACM, https://cacm.acm.org/blogs/blog-cacm/180512-is-there-a-10x-gap-between-best-and-average-programmers-and-how-did-it-get-there/fulltext
17. Holmes, V., Kureshi, I.: Developing high performance computing resources for teaching cluster and grid computing courses. Procedia Computer Science **51**, 1714–1723 (2015). https://doi.org/https://doi.org/10.1016/j.procs.2015.05.310, international Conference On Computational Science, ICCS 2015
18. Jarzabek, L., Czarnul, P.: Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications. J. Supercomput. **73**(12), 5378–5401 (2017). https://doi.org/10.1007/s11227-017-2091-x, https://doi.org/10.1007/s11227-017-2091-x
19. de Jesus Oliveira Duraes, T., Sergio Lopes de Souza, P., Martins, G., Jose Conte, D., Garcia Bachiega, N., Mazzini Bruschi, S.: Research on parallel computing teaching: state of the art and future directions. In: 2020 IEEE Frontiers in Education Conference (FIE). pp. 1–9 (2020). https://doi.org/10.1109/FIE44824.2020.9273914


ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29

20. Kurniawati, R.: Teaching parallel programming with java and pyjama. In: Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2. p. 1109. SIGCSE 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3478432.3499115
21. Lamprou, A., Repenning, A.: Teaching how to teach computational thinking. In: ITiCSE. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3197091.3197120
22. López, P., Baydal, E.: Teaching high-performance service in a cluster computing course. Journal of Parallel and Distributed Computing **117**, 138–147 (2018). https://doi.org/https://doi.org/10.1016/j.jpdc.2018.02.027
23. Marzulo, L., Bianchini, C., Santiago, L., Ferreira, V., Goldstein, B., França, F.: Teaching high performance computing through parallel programming marathons. In: IPDPSW. pp. 296–303 (2019). https://doi.org/10.1109/IPDPSW.2019.00058
24. Neeman, H., Lee, L., Mullen, J., Newman, G.: Analogies for teaching parallel computing to inexperienced programmers. SIGCSE Bull. **38**(4), 64–67 (jun 2006). https://doi.org/10.1145/1189136.1189172
25. Pan, Y.: Teaching parallel programming using both high-level and low-level languages (2001)
26. Pfalzgraf, A.M., Driscoll, J.A.: A low-cost computer cluster for high-performance computing education. In: IEEE International Conference on Electro/Information Technology. pp. 362–366 (2014). https://doi.org/10.1109/EIT.2014.6871791
27. Raj, R.K., Romanowski, C.J., Impagliazzo, J., Aly, S.G., Becker, B.A., Chen, J., Ghafoor, S., Giacaman, N., Gordon, S.I., Izu, C., Rahimi, S., Robson, M.P., Thota, N.: High performance computing education: Current challenges and future directions. In: ITiCSE-WGR '20. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3437800.3439203
28. Rościszewski, P., Czarnul, P., Lewandowski, R., Schally-Kacprzak, M.: Kernelhive: a new workflow-based framework for multilevel high performance computing using clusters and workstations with cpus and gpus. Concurrency and Computation: Practice and Experience **28**(9), 2586–2607 (2016). https://doi.org/https://doi.org/10.1002/cpe.3719
29. Rüde, U., Willcox, K., McInnes, L.C., Sterck, H.D.: Research and education in computational science and engineering. SIAM Review **60**(3), 707–754 (2018). https://doi.org/10.1137/16M1096840
30. Shamsi, J.A., Durrani, N.M., Kafi, N.: Novelties in teaching high performance computing. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop. pp. 772–778 (2015). https://doi.org/10.1109/IPDPSW.2015.88
31. Shaw, A., Varon, D.: Numerical verification of goldbach's conjecture with parallel computing (2018), harvard. CS205: Extreme Scale Data and Computational Science. https://github.com/djvaron/Goldbach
32. Sitsylitsyn, Y.: A systematic review of the literature on methods and technologies for teaching parallel and distributed computing in universities. Ukrainian Journal of Educational Studies and Information Technology **11**(2), 111–121 (Jun 2023). https://doi.org/10.32919/uesit.2023.02.04
33. Trejo-Sánchez, J.A., Hernández-López, F.J., Uh Zapata, M.A., López-Martínez, J.L., Fajardo-Delgado, D., Ramírez-Pacheco, J.C.: Teaching high-performance computing in developing countries: A case study in mexican universities. In: IPDPSW. pp. 338–345 (2022). https://doi.org/10.1109/IPDPSW55747.2022.00066
34. Xu, Z., Chi, X., Xiao, N.: High-performance computing environment: a review of twenty years of experiments in China. National Science Review **3**(1), 36–48 (01 2016). https://doi.org/10.1093/nsr/nww001, https://doi.org/10.1093/nsr/nww001


ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63783-4_29