# Parallel Programming and Languages

Moderator: Henry Dietz[0000-0002-5878-881X]

Panelists: Frithjof Gressman, Mark Marron, Rudolf Eigenmann

University of Kentucky, Lexington KY 40506, USA
hankd@engr.uky.edu

**Abstract.** Thirty-six years ago, when this workshop series was born, "languages" was given top billing in the name: Languages and Compilers for Parallel Computing. Within this speedup-oriented parallel-processing community, there was a strong belief that parallel programming was more difficult than it should be, and that new parallel programming languages would play a leading role in making parallel programming a skill that all programmers would be able to claim. However, that is not how things have progressed. Parallel programming languages have not become dominant. Not only is parallel programming rarely taught in introductory programming courses, but it is common that the only parallel programming language undergraduate students are required to write programs in is Verilog, which is intended to be a hardware description and simulation language. Arguably, parallel programming has become even more difficult as computer architectures have evolved into heterogeneous structures and programming has often degenerated into using languages like Python to create larger applications by "duct taping" existing "black box" applications together. This panel examines how programming languages and libraries could deliver on the promise of making efficient parallel programs easy to construct and debug.

**Keywords:** Parallel Programming, Programming Language, Libraries, Software Tools, Software Engineering, Parallel Architecture.

## 1 Introduction

This panel is bringing together … to discuss how programming languages, and more broadly libraries and software tools, can qualitatively simplify parallel programming.

The panel discussion at LCPC (and this paper) begins with a brief introduction by the moderator followed by ten-minute position presentations from each of the panelists. Each panelist independently determines the content of their position presentation and submits a section for this paper after the workshop. To ensure collection of opinions on some specific topics, the moderator prepared and distributed this "Introduction" section of the paper before the workshop, giving three prompts to be addressed by each panelist. The conclusion is written by the moderator after the workshop, summarizing the discussion that followed the position presentations.

## 1.1 Getting Started in Parallel Programming

There is a very old joke in this field that there is something wrong when it takes a PhD to write an optimized matrix multiply routine for a particular parallel computer and you can earn a PhD for doing just that. Four decades later, that joke still is not funny, but remains disturbingly close to reality. Computing Curricula 2020 (CC2020) from ACM and the IEEE computer society[1] lists "Parallel and Distributed Computing" as area 3.4 within "Systems Architecture and Infrastructure." Thus, parallel programming is a skill that graduates of an undergraduate program in either Computer Engineering or Computer Science are theoretically expected to have. Unfortunately, it is clear that very few students are comfortable with parallel programming by the time they graduate.

Thus, the first prompt given to the panelists is:

> ***What should parallel programming languages or libraries do differently in order to make parallel programming accessible to all programmers? Put another way, how can the core skills of parallel programming be simplified enough to be presented to students in an introductory programming class?***

## 1.2 Dealing with Parallel Heterogeneity

Every computer programmer has heard of Moore's Law[2], but many fail to appreciate all that it means. Fundamentally, the exponential growth over time in the number of circuit elements that can be cost-effectively placed on a chip is a key reason that speedup tends to come from parallel processing. However, the fact that power consumption per unit circuitry has not been decreasing as quickly has led to the concepts of so-called "dark silicon": the idea that building somewhat specialized hardware that is only powered and used when appropriate can give significant improvements in performance. The low cost in adding specialized attached processors to a system is why most computers now have GPUs… and might be why future systems have attached quantum computing systems.

However, most parallel programming languages have been designed to only target a specific flavor of parallel architecture. For example, CUDA primarily targets GPUs, and especially those marketed by NVIDA. The result has been that programming to use all the heterogeneous types of parallelism supported by the hardware often means awkwardly interfacing multiple different programming environments.

Thus, the second prompt given to the panelists is:

> ***Not only supercomputers, but even cell phones, now have heterogeneous parallel architectures including multi-core processors and GPUs. How should programming languages and libraries deal with heterogeneous parallel target architectures?***

### 1.3 Duct Tape Holding Black Boxes Together

As a computer engineering systems researcher, it used to be that there were multitudes of potential collaborators always lined up to leverage our new systems technologies to speed up their applications. However, as core application codes became stable, and then in many cases became commercial and proprietary products, it became increasing common that developing a new application was mostly about how to control execution and feeding of data between “black box” programs available only as executable code images. Similarly, going back to the matrix multiply routine mentioned above, the truth is most programmers will not write their own code for such things, but simply call a library routine which is often vendor-provided proprietary code. In sum, most programs are not written from scratch any more, but are created bottom-up from existing code.

Thus, the third and final prompt given to the panelists is:

> ***Programming now rarely means writing a program from scratch, and often involves controlling execution of already-compiled programs as modules within a new program. Are there language, library, or software tool mechanisms that can help manage the parallelism in programs constructed from “black box” components, making these types of programs easier to write and maintain?***

The following sections are provided by each of the panelists to summarize their responses to the above prompts and to generally state their position with respect to the future of parallel programming and languages.

## 2 Panelist: ...

Position statement contributed by each panelist...

## 5 Conclusion

To be written by the moderator after the panel.

## References

1 CC2020 Task Force, Computing Curricula 2020: Paradigms for Global Computing Education, Association for Computing Machinery, New York, NY, USA, ISBN 9781450390590 (2020), DOI 10.1145/3467967
2 R. R. Schaller, "Moore's law: past, present and future," in IEEE Spectrum, vol. 34, no. 6, pp. 52-59, June 1997, doi: 10.1109/6.591665.