The Future of Parallelism and Concurrency in C++ and How It Relates to DOE Abstraction Layers

David Hollman,
Sandia National Laboratories

Abstract

The end of single-processor speedup in the early 2000s ushered in an era of unprecedented growth for C++, owing largely to a revived focus on minimization of overheads at the software level. The demand for better expressions of parallel and concurrent application code targeting multi-core processors drove the design of large portions of the C++11 standard. Much like the first decade of the 21st century, the past decade has also seen a shift in the demands of hardware—most prominently due to the emergence of accelerators and many-core architectures. As a result, the C++ committee has put significant effort recently into the language and library abstractions needed to achieve performance on modern hardware for C++20 and beyond.

One of the largest and widest-reaching library design efforts undertaken by the C++ committee in recent years is the proposal of a generic abstraction for execution, known as an executor. Dating back to at least 2012, the effort to design an executors library for standard C++ has come to a head recently with its inclusion in the list of priorities for the C++20/23 time frame. The effort to construct a concept-driven design that meets the needs of vastly different domains with vastly different approaches to execution and vastly different needs is one of the most ambitious generic programming exercises of its kind ever. In the past year in particular, the executors design effort has undergone significant changes to accommodate the needs of yet another domain and yet another perspective on execution. The authors of the proposals have had to invent new mechanisms for expressing the vast outer product of behaviors, semantics, performance hints, and other properties of executors in generic contexts.

Such advances demand careful consideration from current and future DOE abstraction layers. Performance-portable expression of HPC applications requires careful use of generic programming that balances readability, productivity, and optimization potential. By designing abstraction layers that conform to current and upcoming international standards, applications can benefit from the massive investment outside of HPC in this area, allowing them to take advantage of common optimization effort, teaching materials, and programming model understanding.