Abstract

The fundamental difficulty with scientific software is that the existing languages are either too general-purpose and too low-level, or both. Thus, physics equations are entangled with the way they are implemented, even to low level details such as grid structure, cache optimization, etc. Because everything is coupled, it is difficult for the various scientific communities to adapt to algorithm or hardware advances. More importantly, it means that researchers cannot specialize as easily. Everyone must understand physics, software engineering, and computer science to a very deep level, and the learning process, especially for students, is much more arduous than it needs to be.

In principle, it should be possible to independently specify (1) the scientific equations to be solved; (2) the type of grids; (3) the type of numerical methods, e.g. time integrators, elliptic solvers; (4) the intended execution platform (desktop, accelerated cluster, etc.); (5) performance goals, e.g. as fast as possible, minimum cost, etc.

If these layers could be properly separated, physicists could readily explore new science without having to become software experts, computer scientists would be able to accelerate and improve existing codes without having to become deeply entangled with a physics problem, and hardware vendors could accelerate a host of relevant physical problems by concentrating on a very limited part of the software stack. Indeed, vendors could even target specific classes of problems for hardware acceleration.

The Chemora project is an effort to isolate some of these languages in the context of an existing code framework with a well-established community. While it cannot claim to fully solve the issues involved in isolating the layers, it is an effort to move development in that direction.

Our experience with the Cactus project has told us that modularization is the key to building lasting and useful code. The Cactus framework[1], even from its inception, has been effective at modularizing concerns such as the time integrator, coarse (MPI level parallelism), I/O, etc. It accomplished this by allowing scientists to focus on writing code for block-shaped domains, e.g. kernel updates.

This has led to collaboration between scientists in multiple countries, both in fundamental physics research and computer science innovations.

The advent of OpenMP broke this model to some extent, causing parallel directives to intrude into the scientific code. While this was not too painful, the introduction of accelerators made the problem significantly worse.

Even before accelerators became interesting for large-scale scientific computations, the sheer complexity of Einstein’s equations of general relativity introduced a new level of complexity into the existing software framework. These equations can be written down in very compact form, but any low-level implementation in software results in literally tens to hundreds of pages of mathematical equations. Implementing and debugging these by hand was painful, and experimenting with even slight modifications was virtually impossible. Out of this need, a new layer
called Kranc\cite{2} was created that understands equations at a high level, and from this generates low-level code that can be understood by a standard compiler.

The benefits of such a layer do not stop at the ease at which equations can be modified without re-writing a major part of low-level code. It also means that, given a motivated team of people, the same high-level physics code can be used to generate similar code for different hardware architectures. Even beyond that, automatic (transparent for the end-user) tuning can be applied to run efficiently on a majority of the available supercomputing resources.

Using a high level description and Kranc has already made some performance benefits possible, specifically vectorization. Most compilers will not attempt to vectorize equations as large as those appearing in the Einstein equations. Manual calls to vector intrinsic instructions are required.

Work at making Cactus code run optimally on GPU’s is, however, a harder task and requires deeper changes both to the Cactus infrastructure and to the Kranc code generation system. However, the end result of this effort should result in something modular which can more easily adapt to hardware changes.

References
