Achieving Genericity and Performance using Embedded Domain Specific Languages

Vincent Reverdy[a,b], Jean-Michel Alimi[a]

[a] *LUTh, CNRS, Observatoire de Paris, Univ. Paris Diderot; 5 place Jules Janssen, 92190 Meudon, France*
[b] *Department of Astronomy, University of Illinois at Urbana-Champaign, MC-221, 1002 West Green Street, Urbana, IL 61801, USA*

Abstract

We describe the design and use of Embedded Domain Specific Languages (EDSL) to achieve both genericity and performance required by modern simulation codes, taking an example in cosmology. We highlight the importance of good software architectures to tackle the increasing complexity of these codes, in a context of an increasing hardware heterogeneity and in a context in which physicists want to explore more theoretical models than ever.

Key words: High Performance Computing, Exascale, Cosmological simulation, Programming Languages

1. Why exascale and big data?

Let us ask the question "In the first place, why do we need exascale and big data approaches?". Only as a domain expert in cosmology, running on one million nodes to achieve one exaflop or analyzing petabytes of data are not goals in themselves. Asking "Why?" is asking for scientific motivations. And yes, of course, there are plenty of scientific motivations for exascale and big data. Getting more computing power provides physicists with the ability of:

- running bigger simulations to probe more scales in time and in space
- running more realizations for statistical purposes
- exploring parameter spaces with more accuracy
- enrich simulations with more physical phenomena
- test for the emergence of complexity with *ab initio* approaches

and having the possibility of running analyses on a very large number of heterogenous datasets help them to refine models and search for anomalies. Consequently, the accumulation of data in physics and their analyses is not the end of the story. It is the starting point of a deeper understanding of the processes we are interested in. That being said, as we have relevant science cases for exascale and big data, the next question we may ask is about the technical means that will be necessary to get there. And this is where problems arise because it may require a deeper knowledge of hardware and software from domain experts.

2. How will we achieve exascale computing and big data analyses?

If we are looking to current trends in high performance computing, it is very likely that we will not get to exascale with standard CPUs. The most powerful machines are equiped with accelerator cards, either many-cores or GPUs. Getting the best out of these heterogeneous hardware architectures requires more and more knowledge: networking for inter-node communications, asynchronism and threading for intra-node parallelization or data alignment techniques to maximize SIMD efficiency. Reaching exaflop is one problem, reaching exascale is a different one. And if we do not have a technical breakthrough in the coming years regarding to memory, the data transfer rate between memory and the central processing units is likely to become one of the most limiting factor of computing. In other terms: computing time could be ultimately considered as negligible compared to data transfer time. Solving these kind of issues require an understanding of data structures, memory models, cache misses and branch prediction. To summarize: designing exascale scientific codes will not be an easy task. And if, in the same time, physicists want to increase the complexity of their codes to solve new interesting problems, it will become a nearly impossible one. This raises the question of how scientific applications are written, how languages are designed and what kind of software architecture are chosen.

3. Issues with standard programming models

In order to address these problems, a first (probably naive) approach would be to bring together computer scientists, applied mathematicians and physicists together to create an ideal application to answer a particular scientific question. The major issue here, is that the implementation time can exceed by several orders of magnitude the life time of the scientific question. Scientific codes are not static: the underlying models are not fixed once for all and are intended to evolve during the application lifetime. If one looks to an existing scientific application, it can generally identify four problems regarding to software architecture.

The first one is related to the interwining of completely different aspects: physics, numerical algorithm, and optimization/parallelization. As these aspects are interdependent, it becomes more and more difficult, as the software grows in complexity, to change just one aspect of the code, for example, the physical model. This could be easily illustrated by a simulation code based on a mesh in which the physical variables are stored once for all in separate global arrays. When one wants to add a new physical variable it needs to modify all the internal loops of the code. The interdependence of the physics and algorithms makes the code difficult to modify.

The second one, which is a consequence of the first one, is the non-commutativity of modifications: two different scientific teams starting from the same original version of a scientific code but implementing changes in different orders (e.g. changing the model, or changing the parallelization) are unlikely to be able to merge their versions in the end. This leads to an increasing number of forks of scientific codes.

The third one is the problem of expertise: as all the aspects are intertwined, it is complicated to improve a particular element of the code, without understanding the rest of it. As a consequence, it is difficult to make the most of people with different backgrounds. For example, a software in which the mesh structure and the physical solvers are intertwined at the most fundamental level is barely modifiable by a non-domain specialist.

And the last, and maybe the most problematic one, is the inability to manage the combinatorial explosion of the lines of code. In fact, with traditional programming models, to achieve the best performances for any combination of P data types, Q algorithms and R architectures the required lines of codes (SLOC) scales as:

$$\text{SLOC} \propto O(P \times Q \times R) \tag{1}$$

In other words, if one wants to achieve the best possible performance, it will need to write all the versions of all functions, leading to a combinatorial explosion of code complexity. As an example, a software based on two data types (scalars and vectors), providing three algorithms (sum, mean value, maximum norm) for three architectures (sequential, multicore threading, GPUs) may require up to eighteen times more lines of code than a single combination of these elements. Consequently, each research team tends to only implement the specific case it really needs, leading to a loss of genericity and long term maintainability.

4. A case for Embedded Domain Specific Languages

In the same manner as one could try to optimize an algorithm, one can wonder whether it would be possible to manage code complexity by changing the way the number of SLOC scales:

$$\text{SLOC} \propto O(P + Q + R) \tag{2}$$

Converting the original product to a sum is the task of a compiler: taking different pieces of codes, putting them together in a coherent way and optimizing the result at the assembly level. Doing so, it solves the above mentioned issues of standard programming models by taking apart the different aspects of codes, therefore letting the expert to focus on their domain of expertise and letting the compiler generating the glue code to construct the executable. Instead of bringing together computer scientists, applied mathematicians and physicists to write the application, they can cooperate at a more abstract level: the software architecture level.

Technically, designing a language and a compiler for a particular domain is not an easy task. In fact, there are three possibilities to solve the four problems: (i) to create a Domain Specific Language (DSL) from scratch, to create a precompiler or (ii) a language translator and (iii) to create an Embedded Domain Specific Language. The first possibility offers more flexibility but as the scientific targetted communities can remain small, long term maintenance and high quality optimizations can be difficult to achieve. The second one consists in writing a program which will translate a language dedicated to a particular application into a more general language. It can be, for example, to automatically generate code for matrix operations. The community remain small, but implementing a translator is far easier than implementing a whole compiler. But it can deeply modifies the compilation process. The third one directly incorporate itself in an existing language, being completely

transparent for the final user. It is this third possibility that we focus in this "white paper". The general idea is to implement a language inside another language which will act as an active library in between the user code and the assembly code. Languages like C++ or D are particularly adapted to these tasks. Using template metaprogramming, compile-time reflection, and generative programming, one can either precompute results at compile-time or control the instanciation process using the properties of types thus achieving both performance and genericity.

5. Lessons learned from experimentation

We have successfully created an EDSL in C++11 to implement a 3D raytracing code based on the equations of general relativity in a weak-field metric. In this code, the underlying data structures, the integrators, the physics equations to be solved, and the parallelization algorithms are implemented separately. The combination of these elements is done at the end to generate the raytracing code, but other applications as the development of a full Cosmological code, are possible with no modifications of these fundamental bricks.

It relies heavily on template metaprogramming to perform operations during the compilation process. These operations can be of two types: either operations on values or operations on types. The idea of value metaprogramming is to compute as much as possible at compile-time to avoid unnecessary computations at run-time. It can be for example to simplify as much as possible series expansion, keeping the lowest possible number of operations for execution. Or it can be to precompute the geometric properties of a structured mesh only knowing the number of dimensions, therefore dramatically improving performances relying on compiler optimizations. But it implies that the final user have to recompile its program every time it changes the number of dimension. The idea of type metaprogramming is to control the instantiation process using properties of types. In practice it consists to branch to portions of codes when a type satisfies certain characteristics.

In the case of our 3D raytracing code, we have been able to produce generic meshes and generic integrators, acting in the same way no matter what the physical contents is nor the integrated vectors are. In our solution, the hyperoctree used to partition the physical space is effectively implemented separately from the containers of the physical variables (for a cosmological simulation: the matter density ρ, the gravitational potential Φ and its derivatives...). As the second one can be passed as a template parameter to the first one in the user code, the compiler can generate the optimal tree for the specific underlying physics at compile-time. The same kind of techniques can be used to implement separately the parallelization schemes.

By doing so, EDSL creators provide application creators fundamental generic pieces of code. Each application can combine these pieces in its own way, letting the compiler to find the best possible implementation and optimization for the final code. These techniques have been successfully tested for years by the Boost library community and our experience let us say that they can be perfectly used to write scientific codes. Our 3D raytracing code has been used to analyze the light propagation in the Dark Energy Universe Simulation: Full Universe Runs on the Curie supercomputer. It managed without any scalability problems, hundreds of billions of mesh cells distributed over thousands of CPUs, allowing us to examine the imprints of large scales structures on cosmological measurements.

6. Conclusion

Standard programming models are unlikely to be able to manage the increasing complexity of codes coming from both the hardware heterogeneity and the physics to explore in the exascale and big data era. Focusing on abstractions and on domain specific languages one can reduce this complexity at the programming level achieving both genericity and performance to get the most out of the machines of the next generation. This is still an exploratory direction, but as the C++ committee seems to push in the direction of generic programming and compile-time reflection, it is worth to be considered with attention.