Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 7, Number 1, pages 5–8, January 2018

Industrial proof of HPC application design for long term maintenance

Francois Letierce
CEA
CEA, DAM, DIF
F-91297 Arpajon Cedex, France
francois.letierce@cea.fr

***Abstract*— For the past 20 years, we developed CFD software in an industrial context. These applications contain several highly complex physics and can use a lot of resources. To offer the best user experience, these software were designed or modified to use super-computers' power in order to achieve reasonable simulation times. Therefore, although developers were not computer scientists (but mostly physicists and mathematicians) they learned HPC the hard way. Through a historical journey, we propose to share our experience of making a large and heterogeneous application HPC compliant, and how this work has influenced our choices for developing its successor. Then, we present this new HPC orientated software. We explain its architecture: object-orientated approach and proprietary middleware which can abstract low level HPC related problems; and its development methodology: UML modeling with code generation and fully integrated testing environment for results and performance analysis. Finally, we give some feedbacks on its development experience and show some early results. These results are behavior and performance comparison between new and old "hand tuned" code, on mainframe processors.**

Keywords—legacy application migration; software design; HPC; object-orientated framework; development methodology

I. A HISTORICAL INTRODUCTION

Simulation has always been an important part of our job, but, for the last couple of decades, it has become our main focus. We use an iterative scientific approach to solve our industrial problems. It comes with 3 main steps:

- First, we try to model real and highly complex phenomena with physical laws which are translated into mathematical equations;
- Then, numerical simulations are used to solve these discreet equations (in time and space);
- Finally, simulations results are compared with experimental ones to validate models.

Although this methodology is working fine, it has evolved over time. In 20 years, every engineer has become very familiar with computers and it's now unthinkable to do your scientific job without one. In numerical simulation field, even more complex computers are used: supercomputers. And they come with different flavors over the ages.

A. Part 1: Supercomputers

Not only these engineers need to be fluent in some programing language, but they also need to master some low level computer skills in order to get the most of supercomputers' power. These skills depend on supercomputers architectures and internal technologies. Here are some examples of what we've been through in our company:

- The 1st supercomputer bought to run our large industrial simulation software was a Cray T90, providing roughly 50Gflops. As a vector processing machine, developers needed to learn how to write code specifically to use its maximum potential.[1]
- The 2nd one had a totally different architecture. It was a SMP cluster. Hundreds of nodes composed of several CPUs with shared memory, providing roughly 5Tflops. No more need to use a vector friendly approach. But this time, developers were trained to grasp *message passing parallelism*, rewriting software to get a chance to use all this power.
- Thankfully, the 3rd one used similar cluster architecture, providing roughly 50Tflops. Nevertheless, as CPUs were Intel Itanium, developers' knowledge of computers was extended, almost as much as these CPUs pipelines. This should have needed some code rewrite to achieve better performance. But this time, we passed.
- With the 4th one, we thought we finally get a hand on what will ever be supercomputers. Even refined (Intel Xeon Nehalem, Fat Tree Infinity Band, etc.) its architecture was familiar. System was much more mature and stable (Linux, SLURM, LUSTRE, etc.). Still, something was different, multi-core processors. Some developers did not grasp the need to redesign their code to use concurrency [1], some others tried to abstract parallelism between CPU nodes and CPU cores [2], the rest tried to rewrite code, entirely or partially, using multithreading (sometimes nested inside message passing parallelism) to achieve performance (1Pflops).
- Now, the 5th one is at door. Composed of 2 distinct parts. First part is an upgraded "clone" of the 4th one. But again, a "little" change in CPUs' (Intel Xeon

[1] Fun fact, this supercomputer was like a very big black coffin, filled with *Fluorinert* so every pieces of it can be liquid-cooled.

Haswell) design: vector processing units are back…To achieve maximum performance (roughly 2.5Pflops), multilayers of parallelism are needed. One for message passing, one for multithreading, one for vector processing. And this is even more mandatory to fully exploit the 20Pflops of the second part of this supercomputer: hundreds of nodes of Intel Xeon-Phi (KNL).

B. *Part 2: HPC applications*

With this history in mind, needlessly to say that legacy HPC application migration of has been very important to us. In this talk, we will speak about one of our fast *Computational Fluid Dynamic* software.

Almost 20 years ago, researchers and engineers, physicists, mathematicians and some found of computers people were gathered in a lab to develop CFD codes. Separated in 2 teams, each team developed a code. These programs needed to implement complex physical models, each one with a specific goal. They were designed to be run one after the other and aim to be highly performant.

The 1st one, *A1*, was originally written in FORTRAN and was a sequential program. As supercomputers evolved, the 1st major HPC migration was to make it parallel. Writing message passing parallelism, lots of "glue code" was added, written in C and later in C++. Core data structures were mostly arrays and helpers functions. The rewriting process took a lot of time and resources. It ended making the code more complex to understand and to maintain. Changes were scattered everywhere and mostly undocumented. A1 program was developed, maintained and used in an industrial context for almost 20 years. Of course, new features were introduced along the road. Developers designing them were more aware of HPC needs and tend to think of them when creating new algorithms. But as we have seen though supercomputers' history, parallelism needs of a time may be different from one of another time. Eventually, performance bottleneck were studied. For example, it took a couple of years to rewrite a complex feature very sequential by nature so it can be run in parallel. In an industrial context, you can't remove a much needed feature because it destroys performance. Sometimes, you can't even switch to newer or different technologies because it would imply changing years of simulation results, used as data bases for users' work.

The 2nd one, *A2*, was entirely written in FORTRAN. Although it has known a very similar fate to A1, glue code to make it parallel, changes and new features have always been written in FORTRAN. This choice made maintenance quite easier.

With the new era of supercomputers based on many cores architecture, it has become inevitable to rewrite codes to use concurrency. A1 or A2 have been migrated too many times already, and further internal changes cannot be afforded. This would cost too many resources, too much time and would end up giving even complexes codes and be harder to maintain. Then, decision was made to write a new code: *A3*, replacing both A1 and A2 codes. Along the stakeholder requirements, it had to be HPC compliant, ready to use future exascale supercomputer. We still don't know what form these future supercomputers will take, but with feedbacks and experiences gathered for 20 years, one thing is certain: we need abstraction. This abstraction layer must allow us to address underneath changes without rewriting core level code.

II. Choices to design an industrial HPC application "migration ready"

Now part of the teams, computer scientists have bring along them some professional knowledge about software design, algorithms implementations technics, low level computers functionality, etc. With them, we tried to define what we want through this much needed abstraction layer and what shape it would take.

A. *Programming language*

Experience has shown us that using only one programming language is easier to maintain. We want abstraction and need performance. We choose C++. It has object-orientated programming features. Still, you can access low level API to achieve good performance. C++ ISO Committee is also becoming more active lately (C++'11, 14, 17 standards). Especially expanding language for better abstraction and working on unifying concurrency API.

B. *Abstraction layer*

Although DSLs are very attractive because they provide very high level of abstraction and good performance (*via* specific generated code optimization), one of stakeholders need was long-term maintenance. Most considered DSLs were developed by small teams or felt too much like research tools. Therefore decision was made it was too risky to base large industrial software on these. Instead, we have chosen a proprietary framework, compliant with our professional needs and developed for over 10 years. It is named Arcane [3] and is co-developed by CEA-DAM [4] and IFPEN [5].

III. Development environment and tools

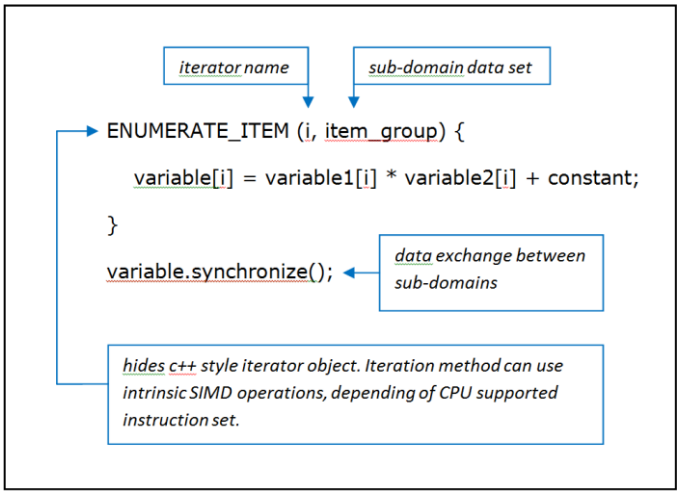

Fig. 1. One example of Arcane's macro which hides complex iterator objects. Theses objects can automate vectorization (using intrinsic operations). Message passing abstraction API is also demonstrated in this figure.

A. *Arcane*

This development platform is designed to provide lots of tools to help writing parallel numerical simulation codes. It is written in C++, uses object-orientated approach, support dynamic configuration through XML configuration files and is well documented. It covers architectural aspects, like mesh associated data structures; parallelism, to abstract technical difficulties; and environment, like input data set, dynamic configuration of code execution, output files, etc. From a HPC migration perspective, here are some interesting items:

- Could it be a mesh item or a core component, everything is well designed as object-orientated classes. You can find common design patterns and it helps developers making quality code, mutualizing and anticipating future features. This is where object-orientated truly shine and it is needed to avoid rewriting everything.
- PODs are redefined, meaning you can easily change floating point precision, integer size or be prepared for a hypothetical 128bits change, without rewriting your code. But that's pretty basic. You can also find "SIMD types" which were designed to use explicitly SIMD methods (when it's available in your CPU' instructions set). This API abstracts the technical difficulties to use intrinsic functions for vector processing functionality.
- Most importantly: parallelism is abstracted. A simple API lets you synchronize mesh data structures over multiples sub-domain (using message passing paradigm). Even more interesting, concurrency and vector processing are also hidden behind a simple API. For example, using a simple macro, "Fig.1", let you iterate over a mesh data structure items. Behind this macro is an iterator object (like standard C++ STL iterator). Used with appropriate SIMD types, this iterator will ensure that compiler will generate SIMD instructions. Even better, compatible CPU' instructions set will be auto detected by the platform, so AVX512 will be used for KNL, AVX for Haswel, etc. No need to learn intrinsic API anymore, which is a big relief to physicists or mathematicians.

B. *Modane*

Obviously, developers need to learn Arcane's concepts and API to make use of it. There are fundamental classes you will need to understand before starting writing your code. Creating these objects is a tedious task. When using Arcane platform, you can have access to an UML-like modeling software called Modane [6]. With this tool, developers can graphically design their software components as they would do by drawing an UML class diagram. Modane is also able to auto-generate associated Arcane/C++ source code and XML input options. The tedious task is gone, and the generated base code is error proof, "Fig.2". Along the multiples options of code generation, you will find the possibility to generate parallel code for some specific functions. For example, specifying that a method will iterate over all nodes of the mesh, Modane can generate a multithreaded loop. All you have to provide is the node processing function you want.

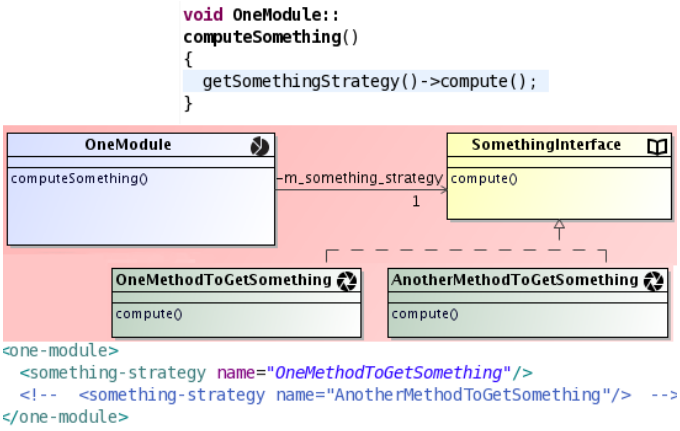

Fig. 2. An example of component design using a typical strategy design pattern with Modane's GUI. C++ auto-generated code is given above model. XML configuration part is given below.

IV. Methodology

These choices have allowed us to design new software using a model driven concept. With capitalized experience of both teams, the 1st development step was to decide what can be shared between old codes, what was useful for both of them and can be reused "as it is", mainly because it has already been worked on and can be "unplugged" with minimum effort. These discussions took place for almost a year. In the end, we grasped how to mutualize our core business models and decided how components will be designed. The design process of each component started with a reverse engineering phase. Depending on how the component was designed in old codes, development could either be:

- To rewrite from scratch, using middleware full potential;
- To partially rewrite, replacing old hard coded parallelism operations or data structures and use framework API;
- To reuse the whole component, only developing an adaptive wrapping layer.

Because we use object-orientated concepts, complying with well-known design patterns, almost every component's models are expendables. One can be replaced with new one easily and switching from one to another is as simple as changing a keyword in a configuration file, "Fig.2". With these possibilities, validation was made easier. We have been able to compare components' behaviors and results from old codes to newly developed one. This was much needed in an industrial context to ensure users experience won't change (too much) and to produce expected results for similar simulations. Once validation is over, new features or improved components can be introduced, tested and used.

V. Results

After almost 5 years of development, we have successfully delivered new software, ready for industrial use, merging all functionalities of both old codes. A lot of time was used to train

developers (who, I recall, are mainly physicists and mathematicians) to object-orientated concepts and UML modeling. Obviously, learning Arcane API and mechanisms took some time too. But once developers were used to these, they spend most of their time designing component, capitalizing years of feedbacks, thinking ahead for new possibilities and focusing on features and algorithms. They can think of how to improve their models and numerical schemes instead of learning low level computer tricks or writing tons of glue code. Coding time was pretty short after all. Validation took a lot of time, and because it was done comparing results with older codes, we encountered one of the main problems of the chosen methodology: performance issues. Simulation times between old and new codes were very different. Using generic middleware, real object-orientated design and auto-generated code tend to cost a lot, performance wise. Because of the HPC context and to deliver a better experience for users, we worked on improving performance the last few years, along the rest of the development process.

Using profiling tools (Arcane internal ones or 3rd party ones), developers learned to optimize their code. Most of the problems were:

- Bad use of some framework operations, mostly due to inexperience during early development stage;
- Sub-optimal use of middleware data structures or parallelism mechanisms;
- Reuse of algorithms with inadequate data flow for Arcane's data structure, heavily penalizing memory access.

With the help of Arcane's team, problems were solved and developers are now writing much more appropriate code. Arcane own developers' team has also fix and enhance their framework. This conjugate work is still in progress and results are very encouraging, improving performance over the years, "Fig.3". Furthermore, as Arcane is used by other applications, every change benefit to all of them.

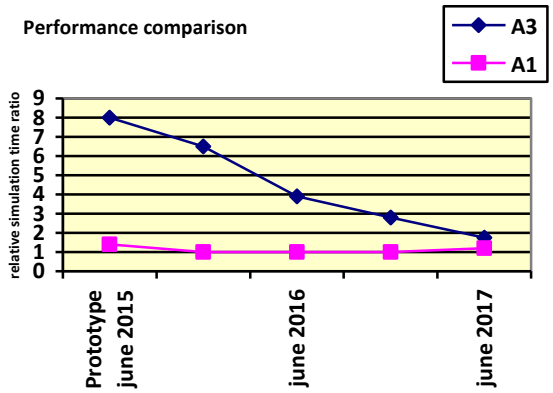

Fig. 3. Performance comparison between old and new code on a typical industrial test case: 2 million elements mesh, divided in 64 sub-domains.

VI. Conclusion

Capitalizing experience through many years of HPC application development, switching from one supercomputer to another, porting parts of codes to adapt to each flavor, we decided to write new software to face the exascale challenge. We needed an abstraction layer to hide low level computer related needs so developers, who are not computer scientists, could focus on their own discipline.

This abstraction comes with a cost. Learning object-orientated approach and UML modeling is not an easy task. Yet, these concepts feel more sustainable than to learn current CPU instruction set. When developing as a team, they are powerful tools to design, share and anticipate features of software.

Using a professional framework alleviate the coding phase and provide abstraction over data structures, parallelism, configuration, etc. Inconvenient is that it aims to be generic, and by doing so, may not be as performant as handmade and finely tuned code. Yet, with good interaction and joint effort, lots of work has been done to improve both framework and application. Arcane's roadmap is promising and more expected HPC features are to come, such as SIMD operations abstraction, dynamic load balancing, task based multithreading, etc.

Overall, experience is very positive, from both developers and users. Performance issues are becoming less and less relevant. We are very optimistic that, with upcoming works (on both software and framework), the new software will be more performant. Especially because it is much more stable and robust, it can handle resource scaling old one could not. And this is exactly what we wanted to hit the exascale road.

[1] H. Sutter, "The Concurrency Revolution", C/C++ Users Journal, 23(2), 2005.

[2] P. CARRIBAULT and M. PERACHE, "The MPC (MultiProcessor Computing) framework", https://sourceforge.net/projects/mpc

[3] G. Grospellier and B. Lelandais, "The Arcane development framework", In Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC '09). ACM, New York, NY, USA, 2009.

[4] CEA-DAM Available at: www-dam.cea.fr.

[5] IFPEN Available at: www.ifpenergiesnouvelles.fr

[6] B. Lelandais and M.-P. Oudot, "Modane: A Design Support Tool for Numerical Simulation Codes", Oil Gas Sci. Technol. – Rev. IFP Energies nouvelles, 71 (4), 2016.