An optimized finite-element library: Akantu

Nicolas.Richart@epfl.ch, collaborateur scientifique, Guillaume.Anciaux@epfl.ch, collaborateur scientifique
& Jean-Francois.Molinari@epfl.ch, professeur EPFL – Laboratoire de simulation en mécanique des solides

Akantu signifie petit élément en kinyarwanda, une langue bantoue. Désormais, c'est également une bibliothèque open-source orientée objets d'éléments finis, qui a pour ambition d'être à la fois générique et performante.

Akantu means a little element in Kinyarwanda, a Bantu language. From now on it is also an open-source object-oriented library which has the ambition to be generic and efficient.

Within LSMS (*Computational Solid Mechanics Laboratory*, lsms.epfl.ch), research is conducted at the interface of mechanics, material science, and scientific computing. We currently work on damage mechanisms, contact mechanics, and micro mechanics. These domains imply studying phenomena at different scales, from atomic (nano-scale) to continuum (macro-scale). In order to understand the physics involved, we need ever more computationally intensive numerical simulations. For the macroscopic scale the finite-element method is a well established numerical method. However, as far as we know, there are no open-source projects that fulfill genericity, robustness and efficiency. Along these requirements **Akantu** was born.

The genericity is necessary to allow the easy exploration of mathematical formulations through algorithmic ideas. Furthermore, we believe that the open-source philosophy is important for any scientific software project evolution. Indeed, the collaboration permitted by shared codes enforces sanity when users (and not only developers) can criticise the implementation details. In addition, the understanding of complex physical mechanisms stands on the manipulation of huge data sets. Therefore, robustness and efficiency permit to push further the limitations imposed by the numerical simulations and more specifically in the context of parallel computation.

In order to achieve these goals, we made noticeable choices in the architecture of Akantu. First we decided to use the object-oriented paradigm through C++. This paradigm is useful in terms of genericity and code factorization. In fact, it relies on the concepts of inheritance and polymorphism, which allow to identify the common interface of objects so as to define high-level classes. These are to be derived in specialized classes. For instance, in the finite-element method, applied to solid mechanics, we need to compute different material laws. Most of them take strain as an input and compute the associated stress. In that case, the common interface to material objects contains a function that compute stresses from strains (see Figure 1).

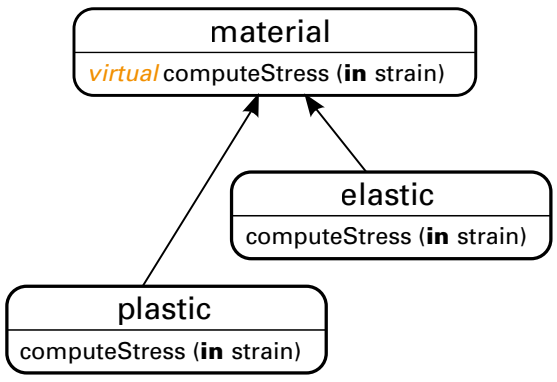


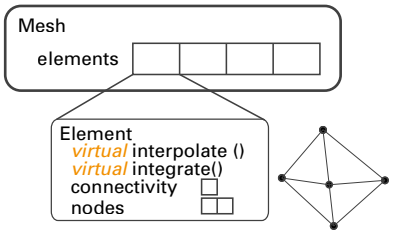
fig. 1 – inheritance schematic of material classes

The constitutive law is obviously not computed in the same way for every materials. Each material has to re-implement the function *computeStress*. The polymorphism mechanism allows the use of a common interface with any kind of material instantiated. It mainly relies on virtual function calls, which consist in finding the right function to invocate from the table containing all the implementations. Even though polymorphism provides an helpful tool to developers, there is an extra cost associated to virtual function calls that affects strongly the efficiency.

Then, virtual function calls should be limited to specific situations and avoided where critical sections of the program are executed. In finite-element algorithms, in order to perform field manipulations, loops over elements are always necessary and form the critical sections. In these loops, virtual calls should be excluded in order to maintain good calculation times. To demonstrate this point, we will use the example of mesh objects, which are naturally part of every finite-element code. Two distinct architectures are now presented: first an all-object approach and then what has been used in Akantu to avoid virtual function calls.

A mesh is a set of elements that connect some nodes. Depending on the meshing process, these elements can be of different types (triangles in 2D, tetrahedra in 3D,...). The natural idea is to define a generic element class that describes a common interface. Then, any element can inherit from this common object description. In this view the element embeds a lot of intelligence. For example, one element should know how to integrate a given field. This approach, which forms a full object architecture, stores for each element a complex object which is also autonomous (see figure 2a). In any processing loop over elements, it will result in a virtual function call per element and lead to a drop of performance. To improve this, while maintaining the usage of object oriented paradigm, we limit in Akantu the virtual calls to be outside of any loop. Inside a loop, the manipulated data structures are vectors. For meshes, it means that elements, as a group, are represented by a vector of nodal coordinates and a vector of connectivities (see figure 2b). Global functions, like the integration procedure, operate on the entire set of elements. The counter part is that genericity is reduced when compared to a full object view: the high level classes contain now more complex functions.

a) Full object architecture



b) Object/vectors architecture

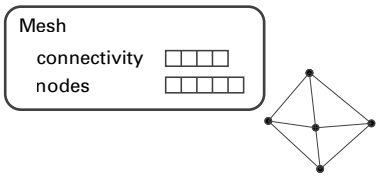


fig. 2 – object architecture versus a mixed object/vector architecture for a mesh class

In the case of the object/vector architecture, there are less virtual function calls but there are still potentially conditional jumps in the critical loops. Indeed, for a mesh containing different element types, an integration loop includes a decision per element to select the appropriate integration method. However, even a single *if* statement can decrease performances. Therefore, to be even more efficient, decisions should be avoided in loop contexts. The solution is to make choices outside of the loops. This will produce functions that are specialized to a typical situation. In other words the code needs to be vectorized.

In Akantu, the connectivities have been sorted by element types so as to break loops over elements as said above. Concerning a complete finite-element sequence, that contains gradient computation, constitutive law call, integration and assembly, the vectorialization imposes a specific task organisation. Global tasks can be divided in *simple* operations and pipelined in order to obtain the desired result, as shown in figure 3.

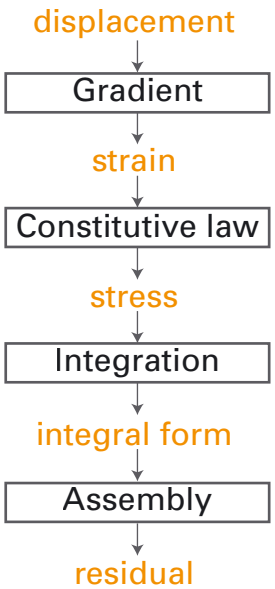


fig. 3 – pipeline of vectorial operations used to compute nodal residuals from nodal displacements

The crucial point here is that these SIMD (Simple Instruction Multiple Data) operations are in fact well optimized by nowadays compilers. These vectorial operations can also be ported *easily* on vectorial architectures such as modern GPUs. Nevertheless, the main drawback of this approach is the memory cost since we have to store partial results through the task pipe (strains, stresses, integral form, see figure 3).

In order to demonstrate the performance of our code and the relevance of our architecture choices, we made a comparison with another C++ finite-element code, OOFEM (*Object-Oriented Finite-Element Model*, www.oofem.org). In OOFEM the authors made the choice to use the object inheritance concept down to the lowest levels. Our comparison test case considers a meshed cube which models steel during a normal compression. Comparative results are presented on figure 4.

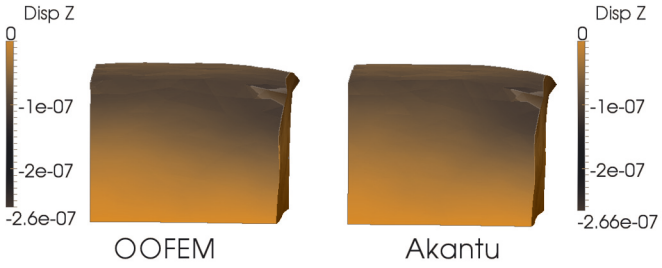


fig. 4 – comparative view between OFFEM and Akantu after 5000 time steps. The color shows the Z-axis displacement field

While the numerical results are very close, performance of Akantu appears to be 25 times faster. Thus, the choice of being very generic and of having a full object view has an important impact on the performance.

We used the same test case, and refined the mesh to get approximately 6.6 million elements, in order to do a scalability test of Akantu. The results are shown on figure 5. The scalability shows good behavior up to 32 processors. We also emphasize that a super scalar effect is observable with 4 processors. This must be due to communications overlap and important processor cache effects. The announced memory usage drawback appears not to be a real limitation. Indeed on a cluster considered as low memory (2Gb per octo-core) we managed to run a reasonably large case with approximately 3.3 million of elements, even in sequential.

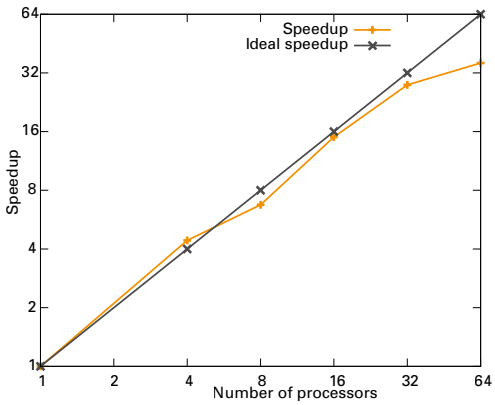


fig. 5 – Speedup of Akantu library. The model manipulated is a cube meshed with 6 567 862 elements

We presented the main choices taken in the development of Akantu to achieve genericity and efficiency. We designed the library as an hybrid architecture with object at the high level layers and vectorialization for the low level layers. Thus, Akantu benefits the inheritance and polymorphism mechanisms without the counter part of having virtual calls within the critical loops. Even if the development is still at its onset, the first results seem encouraging. They tend to prove that these choices show nice performance speedup while our needs for genericity were maintained at a reasonable level. Soon (summer 2011 ?), the first release of Akantu will be out, with a set of tutorials that are thought to be the basis of a future educational program. In particular, Akantu tutorials will be added to the core finite-element classes, at the Bachelor and Master level of the Civil Engineering program. Furthermore, Akantu will be part of several research projects conducted within LSMS. ■