Migration of a Generic Multi-Physics Framework to HPC Environments

P. Dadvand*,**, R. Rossi*,**, M. Gil***, X. Martorell***,
J. Cotela*,**, E. Juanpere***, S.R. Idelsohn* and E. Oñate*,**
Corresponding author: pooyan@cimne.upc.edu

* Centre Internacional de Mètodes Numèrics en Enginyeria (CIMNE)
Gran Capità s/n, Edifici C1 - Campus Nord UPC, 08034 Barcelona, Spain.
** Universitat Politècnica de Catalunya (UPC)
Jordi Girona 1-3, Edifici C1, 08034 Barcelona, Spain.
*** Computer Architecture Department - UPC
Jordi Girona 1-3, Edifici C6, 08034 Barcelona, Spain.

Abstract: Creating a highly parallelizable code is a challenge and development for distributed memory machines (DMMs) can be very different form developing a serial code in term of algorithms and structure. For this reason, many developers in the field prefer to develop their own code from scratch. However, for an already existing framework with large development background the idea of transformation becomes attractive in order to reuse the effort done during years of development. In this presentation we explain how a relatively complex framework but with modular structure can be prepared for high performance computing with minimum modification. Kratos Multi-Physics [1] is an open source generic multi-disciplinary platform for solution of coupled problems consist of fluid, structure, thermal and electromagnetic fields. The parallelization of this framework is performed with objective of enforcing the less possible changes to its different solver modules and encapsulate the changes as much as possible in its common kernel. This objective is achieved thanks to the Kratos design and also innovative way of dealing with data transfers for a multi-disciplinary code. This work is completed by the migration of the framework from the x86 architecture to the Marenostrum Supercomputing platform. The migration has been verified by a set of benchmarks which show very good scalability, from which we present the Telescope problem in this paper.

Keywords: Parallelization, Computational Fluid Dynamics, Domain Decomposition.

1 Introduction

The present work is based on Kratos Multi-Physics [1] a free, open source framework for the development of multi-disciplinary solvers. The complexity of the coupled problems and their large representing models in practice were the motivation to port the code to high performance computing platforms. The preparation was started by parallelizing the code for Shared Memory Machines (SMMs) and then completed by adapting to domain decomposition methodology for Distributed Memory Machines (DMMs).

In this work we describe the methodology and changes made in order to use different high performance platforms.

2 Kratos structure

In this section a brief description of Kratos structure will be given in order to understand better the parallelization procedure and its implications in the code.

Kratos is written in C++ and organized following object-oriented paradigms. We will focus on classes that encapsulate the algorithms involved in Kratos parallelization. A complete description of the classes can be found in [1]. The solution algorithms in Kratos are encapsulated in classes below:

`LinearSolver` encapsulates the algorithms used for solving a linear system of equations. Different direct solvers and iterative solvers can be implemented in Kratos as a derivatives of this class. `LinearSolver` is implemented based on the *Space* class. `Space` defines a matrix and a vector and also encapsulates their operators.

`Strategy` encapsulates the solving algorithm and general flow of a solving process. Strategy manages the building of equation system and then solve it using a linear solver and finally is in charge of updating the results in the data structure.

`BuilderAndSolver` is used by the `Strategy` classes to perform all of the building operations and the inversion of the resulting linear system of equations. `BuilderAndSolver` covers the most computational intensive phases of the overall solution process.

`Scheme` is designed to be the configurable part of `Strategy`. It encapsulates all operations over the local system components before assembling and updating of results after solution.

`Process` is the place for adding new algorithms to Kratos. Mapping algorithms, Optimization procedures and many other type of algorithms can be implemented as a new process in Kratos.

Another important class for our purpose is the `ModelPart` which holds all data related to an arbitrary part of model. It stores all existing components and data such as `Nodes`, `Properties`, `Elements`, `Conditions` and solution data related to a part of model and provides the interface to access them and their data in different ways.

From a very global point of view Kratos implements a kernel and application mechanism. Each application act as a plug-in and is compiled separately as a shared object. This structure of Kratos lets developers to concentrate on their own application meanwhile enabling the use of other applications via Kratos. This mechanism also results to be a key point in the parallelization of the code as we will describe later.

Finally Kratos uses the Python script as its main procedure. This is a large added value in its flexibility and also a very useful tool to handle several platforms by configuring the input script for the specific target without changes in the c++ code.

3 SMMs Parallelization

The first step toward high performance computing is the parallelization for shared memory machines. The OpenMP library is used for this purpose. The ease of use and its portability between different platform constitute the key points for this selection. However, the lack of conformance to the last standards in some compilers results in extra modifications in order to increase the portability of the code.

`LinearSolver` classes were the first part to be parallelized. As mentioned before the operation used in linear solvers is encapsulated in the `Space` classes. So, just by replacing the Space with a parallel version of it, all iterative solvers in Kratos became parallel without further effort. However, in practice some modifications had to be made in `BuilderAndSolver` classes to optimize the memory access for NUMA machines.

Following the parallelization of the linear solvers, the `Strategy` classes were parallelized. As described before the `Strategy` classes use `BuilderAndSolver` and `Schemes` to perform different tasks in the solution. For most of the strategy classes, the parallelization is reduced to changing their `BuilderAndSolver` and `Schemes` to a parallel version.

Finally, some `Process` classes had to be customized in order to parallelize them or to protect them from possible racing conditions.

All these improvements result in a good speedup of the code for multi-CPUs machines, but the memory bandwidth limit in desktop multi-core CPUs prevents the scalability of the solvers in these machines. In Kratos, most of the applications implement only new elements and conditions using standard strategies or provided `BuilderAndSolver` and `Schemes`. One can observe that many applications became parallel without any modification, which is considered as an important added value for the design.

4 DMMs Parallelization

After the preparation for SMMs the next step is to deal with clusters using a standard domain decomposition approach. In this process, the main objective is to have the same code for serial and parallel versions, and also to keep the data transferring part as automatic as possible for the applications. With this two objectives in mind, most of the changes are carried out in the kernel part of the Kratos, and a new `Communicator` class is implemented, which is in charge of transparent data transfer.

4.1 Partitioning

Following a standard domain decomposition approach, the first step is to partition the domain efficently. To this end, the METIS [2] library is used, since it reduces partition interfaces better than other methods such as greedy or spatial bi-sectioning. The possibility of using a balanced kd-tree still exists, although so far remains unexplored.

Partitioning must be completed with a colouring procedure to minimize the transference latency and also to avoid blocking in the processes. In this process, we look for a sequence of data transfers between domains which maximizes the number of simultaneous data transfers at the same time.

Regarding the code structure, the METIS partitioner is added via a new application; so one can compile it as a separate shared library and use it only if needed. METIS partitioning has an interface to Python, so one can call it from the input script file when running in a cluster. Thus, simply by changing the Python script, the same code with minimal changes can be used for both SMMs and DMMs.

4.2 Communications

As mentioned above one of our main goals is to make the data transferring part as automatic as possible for the applications. Another goal is to keep the serial and parallel codes in applications as similar as possible. These two goals are reflected in the design of the `Communicator` class. This class encapsulates all necessary data for domains, their interfaces and the decomposition data transfers in a generic way. The `Communicator` class is the base and it can store the following data:

`NeighbourIndices` Neighbour domains, with respect to the colouring

`LocalMesh` Entities that belong to this domain, including internal entities

`GhostMesh` Stores all entities which are a duplicated of the entities in other domains

`InterfaceMesh` Contains the entities that can be ghost or local but they are in the interface between this domain and other domains

`LocalMesh[i]` Stores all entities that belong to this domain but are duplicated in domain *i*

`GhostMesh[i]` Contains entities which are a duplicated of the entities in neighbour domain *i*

`InterfaceMesh[i]` Contains the entites that can be ghost or local but they are in interface between this domain and domain *i*

And the `Communicator` class defines the following groups of methods:

`Synchronize` Different versions of synchronize are in charge of copying different data from local entities to all their duplicated ghosts in other domains.

`Assemble` Calculates the sum of the data in a local entity and all its ghosts and set the result in the local and the ghosts.

`MaxAll`, `MinAll`, `SumAll`, etc. A group of method reimplementing the MPI communication tasks.

It is important to mention that the `Communicator` base class provides the interface to these methods with an empty version of them. The `MPICommunicator` class derives from `Communicator` and implements these methods for using MPI. This allows us to call a synchronize when necessary. In a serial run, the code uses the `Communicator` base class, which does nothing. In a parallel run, the `MPICommunicator` will be used, and the synchronize will be performed using MPI, without needing to customize the application for each platform.

4.3 Solution

Implementing a MPI version of the solution consists of implementation of `Strategy` classes for MPI. Here again, as in shared memory parallelization, the encapsulation of the solution in `Strategy` classes and the use of a few `BuilderAndSolver` and `Scheme` classes help to minimize the effort required. The new adapted strategies are based on Trilinos [3] library. This library provides a very good performance while providing a very clean interface in comparison with other similar libraries.

5 Benchmarks

We have performed the evaluation of the Kratos migration on the Marenostrum Supercomputer at the Barcelona Supercomputing Center. Marenostrum is built using dual-core PowerPC 970MP processors (2.3 Ghz). It has 2500 blades, for a total of 10000 processors. Each processor has a 64Kb instruction/32Kb data L1, and a 2Mb shared L2 cache memories. For the experiments, all Kratos software has been compiled with GCC 4.4, except the BLAS and LAPACK libraries that were compiled with the XLC10.1 compiler. Using the XLC compiler on those libraries resulted in an overall improvement of 5 to 10% on the Kratos computation time. The experiments presented in this paper use from 4 to 253 nodes, with the 4 cores available on them, for a total of up to 1012 cores. Time restrictions did not allow us to run on 1024 and above, yet.

We have used an input problem named Telescope, computing the airflow on the surroundings of the Canaries Great Telescope in the island of La Palma, in the Canary Islands, Spain (see Fig. 1). The problem has 24 million elements. All the experiments scale perfectly.

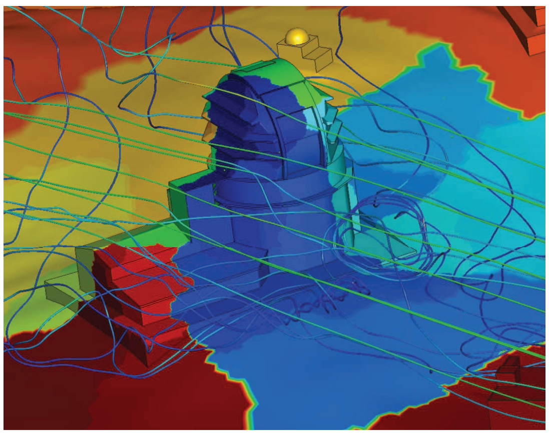

Figure 1: Snapshot of the Telescope solution

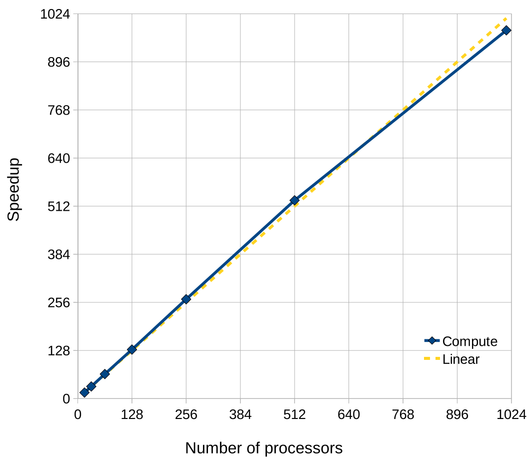

Figure 2: Speedup achieved on the Telescope problem

Using this problem, we have evaluated the cost of (1) reading the input problem and generating the appropriate partitions according to the number of processors, and (2) the computation time taken to compute 100 time steps of the airflow simulation. The first part (1) is done in a single processor, and the computation (2) is performed in parallel. Writing the results also occurs in parallel, and it is intermixed with the computation. This is done by writing the values computed after some of the time steps.

Figure 2 shows the speedup obtained in the computation phase of the Telescope experiment, compared to the perfect linear scaling. As it can be seen, Kratos scales very well when using up to 1012 processors. It is important to note that the data distribution can be done once for a problem, and then several experiments can be launched on it, so that achieving such good scalability is very important for the scientists interested in the solutions given by Kratos to their problems.

6 Acknowledgments

The authors wish to acknowledge the support of the Spanish *Ministerio de Ciencia e Innovación* through the E-DAMS project and the European Commission thorugh the Realtime project. The work of J. Cotela is funded by the Spanish *Ministerio de Educación* through a doctoral grant in the FPU program. This work is supported by the HiPEAC European Network of Excellence (FP7/ICT 217068), the Ministry of Science and Technology of Spain and the European Union (FEDER) under contract TIN2007-60625 and by the Generalitat de Catalunya (2009-SGR-980).

References

[1] P. Dadvand, R. Rossi, E. Oñate. *An object-oriented environment for developing finite element codes for multi-disciplinary applications.* Archives of computational methods in engineering. Vol. 17, pp. 253 - 297, 09/2010 .ISSN 1134-3060.

[2] G. Karypis and V. Kumar, *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0* , University of Minnesota, Minneapolis, MN, 2009.

[3] M. Heroux, R. Bartlett, V. H. Robert Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams *An Overview of Trilinos*, Sandia National Laboratories, SAND2003-2927, 2003.