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 specially for distributed memory machines (DMMs). Moreover, algorithms and data structures suitable for these platforms can be very different from the ones used in serial code. For this reason, many programmers in the field prefer to start their own code from scratch. However, for an already existing framework supported by a long-time expertise 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 high 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.

From a very global point of view Kratos implements a kernel and application mechanism. Each application acts 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.

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].

2.1 Data Structure

Kratos uses an entity based data structure which means that all variables belonging to an entity are stored together in one block of memory. For example, all data related to a certain `Node` are stored together but there is no guarantee that the blocks of nodal data for different nodes are sequentially stored in memory. The entities with data in Kratos are:

`Node` stores nodal historical and non-historical data, reference position and pointer to its degrees of freedom.

`Element` which has pointer to its nodes and also keeps all elemental data and a pointer to its properties.

`Condition` like the `Element` has pointer to its nodes and keeps its data and properties.

`Properties` is also a block of Kratos' data structure, and serves as a shared data between `Elements` or `Conditions`.

The entities mentioned above are the first layer of data structure. These entities are grouped together in the following classes which construct the superior layers of the data structure:

`Mesh` stores a group of `Nodes`, `Properties`, `Elements`, `Conditions`, generally representing a part of model but without additional solution parameters. It provides access interface to its data.

`ModelPart` holds all data related to an arbitrary part of model. It stores `Mesh` (which has arrays of `Nodes`, `Properties`, `Elements`, `Conditions`) and solution data related to a part of model and provides interface to access them in different ways.

Figure 1 shows the organization of data in Kratos. In this structure `ModelPart` has a very important role. All of the processes and algorithms in Kratos are designed to get a `ModelPart` as their input. This unique position provides a clean way to pass communication information to different procedures in time of parallelization.

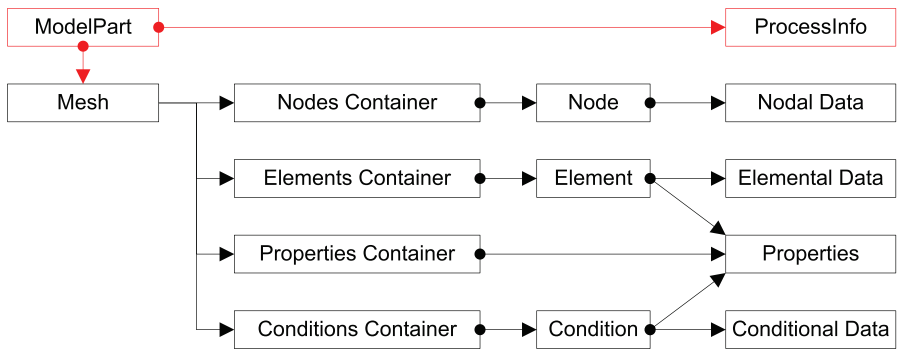

Figure 1: `ModelPart` holds `Mesh` with some additional data referred as `ProcessInfo`

2.2 Classes Encapsulating Solution Algorithms

The solution algorithms in Kratos are encapsulated in the 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. For more flexibility, different steps during solution are deferred to `BuilderAndSolver` and `Scheme`. Figure 2 shows this 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.

Finally, Kratos uses the Python script as its main procedure. This feature significantly improves its adaptability to different solutions and also results to be 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 [2] 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.

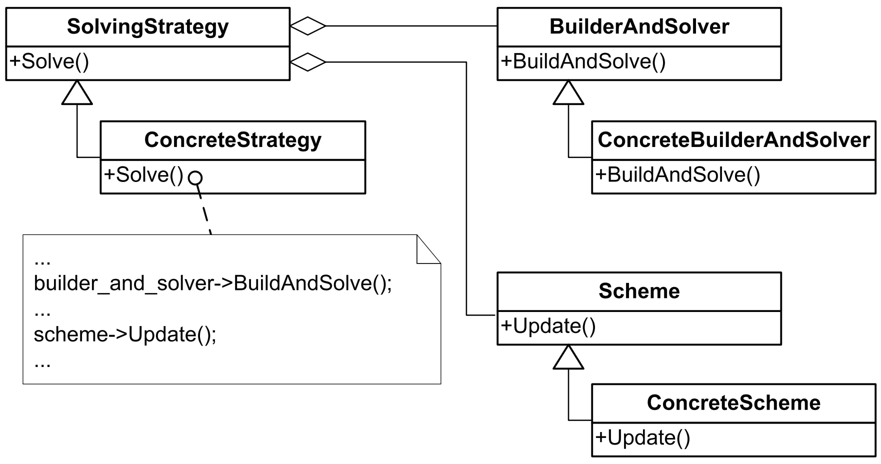

Figure 2: Deferring different parts of the algorithm to `BuilderAndSolver` and `Scheme`. For example in `Solve()` method some steps are deferred to `BuilderAndSolver` via `BuildAndSolve()` method or to the `Scheme` via `Update()` method. The hierarchy in those classes allows the replacement of such steps with another one easily.

3.1 Solver

`LinearSolver` classes were the first part to be parallelized. As mentioned before, the operation used in linear solvers is encapsulated in the `Space` classes. An iterative solver takes the space as their template argument, as shown in the following code:

```
typedef UblasSpace<double, CompressedMatrix, Vector> SpaceType;
typedef UblasSpace<double, Matrix, Vector> LocalSpaceType;

typedef BICGSTABSolver<SpaceType,  LocalSpaceType> BICGSTABSolverType;
```

So, the first step is the implementation of the parallel space that provides the OpenMP parallelized version of matrix and vector operations. Then, just by replacing the Space with a parallel version of it, all iterative solvers in Kratos become parallel without further effort, as can be seen in following code:

```
typedef ParallelUblasSpace<double, CompressedMatrix, Vector> ParallelSpaceType;
typedef ParallelUblasSpace<double, Matrix, Vector> ParallelLocalSpaceType;

typedef BICGSTABSolver<ParallelSpaceType,  ParallelLocalSpaceType> ParallelBICGSTABSolverType;
```

3.2 Build Process

The build process is referred to the part of the solution in which the global system is constructed. The process includes the initialization of the global system, the calculation of the elemental contributions and assembling of these contributions in the global system. Following the parallelization of the linear solvers the build process is parallelized.

The main part of the build process is the calculation of the elemental contributions. This part consists of several operations over elemental data. The performance of the code is optimized by improving the locality of the data and better use of cache, so the overall performance of this part is depend highly to the amount of the cache in the machine. While the elemental calculation is independent for each element, the parallelization of this part is rather easy. As we mentioned before the `BuilderAndSolver` class is in charge of building process. So the main loops `BuilderAndSolver` classes are parallelized, using OpenMP directives. It is important to mention that this approach implicitly enforced the independence of the elemental calculations in each element to the rest.

3.3 Algorithms

In order to complete the parallelization of the code different `Strategy` and `Process` classes were parallelized. 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 respective `BuilderAndSolver` and `Schemes` classes to a parallel version.

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

3.4 NUMA Machines

The shared memory machines are divided in two main categories respect to their memory access mechanism: The Uniformed Memory Access (UMA) and Non-Uniform Memory Access (NUMA) machines. The difference comes from the fact that in UMA machines all CPUs has the same memory access time while in NUMA machines each CPU has faster access to its local memory and slower access to the rest of the memory.

In order to optimize the performance in NUMA machines one has to ensure that each CPU has its process data in its local memory. The OpenMP provides the first touch mechanism for local allocation of the memory. OpenMP allocates each data in the local memory of the CPU that initialize it. So in order to optimize the performance in NUMA machines the initialization part of the `BuilderAndSolver` classes has been modified to ensure the first touch for each CPU as can be seen in following code:

```
int number_of_threads = omp_get_max_threads();
vector<unsigned int> matrix_partition;
CreatePartition(number_of_threads, indices.size(), matrix_partition);

for (int k = 0; k < number_of_threads; k++)
{
  #pragma omp parallel
  if (omp_get_thread_num() == k)
    for (std::size_t i = matrix_partition[k]; i < matrix_partition[k + 1]; i++)
    {
      std::vector<std::size_t>& row_indices = indices[i];
      std::sort(row_indices.begin(), row_indices.end());

      for (std::vector<std::size_t>::iterator it = row_indices.begin();
           it != row_indices.end(); it++)
        A.push_back(i, *it, 0.00);

      row_indices.clear();
    }
}
```

It is important to mention that the above algorithm will be run in serial which permits the use of `push_back` method. Nevertheless it is executed once at the beginning of the calculation to ensure that each processor allocate its data in its local memory. All these improvements increases the speed-up 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 applications implement only new elements and conditions using standard Strategies or provided `BuilderAndSolver` and `Schemes`. One can observe that many applications become 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. The solution algorithms used are based on standard domain decomposition approach [3, 4] and are implemented using MPI library [5]. In this process, the main objective is to have the same code for serial and parallel versions, and also to

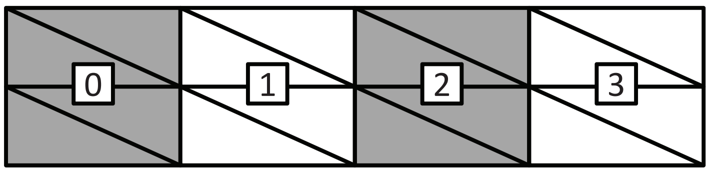

Figure 3: Example of a mesh divided in four partitions.

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 new `Communicator` and `MPICommunicator` classes are implemented, which are in charge of transparent data transfer.

4.1 Partitioning

Following a standard domain decomposition approach, the first step is to partition the domain efficiently. To this end, the METIS [6] 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.

The first step after partitioning process is obtaining the graph of the domains in form of a matrix representing the interface between domain i and domain j with non-zero value in its element a_{ij}. For example for the partitioning shown in figure 3 is:

$$\mathbf{G} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Then this graph matrix is passed to a coloring procedure to determine a sequence of communications between domains which 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. The restriction is the fact that each process can communicate with one other process at each step. For example for partitions P_0, P_1, P_2, P_3 shown in figure 3 the data transfer between P_0 and P_1 can be done simultaneously with the data transfer between P_2 and P_3. So the coloring gives 0 to these transfers and 1 to the data transfer between P_1 and P_2 as shown in figure 4. This means that P_2 will communicate first with P_3 and then in the next step with P_1. Without coloring the latency would increase while in the first step the P_2 wants to communicate with P_1, but has to wait until the data transfer between P_1 and P_0 finishes P_1 becomes available as can be seen in figure 5.

The result of the coloring is a matrix $\mathbf{C}$ where each row i contains the sequence of neighbours to be communicated. For example the coloring matrix for example of figure 3 is:

$$\mathbf{C}_{n_p \times n_c} = \begin{bmatrix} 1 & -1 \\ 0 & 2 \\ 3 & 1 \\ 2 & -1 \end{bmatrix}$$

where the n_p is the number of partitions and n_c is the number of colors. Each element $\mathbf{C}_{ij}$ holds the process MPI rank to be communicated from process i in step j of communication and value -1 stands for no communication in this step.

In general the partitions may have overlapping. In our example the overlapping only exists in nodes of interfaces which are duplicated but this can be extended to other entities. However, according to do-

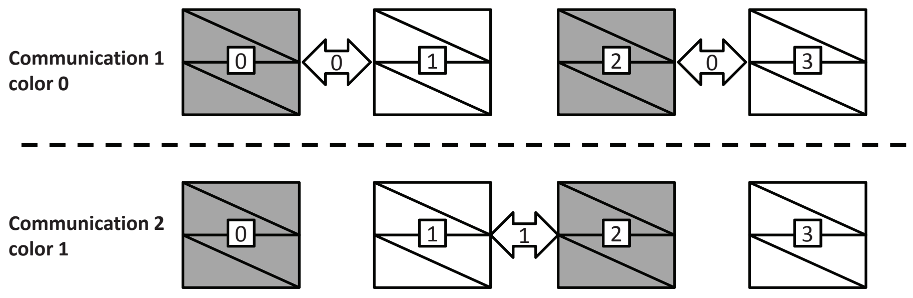

Figure 4: The coloring for the four partitions shown in figure 3.

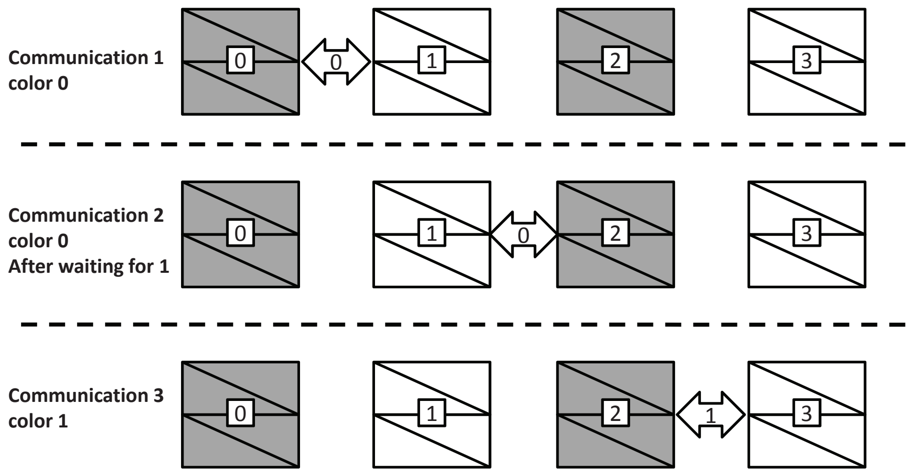

Figure 5: Not optimum coloring causes higher communication time.

main decomposition methodology used, it is convenient to have each entity only belong to one partition and marked as *Local* and mark its duplicated ones in other partitions as *Ghost*. Assigning these categories simplifies the synchronization where the data from each local entity is copied to all its duplicated ghosts.

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:

```python
# we import the metis application from input
# script only in machines where we have
# compiled this application.
from KratosMetisApplication import *
```

Thus, simply by changing the Python script, the same code with minimal changes can be used for both SMMs and DMMs.

4.2 `Communicator` Class

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 coloring. The ith element of this vector contains the MPI rank of the neighbour to be communicated in step i of communication and -1 if there is no process to communicate in this step. In fact, this vector is the row i of the coloring matrix $\mathbf{C}$.

`LocalMesh` Entities that belong to this domain. Local mesh holds all nodes, elements and conditions that are belong to this domain even if they are not directly in the interface.

`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 where i is the color of the neighbours of this domain.

`GhostMesh[i]` Contains entities which are a duplicate of the entities in neighbour domain i where again i is the color of the neighbours of this domain.

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

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 sets the result in the local and the ghosts.

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

In order to increase the extensibility of the design, a new interface is designed for `Synchronize` and `Assemble` methods. The information of variable to be synchronized or assembled is passed to these methods by an additional parameter and there are overloaded versions to operate over all variables as can be seen in following code:

```
// This will assemble only the NODAL_AREA which is a double
strucutral_model_part.GetCommunicator().AssembleCurrentData(NODAL_AREA);

// This will assemble NORAML which is a vector
strucutral_model_part.GetCommunicator().AssembleCurrentData(NORMAL);


// This will synchronize all variables
fluid_model_part.GetCommunicator().Synchronize();
```

In this way any new algorithm with new variables can reuse the existing communicator without any modification. This feature significantly increases the extensibility and generality of the design.

The `Communicator` class only provides the interface to these methods with an empty version of them and the implementation for MPI is delegated to its derived class `MPICommunicator` which will be described in following section.

4.3 MPI Communication

The `MPICommunicator` class derives from `Communicator` and implements its methods for using MPI as follow:

`Synchronize` The implementation consists in loop over `NeighbourIndices` and filling a buffer with data of local entities stored in `LocalMesh[i]` to be sent and also allocating a buffer with ghost entities stored in `GhostMesh[i]` for receiving data. Then performs the send and receive using MPI and finally copies back the received buffer to the ghost entities by performing a loop over entities stored in `GhostMesh[i]`. The following pseudo-code shows this procedure:

```
for(i_color = 0 ; i_color <  neighbours_indices.size() ; i_color++)
  // if there is a neighbour for this color
  if((destination = neighbours_indices[i_color]) >= 0)
  {
    NodesContainerType& local_nodes = LocalMesh(i_color).Nodes();
    NodesContainerType& ghost_nodes = GhostMesh(i_color).Nodes();

    send_buffer_size = local_nodes_size * nodal_data_size;
    receive_buffer_size = ghost_nodes_size * nodal_data_size;

    //Allocating buffers
    double* send_buffer = new double[send_buffer_size];
    double* receive_buffer = new double[receive_buffer_size];

    // filling the send buffer
    for(i_node = local_nodes.begin(); i_node != local_nodes.end(); ++i_node)
      copy_nodal_data_to_buffer(i_node, send_buffer);

    // performing the send and receive
    MPI_Sendrecv( ... )

    // Updating nodes
    for(i_node = ghost_nodes.begin() ; i_node != ghost_nodes.end() ; i_node++)
      copy_buffer_to_nodal_data(i_node, send_buffer);

    delete [] send_buffer;
    delete [] receive_buffer;

  }
```

It can be observed that the stored communication information is designed to optimize the performance of communication by avoiding any search for related entities in each synchronization.

`Assemble` The implementation of this method is very similar to the `Synchronize` method with two differences: first, the loops for allocate and filling buffer are done over `InterfaceMesh[i]` instead of `LocalMesh[i]` and `GhostMesh[i]`; second, we have to keep the received buffer for each color until all communication steps are finished and then update the nodes to avoid accumulative sum between different copies.

`MaxAll`, `MinAll`, `SumAll`, etc. These methods are implemented by calling the corresponding MPI function.

It is important to mention that having the empty version of above methods in the `Communicator` class and their MPI implementation in the `MPICommunicator` gives the very important advantage that one can use the same code for SMM and DMM machines. Consider the following sample code to calculate nodal area for a triangular mesh:

```
double area = 0.0;
for(i = mr_model_part.ElementsBegin();i != mr_model_part.ElementsEnd(); i++)
{
  // calculating the area of the element
  area = i->Area();
  area *= 1.00 / 3.00;

  // adding one third of the area to the nodal area of each node
  i->Node(0)[NODAL_AREA] += area;
  i->Node(1)[NODAL_AREA] += area;
  i->Node(2)[NODAL_AREA] += area;
}

// to have the correct result in parallel we have to
// assemble the results obtained in each partition
model_part.GetCommunicator().AssembleCurrentData(NODAL_AREA);
```

In a serial run, the code uses the `Communicator` class, where `AssembleCurrentData` is an empty method and does nothing. In a parallel run, the `MPICommunicator` will be used, (by setting the model part's communicator via input script to `MPICommunicator` in run time) and the assembling process will be performed using MPI, without needing to customize the application for each platform.

Another observation is respect to the interface of these methods which is considered to be very clean and completely hides the MPI and partition information from the user, which was one of the main goals in the design. The interface is also very extensible to new variables, which conforms with requirements for the multi-physics design of the code. It lets new variables to be synchronized without any modification in the communication part. This high level of extensibility is achieved by the innovative way of passing the variable information to the communicator through an argument and using the C++ features to guarantee the type-safety and robustness of the design.

4.4 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 [7] library. This library provides a very good performance while providing a very clean interface in comparison with other similar libraries.

5 Benchmarks

5.1 An initial SMM example

A first scalability test was run on a node in a local cluster at CIMNE containing two Intel Xeon E5645 CPUs (2.4GHz). Each CPU has 6 cores with 12MB L3 cache, shared between all cores and each

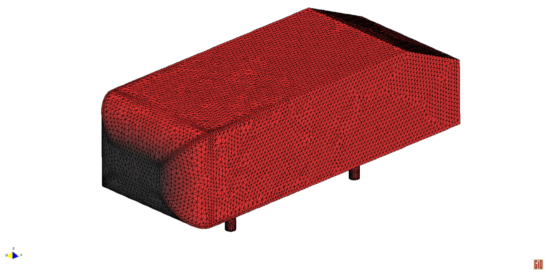

Figure 6: Representation of the Ahmed's body benchmark.

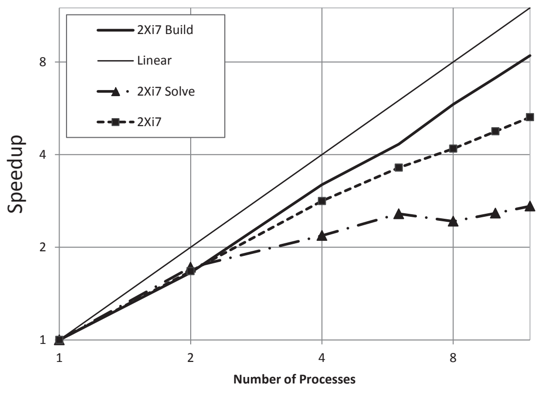

Figure 7: Speedup for the OpenMP example

core has $32+32kB$ L1 cache and $256kB$ L2 cache. This test was performed using the geometry of a widespread benchmark problem for turbulent flows around bluff bodies called Ahmed's body, pictured in Figure 6. The interested reader is encouraged to consult the original publications on the subject, such as [8] and [9] for the details of its geometric definition (our example was computed on the variant with a 25^o rear angle).

This example was simulated using an incompressible Navier-Stokes solver that has been recently implemented in Kratos, described in [10]. This solver is based on the stabilization of the Navier-Stokes equations using the algebraic subgrid scale method (see [11]) and a pressure Schur complement based approach (see [12]) to the separation of velocity and pressure unknowns. Note that this solver does not include a turbulence model and therefore does not account for turbulent dissipation. Obviously, the results obtained with this kind of formulation can not be expected to coincide with the experimental ones, but they should at least agree qualitatively. In any case, this issue is secondary, as our present aim is to verify the parallel performance of the code.

The model was described using a mesh of 344882 nodes and 1699019 elements. The simulation is performed using 1, 2, 4, 8 and 12 OpenMP processes, and its parallel performance is reflected in the results presented by Figure 7.

As can be seen in the figure, the total speedup obtained is not impressive, but a more detailed analysis reveals the causes of this result. If we concentrate our atention on the time spent in finite element operations, including the computation of the local finite element matrices and its assembly to compute the global matrices, labelled as *build* in Figure 7, it can be seen that they scale almost linear. By comparison, the time spent solving linear systems, labeled as *solve* in the figure, has a significantly worse parallel performance, suggesting that the lack of scalability is due to the specific linear solver used for this example. This suspicion is reinforced by the results obtained in the following distributed memory benchmarks, which exhibit a better performance.

5.2 DMM test cases

We have performed the evaluation of the Kratos migration on the Marenostrum Supercomputer at the Barcelona Supercomputing Center [13]. Marenostrum is built using dual-core PowerPC 970MP processors ($2.3GHz$). 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. Marenostrum nodes are linked through a Myrinet network. 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 first example on Marenostrum uses the same geometry of the Ahmed's body benchmark pre-

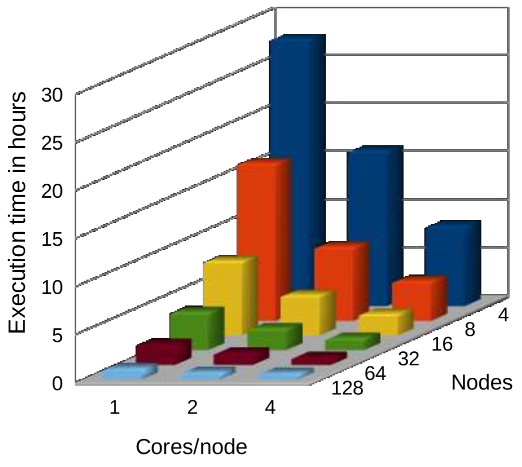

Figure 8: Execution time results for the Ahmed experiment

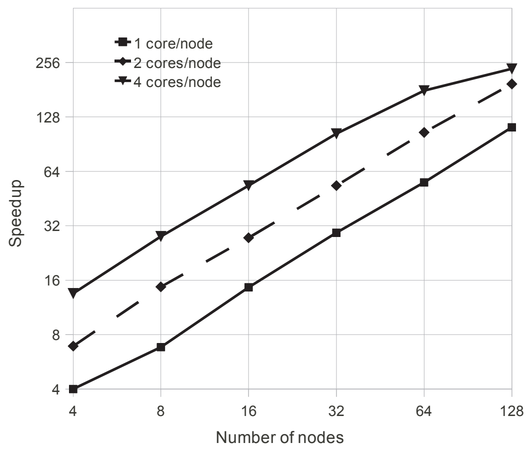

Figure 9: Speedup achieved on the Ahmed problem

sented in the previous example, this time using a finer mesh containing 1.7 million elements and 345 thousand nodes. A different incompressible Navier-Stokes solver was used in this case, based on the fractional step procedure (see for example [14]), again without any modification to account for turbulent dissipation.

Figure 8 shows the execution time in hours obtained in the computation phase of the Ahmed experiment. Figure 9 shows the corresponding speedup. This experiment was run in Marenostrum using 1 to 4 cores per node, and from 4 to 128 nodes. As it can be seen, Ahmed maintains an almost linear scalability when using up to 64 nodes. Note that no matter the cores belong to the same node or not (i.e. in the combinations of 64x1, 32x2, 16x4), an equivalent scalability is achieved. When using 128 nodes, and only when using 1 and 2 cores per node, the solver scales at the same ratio. At 128x4 the efficiency decreases: at this point, the data distribution of 1.7 million elements across 512 cores (=128x4), leaves only 3125 elements per core. The time spent on the computation of the time step is then around 0.5 seconds. This is now very similar to the communication time spent with this amount of cores, which is around 0.6 seconds. In this situation, the benefits obtained through parallelism in the computation part do not compensate the time spent in the communication part, and this is the reason to have a reduction in the efficiency of the execution.

After the Ahmed's body benchmark, we have performed additional experiments using as input a model named Telescope. This model was originally designed to compute the turbulent air flow on the site of the Canaries Great Telescope in the island of La Palma in the Canary Islands, Spain. Its geometry can be observed in Figure 10, but the interested reader is directed to [15] for details on the original problem this model was designed for and additional details on its geometry. For our purposes it is sufficient to mention that the problem has 24 million elements and that we used the same fractional step formulation for the fluid as in the previous example.

Figure 11 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 almost linearly when using up to 1024 processors. However, the speed-up of the performance starts to fall above 1024 processors. In this case, the problem declines it scalability when the elements per core are less than 24,000 (in the 1024-core experiment). It is also when the amount of communications becomes too big compared to the computation time for each time step. Even that there are still 11,700 elements per core in the 2048-core configuration, the increased size of the interconnection network used and the unbalances among cores cause the performance penalty.

Load balancing and network performance are the two factors we have determined that allow to get

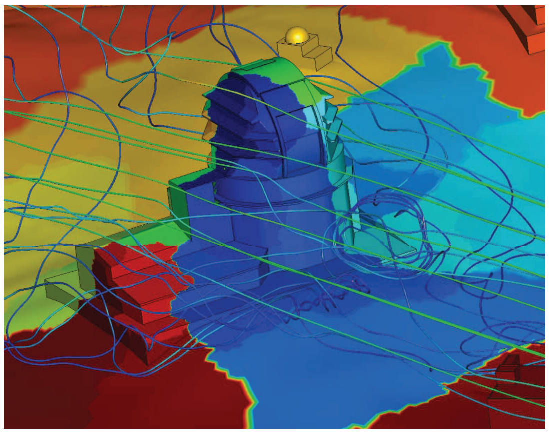

Figure 10: Snapshot of the Telescope solution

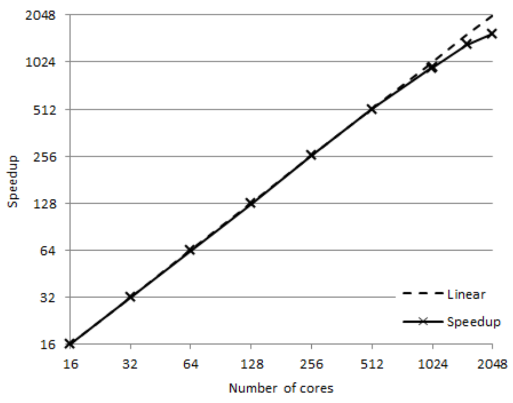

Figure 11: Speedup achieved on the Telescope problem

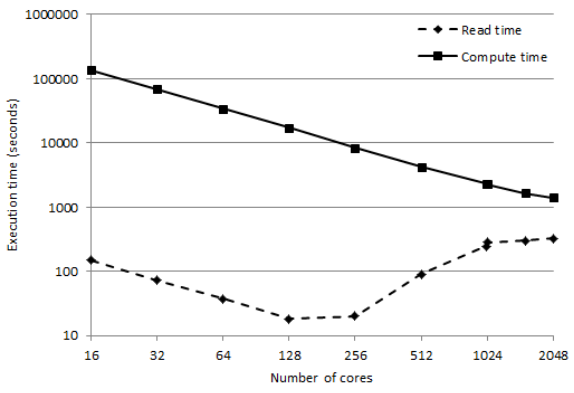

Figure 12: Comparison of read vs. computation times

good scalability up to 512 cores. With the METIS partitioning applied to the problem, each processor core spends a similar amount of time in the computation phase. This avoids the loose of performance that happens because some processor cores are still working on the computation phase, while others are idle waiting for the former to finish. Network performance is equivalently important. With the METIS partitioning, there is a single phase for data exchange. As the node count grows, the amount of messages sent from each node also grow. The Marenostrum network (Myrinet) supports well the amount of data transferred for the Telescopio problem up to 512 nodes. Above that, the amount of messages has increased up to a point that the communication time is in the same order of magnitude that the computation time. The performance of the network cannot be further improved, and it becomes the bottleneck to achieve scalability. Even though the computation time of the computation phase still decreases with the increase of the number of cores.

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 a better overall scalability is achieved, which is very important for the scientists interested in the solutions given by Kratos to their problems. This can be seen in Figure 12, as the sequential reading phase takes nearly a constant execution time across the number of nodes, while the parallel execution time of the computation keeps decreasing.

Conclusion

We have proposed a modular way to migrate a complex multi-physics framework to high performance platforms. The proposed way takes advantage of some particular specifications of the framework but can be extended to migrate other codes with similar design characteristics.

Most of the required implementation involves the `BuilderAndSolver` and `Communicator` classes and few modifications has to be made in the rest of the code. In this way we avoid introducing new errors in already tested part of the code due to parallelization. Most of the applications in Kratos are migrated to SMMs and even DMMs without any modifications which shows the effectiveness of the approach used. Also we like to emphasize the large impact of using external libraries for partitioning and linear algebra in reducing the migration effort.

An innovative interface for communication is presented with clear and very extensible results. All the partitioning information is hidden in standard operation but still available for advanced operations. The interface can extended to any new variable without problem which makes it suitable for the multi-physics nature of the framework.

The Kratos framework has been successfully migrated to the different high performance platforms by using the presented approach. The migration is tested with benchmarks over different platforms. The scalability of the system generally depends on the model size and hardware, specially to memory bandwidth. For SMMs the build process scales as expected but linear solvers suffer from limitation in memory bandwidth. For DMMs the size of the problem also becomes important. The reason is the fact that the communication depends to the area of the partition while calculation depends to its volume. So with less element there is more surface respect to the partition volume and increases the importance of the data transfer time.

The implementation of the hybrid OpenMP-MPI is considered to be the next step in order to adapt the code to modern clusters with multi-cores nodes.

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. J. Cotela gratefully acknowledges the support of the Spanish *Ministerio de Educación* through a doctoral grant in the FPU program and the *Col·legi d'Enginyers de Camins, Canals i Ports*. 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). The author thankfully acknowledges the computer resources, technical expertise and assistance provided by the Red EspaÃśola de Supercomputación.

References

[1] Pooyan Dadvand, Riccardo Rossi, and Eugenio Oñate. An Object-oriented Environment for Developing Finite Element Codes for Multi-disciplinary Applications. *Archives of Computational Methods in Engineering*, 17:253–297, 2010.

[2] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: portable shared memory parallel programming*. The MIT Press, 2007.

[3] Barry F. Smith, Petter Bjørstad, and William Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[4] Alfio Quarteroni and Alberto Valli. *Domain Decomposition Methods for Partial Differential Equations*. Oxford University Press, Oxford, UK, 1999.

[5] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[6] Vipin Kumar George Karypis. *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0*, 2009.

[7] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An overview of trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[8] S. R. Ahmed, G. Ramm, and G. Faltin. Some salient features of the time-averaged ground vehicle wake. *Scientific Programming*, 1984.

[9] Hermann Lienhart and Stefan Becker. Flow and turbulence structure in the wake of a simplified car model. *Society of Automotive Engineers*, 112:785–796, 2003.

[10] Jordi Cotela, Riccardo Rossi, Eugenio Oñate, and Pooyan Dadvand. A comparison of different parallel techiques applied to the solution of the navier-stokes equations. In P Iványi and B. H. V. Topping, editors, *Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*. Saxe-Coburg Publications, April 2011.

[11] Ramon Codina. A stabilized finite element method for generalized stationary incompressible flows. *Computer Methods in Applied Mechanics and Engineering*, 190(20-21):2681–2706, 2001.

[12] S. Turek. *Efficient solvers for incompressible flow problems: an algorithmic and computational approach*. Springer, Berlin, 1999.

[13] Barcelona Supercomputing Center. Marenostrum. http://www.bsc.es.

[14] R. L. Sani P. M. Gresho. *Incompressible flow and the finite element method*. Wiley, 2000.

[15] Ramon Codina, Joan Baiges, Daniel Pérez-Sánchez, and Manuel Collados. A numerical strategy to compute optical parameters in turbulent flow. application to telescopes. *Computers & Fluids*, 2009.