CHARACTERIZATION OF APPLICATIONS IN NEW ARCHITECTURES

[Amoto quaeramus seria ludo]
CHARACTERIZATION OF APPLICATIONS IN NEW ARCHITECTURES
# Table of contents

1. Introduction .................................................................................................................. 5  
   1.1. Numerical applications .......................................................................................... 7  
   1.2. Supercomputing .................................................................................................... 10  
   1.3. Motivation and goals ............................................................................................ 14  
   1.4. Initial planning and economic cost of the PFC ..................................................... 15  
2. Kratos and Mare Nostrum .......................................................................................... 17  
   2.1. Kratos .................................................................................................................. 17  
   2.2. Mare Nostrum .................................................................................................... 21  
3. Learning more about Kratos ....................................................................................... 25  
   3.1. Kratos structure .................................................................................................... 25  
   3.2. PureConvectionEdgeBased .................................................................................. 26  
      3.2.1. EdgeStructureType ..................................................................................... 26  
      3.2.2. MatrixContainer ......................................................................................... 27  
      3.2.3. PureConvectionEdgeBased ......................................................................... 28  
   3.3. Installation in a laptop ......................................................................................... 29  
4. Kratos application requirements ................................................................................. 32  
   4.1. Necessary libraries and utilities ........................................................................... 32  
      4.1.1. System libraries ............................................................................................ 33  
      4.1.2. Compilers .................................................................................................... 34  
      4.1.3. Interpreters .................................................................................................. 35  
      4.1.4. Basic additional libraries ............................................................................. 35  
      4.1.5. Main libraries ............................................................................................... 37  
   4.2. Library dependencies ............................................................................................ 38  
5. Migrating Kratos to Mare Nostrum .......................................................................... 39  
   5.1. Configuration environment ................................................................................. 40  
   5.2. Build executable ................................................................................................... 41  
   5.3. Run ....................................................................................................................... 41  
   5.4. Scalability ............................................................................................................ 42  
   5.5. Costs .................................................................................................................... 44  
6. Data collection and comparative .............................................................................. 45  
   6.1. Kratos execution flow ........................................................................................... 46  
   6.2. Ahmed_25 .......................................................................................................... 49
6.3. TELESCOPIO………………………………………………………………………………………………………51
7. Performance improvements after migration…………………………………………………………………………55
  7.1. It’s the software, stupid………………………………………………………………………………………………55
  7.2. Standards not accomplished…………………………………………………………………………………………58
  7.3. GNU less restrictive than IBM…………………………………………………………………………………………59
  7.4. IBM XL bugs………………………………………………………………………………………………………………60
8. Final planning and economic cost of the PFC…………………………………………………………………………63
9. Conclusions……………………………………………………………………………………………………………………65
10. Future extensions of the work done……………………………………………………………………………………68
11. Appendix ………………………………………………………………………………………………………………………70
  11.1. Pseudo code of Kratos …………………………………………………………………………………………………70
    11.1.1. EdgeStructureType…………………………………………………………………………………………………70
    11.1.2. MatrixContainer…………………………………………………………………………………………………74
    11.1.3. PureConvectionEdgeBased…………………………………………………………………………………………82
  11.2. Laptop’s /proc/cpuinfo …………………………………………………………………………………………………88
  11.3. env64.sh……………………………………………………………………………………………………………………..90
  11.4. do-configure………………………………………………………………………………………………………………91
  11.5. Jamroot…………………………………………………………………………………………………………………….93
  11.6. kratos-submit.sh …………………………………………………………………………………………………………..96
  11.7. Compilers and their flags ………………………………………………………………………………………………98
  11.8. Automatic installation script…………………………………………………………………………………………..100
  11.9. Snapshots of Ahmed_25 ……………………………………………………………………………………………..106
  11.10. Snapshots of TELESCOPIO…………………………………………………………………………………………108
  11.11. Snapshots of Ferrari …………………………………………………………………………………………………109
Acknowledgements ………………………………………………………………………………………………………………110
References …………………………………………………………………………………………………………………..111
1. Introduction

Computer science is continuously evolving to improve the development of applications, and to allow programmers to achieve better productivity. One key issue to solve is the ability to reuse the work previously done by others. Currently, the same or similar algorithms and libraries are used in many different kinds of applications (weather forecasting, physics simulations, artificial intelligence decision, entertainment programs, etc.), despite of which kind of input data is processed and how [1].

Users and programmers deal with bigger problems and designers have more logic gates than before. Three main kinds of concerns have been appearing for them [2]:

- How to solve large problems that run too slowly even on the fastest computer, as many scientific and engineering applications do?
- How to obtain solutions to problems that could be done on one of today’s fastest computer when the budget is limited?
- How to increase programmer productivity?

It is becoming increasingly difficult to obtain more performance from von Neumann model, because many of the technological constraints that influenced its design over 40 years ago have been surpassed [2]. This old but revolutionary model at the beginning of computer science was based on a serial processor. Arguments for processing a single instruction at a time have been overcome and nowadays even commercial processors are parallel (dual core and even quad core). Hardware technology improvements repealed Grosch’s law [50], which had stated the best cost/performance was obtained with the most powerful uniprocessor. Therefore, it is not true that “a collection of smaller processors will always have less performance than a single large processor of the same total cost” as Grosch stated decades ago.

That is why, in recent years, better performance is a basic driving force also in parallel processing research. The final goal is to have a large collection of processing elements cooperating to solve large problems in a faster way [2].

In short, parallel processing is perceived as having the potential to improve performance,
cost/performance ratio and productivity, which are directly linked with the three main concerns asked above. But in order to achieve this, we must know which elements are involved: problem we want to solve, algorithm we want to use, high level language we are going to code with and platform we have with its operating system and architecture of its own [2]. The sum of algorithm and its implementation in a high level language give us an application and the sum of architecture and its operating system gives a platform.

![Diagram of computing system elements and interrelationships](image)

**Figure 1:** Key elements of a computing system and their interrelationships

Another important part of an architecture is how memory and I/O (Input/Output) are managed, because it is how all processing elements communicate. About memory, the question is whether each processor should store its own program or whether each processor should have exclusive access to its part of the data. There are three main possibilities or points of view which are explained below: SIMD, MIMD with private memory and MIMD with shared memory.

So, at this point, we know there are two main elements: application, which demands its necessities, and platform, which covers those necessities. The increasing number of requirements are the reason that only one processor is not enough for high performance applications, forcing processors to cooperate in a synchronized way. For this, there is the idea of granularity [2]: a program must be divided in subtasks (a.k.a. threads) which need to communicate to exchange data and coordinate their activities in order to distribute workload and improve application performance. Also, the time spent on such communication should be
small compared to the time spent executing the subtasks. In fact, it is a divide-and-conquer approach to solve the problem.

For example, one of the limitations of an application is the size of data input. Sometimes, problem complexity requires high amount of memory and computation elements. In these situations, applications should be executed in High Performance Computer (HPC) platforms, or supercomputers, as explained below.

So, when those necessities and requirements are high enough, migration of the application to a new and more powerful platform is needed, with inappreciable modifications or without any changes either in the algorithm or in the source code. Throughout this document we show how to achieve that goal and what techniques, resources and steps have been chosen around application and platform – Kratos and Mare Nostrum would be the main elements in this document– in order to help make easier future migrations, either with others applications on similar or different platforms.

In this Section we present all theory material that will make easier for you to understand further Sections. In Second Section real application and platform selected will be shown. In Third Section first step, learning will be detailed and in Fourth Section we present and see the application’s inside. Fifth, Sixth and Seventh Section are about effective migration, execution results and improvements tools used respectively. Rest of the document is dedicated for conclusions, economic costs of the project, future extensions or projects using the work done in present document and appendixes.

1.1. Numerical applications

Numerical applications are those applications applied to solve complex and difficult problems through an algorithm. As more powerful hardware platforms have appeared during last decades, numerical applications have grown with same or more proportion in necessities, features and accuracy. Fortunately, new computer science techniques have improved the architecture and use of hardware; so, consequently, applications have been adapted to these new and more efficient platforms.
Normally, numerical applications rely greatly on the availability to scale in distributed memory machines. This is due to the amount of memory needed when data input increases. Creating a highly parallelizable code is a challenge in terms of algorithms and structure, as well as the resource management required to obtain a good performance. Usually, an intermediate step is needed since the application must migrate to a different platform that is often quite different as regards hardware and software.

The problem to be solved can be viewed as an algorithm applied to a set of data points, and its domain can be divided into parts and assigning each part to a different processor, using the computer science technique known as domain decomposition or data parallelism when working in parallel. These applications are usually solved by numerical finite elements method, often following Laplace’s or Poisson’s equation.

For example, in weather simulation, the weather over California evolves independently from that over Indiana. For short time extrapolations, they can be simulated on separate processors through data decomposition technique and, eventually, information flows between these sites and their dynamics are mixed through their data set boundaries. The weather in California would be represented as a set of data points and the weather in Indiana would be another set of data points. Of course, communication of data between the processors (e.g., through shared memory) is what makes this eventual mixing, updating data of each data set every time it is needed and synchronizing all processors (see Figure 2) [6].
Two simulation methods rise from application point of view: time-stepped and event-driven simulations, both parallelizable. Time-stepped simulations consists in solving the problem temporally, recalculating the problem every time a time step is done and synchronizing all processors (e.g., in weather forecasting a time step would be a fraction of time, one minute, one hour and so on). On the other hand, event-driven simulations consist in simulating systems where objects interact with events (e.g., in military tactics an event would be a tank hit from behind).

Other classification divides applications depending on how is shared the workload [6]:

- **Synchronous.** Each point can be evolved in point-by-point synchronous mode, as is on a SIMD architecture (explained below).

- **Loosely synchronous.** Temporal synchronization is on a subdomain. Most of today’s major applications are of this type, because finite element problems and finite difference codes with adaptive meshes are loosely synchronous. In fact, domain decomposition (see previous figure) has this structure, as does a fast multiple approach. MIMD architecture (explained below) fits really well for this method.

- **Asynchronous.** Event-driven simulations fall into a class that includes problems not formulated in terms of a stepped time or iterator that is associated with each system entity. In fact, asynchronous problems can be very hard to parallelize. One example is The Joint Theater Level Simulation [69].

- **Pleasingly parallel.** The time or iteration evolution structure of a problem can greatly impact the appropriate software and hardware architecture, using a cluster or farm with worker nodes that receive chunks of the simulation to do as they finish their previous assignments. A good example of this is the search engine used by Google [51].

The algorithm of the numerical application can be transformed into a parallel program which consists of a loop over iterations divided into two phases: communicate (at the start of each iteration, any outside data values needed to update the data values owned by a processor must be communicated) and compute (each processor operates without the need to synchronize with other machines and updates its data values) [6].
Synchronous and loosely synchronous problems perform well on large parallel machines if the problem is large enough. For a given machine, there is a typical subdomain size above which one can expect to get good performance. If problem is scaled with fixed subdomain size, there will be a certain ratio of parallel speedup to $N_{\text{proc}}$. So, the majority of large-scale scientific and engineering codes can be parallelized [6].

For example, one method to do this is Jacobi’s method for Poisson’s equation. If we ignore some issues as bus or switch contention, we obtain next equation for speed up developing Amdahl’s law:

$$S(N_{\text{proc}}) = N_{\text{proc}} \cdot \left(1 - \frac{2}{\sqrt{n \cdot N_{\text{proc}}}}\right)^2; \quad n = \#\text{points updated per processor}$$

If we add communication overhead to the formula we get next equation:

$$S(N_{\text{proc}}) = N_{\text{proc}} \cdot \left(1 - \frac{2}{\sqrt{n \cdot N_{\text{proc}}}}\right)^2 \cdot \left(1 + \frac{t_{\text{comm}}}{\sqrt{n \cdot t_{\text{float}}}}\right)$$

We can simplify all this in the equation:

$$S(N_{\text{proc}}) = \varepsilon \cdot N_{\text{proc}} = \frac{N_{\text{proc}}}{1 + f}; \quad \varepsilon = \text{efficiency}; f = \text{overhead}$$

Last equation gives us a simple tool to evaluate the efficiency of applications in a particular platform or the associated overhead once we have calculated its speedup.

Anyway, algorithms of the applications sometimes cannot be migrated to all platforms and their portability sometimes is elusive [6], so migration without any changes might not be possible in some occasions; and, in few cases, migration is simply not possible. Their scientific models should be reformatted when porting to a parallel mechanism.

1.2. Supercomputing

Since the creation of the first computer, complexity and resource requirements of the applications have not stopped to grow. In consequence, faster computers and more efficient
algorithms and techniques are necessary as we have shown above. The fastest computers are known as supercomputers or HPCs which are really powerful machines beyond any common computer.

From the software point of view, it is possible to improve the performance of applications through languages with their programming models, compilers and interpreters. Choose the most suitable language for an application can affect greatly the performance of final program. So, what kind of languages can be found and chosen?

- **Compiled** languages are those whose source code is transformed to an executable binary file like C [3].
- **Interpreted** languages are those whose source code is read by an executable who interprets and executes all commands line by line like Maple [4] or Python [31].

Compiled languages normally have faster executions than interpreted languages because they are more efficient.

As said above, we can improve applications through compilers and interpreters, the program “translators”:

- A **compiler** is a computer program that transforms source code written in a programming language into another computer language (often having a binary form known as object code), in order to create an executable program [67]. A compilation process has seven stages [2]: lexical analysis, syntax analysis, intermediate code generation, code optimization, code generation, table management and error handling. Some compilers can look for parallelism in order to automatically improve the performance of the program –as commented above- that is being compiled in code optimization stage using vectorizers, analyzing dependency graphs and etcetera. Nevertheless, some of these automatic optimizations must be activated through compiler flags.
- An **interpreter** normally means a computer program that executes instructions written in a programming language [68], casting a command after another until the program finishes or crashes.
Also, it is important to mention OSs (Operating System) because their tasks can affect the performance of programs [2]. But numerical applications do not do many I/O explicit petitions to the OS, they only do implicit ones. However, this is not the goal of this project and, thus, the OS is not taken in consideration.

Finally, we can make two different approaches to parallelize the applications [2]: automatic parallelization which means the programmer does not have to modify the program or learn any new language (compilers are who have to work in this case) and parallel extensions that means the programmer must learn an extension to a language that he or she already knows (like OpenMP [5]).

From the hardware point of view, we can classify advances in Table 1 [2]:

<table>
<thead>
<tr>
<th>Level</th>
<th>Technological advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Instructions are divided in stages <em>(fetch, decode, execute, memory and write back)</em></td>
</tr>
<tr>
<td>CPU</td>
<td>Multi core processors (more than 1 core by chip):</td>
</tr>
<tr>
<td></td>
<td>* dual core and quad core *</td>
</tr>
<tr>
<td>Memory</td>
<td>Symmetrical processors, that is, which shares through a bus the same memory,</td>
</tr>
<tr>
<td></td>
<td>NUMA <em>(Non-Uniform Memory Access)</em>, NORMA <em>(NO Remote Memory Access)</em>,</td>
</tr>
<tr>
<td></td>
<td>COMA <em>(Cache-Only Memory Architecture)</em>...</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Ways to connect processors: cluster, MPP <em>(Massive Parallel Processing)</em>, grid</td>
</tr>
<tr>
<td></td>
<td>computing, omega networks...</td>
</tr>
<tr>
<td>Specialization</td>
<td>Specialized processors and/or architectures:*</td>
</tr>
<tr>
<td></td>
<td>FPGA <em>(Field Programmable Gate Array)</em>, GPGPU <em>(General Purpose computing on</em></td>
</tr>
<tr>
<td></td>
<td>Graphics Processing Unit)*, SIMD <em>(Single Instruction Multiple Data)</em>, MIMD</td>
</tr>
<tr>
<td></td>
<td><em>(Multiple Instruction Multiple Data)</em>, ASIC <em>(Application Specific Integrated Circuit)</em>,</td>
</tr>
<tr>
<td></td>
<td>SPU <em>(Synergistic Processing Unit)</em>...</td>
</tr>
</tbody>
</table>

*Table 1: Technological advances sorted by approach*

Most of these advances have recently appeared. For example, we could say 20 years ago specialization only consisted in SIMD, MIMD and a combination of both (nowadays these continue being the major computational models) [2].

Thanks to SIMD and MIMD architectures, supercomputers achieved GFLOP *(Float Operation)* level. Current supercomputers easily achieves TFLOP and PFLOP level as Moore’s law stated [6] and it is shown by Top500 webpage which is updated twice a year from 1993 [7]. This means every year computer benefits increase, at least parallel architecture continues to be an active area of research [6].
SIMD and MIMD are computational models based on processor distribution and its interconnection:

- **SIMD computational model** corresponds to a single stream of instructions, each of which is applied to multiple data items. It means that a single instruction causes the execution of identical operations on multiple pairs of data; this is why SIMD, vector processors and array processors are considered synonymous. But this is not exactly true because vector processors have replicated many times part of its calculation hardware, while SIMD processors are pipelined and much cheaper [2].

- On the other hand, we have **MIMD computational model** that has two branches and is like an improvement of SIMD architecture. Basically, in SIMD architecture, if we expand each processor’s memory so that it can hold a substantial number of instruction data, we get MIMD architecture. This creates some new concerns about applications when they are broken in subtasks because now every process can manage its own memory. In consequence, all executing processors need to be synchronized or coordinated in some intelligent and flexible way. Also, they need somewhat of flexible communication. This is when the two different branches concept gets sense: private memory (message passing) vs. shared memory.

Thanks to the methods of data hiding used by modern programming languages (like MPI) that prevents of undesired data sharing; processors can communicate in a simple and unconstrained fashion by sharing data using commonly known addresses. This is why programmers normally prefer MIMD architecture [2].

So the sum of software and hardware advances is what has brought us to current techniques, languages and tools. If we put some all those advances to the limit, therefore, we reach supercomputing or HPC level that is a computer science technique which is born in order to satisfy a real necessity that appears in the knowledge society, especially when simulating large, varied and complex scenarios and their evolutions in a very short time: meteorology, genetics, engineering, quantum thermodynamics and so on [6].
1.3. Motivation and goals

A migration is the process of making an effective installation of an application in a target platform. In this project we present a real migration process of a complex numerical application (called Kratos) into a real supercomputer (called Mare Nostrum). The whole present document talks about this process, trying to think about it, to gather conclusions and to share knowledge gained during the project.

The main motivation is to show that a migration has specific, well-defined steps or stages. It would be a sort of template for migrations through a particular case (Kratos and Mare Nostrum, detailed in next Section).

There are studies about applications (algorithms, software engineering, object modeling, source code, program improvements and so on), about hardware platforms (transistors, architecture design, instruction stages, memory, I/O, core and processor distribution and so on) and about how to install a program or package in specific platform or platforms. But about migrations in general there is not many information. So, in this project our goals are:

- **Main goal is focused on the results on installation, execution and its improvement.** Few times the difficulty in these efforts are considered; so in this document we are trying to reflect that work, including the necessary tasks in order and time involved in the process to facilitate any kind of migration, which could be done with another applications and/or target platforms.

- **We want to help to fill this information gap in order to make easier any migration and finally propose the main steps followed in any migration process.**

- **Also, another goal is to check if the execution of the application is correct.** During all the process, we did parallel debugging that means we ensured the application produced correct answers because it satisfied two criteria [6]: absence of non determinism (always same answers are produced on same inputs) and equivalence between executions with different amount of resources. This means not only the work was done but all results of every execution were correct.
As a summary of this part of the document, what we want to show is how an already scalable, relatively complex, real framework, but with modular structure, can be migrated to a HPC (*High Performance Computer*) platform. Also, we want to explain we can use software tools installed in our HPC platform target in order to improve the performance of the migrated numerical application.

### 1.4. Initial planning and economic cost of the PFC

In order to estimate the initial economic and temporal costs of the PFC, we suppose two main conditions:

- University estimates a PFC takes about 600.
- In our case, Mare Nostrum computing time is free of charge because of our project is non commercial and a research. However, we add that cost in order to be more realistic.

For the first condition, the work done in the PFC is divided in three profiles: Analyst, Programmer, and Documenter. Programmer time is estimated in 200 hours and 10€ per hour. Also, Analyst time is estimated in 200 hours in total and 20€ per hour, divided in: application, target platform, test installation, benchmark, migration, execution flow, analysis of results and improvement. Finally, Documenter time includes the redaction of present document (50 hours, 15€ per hour). Then, total economic cost of human resource is 8750€. In Table 2 and attached Gantt diagram this is more detailed:
For the second condition, Mare Nostrum economic cost is confidential. Because of this, we looked for information about similar platforms and we found a distributed machine with Mare Nostrum characteristics costs about 0’20€ per cpu-hour [63, 64, 65, 66]. As we initially estimate 20000 cpu-hours, total economic cost of Mare Nostrum computing time is 4000€. So, initial total economic cost is 12750€.
2. Kratos and Mare Nostrum

2.1. Kratos

Kratos [8, 21] is written in C++, uses Python as a kind of interface and it uses object-oriented philosophy organization. Kratos implements a kernel and uses its own applications as plug-in because they are compiled as shared libraries, every one of them with its specific solvers. Thus, Kratos can be considered as a framework. Also, Kratos is developed by CIMNE [22] (International Center for Numerical Methods in Engineering) staff and uses a tool called GiD [23] to visualize the results.

Now we present an overview of all main classes of Kratos: first the data class, then the solver classes and, finally, the classes that achieve the parallelization of Kratos.

**Modelpart** contains data structures and all data related to the model. Its components are Nodes, Properties, Elements, Conditions and solution data. It also provides the procedures that give access to all data contained in it.

This organization allows Kratos developers to have independent applications in order to developers decide what application or applications use in every moment without loading all the rest in memory. Each application has a well defined structure, divided in three classes:

- **LinearSolver** has the algorithms used for solving a linear system of equations and it is based on the Space class which defines a matrix and a vector and also contains their operators.

- **Strategy** has the general flow of a solving process and the solving algorithm itself. Also, it has to update the results in the data structure which is called Modelpart as we are going to see forward. In order to configure Strategy, Scheme is the class which must be modified, because contains all operations of local system components before assembling and updating of results after solution.

- **Process** is where new algorithms have to be added.
Originally, Kratos was a serial program but, because of this modular structure, when these classes are parallelized Kratos becomes parallel. It uses mainly MPI [11] (*Message Passing Interface*) for parallelization and some OpenMP which usually is deactivated in compilation process, because performance is better with all the program flow being parallel instead of only some loops.

When executed in parallel with MPI, Kratos uses data decomposition in order to give a divide-and-conquer solution. Strategy class changes a bit in order to give to Kratos parallel execution. MPI adapted classes are based on the Trilinos [10] library that gives a very good performance. The parallelization is performed trying to encapsulate the changes as much as possible in its common kernel and trying to minimize changes into any module.

Communication and coordination between elements it is possible thanks to **Communicator** class which uses MPI system and classes. It transfers data as any MPI program, encapsulating domains, and their interfaces and data decomposition data transfers. For communication, it is important to know the concept local and ghost elements in Kratos.

As we told, in domain decomposition the whole data is divided in data sets. Each boundary

---

**Figure 3:** Classification of Kratos main classes

<table>
<thead>
<tr>
<th>KERNEL</th>
<th>APPLICATION</th>
<th>IO</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW PROCESS</td>
<td>NEW STRATEGY</td>
<td>NEW FORMULATION</td>
</tr>
<tr>
<td>PROCESS</td>
<td>STRATEGY</td>
<td>FORMULATION</td>
</tr>
<tr>
<td>MODEL</td>
<td>MODEL PART</td>
<td>MESH</td>
</tr>
<tr>
<td>NEW ELEMENT</td>
<td>NEW CONDITION</td>
<td></td>
</tr>
<tr>
<td>NODE</td>
<td>PROPERTIES</td>
<td>ELEMENT</td>
</tr>
<tr>
<td>CONDITION</td>
<td>DEF</td>
<td></td>
</tr>
<tr>
<td>GEOMETRY</td>
<td>LINEARSOLVER</td>
<td>DATA CONTAINERS</td>
</tr>
<tr>
<td>VECTOR</td>
<td>QUADRATURE</td>
<td>MATRIX</td>
</tr>
<tr>
<td>BASIC TOOLS</td>
<td>FINITE ELEMENT</td>
<td>BASE FINITE ELEMENT</td>
</tr>
</tbody>
</table>
has elements which belong to one or other data set. From the point of view of a data set, a local element is the boundary element which belongs to the data set. On the other hand, a ghost element is the boundary element which belongs to a neighbor data set. Every element that belongs to a boundary is a local element in a data set and a ghost element in another data set.

Knowing all this, we show the data structures contained in the class:

- **NeighbourIndices**: neighbor domains, including the coloring.
- **LocalMesh**: entities that belong to this domain, including those who are internal. LocalMesh\[i\] stores all entities that belong to the current domain but are duplicated in domain \(i\).
- **GhostMesh**: entities which are a duplicated of the entities in other domains. GhostMesh\[i\] contains entities which are a duplicated of the entities in domain \(i\).
- **InterfaceMesh**: contains the entities that can be ghost or local but they are in the interface between this domain and other domains. InterfaceMesh\[i\] contains the entities that can be ghost or local but they are in interface between this domain and domain \(i\).

Also, the **Communicator** class has the following main methods which are empty when Kratos is executed in serial mode:

- **Synchronize**: copies different data from local entities to all their 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** and so on: implementations of MPI communication tasks.

The **MPICommunicator** class, which derives from **Communicator**, implements these methods using MPI. Thus, Kratos serial and parallel modes perfectly fit in an automatic way.

To partition the domain efficiently, the METIS [9] library is used, because it reduces partition interfaces better than other methods such as greedy algorithm method. METIS partitioner is added as an application, so it is used only when the user decides it.
The use of a Python scripting as the main procedure makes the Kratos code -applications and solvers- easier to use without modify C++ source code: calls procedures which read data, loads Kratos kernel, selects applications and solvers and defines the way to use those selected applications and solvers. For example, MetisApplication –which uses a partitioning library commonly used in grid systems and called METIS– has an interface to Python and can be casted up from the script file when running in any kind of distributed machine.

Thanks to Kratos versatility we can use an HPC to run Kratos in parallel in an easy way a priori. In fact, at the end of the project, Kratos has been running in:

- Intel and IBM processors as seen below.
- GPGPU, some solvers of Kratos use CUDA [12] (Compute Unified Device Architecture).
- Little and big HPCs as we are going to see forward.
2.2. Mare Nostrum

Mare Nostrum [13] is a supercomputer that can be classified as a MPP in a cluster which uses dual core processors. It was the European most powerful supercomputer in 2005 (it was inaugurated at 4/12/2005), belongs to BSC [14] (Barcelona Supercomputing Center) and it is located in a church at the Campus Nord of the UPC [15] (Universitat Politècnica de Catalunya, Polytechnic University of Catalonia in English). It was improved in 2006 and nowadays is in 77th position of Top500 ranking.

Its specifications are detailed in Table 3:

<table>
<thead>
<tr>
<th>Token</th>
<th>Mare Nostrum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microchip</td>
<td>IBM PowerPC 970MP (64 bits, dual core 2.3 GHz)</td>
</tr>
<tr>
<td>Memory</td>
<td>8 GB per node</td>
</tr>
<tr>
<td>Hard disk</td>
<td>36 GB per node</td>
</tr>
<tr>
<td>Queuing system</td>
<td>Moab client 5.2.3 and SLURM 2.1.9</td>
</tr>
<tr>
<td>Interconnection network</td>
<td>Myrinet express 1.2.7</td>
</tr>
<tr>
<td>Units per node</td>
<td>2</td>
</tr>
<tr>
<td>Number of nodes</td>
<td>2560 (BladeCenter JS21)</td>
</tr>
<tr>
<td>Number of processors</td>
<td>10240</td>
</tr>
<tr>
<td>Total memory</td>
<td>20 TB</td>
</tr>
<tr>
<td>Total Hard disk</td>
<td>280 TB (GPFS)</td>
</tr>
<tr>
<td>Operating system</td>
<td>Linux Suse</td>
</tr>
</tbody>
</table>

Table 3: Mare Nostrum specifications

The main reason to choose Mare Nostrum as our target platform is because with it we have more total memory and computing power. With that enormous amount of resources bigger simulations can be computed and, so, solved.
Mare Nostrum, as it is in previous table, uses IBM PowerPC 970MP, each is a dual core processors of 64 bits at 2’3 GHz; 5120 in total. Thus the total number of processors in Mare Nostrum supercomputer is 10240. Each node has 2 dual core processor and 8GB of memory; 4GB for each dual core processor.

Queuing system is supported by Moab client [16] and SLURM [17] (Simple Linux Utility for Resource Management).

Moab client is a utility of Moab Workload Manager which is a highly advanced scheduling and management system designed for clusters, grids, and on-demand/utility computing systems. In our case, HPC, Moab applies site policies and extensive optimizations to orchestrate jobs, services, and other workload across the ideal combination of network, compute, and storage resources. Moab enables true adaptive computing allowing compute resources to be customized to changing needs and failed systems to be automatically fixed or replaced. Moab increases system resource availability, offers extensive cluster diagnostics, delivers powerful QoS/SLA features, and provides rich visualization of cluster performance through advanced statistics, reports and charts. Therefore, Moab is the resource monitor of Mare Nostrum.

SLURM is an open-source resource manager designed for Linux clusters of all sizes. It provides three key functions. First it allocates exclusive and/or non-exclusive access to resources (computer nodes) to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work (typically a parallel job) on a set of allocated nodes. Finally, it arbitrates contention for resources by managing a queue of pending work. SLURM's design is very modular with dozens of optional plug-ins. Therefore, SLURM is the resource manager of Mare Nostrum because it is designed to operate in heterogeneous clusters with up to 65,536 nodes and thousands of processors and can sustain about 120,000 jobs per hour.

Interconnection network is supported by Myrinet express [18], created by a company called Myricom (founded in 1994). Created for supporting HPC platforms’ Ethernet at 10 Gbps rate, its networking protocol is Infiniband which is prepared to use MPI with measured latencies of
2.5 microseconds. It supports TCP/IP and UDP/IP traffic. Therefore, Myrinet express is the network hardware and network manager of Mare Nostrum.

Mare Nostrum offers for the programmer a simple submission system oriented to make easier queuing jobs [19]. A job is a script which demands to the machine to execute a couple of commands (e.g., an execution) with required resources by the programmer. It is the execution unit for the SLURM and its queuing system is who decides when it can be executed; it may be the job never runs if its demands are too high. To make it clear, take a look to a script example:

```bash
#!/bin/bash
#
# @ initialdir = .
# @ job_name = TELESCOPIO-128
# @ total_tasks = 128
# @ tasks_per_node = 4
# @ noderset = clos
# @ wall_clock_limit = 5:59:50
# @ queue
source /home/bsc15/bsc15245/kratos-submit/env64.sh
hostname
uptime
sl_get_machine_list
sl_get_machine_list > machinefile.$$
echo EXECUTING...
echo "###################################################################"
cd /home/bsc15/bsc15245/kratos-submit
mpirun -np 128 -machinefile /home/bsc15/bsc15245/kratos-submit/machinefile.$$ /home/bsc15/bsc15245/kratos-submit/kratos-submit-TELESCOPIO.sh
echo "###################################################################"
echo EXECUTED!!!
hostname
uptime
```

Inside blue square we can see the required resources by the programmer and some options, also known as job directives: initial directory, the name of the job, total number of cores required, number of cores per node (1, 2 or 4), if the cores must be physically close, wall clock time (maximum of time the job can be running) and so on.

Inside red square is the commands of the script. There is a mandatory command: sl_get_machine_list which is a call to SLURM in order to get the list of servers which will interconnect the cores during the execution. Also, it is needed by mpirun which is a tool to launch an MPI program.
Basically, Mare Nostrum has a queue where jobs wait their turn to enter in execution. The most important commands to launch jobs are shown in Table 4:

<table>
<thead>
<tr>
<th>Command</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>mnsubmit &lt;job_script&gt;</td>
<td>Submits a job to the queue system</td>
</tr>
<tr>
<td>mninteractive &lt;command&gt;</td>
<td>Submits an interactive job, executing the command or applications specified as argument</td>
</tr>
<tr>
<td>mnq</td>
<td>Shows all the jobs submitted, classified in three groups (active, eligible and blocked)</td>
</tr>
<tr>
<td>mncancel &lt;job_id&gt;</td>
<td>Removes a job from the queue system</td>
</tr>
<tr>
<td>checkjob &lt;job_id&gt;</td>
<td>Obtains detailed information about a specific job</td>
</tr>
<tr>
<td>mnstart &lt;job_id&gt;</td>
<td>Shows information about the estimated time for the specified job to be executed</td>
</tr>
<tr>
<td>mnhold –j &lt;job_id&gt;</td>
<td>Sets a block to the specified job. To release it, the same command must be run with –r option</td>
</tr>
</tbody>
</table>

Table 4: Mare Nostrum queue commands

Also, in Mare Nostrum it has already installed tools as GCC, G++ and GFORTRAN (GNU), XLC, XLC++ and XLF (IBM XL) or system libraries as MPICH (Message Passing Interface Chameleon) which is a standard for MPI in distributed memory applications used in parallel computing [20]. In addition, while there is space available we can install additional libraries and tools that would be needed to compile and execute our programs. So, Mare Nostrum is a suitable HPC platform to work with.
3. Learning more about Kratos

An extensive code analysis and classes of the target application, this is, of Kratos is mandatory in order to understand Kratos executing flow and needs. For achieving this, one of the solvers of Kratos was chosen: PureConvectionEdgeBased, used to calculate no compressible fluids dynamics. So, in this Section, we present the picture of the insides of one of Kratos’ applications.

3.1. Kratos structure

As it was shown in the previous Section, Modelpart is the class related with data structures and holds all data related to an arbitrary part of model. This means this class holds the data of a piece of the model. Therefore, Modelpart saves a part of the whole problem if Kratos is being executed in parallel mode and all its connections with the rest of the problem. However, in Figure 4 we can see how all that data is hold.

![Figure 4: Main classes and attributes of Kratos](image)

The goal of this Section is understanding Kratos structure by analyzing one of its solvers, more than showing the parallel behavior of Kratos as a whole.
3.2. PureConvectionEdgeBased

This class needs two auxiliary classes in order to the analyzed solver can work. The analysis followed bottom-up method, so the basic classes are analyzed first. Figure 5 shows the three classes involved in the solver.

![Figure 5: PureConvectionEdgeBased classes](image)

All the code of these classes can be found in pseudo code at the appendix of the present document. It must be remarked that EdgeStructureType basically represents the data (3D axis: x, y and z) contained in one single position of MatrixContainer, which has the effective data of the model.

3.2.1. EdgeStructureType

Data contained in a single position of a vector or matrix.

![Figure 6: EdgeStructureType variables and procedures](image)
3.2.2. MatrixContainer

This class is the definition of a CSR [24] (Compressed Storage Row) matrix which is the format used by the solver in order to read and write the data in disk. This format puts the subsequent non-zeros of the matrix rows in contiguous memory locations in order to save space, because real problems has a lot of empty positions in their data matrices.

It works as follows. Assuming we have a non-symmetric sparse matrix called $A$, we create 3 vectors: one for floating-point numbers (val), and the other two for integers (col_ind, row_ptr). The val vector stores the values of the nonzero elements of the matrix, as they are traversed in a row-wise fashion. The col_ind vector stores the column indexes of the elements in the val vector. Both vectors have same number of positions. That is, if $\text{val}(k) = A_{i,j}$ then $\text{col\_ind}(k) = j$. The row_ptr vector stores the locations in the val vector that start a row, that is why row_ptr vector has as positions as rows has the matrix $A$.

Thus, the amount of units of used space with CSR format is $2 \times \text{nnz} + n + 1$ where $\text{nnz} = \#\text{non - zeros in matrix } A$ and $n = \#\text{of rows of matrix } A$, in front of $n^2$ units of used space with common formats.

In consequence, the amount of saved memory with this matrix format helps to compute bigger problems and that is why is very used in HPC platforms.
3.2.3. PureConvectionEdgeBased

This is the main class of the solver that uses MatrixContainer which is the CSR matrix class and contains the main action: solve (common to all solvers of Kratos).
3.3. Installation in a laptop

In order to migrate an application to an HPC platform, first we must understand, analyze and characterize which system utilities and tools, compilers, interpreters and libraries—including their dependencies—are needed in the application that we want to migrate. I did a test installation of Kratos in a laptop with Linux Ubuntu 8 that helped us to get the “photograph” or to know application requirements, following the guide in Kratos wiki [21]. It is important to get in mind Ubuntu’s Synaptic tool automates much of the installation work (except Kratos and some specific and complex libraries). Table 5 shows the specifications of the laptop:

<table>
<thead>
<tr>
<th>Token</th>
<th>Laptop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microchip</td>
<td>Intel Core 2 Duo T7200 (64 bits, dual core 2 GHz)</td>
</tr>
<tr>
<td>Memory</td>
<td>2 GB</td>
</tr>
<tr>
<td>Hard disk</td>
<td>30 GB</td>
</tr>
<tr>
<td>Queuing system</td>
<td>N/A</td>
</tr>
<tr>
<td>Interconnection network</td>
<td>N/A</td>
</tr>
<tr>
<td>Units per node</td>
<td>N/A</td>
</tr>
<tr>
<td>Number of nodes</td>
<td>N/A</td>
</tr>
<tr>
<td>Number of processors</td>
<td>2</td>
</tr>
<tr>
<td>Total memory</td>
<td>2 GB</td>
</tr>
<tr>
<td>Total Hard disk</td>
<td>30 GB</td>
</tr>
<tr>
<td>Operating system</td>
<td>Linux Ubuntu 8</td>
</tr>
</tbody>
</table>

Table 5: Laptop specifications

Once the application seems that is correctly installed, it is important to make some tests to demonstrate the application runs correctly. In order to prove the installation of Kratos in the laptop, we executed the benchmarking included in the application and one little problem which used PureConvectionEdgeBased solver, courtesy of Riccardo Rossi (member and creator of Kratos).
Finally, it is good to annotate or have in mind all the dependencies and problems found during the test installation of the application in order to avoid problem repetition when migrating the application. One of the most important tips is the installation order, in test installation of Kratos case:

1. System libraries.
2. SuperLU_DIST.
3. CMake and SWIG.
4. Trilinos.
5. Boost/Bjam.

Every library and tool will be explained in next Section. Anyway, main problems found during test installation are the following:

- Paths must be coherent and set before any execution. It is important to install tools and libraries without make a mess, selecting appropriate directories of all things that will be manually installed.
- SWIG must be 1.38 version or higher.
- A lot of times, when installing a new library it is normal to put wrong paths or dependencies. In the case of tools and libraries which have a Makefile file, it is important to do the command make clean before repeat any compilation (common error in any manual installation in a Linux environment).
- Kratos must be compiled with Bjam tool, included in boost. Thus, in order to compile Kratos, the bjam command is used which looks for a file called Jamroot; it is similar to Makefile of the make command. However, in this file it is mandatory to write all the paths of the most part of tools and libraries previously installed.
- Every solver of Kratos is independent and it is indicated in Jamroot file. Because this modularity, it is recommended to install Kratos partially and add solvers gradually every time we compile successfully (solvers that uses MPI and Trilinos could be the most problematic).
- Some indispensable libraries for Kratos could be lost when upgrading our OS. For example, in our case Linux Ubuntu was upgraded from v8.04 to v10.10 LTS when
Kratos was already installed. Python, collaterally, was upgraded from v2.5 to v2.6. This made Kratos lose its path to Python libraries and crashed when executing (Kratos could not make `import Kratos *` command). In order to fix this issue, once the problem was detected, I entered libs directory of Kratos and executed the command `ldd Kratos.so` (where Kratos is linked to its main libraries). The output was (error is in yellow):

```
edgar@takashima:~/Escritorio/temp/kratos/kratos/libs$ ldd Kratos.so
linux-vdso.so.1 => (0x00007fff491ff000)
libutil.so.1 => /lib/libutil.so.1 (0x00007f52aec08000)
libpthread.so.0 => /lib/libpthread.so.0 (0x00007f52ae9eb000)
libdl.so.2 => /lib/libdl.so.2 (0x00007f52ae7e6000)
libboost_python-gcc42-mt-1_38.so.1.38.0 =>
/usr/local/lib/libboost_python-gcc42-mt-1_38.so.1.38.0
(0x00007f52ae594000)
libpython2.5.so.1.0 => not found
librt.so.1 => /lib/librt.so.1 (0x00007f52adeda000)
libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x00007f52adbc6000)
libm.so.6 => /lib/libm.so.6 (0x00007f52ad943000)
libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x00007f52ad72b000)
/lib64/ld-linux-x86-64.so.2 (0x00007f52af0000)
/libssl.so.0.9.8 => /lib/libssl.so.0.9.8 (0x00007f52ad159000)
/libcrypto.so.0.9.8 => /lib/libcrypto.so.0.9.8 (0x00007f52acdd1000)
/libz.so.1 => /lib/libz.so.1 (0x00007f52acbb1000)
edgar@takashima:~/Escritorio/temp/kratos/kratos/libs$ ldd Kratos.so
```

This problem was solved modifying the Jamroot and compiling Kratos again with the command `bjam -j2 threading=multi` that updated all Kratos.so paths.

If you want to make a look into Jamroot, you can find it in the appendix.
4. Kratos application requirements

Once test installation works, we can proceed to install the application in the target HPC platform just after know the application requirements of the application. This means first we should use all the information previously gathered with the test installation and know this step will be re-visited as we will see in further Sections. That is because the HPC platform could force us to cycle and return to this step every time we find a lack in our current application requirements.

So, in this Section, we explain you the final Kratos installation requirements, all libraries and tools needed.

4.1. Necessary libraries and utilities

Kratos, as explained above, is modular and every of its applications and solvers is independent from the rest. Also, new applications and solvers can be added or existing ones can be removed; so we can draw Kratos in a really simple way with all solvers encircling Kratos. However, we must insist in the fact that every application can contain one or more solvers.

It is important to remark that the picture above in this document includes only some of Kratos solvers. We explain below how it works.
In the next Subsection, we introduce the libraries and utilities used and their installed version in each one of the platforms involved in the migration process. These platforms are:

- Acuario: the original CIMNE HPC cluster (see Section 5.4).
- Intel laptop: to test and learn basic installation and execution issues.
- Marenostrum BSC Supercomputer: the target of our migration work.

### 4.1.1. System libraries

As we know, system libraries can be static or shared (dynamic). This references at how an object or group of objects is linked [25]. A static library contains all objects and symbols (structures, procedures and so on) needed for a program or library; in fact, linking against it means to add those symbols to the program or library which uses it, because it is like a compressed file. On the other hand, a shared library holds only the symbols of the library and static libraries used; referencing to other shared libraries for the symbols not resolved (known as undefined). The difference between both kinds of libraries is in the spent time when loading the library in memory, being static libraries faster than shared libraries. However, static libraries have the problem of being a bit more difficult to administrate because it is difficult to know against which libraries were linked. This administration can be partially done through *nm* and *ldd* Linux commands:

- **nm** enumerates all symbols contained in a library and indicates if a symbol is resolved or not.
- **ldd** shows all libraries where a library looks for its unresolved symbols. This command only works for shared libraries.

Basically, source code of compiled languages first is compiled and then linked into a library or a binary executable. It is in linking stage when the programmer decides if a library must be static or dynamic.

To make the installation easier, all libraries of the installation have been dynamically linked, this means libraries are shared (.so extension). Now, we must see which libraries could or should be installed before installing Kratos.
Kratos and its libraries use three different languages: C/C++, FORTRAN [30] (*FORmula TRANslation*) and Python. C/C++ and FORTRAN languages are compiled languages while Python is an interpreted language. Therefore, their respective compilers (interpreter when using Python) and libraries must be installed in the system. In our case, we used GNU compilers and the Python interpreter. Also, if Kratos is going to work in parallel mode, MPI libraries have to be installed in the target platform. As our target HPC platform is Mare Nostrum, we have to use MPICH libraries.

### 4.1.2. Compilers

4 compilers are used to install Kratos: SWIG [26] (*Simplified Wrapper and Interface Generator*), GCC [27], Bjam [28] and CMake [29] (*Cross platform Make*).

<table>
<thead>
<tr>
<th>Name</th>
<th>Acuario</th>
<th>Laptop</th>
<th>Mare Nostrum</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWIG</td>
<td>1.3.36</td>
<td>1.3.40</td>
<td>-</td>
</tr>
<tr>
<td>GCC</td>
<td>4.1.2 (ICC 11.1)</td>
<td>4.4.3</td>
<td>4.4.0</td>
</tr>
<tr>
<td>Bjam</td>
<td>3.1</td>
<td>3.1.17</td>
<td>3.1</td>
</tr>
<tr>
<td>CMake</td>
<td>N/A</td>
<td>2.8.1</td>
<td>2.7</td>
</tr>
</tbody>
</table>

*Table 6: Version of compilers*

SWIG is a software development tool that connects programs written in C/C++ with interpreted languages as Perl, PHP, Python, Tcl and Ruby. With it, we can create C/C++ objects or call C/C++ procedures from Python. It must be remarked Boost has a very similar tool to swig called Boost::Python. The main difference between them resides in the fact SWIG creates an intermediate Python file to work, while Boost::Python goes straight against C/C++ libraries but it adds a lot of extra C/C++ code. Kratos uses Boost::Python and CMake uses SWIG.

GCC is an open source C compiler made by GNU. With this I am also referring to G++ (GNU’s C++ compiler) and GFORTTRAN (GNU’s FORTRAN compiler).

Bjam is the tool of Boost which helps to compile enormous projects in an easy with, using a main file called Jamroot and support files called Jamfile. This is similar to Makefile idea, but eases in a lot of ways the use of different C/C++ compiler brands when compiling. In Kratos case, we have a Jamfile for every solver and we can deactivate every one when commenting its particular line placed in the Jamroot.
CMake is an open source utility that is used to compile, link, build, and test software. It automatically generates Makefile files in order to easily compile. In some way, it is similar to Bjam because it helps to organize and compile enormous project and it uses C/C++ and FORTRAN compilers. Both utilities are thought from the point of view of distributed projects in the cloud.

4.1.3. Interpreters

2 interpreters are needed in order to execute Kratos: Python and Csh [32] (C shell, tcsh in its improved form).

<table>
<thead>
<tr>
<th>Name</th>
<th>Acuario</th>
<th>Laptop</th>
<th>Mare Nostrum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>2.4</td>
<td>2.6.5</td>
<td>2.5.5</td>
</tr>
<tr>
<td>Csh</td>
<td>6.14</td>
<td>20070713-2ubuntu</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 7: Version of interpreters

Python, as explained above, is an interpreted language but it is also the name of the interpreter itself. Any person who wants to use Kratos must know, at least, a bit of Python because it is the interface for the user-programmer to execute any solver. First, in order to load all Kratos initial possibilities, an import Kratos * command is required and, then, the user can load the solvers needed.

Csh is a Linux command interpreter. Obviously, it is essential to administrate and execute commands in Mare Nostrum as in all possible platforms that use Linux OS.

4.1.4. Basic additional libraries

In this document, when talking about basic libraries or basic additional libraries, I am referencing to the libraries which are needed by Kratos or that are essential for more complex libraries. These libraries are four: BLAS [33] (Basic Linear Algebra Subprograms), METIS/PARMETIS [34], LAPACK [35] (Linear Algebra PACKage) and SuperLU_DIST [36].

<table>
<thead>
<tr>
<th>Name</th>
<th>Acuario</th>
<th>Laptop</th>
<th>Mare Nostrum</th>
</tr>
</thead>
<tbody>
<tr>
<td>METIS/PARMETIS</td>
<td>3.1</td>
<td>3.1.1</td>
<td>3.1.1</td>
</tr>
<tr>
<td>BLAS and LAPACK</td>
<td>3.3.0</td>
<td>3.3.0</td>
<td>3.3.0</td>
</tr>
<tr>
<td>SuperLU_DIST</td>
<td>2.0</td>
<td>2.3</td>
<td>2.3</td>
</tr>
</tbody>
</table>
BLAS is a library that contains a lot of basic linear algebra vector and matrix operations. It was published in 1979 and it is used to build larger packages or libraries. It is very used in HPC platforms, where it can be found optimized; in fact, Intel and AMD distribute them in its particular optimized version. You can found it written either in C (CBLAS in this document) or in FORTRAN (FBLAS in this document). In Mare Nostrum installation I chose FBLAS, because the version in FORTRAN I got includes all CBLAS procedures and has a lot more that are needed by other libraries.

METIS is library that holds a set of programs for partitioning graphs, finite element meshes and producing reducing orderings for sparse matrices. PARMETIS is the version for parallel computing (METIS is serial), but exists other specialized METIS libraries like HMETIS. In our case, METIS and PARMETIS are the libraries that our application is using. It is written in C.

LAPACK is also a library for numerical linear algebra, and provides routines for solving systems of linear equations and linear least squares, eigendecomposition of matrix problems, singular value decomposition and more algebra stuff. In our case it is written in FORTRAN but you can found LAPACK distributions written in C/C++. It needs the library FBLAS in order to work.

SuperLU_DIST is a parallel library used to solve systems of linear equations for distributed memory machines. Basically, it performs LU decomposition [37] (transform a matrix into the product of 2 matrices with a half of each full of zeros) in order to solve the equation system. I use its C version but FORTRAN version exists. In our installation it needs FBLAS and METIS/ Parmetis libraries.

Also, in this level, MPI is also included but in my opinion nowadays it should be considered as a system library. Without MPI libraries we would lose much of the potential of platforms with parallel features, and in this case it is also needed by almost all libraries. About Python Numpy [38] (Numerical Python), it is a Python extension used by Trilinos (focused in scientific computation with Python) but finally was not necessary.

<table>
<thead>
<tr>
<th>Python Numpy</th>
<th>1.2.1</th>
<th>1.5.0</th>
<th>1.5.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>OpenMPI</td>
<td>OpenMPI 1.4.1</td>
<td>MPICH 1.2</td>
</tr>
</tbody>
</table>

**Table 8**: Version of basic additional libraries
4.1.5. Main libraries

Main libraries are those libraries which can be found at top level of any installation, normally are packages of libraries. In Kratos case, we have four packages: Python, Boost [39], Trilinos and Kratos itself.

<table>
<thead>
<tr>
<th>Name</th>
<th>Acuario</th>
<th>Laptop</th>
<th>Mare Nostrum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>2.4</td>
<td>2.6.5</td>
<td>2.5.5</td>
</tr>
<tr>
<td>Boost</td>
<td>1.37</td>
<td>1.38.0</td>
<td>1.38.0 or higher</td>
</tr>
<tr>
<td>Trilinos</td>
<td>9.0</td>
<td>10.2.1</td>
<td>10</td>
</tr>
</tbody>
</table>

*Table 9: Version of main libraries*

We have overviewed above which paper Python plays in Kratos application. Python, as a library, is needed for importing modules, some basic I/O procedures, initializing variables, choose which solver has to be used, control solver iterations (called steps) and so on.

Boost is really important for Kratos, because it is the package of libraries that holds all high level procedures for Kratos as, for example, interconnect Python with C++ (Boost::Python) or use MPI (Boost::MPI). Without this library Kratos cannot execute any solver at all. It only needs Python library in order to be compiled and used.

Trilinos is, as Boost, an essential package of libraries, but without it Kratos only would lose some of its solvers and related procedures. In our case, we need those solvers and procedures so it has to be installed in Mare Nostrum. Trilinos is a project that tries to implement robust algorithms and enabling technologies using object-oriented paradigm, upgrading establish libraries as PETSc [40] (*Portable, Extensible Toolkit for Scientific Computation*), METIS/PARMETIS, SuperLU_DIST, Aztec [41], BLAS and LAPACK. Trilinos is compiled with CMake tool and, because of its broad scope; Sandia National Laboratories has recently divided Trilinos in eight capability areas [42]: Framework & Tools, Software Engineering Technologies & Integration, I/O Support, Discretizations, Meshes, Geometry & Load Balancing, Scalable Linear Algebra, Linear & Eigen Solvers and Embedded Nonlinear Analysis Tools. It needs basic libraries to be compiled and its configuration is set with a file called do-configure which makes a *cmake* command.
Finally, in order to create Kratos libraries, we must have previously compiled the basic libraries, Boost, Trilinos and Python.

4.2. Library dependencies

Once we know every library and its dependencies, it is possible to draw a graph of library dependencies; from bottom to top, from basic to complex, making clear what installation order must be followed in order to install the target application. In Figure 10 we can observe the graph of library dependencies of Kratos:

![Graph of library dependencies of Kratos](image)

Figure 10: Graph of library dependencies of Kratos

With this picture in mind, we can try to make effective the migration the application in the target platform.
5. Migrating Kratos to Mare Nostrum

As commented in the beginning of previous Section, now we can proceed to install the application in the target HPC platform. As has happened with Kratos in Mare Nostrum, application requirements stage could be re-visited if we find, during installation of the application in the target platform, any lacks in our graph of library dependencies in order to improve it.

In any migration we can find up to three steps as showed in Figure 11:

Configure environment is all the related to compilation flags, paths, tools and libraries in general. It includes the customization of configure files too (in Kratos case: Makefile, do-configure and Jamroot files).

Build executable is all the related to the proper include and libraries location and undefined symbols in the compiling and linking stages. In this step we can partially check if the environment step is correct.

Finally, Run is all the related to the proper loading of utilities and libraries. In this step we can check if the environment and build steps are finally correct, when execution results exist and are consistent and correct (in Kratos case, running one or more of its solvers).
5.1. Configuration environment

The installation of Kratos in Marenostrum differs greatly if compared to the test installation in the laptop. First of all, most of system libraries could be installed manually, but it is preferable to use in some cases libraries already installed in the target platform (e.g., MPICH is mandatory if we want interconnection network to work correctly). In Kratos case, we used Mare Nostrum versions of MPI, GCC, GFORTRAN, MPICC (MPI wrapper for C/C++), MPIF77 (MPI wrapper for FORTRAN77), SWIG and CMake. The installation sequence for these packages is:

1. FBLAS.
2. METIS/PARMETIS.
3. LAPACK.
4. SuperLU_DIST.
5. Trilinos.
6. Python.
7. Boost/Bjam.
8. Kratos.

First problem of this step was found when we found all libraries where compiling in 32 bits thanks to use file and objdump commands but the solution was simple: add –m64 flag in all configuration files.

Also, in Mare Nostrum paths are not saved from session to session, so env64.sh script was created. Every time Kratos is executed or it is necessary to compile with different paths and environment variables, it must be casted source env64.sh command. In the appendix an env64.sh script example can be found.

If a lower version of GCC 4.1.2 is used, a lot of OpenMP directives and Python commands cannot be compiled. Also, we have to set some global variables through env64.sh script in order to use MPI for GNU instead of MPI for IBM XL.
5.2. Build executable

Some basic libraries already exist in Mare Nostrum and are prepared to use IBM XL primitives (BLAS, LAPACK and METIS/PARMETIS), thus we must configure paths and configure files to link with our own basic libraries.

In order to make a faster detection of unresolved symbols in Run stage, it is highly recommended to compile all libraries as shared because of two main reasons:

- `ldd` Linux command can be used, so easily we can see against what libraries has been compiled the library which causes the execution errors.
- Linking with static libraries may force to change configuration files, mainly Makefile, because this kind of library normally has to be placed just after all object files (remember: it is like a compressed file of objects).

To install Trilinos 10, it is recommended to erase all target directories every time the "do-configure" file changes. Also, it is a good idea to use `make VERBOSE=1` command in order to see all compilation and linking messages in the case there are compilation errors.

And one more important thing: while compiling, we must be aware about how much space we will need and estimate this amount of free space in disk. In Mare Nostrum, we need about 6 GBs of minimum free disk quota for all necessary data during the compilation and linking steps. As we know there exists also Run stage, we must add more space to that minimum disk quota to execute Kratos application. However, this extra free space for executing is variable because depends on the size of the problem we want to solve with the application.

5.3. Run

MPI loading must be unique in every execution. Because of a wrong default configuration in Kratos structure, MPI was loaded twice and, so, the program failed in every execution. We had to change this configuration following CIMNE’s supervision.
Although configure environment and build executable finish correctly, we can find undefined symbols when executing Kratos. This case is the most uncomfortable of all, because an undefined symbol can appear when executing the target application for various reasons:

- Wrong configuration when compiling a library: mix 32 and 64 bits libraries gives compilation errors (e.g.).
- Wrong compilation flags: it is common some troubles appear when compiling C/C++ and FORTRAN source files together (e.g.). This is because of standard rules says any program in C/C++ that casts FORTRAN procedures, has to add a trailing underscore at all procedure names (in GCC, it is added by default but it can be removed with --fno-underscoring flag).
- Wrong library in linking stage: it could be possible that Trilinos and Kratos links with two different and incompatible MPI libraries with the used MPI wrapper (e.g.).
- New system libraries needed with certain compilation flags: optimizations might require additional libraries (e.g.).

Once Run stage is correct, we could say the target platform has a stable installation of the application, so migration of the application into target platform has been successful. If it is not correct, we should review the graph of library dependencies, Configure environment stage or Build executable stage; because in one or more of them the problem should be found.

5.4. Scalability

Now, the situation is we have in the target platform a functional version of the migrating application. As in our case we have installed it in an HPC platform, Mare Nostrum, therefore we can prove its scalability [43]. It is directly related to Amdahl’s law which calculates the speed up of an application. Kratos has been tested in a laptop (up to 2 cores used) and Acuario (up to a 90 – 100 cores). In both cases, Kratos scales perfectly; but we want to test it with a greater number of cores (Mare Nostrum has more than 10.000 cores) in order to see if scalability maintains.
In Table 10, Acuario specifications are shown along with the other two platforms:

<table>
<thead>
<tr>
<th>Token</th>
<th>Laptop</th>
<th>Acuario</th>
<th>Mare Nostrum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microchip</td>
<td>Intel Core 2 Duo T7200 (64 bits, dual core 2 GHz)</td>
<td>Intel® Xeon® CPU E5410 @2.33GHz</td>
<td>IBM PowerPC 970MP (64 bits, dual core 2’3 GHz)</td>
</tr>
<tr>
<td>Memory</td>
<td>2 GB</td>
<td>32 GB or 16 GB</td>
<td>8 GB per node</td>
</tr>
<tr>
<td>Hard disk</td>
<td>30 GB</td>
<td>120 GB</td>
<td>36 GB per node</td>
</tr>
<tr>
<td>Queuing system</td>
<td>N/A</td>
<td>N/A</td>
<td>Moab client 5.2.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Slurm 2.1.9</td>
</tr>
<tr>
<td>Interconnection network</td>
<td>N/A</td>
<td>Infiniband 4*</td>
<td>Myrinet express 1.2.7</td>
</tr>
<tr>
<td>Units per node</td>
<td>N/A</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Number of nodes</td>
<td>N/A</td>
<td>9x (+2AMD)</td>
<td>2560 (BladeCenter JS21)</td>
</tr>
<tr>
<td>Number of processors</td>
<td>2</td>
<td>72</td>
<td>10240</td>
</tr>
<tr>
<td>Total memory</td>
<td>2 GB</td>
<td>172 GB</td>
<td>20 TB</td>
</tr>
<tr>
<td>Total Hard disk</td>
<td>30 GB</td>
<td>3 TB (shared)</td>
<td>280 TB (GPFS)</td>
</tr>
<tr>
<td>Operating system</td>
<td>Linux Ubuntu 8</td>
<td>Redhat 5</td>
<td>Linux Suse</td>
</tr>
</tbody>
</table>

Table 10: Specifications of each platform

But there are various kinds of scalability, so what kind of scalability are we going to try? In our case we are doing horizontally scalability, this means we want to add more cores and watch if or when this equation is true:

\[
S(N) = \frac{N}{n}; N = \text{cores}, n = \text{cores in baseline case}
\]

This formula corresponds to linear scalability case. So in next Section we tell how to collect data with different number of nodes, doing it systematically in order to know how Kratos scales and its efficiency.
5.5. Costs

We do not explain in this part of the document the economic cost of the project (it is in a further Section), but remember the efforts made during migration of a numerical application to a target platform as a kind of template:

- **Learning** how it is structured the source code of the application which consists in analyzing the code, making a bit of reverse engineering [44], and drawing in the process part or whole the object model of the application. Also, we should do a test installation in a small system in order to start observing how to install the application.

- **Application requirements** of the application must be known, looking for information about libraries and utilities we are going to use in order to draw its graph of library dependencies.

- **Migrating** the application where we can observe that the complexity of a migration depends a lot on the migrating application though, but there are 3 steps that feed off each other and easily identifiable: Configure environment, Build executable and Run. Here is where we might have to review the study of the application requirements we did previously if anything is incorrect.

Also, when talking about time consumption in compiling and linking needed libraries, an automatic standard installation can last compiling and linking between seven and ten hours in Kratos case, depending on the flags and utilities used. We will talk more about this in a further Section too.
6. Data collection and comparative

Although we have an installation which can be executed successfully, our job is not finished yet. If only the migration is done, the work would be a bit absurd and unnecessary. At this point, we have some options beyond only executing and using the application in a productive way. We can also improve the application temporarily talking; we are referring here to reduce its elapsed time \([45]\) \(t_{\text{elapsed}} = t_{\text{user}} + t_{\text{system}}\); includes time consumed by the program in CPU, I/O, and memory access and so on). We could try to get this goal by improving the source code of the application, but in most occasions we are migrating applications that are not programmed by us, our team or our staff. In these common occasions it is very difficult or impossible to improve the program, but there are some ways or computer science techniques to improve the application without changing its execution flow. However, these techniques can be used in addition to the possibility of changing our code.

We should take advantage of the work done in learning Section, but from the point of view of looking how the application executes. In Kratos case, there is an initial Python script that loads and executes all the process. How it works is extended below.

As said in First Section, in the case of executing the application in an HPC platform, it is obligatory to do parallel debugging. Two criteria must be accomplished [6]:

- When repeating the execution of a problem with the same solver, the solutions given must be always the same (this is called absence of non determinism).
- In addition, these executions must be equivalent when changing the resources (more or less cores, more or less memory, different core distributions and so on). As above, this is extended further.

Let’s see how this is done in Kratos case.
6.1. Kratos execution flow

When we are not the programmers of the migrating application, understanding how behaves the application is highly recommended in order to know what resources needs. So, understanding how an application executes is essential in a migration. We must analyze the source code that casts the most abstract and important procedures, following basic reverse engineering techniques. For example, if we have the source code of the application, we could use UML [47] (Unified Modeling Language) tools as UMLGraph [48] in order to get the object model of the application. But in Kratos case, the top procedures are all called from a simple Python script, so we can do it manually.

If Kratos is executed in serial mode, we can get following pseudo code simplification:

```python
import Python libraries
import kernel
import solvers
read input_data
create ModelPart
select options and solver
time_step_number = 1
while time_step_number ≤ max_time_step do
    if time_step_number ≤ initial_time_step_number do
        initialize data
    else
        Solver.solve()
    endif
    time_step_number += 1 endwhile
finalize results
```

It must be remarked `solve()` actions actualizes data in every step, because Kratos is a time-stepped and loosely synchronous numerical application as we can see. Anyway, when Kratos executes in parallel, MPI launcher goes first as was shown previously (example in Section 2.2). In Mare Nostrum, this launcher is casted through `mpirun` command [46]. This command executes \( n \) threads (\( n = \#\text{cores} \)) of the script placed in its command line. In our case, this
script should make a command similar to `python my_python_script_using_kratos.py` which loads Python modules, Kratos’ kernel and solvers needed; then solves with specific options the problem through the selected solver by the user-programmer.

We automated with a script (`kratos-submit.sh`) the submission to Mare Nostrum of Kratos jobs (you can find its source code in the appendix). Basically, it creates the submission script (called `kratos-submit.q`) with some options (e.g., problem we want to execute) and submits it:

---

**Figure 12:** Execution steps of `kratos-submit.sh` and `kratos-submit.q`

When Mare Nostrum executes Kratos, it creates \( n \) executions of the Python script. Python has a specific MPI library which gives parallel capacities. It helps to synchronize all those threads through Communicator class. This class, as explained in Second Section, does not do anything in serial mode but organize threads in parallel mode. Then, Kratos starts to solve the problem.

Simplifying how it works, when solve action is casted, the solver algorithm is executed and then every thread waits in a MPI barrier until all threads has finished. But, another essential doubt may come in mind: how does Kratos share the work loading in parallel mode? Kratos uses METIS library to divide the data in order to do domain decomposition technique. But it can only be done by one processor, thus dividing data is a serial process.
We show below if we can do some improvement for this inconvenient. So, if we add this information to what we have just explained, Kratos flow in parallel mode is as follows:

Let’s see an example. Imagine we have a sport car modeled in the polygonal format used by Kratos (.mpdpa extension). Also, you added wind information to this model in order to see wind pressure in the aerodynamics of the car [49] and your computer has 4 cores, so you can solve the problem with 4 threads. Rank 0 processor would use METIS to divide the mesh after libraries are loaded and data is read. Then, every processor would solve its part of the problem.
Once our platform has executed Kratos application, we can proceed to use GiD program, which unite all solution pieces, in order to visualize the solution from many points of view: the partitions made by METIS, pressure, and wind flow and so on.

![Figure 15: Example of Kratos execution (2)](image)

Below we explain about the three main elements that belong to an application execution in an HPC platform: chosen resources, needed libraries and problem we want the application to solve.

### 6.2. Ahmed_25

Ahmed_25 [52] is almost considered as a benchmark problem in numerical applications that uses finite element method and, consequently, data decomposition technique. It consists in a kind of vehicle inside a sort of wind tunnel. In fact, it derives from the Ahmed body that was first defined and its characteristics described in an experimental work. Configurations with slant angles of 25° and 35° are considered as a test case, so Ahmed_25 is the Ahmed body with 25° slant angle. It has about 1,600,000 elements.

Once the first stable installation in Mare Nostrum was a reality, we proved Ahmed_25 problem with different amount of resources. Every different resource configuration was proved three times at least and each solution was visualized in GiD in order to make parallel debugging, assuring this way results and installation were correct.
As Mare Nostrum gives us the possibility of demand different number of cores and different number of cores per node, we used for each solution of this problem:

- \(2^n\) cores; \(2 \leq n \leq 9\)
- number of cores per node = \(\{1, 2, 4\}\)

Therefore, with these resource premises, we executed the problem submitting it to Mare Nostrum a couple of times for every resource configuration.

![Figure 16: Ahmed_25 solution executed with 128 cores](image)

We used as fictitious baseline 1 core, because this way the different presentations of the results are easier to read and understand. Figure 17 shows speedup and Figure 18 shows execution time in hours for each configuration.

![Figure 17: Speed up from 1 up to 512 cores executing Ahmed_25](image)
As we can observe, in this first figure we see two main behaviors in different configurations:

- The best number of cores per node is four because speedup is a slightly higher than other configurations. This is because with four cores per node the distance shortens.
- As we can see, speedup stops to double when higher number of processors is required.

![Figure 18](image)

**Figure 18:** Execution time from 1 up to 512 cores executing Ahmed_25

If we remember, in Kratos execution flow, the part where the problem is divided is serial. As Amdahl noted, the compute time can be divided into the parallel portion, and no matter how high the degree of parallelism in the former, the speedup will be asymptotically limited by the latter, which must be performed on a single processing element [2]. This means that Kratos is limited by that serial part, forcing to stop to go faster and identifying Kratos speedup bottleneck. Nonetheless, there is a tricky solution in order to solve this: separating the execution in two parts. Thanks to data decomposition, we can divide the problem in \( n \) pieces and then reuse that division so many times as we want because number of time steps, solver used and more Kratos problem configuration options are external. However, for every number of processors, serial part must be executed at least once. We proved deeply this way to solve a problem with Kratos with next problem executed in Mare Nostrum.

### 6.3. TELESCOPIO

TELESCOPIO [53] problem is about a real case. The GTC (*Gran Telescopio CANARIAS*) is a telescope with a primary segmented mirror of 10‘4 meters of diameter. It is installed in the Roque de los Muchachos Observatory (La Palma, Canary Islands). This telescope is a Spanish initiative, lead by the IAC (*Instituto de Astrofísica de Canarias*) and helped by the FEDER
(Fondos Europeos de Desarrollo Regional) of European Union, the IA-UNAM (Instituto de Astronomía de la Universidad Nacional Autónoma de México), INAOE (Instituto Nacional de Astrofísica, Óptica y Electrónica) and the University of Florida. TELESCOPIO contains the 3D model of the GTC and its surroundings (about 1 km$^2$) and tries to know wind behavior in a determined time window, because optics and light can be disturbed a lot in high precision astronomy. It has about 24,000,000 elements.

So, in this case, we divided the problem execution in two parts after to prove again that speedup stopped as higher number of processors we used. We used for each solution of this problem:

- $2^n$ cores; $4 \leq n \leq 11$
- \textit{number of cores per node} = 4

In fact, we found a problem that forced us to divide the execution. Mare Nostrum has an important limitation: it is not possible to have opened more than 1012 files, crashing the execution of the application. Nonetheless, CIMNE staff helped us dividing the problem for us in 1024, 1536 and 2048 files (remember: it is serial) and then we upload them to Mare Nostrum.
In Figure 20, it can be observed that speedup also slows at some point (about 1024 processors). This is because the number of elements of each piece of the problem is not enough to fill processors time when taking 1024 or more processors.

Also, we can see that until 1024 processors executions the speedup is linear. As we explained during First Section and knowing speedup of Kratos with TELESCOPIO problem, we can evaluate the efficiency of Kratos with TELESCOPIO problem in Mare Nostrum or the associated overhead with this simple formula:

$$S(N_{proc}) = \varepsilon \times N_{proc} = \frac{N_{proc}}{1 + f}; \quad \varepsilon = \text{efficiency}; f = \text{overhead}$$

Figure 20: Speed up from 1 up to 2048 cores executing TELESCOPIO

Figure 21: Efficiency from 1 up to 2048 cores executing TELESCOPIO
We can state that the optimum amount of cores for TELESCOPIO problem is 256, because it is the maximum number of cores with the maximum efficiency.

In the appendix there are more snapshots of Ahmed_25 and TELESCOPIO solutions. Also, there are snapshots of a problem with more than 100,000,000 elements which shows resistance to wind of a Ferrari.
7. Performance improvements after migration

Once we have a functional installation, it is normal to try to reduce elapsed time used by the application when solving problems. This can be done in some ways, as improving the source code of the application. But, as we stated at the beginning, that is not an option. So we have two more possibilities left: improving hardware or improving our installation.

7.1. It’s the software, stupid

It is a valid option to improve the hardware of the target platform, but it is a really expensive way to reduce elapsed time of our applications. A successful high-end computing technology must have a stable, effective programming model that persists over the lifetime of the application [6]. Thus, we must discard that option.

On the other hand we can improve our installation from code generation and changing compilers and their flags. For this, we must annotate which compiler and flags we have used in our first installation. Then, in consequence, we have now the possibility not only to change the resources in an execution but also to select which installation we want to use. From now on, an execution configuration is the sum of demanded resources and selected installation.

Improving code generation is a cheap way to gain some speedup over same application with same amount of resources. The easiest known way for this it is as simple as adding –O3 flag in all compiler sentences which optimizes automatically much part of the code. However, more optimizations can be tried in order to speedup the migrating application. Also, we tried IBM XL [54] compilers which are supposed to fit better than GNU, although it generated some problems which are explained forward.

Figure 22: Code generation vs. Hardware
We decided to test different installations in Mare Nostrum, each with different characteristics (flags, compiler and even different libraries in some cases). The characteristics of each one are overviewed below and summarized in next figure:

1. **GCC**: all Kratos software is compiled with GCC 4.4.
2. **GCC with native libraries**: all Kratos software is compiled with GCC 4.4, except the BLAS and LAPACK libraries that are compiled with the XLC 10.1 compiler.
3. **GXLC**: Kratos and Boost libraries are compiled with GCC 4.4. The rest of the libraries are compiled with IBM GXLC compilers (a kind of mask which translates GCC flags into IBM XL flags; e.g.: -m64 is translated as -q64).
4. **XLC**: Kratos and Boost libraries are compiled with GCC 4.4. The rest of the libraries are compiled with IBM XL compilers.
5. **Optimized XLC**: same as installation 4, but adding a –qhot (high-order transformations) flag which lets IBM XL compiler to do more complex transformations.

In the appendix, compilers and their most critical flags can be found in a table. Of these installations we found the fastest is installation 2, reducing executing time of TELESCOPIO problem about 8-10% respect baseline installation which is installation 1 and installation 3 (their executing time is very similar). The second in the ranking is installation five with 4-5%, followed by fourth installation with a little 2-3%. Curiously, if –qhot flag is removed from Trilinos compilation in installation 5, it reduces a little more its time up to 6-7%.

**Figure 23**: Installations of successfully executed installations of Kratos in Mare Nostrum

In order to manage and test different execution configurations, we have an automatic installation script which can be found in the appendix. Basically, this script installs the
different libraries in order. It follows installation order shown by the Kratos graph of library dependencies. It works as shown in Figure 24:

Once it was functional, we can talk about installation consumed time. This means we can observe if there is any substantial difference between these elapsed times, and we can save exact configuration files:

<table>
<thead>
<tr>
<th>Installation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>6 - 7 hours</td>
<td>7 - 8 hours</td>
<td>6 - 7 hours</td>
<td>10 - 11 hours</td>
<td>10 - 11 hours</td>
</tr>
</tbody>
</table>

*Trilinos compiles with C and FORTRAN compilers. If C is compiled with –qhot option, the execution crashes with the message “Not enough memory allocated for overlapping Error: Out of space due to poor estimate of memory needed” and installation lasts between 4 and 5 days because of Trilinos compilation

Table 11: Elapsed time of automatic installation script for each tested installation

However, we compiled in interactive mode in Mare Nostrum so far. With IBM XL compilers this is not possible because when Trilinos is compiled, some source code can take more than ten minutes that is the maximum amount of time a process can execute in interactive mode. So we created a simple submission script that executes the installation.
First, we changed all GNU flags by IBM XL flags (e.g., `–m64` by `–q64`). In addition, we modified some Makefile files in order to compile shared versions (`–qpic flag`) of some libraries (FBLAS, LAPACK and METIS) and to add some additional essential static libraries which are needed when activating `–O5` (`libxlopt.a`) and `–qhot` (`libxl.a`) flags of IBM XL compilers. Therefore, deeper and wider analysis of how libraries compile was needed in installations 2, 4 and 5. Also, when compiling FORTRAN source code, it is mandatory to add `–qextname` flag in order to add a trailing underscore in every cast from C source code to FORTRAN procedures (and in SuperLU_DIST add `–DAdd_` in order to activate a definition). This is because GNU default configuration adds trailing underscore when casting FORTRAN procedures from C source code.

But we found many important problems when using IBM XL compilers. In particular, we classified them in three main groups: standards not accomplished, GNU less restrictive than IBM and IBM XL bugs not resolved. Also, the main reason of Kratos and Boost to be compiled only with GNU is because it is impossible to compile their libraries with IBM XL. In fact, compilation of Kratos with IBM XL finishes with -1 return code because the compiler simply crashes, and Boost compilation has errors that cannot be resolved without great modifications of the source code.

We also must add we ensured all little modifications we did to source code (basically from Trilinos) don’t change execution flow in any way. In fact, we contacted with Trilinos staff in order to confirm those modifications we did were correct. Anyway, every time an error was found we had to erase all built libraries and start compilation again from the beginning because of Trilinos nature.

7.2. Standards not accomplished

Some not respected C standard directives are not treated as errors by GNU because are not critical, but in IBM XL compilation fails and, in consequence, some additional errors arose. In fact, GNU should respect those standards as IBM XL does.

In "../include/python2.5/pyconfig.h", line 939.9: 1540-0848 (S) The macro name "_POSIX_C_SOURCE" is already defined with a different
"/usr/include/features.h", line 154.10: 1540-0425 (I)
"_POSIX_C_SOURCE" is defined on line 154 of "/usr/include/features.h"

In this case, Python is defining a standard variable that tells which 
_POSIX_C_SOURCE [57] is being used in order to force compiler to use some standards. As we cannot change system libraries in Mare Nostrum and it is not recommended to change Python code, we made a copy of features.h file, placed it in Trilinos source code directory and commented the variable definition. This way we cheat the compiler.

### 7.3. GNU less restrictive than IBM

Some errors are avoided by GNU because it supposes some premises IBM XL not. The main premise is if duplicated name is found, last one is the correct. Another premise is a variable can exists with duplicated name because it lasts until code where it is nested finishes.

In "/packages/tpetra/inout/Tpetra_MatrixIO.cpp", line 214.15: 1540-0403 (S) "ind" is already defined

Repeated variable name is changed inside loop where it is placed.

In "/packages/tpetra/inout/Tpetra_MatrixIO.cpp", line 249.18: 1540-0400 (S) "val" has a conflicting declaration

Same as previous problem: repeated variable name is changed inside loop where it is placed. Pitifully, we must say we finally deactivated Tpetra package of Trilinos after talking with CIMNE staff because this problem was not possible to solve.

In "/opt/ibmcmp/vacpp/10.1/include/stdlib.h", line 22.14: 1540-0400 (S) "abs(int)" has a conflicting declaration.
".../packages/epetra/src/Epetra_ConfigDefs.h", line 129.1: 1540-0424 (I) "abs" is declared on line 129 of ".../packages/epetra/src/Epetra_ConfigDefs.h"

IBM XL uses its own standard library. When it founds a repeated name, it demands to solve confusion. On the other hand, GNU supposes last name found is the correct one. However, what we did is to comment using std::abs; line in Trilinos file.

7.4. IBM XL bugs

IBM XL compiler has, at least, one bug we found when compiling Trilinos.

In ".../packages/epetra/src/Epetra_C_wrappers.cpp"

Some kind of castings in Trilinos, which involve defines of types, cannot be compiled because an IBM XL bug. In fact, they recommend change the code as we did in a Trilinos source file [56]. In our case we had to use memcpy [59] system procedure instead of original casting in order to cheat the compiler in every case found in the file, for example:

```
return((EPETRA_OBJECT_PTR ) comm_);
```

is changed by

```
EPETRA_OBJECT_PTR tmp;
memcpy ( &tmp, &comm_, sizeof(Epetra_Comm *) );
return(tmp);
```

In ".../packages/thyra/src/interfaces/operator_vector/fundamental/Thyra_VectorSpaceBase_decl.hpp", line 545.37: 1540-0416 (S) "createMember" cannot be declared because its name has already been used.

".../packages/thyra/src/interfaces/operator_vector/fundamental/Thyra_VectorSpaceBase_decl.hpp", line 468.3: 1540-0426 (I) The name
"createMember" is used on line 468 of ".../packages/thyra/src/interfaces/operator_vector/fundamental/Thyra_VectorSpaceBase_decl.hpp"

Similar as duplicated variables, in some cases IBM XL finds an error with duplicated procedure names which are not real errors [58]. What we did is to change one of the names, but we must say we finally deactivated Thyra package of Trilinos after talking with CIMNE staff because this problem was not possible to solve. Here we show which sequence of errors we found before deciding to deactivate Thyra package:

In ".../packages/thyra/src/interfaces/operator_vector/fundamental/Thyra_VectorSpaceBase_def.hpp", line 134.34: 1540-0217 (S) "createMember" is not a member of "class VectorSpaceBase<double>"

Name is changed.

In ".../packages/thyra/src/interfaces/operator_vector/fundamental/Thyra_VectorSpaceBase_decl.hpp", line 545.37: 1540-1112 (I) "Thyra::VectorSpaceBase<double>::createMember1() const" is a pure virtual function

At this point is where we were forced to deactivate Thyra package because the conflicting procedure was virtual.

In ".../packages/ml/src/Coarsen/ml_agg_METIS.c", line 60.10: 1506-296 (S) #include file "metis.h" not found

In ".../packages/ml/src/Coarsen/ml_agg_VBMETIS.c", line 59.10: 1506-296 (S) #include file "metis.h" not found

In ".../packages/ml/src/Operator/ml_op_utils.c", line 437.10: 1506-296 (S) #include file "metis.h" not found

Because a reason we don’t really know, IBM XL does not find metis.h file. We reviewed configuration files and include directory and they were correct. Curiously, we substitute in
every source code the `#include file "metis.h"` line by `#include file "parmetis.h"`, solving this way the errors.

In "../packages/ml/examples/BasicExamples/ml_user_smoothing.cpp"

The error consists in a procedure casting that doesn’t find any coincidence. This is because IBM XL doesn’t notice the real type of the variable implied. We added in the source code a casting to force IBM XL to know, substituting `MLList.set("smoother: user-defined function",userSmother);` by `MLList.set("smoother: user-defined function",(char *)userSmother);`

In "../packages/sacado/src/template/Sacado_TemplateManager.hpp", line 82.51: 1540-0109 (S) The "mutable" specifier must be applied only to non-reference class data members

It is not sure if it is a compiler bug, but GNU can compile it. What we did is to deactivate Sacado package.
8. Final planning and economic cost of the PFC

In this Section we present final costs. As it was done in Introduction Section, we have divided human resources cost in program, analysis and documentation; all sum up to 10650€. In Table 12 and attached Gantt diagram this is detailed:

<table>
<thead>
<tr>
<th>Human resource and cost per hour</th>
<th>#hours</th>
<th>Total hours</th>
<th>Total economic cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Analysis (20€/hour)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application</td>
<td>40</td>
<td>300</td>
<td>6000€</td>
</tr>
<tr>
<td>Target platform</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test installation</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application requirements</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Migration</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Benchmark</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution flow</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analysis of results</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Improvement</td>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Program (10€/hour)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test installation</td>
<td>60</td>
<td>360</td>
<td>3600€</td>
</tr>
<tr>
<td>Migration</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Benchmark</td>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution flow</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Improvement</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Documentation (15€/hour)</strong></td>
<td>70</td>
<td>70</td>
<td>1050€</td>
</tr>
<tr>
<td><strong>Total hours</strong></td>
<td>300+360+70=730</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 12: Final human resource cost

Finally, in Mare Nostrum we have consumed 36415 cpu-hours in total and therefore economic cost is 7283€. If we add this cost to human resources cost, total economic cost is 17933€.
When compared with initial planning, we can observe some divergences in time costs and, therefore, in economic cost:

- **Extensions of human resource cost:**
  
  o Migration time was extended because IBM XL bugs were found during Trilinos compilation and we decided to create the automatic installation script for Kratos in Mare Nostrum.
  
  o Benchmark time was extended too because we decided to test more different installations (from 3 to 5).
  
  o Improvement time also was affected by new installations we decided to test.
  
  o Application and its requirement analysis were more complex than expected at the beginning.
  
  o Analysis of results was enlarged because of test up to 5 different installations instead of the initial 3.

- **Extensions of Mare Nostrum consumption:** initially, we planned to test up to 3 different installations. As seen in Improvements Section, we finally did 5 installations and, in addition, IBM XL compilers needed to submit Trilinos compilations because in interactive mode there is a limitation in Mare Nostrum of 10 minutes. Thus, more time of Mare Nostrum was needed every time we wanted to test different installations with different libraries or flags.
9. Conclusions

Nowadays we have powerful machines that are capable of executing really complex and useful applications in a more than reasonable elapsed time, thanks to latest computer science techniques as parallelization and data decomposition for distributed machines. Because data input can be more realistic than ever, billions and even trillions of information are used when applications run in supercomputers like Mare Nostrum, which is a big and very suitable HPC for this work. Kratos is a good proof of this, thanks to its object-oriented structure and GiD which feeds it with new and detailed 3D models.

We know how data decomposition, through METIS library in this case, gets really good performances when used in HPC platforms. In addition, thanks to execution flow analysis, it is possible to raise those performances simply amplifying parallel part in execution and avoiding any serial execution when it is possible. If these two objectives are achieved (good data decomposition and parallelization), applications can get almost linear speedup and obtain a 100% efficiency.

In every migration process, application and target platform must be known. In order to do this, information must be gathered and analyzed. This process can be defined, following and also systematizing it, creating a kind of template which can be really useful in every migration which involves an application and a target platform.

This template points to document all elements involved, as happens for example in software engineering, and helps to decide if the application could be installed and executed in target platform before starting the migration. Therefore, time and money can be saved thanks to the steps presented in this work, because the cost of a migration of these characteristics is clearly not zero even if there is no change in the application.

Also, administration of different installations is easier when following same process in each of them. If graph of library dependencies is followed and installation is automated, installation configuration is more tractable and, thus, different configurations can be tested in order to improve the performance of the application in the target platform. Then, this means it is easy to produce better and more productive installations with this automation.
So, this template that has made easier our work consists in:

- **Learning** how the application and target platform are structured and how they work, using reverse engineering techniques and doing a test installation (e.g., it can be done in a different platform). Also, it is better to be in contact with the creators of the application and the used libraries during all the migration. If any problem appears or if any bug is found, this contact could be useful for us and for other present and future users of them, solving doubts, gathering tips and so on.

- **Application requirements** of the application must be known in order to guess its needs (libraries and utilities) and behavior, drawing the graph of library dependencies of the application and defining its installation order.

- **Migrating** the application to the target platform, modifying the graph of library dependencies and coming back to application requirements step if necessary. It must be decided which libraries and utilities we are going to use and install those which are not installed in the target platform yet. Also, it is highly recommended to do an automatic script where all of our actions are recollected that can make easier future modifications of the current configuration. This step basically involves three elements that feed off each other:

  ✓ **Configure environment**: flags, paths, tools and libraries in general.
  ✓ **Build executable**: define properly include and library location.
  ✓ **Run**: test the application and ensure it is working correctly, checking also environment and executable steps.

Once this is finally done, we have to save current installation configuration.

- **Improving** the performance of the application from code generation (compiler and its flags, testing other versions of the same library and so on), doing some tasks:

  ✓ **Analyze executing flow**: we must know first how the application behaves, identifying critical and important parts of the execution.
  ✓ **Test different configurations**: we should create more installations with different configurations, libraries or utilities. If there is an automatic
installation, it is much easier to save, administrate and manage all those configurations.

✓ **Collect and compare data:** collect all information about executions we have done, compare them and select the most appropriate for us (e.g., the fastest).

Therefore, there are 4 main steps in any migration: learning, application requirements, migrating and improving. Each of them has one or more main tasks which are intertwined and perfectly define every step. Then, Figure 25 summarizes all this process:

![Migration template diagram](image)

**Figure 25:** Migration template
10. Future extensions of the work done

From this project, after conclusions are obtained, we can extract more ideas for future projects. Different points of this work can be improved or developed deeper, about target platform used, automatic installation process and improvement methods.

We have run Kratos in a HPC platform (Mare Nostrum) which uses MIMD distribution. As said, Kratos have some solvers which use CUDA libraries so it can run in GPGPUs. It could be interesting to transform, prepare and adapt some solvers for execution in those platforms, trying to compare the results and their behaviors with others obtained from other executed in completely different platforms.

As we have explained along this document, we created an automatic installation for Kratos as migrating application and Mare Nostrum as target platform. If we extrapolate the main logic of the script described in Section 7.1, we can get the flow shown in Figure 26:

![Figure 26: Proposal for auto installation script for any application](image)

Kratos automatic installation could be transformed in iterative from its present serial form, and then transform it into a generic script as shown in previous figure, passing every configuration and the installation order somehow. For example, the automatic installation script could be fed from a text file, maybe like HTML [60] (Hyper Text Markup Language) or XML [61] (eXtensible Markup Language), in order to parse the graph of library dependencies and installation configuration of each package. Also, it could be programmed in any language...
and even create an object-oriented structure, which also had the option to be fed from a graphical interface (e.g.).

Finally, improvement achieved and problems or errors with utilities open a wide range of options:

- More installation configurations can be tested in order to improve the performance of installation 2.
- Create a debug installation would be very interesting. We created one with a tool called Valgrind [62], but the execution crashed.
- Some Trilinos packages that are not used in Kratos have been deactivated in order to compile the rest of libraries when compiling with IBM XL, which has some bugs not resolved. It would be interesting to know why this happens.
- Also, compiling Trilinos with certain flags of IBM XL C/C++ compiler produces some errors. The –O4 and –O5 flags cannot be activated because causes the compiler to fail because of memory problems. Also, –qhot flag in same compiler creates a problem when executing because of poor estimation of memory needed and forces the execution to fail.

Therefore, there are enough lines of study and research with Kratos migration and IBM XL compiler that help to start new projects from present work.
11. Appendix

11.1. Pseudo code of Kratos

11.1.1. EdgeStructureType

Public:

Variables:

double Mass;

\[ \text{Mass} = \frac{\partial N_i}{\partial x_k} \cdot \frac{\partial N_j}{\partial x_i} \ast \partial \Omega \] (k components of the laplacian matrix with edge ij)

boost::numeric::ublas::bounded_matrix<double,TDim,TDim> LaplacianIJ;

LaplacianIJ = Laplace matrix with double resolution

array_1d<double, TDim> Ni_DNj;

\[ N_i \ast \frac{\partial N_j}{\partial x_i} \ast \partial \Omega \] (k components of the gradient matrix with edge ij)

array_1d<double, TDim> DNi_Nj;

\[ D N_i N_j = N_j \ast \frac{\partial N_i}{\partial x_i} \ast \partial \Omega \] (k components of the transposed matrix with edge ij)

Actions:

inline void Add_Gp(array_1d<double,TDim>& destination, const double& p_i, const double& p_j);

\[ \text{destination}[i] += -N_i \ast D_Nj[i] \cdot p_j + D_Ni \ast N_j[i] \cdot p_i; \quad i \in [0..TDim] \]

inline void Sub_Gp(array_1d<double, TDim>& destination, const double& p_i, const double& p_j);

\[ \text{destination}[i] += N_i \ast D_Nj[i] \cdot p_j - D_Ni \ast N_j[i] \cdot p_i; \quad i \in [0..TDim] \]

inline void Add_D_v(double& destination, const array_1d<double,TDim>& v_i, const array_1d<double,TDim>& v_j);

\[ \text{destination}[i] += N_i \ast D_Nj[i] \cdot (v_j[i] - v_i[i]); \quad i \in [0..TDim] \]

inline void Sub_D_v(double& destination, const array_1d<double,TDim>& v_i, const array_1d<double,TDim>& v_j);

\[ \text{destination}[i] += -N_i \ast D_Nj[i] \cdot (v_j[i] - v_i[i]); \quad i \in [0..TDim] \]

inline void Add_grad_p(array_1d<double,TDim>& destination, const double& p_i, const double& p_j);

\[ \text{destination}[i] += N_i \ast D_Nj[i] \cdot (p_j - p_i); \quad i \in [0..TDim] \]

inline void Sub_grad_p(array_1d<double,TDim>& destination, const double& p_i, const double& p_j);

\[ \text{destination}[i] += -N_i \ast D_Nj[i] \cdot (p_j - p_i); \quad i \in [0..TDim] \]

inline void Add_div_v(double& destination, const array_1d<double,TDim>& v_i, const array_1d<double,TDim>& v_j);

\[ \text{destination}[i] += -N_i \ast D_Nj[i] \cdot v_j[i] + D_Ni \ast N_j[i] \cdot v_i[i]; \quad i \in [0..TDim] \]
inline void Sub_div_v(double& destination, const array_1d<double,TDim>& v_i, const array_1d<double,TDim>& v_j);

destination[i] += Ni_DNj[i] * v_j[i] - DNi_Nj[i] * v_j[i]; i ∈ [0..TDim]

inline void CalculateScalarLaplacian (double& l_ij);

\[ l_{ij} = \sum_{i=0}^{TDim} LaplacianJ(i, i) \]

inline void Add_ConvectiveContribution(array_1d<double,TDim>& destination, const array_1d<double,TDim>& a_i, const array_1d<double,TDim>& U_i, const array_1d<double,TDim>& a_j, const array_1d<double,TDim>& U_j);

#ifdef USE_CONSERVATIVE_FORM_FOR_VECTOR_CONVECTION

temp = \sum_{i=0}^{TDim} a_i[i] * Ni_DNj[i]

destination[i] += temp * (U_j[i] - U_i[i]); i ∈ [0..TDim]
#else

aux_i = \sum_{i=0}^{TDim} a_i[i] * Ni_DNj[i]

aux_j = \sum_{i=0}^{TDim} a_j[i] * Ni_DNj[i]

destination[i] += aux_j * U_j[i] - aux_i * U_i[i]; i ∈ [0..TDim]
#endif

inline void Sub_ConvectiveContribution(array_1d<double,TDim>& destination, const array_1d<double,TDim>& a_i, const array_1d<double,TDim>& U_i, const array_1d<double,TDim>& a_j, const array_1d<double,TDim>& U_j);

#ifdef USE_CONSERVATIVE_FORM_FOR_VECTOR_CONVECTION

temp = \sum_{i=0}^{TDim} a_i[i] * Ni_DNj[i]

destination[i] += -temp * (U_j[i] - U_i[i]); i ∈ [0..TDim]
#else

aux_i = \sum_{i=0}^{TDim} a_i[i] * Ni_DNj[i]

aux_j = \sum_{i=0}^{TDim} a_j[i] * Ni_DNj[i]

destination[i] += -aux_j * U_j[i] + aux_i * U_i[i]; i ∈ [0..TDim]
#endif

inline void Sub_ConvectiveContribution(double& destination, const array_1d<double,TDim>& a_i, const double& phi_i, const array_1d<double,TDim>& a_j, const double& phi_j);

#ifdef USE_CONSERVATIVE_FORM_FOR_VECTOR_CONVECTION
inline void Add_ConvectiveContribution(double& destination, const array_1d<double, TDim>& a_i, const double& phi_i, const array_1d<double, TDim>& a_j, const double& phi_j);

#ifdef USE_CONSERVATIVE_FORM_FOR_VECTOR_CONVECTION
inline void CalculateConvectionStabilization_LOW(array_1d<double, TDim>& stab_low, const array_1d<double, TDim>& a_i, const array_1d<double, TDim>& U_i, const array_1d<double, TDim>& a_j, const double& p_j);

#endif

inline void CalculateConvectionStabilization_LOW(array_1d<double, TDim>& stab_low, const array_1d<double, TDim>& a_i, const array_1d<double, TDim>& U_i, const array_1d<double, TDim>& a_j, const array_1d<double, TDim>& U_j, const double& p_j);

const TDim TDim = ...

// Calculate the contribution from the convective term
for (int i = 0; i < TDim; i++)
for (int j = 0; j < TDim; j++)

const double temp = \sum_{i=0}^{i<TDim} \sum_{j=0}^{j<TDim} a_i[i] * a_i[j] * Laplacian(i, j)

stab_low[i] = conv_stab * (U_j[i] - U_i[i]); i \in [0..TDim]

const double press_diff = p_j - p_i
inline void CalculateConvectionStabilization_LOW(double& stab_low, const array_1d<double,TDim>& a_i, const double& phi_i, const array_1d<double,TDim>& a_j, const double& phi_j);

\[
\text{conv}\_\text{stab} = \sum_{i=0}^{TDim} \sum_{j=0}^{TDim} a_i[i] \cdot a_i[j] \cdot \text{Laplacian}(i,j)
\]

\[
\text{stab}\_\text{low} = \text{conv}\_\text{stab} \cdot (\phi_j - \phi_i)
\]

inline void CalculateConvectionStabilization_HIGH(array_1d<double,TDim>& stab_high, const array_1d<double,TDim>& a_i, const array_1d<double,TDim>& pi_i, const array_1d<double,TDim>& a_j, const array_1d<double,TDim>& pi_j);

\[
\text{temp} = \sum_{i=0}^{TDim} a_i[i] \cdot \text{N_i}\_\text{DNj}[i]
\]

\[
\text{stab}\_\text{high}[i] = -\text{temp} \cdot (\pi_j - \pi_i)[i]; \quad i \in [0..TDim)
\]

#define USE_CONSERVATIVE_FORM_FOR_VECTOR_CONVECTION

#define USE_CONSERVATIVE_FORM_FOR_VECTOR_CONVECTION

#else

aux_i = \sum_{i=0}^{TDim} a_i[i] \cdot \text{N_i}\_\text{DNj}[i]

aux_j = \sum_{i=0}^{TDim} a_j[i] \cdot \text{N_i}\_\text{DNj}[i]

\[
\text{stab}\_\text{high}[i] = -(aux_j \cdot \pi_j[i] - aux_i \cdot \pi_i[i]); \quad i \in [0..TDim)
\]

#endif

inline void CalculateConvectionStabilization_HIGH(double& stab_high, const array_1d<double,TDim>& a_i, const double& pi_i, const array_1d<double,TDim>& a_j, const double& pi_j);

#define USE_CONSERVATIVE_FORM_FOR_VECTOR_CONVECTION

#else

aux_i = \sum_{i=0}^{TDim} a_i[i] \cdot \text{N_i}\_\text{DNj}[i]

aux_j = \sum_{i=0}^{TDim} a_j[i] \cdot \text{N_i}\_\text{DNj}[i]

\[
\text{stab}\_\text{high} = -(aux_j \cdot \pi_j - aux_i \cdot \pi_i)
\]

#endif

inline void Add_StabContribution(array_1d<double,TDim>& destination, const double tau, const double beta, const array_1d<double,TDim>& stab_low, const array_1d<double,TDim>& stab_high);

destination[i] += tau * (stab_low[i] - beta * stab_high[i]); \quad i \in [0..TDim)
inline void Add_StabContribution(double& destination, const double tau, const double beta, const double& stab_low, const double& stab_high);
    destination += tau * (stab_low - beta * stab_high)
inline void Sub_StabContribution(array_1d<double,TDim>& destination, const double tau, const double beta, const array_1d<double,TDim>& stab_low, const array_1d<double,TDim>& stab_high);
    destination[i] += -tau * (stab_low[i] - beta * stab_high[i]); i ∈ [0..TDim)
inline void Sub_StabContribution(double& destination, const double tau, const double beta, const double& stab_low, const double& stab_high);
    destination += -tau * (stab_low - beta * stab_high)
inline void Add_ViscousContribution(array_1d<double,TDim>& destination, const array_1d<double,TDim>& U_i, const double& nu_i, const array_1d<double,TDim>& U_j, const double& nu_j);
    \[ L = \sum_{i=0}^{i<TDim} \text{Laplacian}I(i, i) \]
    nu_avg = 0.5 * (nu_i + nu_j)
    destination[i] += nu_i * L * (U_j[i] - U_i[i]); i ∈ [0..TDim)
inline void Sub_ViscousContribution(array_1d<double,TDim>& destination, const array_1d<double,TDim>& U_i, const double& nu_i, const array_1d<double,TDim>& U_j, const double& nu_j);
    \[ L = \sum_{i=0}^{i<TDim} \text{Laplacian}I(i, i) \]
    nu_avg = 0.5 * (nu_i + nu_j)
    destination[i] += -nu_i * L * (U_j[i] - U_i[i]); i ∈ [0..TDim)

11.1.2. MatrixContainer

Class redefinitions:

//name for the self defined structure
typedef EdgesStructureType<TDim> CSR_Tuple;
typedef std::vector<CSR_Tuple> EdgesVectorType;
//name for row start and column index vectors
typedef std::vector<unsigned int> IndicesVectorType;
//names for separately stored node based values
typedef std::vector<double> ValuesVectorType;
typedef std::vector<array_1d<double,TDim>> CalcVectorType;

Public:

Actions:
//constructor and destructor
MatrixContainer();
\~MatrixContainer();
void ConstructCSRVector(ModelPart& model_part);

\[ n_{\text{nodes}} = \text{nodes of the input variable model part} \]
\[ mNumberEdges = \sum_{i=1}^{N} \text{neighbours of the i node}; \ N = \text{nodes in model part} \]
\[ i_{\text{node}} = \sum_{i=1}^{N} 1; \ N = \text{nodes in model part} \]

if \( i_{\text{node}} \neq n_{\text{nodes}} \) then ERROR endif

\[ m\text{NonzeroEdgeValues}.\text{resize}(m\text{NumberEdges}) \]
\[ m\text{ColumnIndex}.\text{resize}(m\text{NumberEdges}) \]
\[ m\text{RowStartIndex}.\text{resize}(n_{\text{nodes}} + 1) \]
\[ m\text{LumpedMassMatrix}.\text{resize}(n_{\text{nodes}}) \]
\[ m\text{InvertedMassMatrix}.\text{resize}(n_{\text{nodes}}) \]
\[ m\text{DiagGradientMatrix}.\text{resize}(n_{\text{nodes}}) \]
\[ m\text{Hmin}.\text{resize}(n_{\text{nodes}}) \]

row_start_temp = 0
for all nodo i in model_part do

neighb_nodes = neighbour nodes of the i node
i_node = FastGetSolutionStepValue of the i node
n_neighboors = neighb_nodes.size()
work_array.clear()

for all j node in neighb_nodes do
work_array.push_back(FastGetSolutionStepValue of the j node)
endfor

sort(work_array)

mRowStartIndex[i_node] = row_start_temp

counter = 0
while counter < n_neighboors do
j_neighbour = work_array[counter]
csr_index = mRowStartIndex[i_node] + counter
mColumnIndex[csr_index] = j_neighbour
mNonzeroEdgeValues[csr_index].Mass = 0
mNonzeroEdgeValues[csr_index].LaplacianIJ = ZeroMatrix(TDim, TDim)
mNonzeroEdgeValues[csr_index].Ni_DNj = ZeroVector(TDim)
mNonzeroEdgeValues[csr_index].DNi_Nj = ZeroVector(TDim)
counter++ endwhile
row_start_temp += n_neighbours
endfor

mRowStartIndex[n_nodes] = mNumberEdges
mLumpedMassMatrix[i_node] = 0; i ∈ [0..n_nodes]
mHmin[i_node] = 1 * e^10; i ∈ [0..n_nodes]

mDiagGradientMatrix[i_node] = ZeroVector(TDim); i ∈ [0..n_nodes]

void BuildCSRData(ModelPart& model_part);

weighting_factor = \frac{1}{TDim + 1}

forall i node in model_part do

heights[ie_node] = \frac{1}{\sum_{comp=0}^{TDim} dN_dx(ie_node,comp) * dN_dx(ie_node,comp)}

;ie_node ∈ [0..TDim]

CalculateMassMatrix(mass_consistent, volume)

mass_lumped = ZeroVector(TDim + 1)

mass_lumped[ie_node] = \sum_{je_node=0}^{TDim} mass_consistent(ie_node,je_node)

;ie_node ∈ [0..TDim]

weighted_volume = volume * weighting_factor

nodal_indices[ie_node] = GetGeometry()[ie_node].FastGetSolutionStepValue(AUX_INDEX) of the i node

;ie_node ∈ [0..TDim]

ie_node = 0

while ie_node ≤ TDim do

mHmin[nodal_indices[ie_node]] = MIN(SELF,heights[ie_node])

je_node = 0

while je_node ≤ TDim do

if ie_node ≠ je_node then

csr_index = GetCSRIndex(nodal_indices[ie_node], nodal_indices[je_node])

mNonzeroEdgeValues[csr_index].Mass += mass_consistent(ie_node,je_node)

laplacian ≡ mNonzeroEdgeValues[csr_index].Laplacian

laplacian(l_comp,k_comp) += dN_dx(ie_node,l_comp) * dN_dx(je_node,k_comp)

* volume; l_comp,k_comp ∈ [0..TDim]

gradient ≡ mNonzeroEdgeValues[csr_index].Ni_DNj

gradient[l_comp] += dN_dx(ie_node,l_comp) * weighted_volume

; l_comp ∈ [0..TDim]

transp_gradient ≡ mNonzeroEdgeValues[csr_index].DNi_Nj

transp_gradient[l_comp] += dN_dx(ie_node,l_comp) * weighted_volume

; l_comp ∈ [0..TDim)

endif

je_node ++ endwhile

ie_node ++ endwhile

mLumpedMassMatrix[nodal_indices[ie_node]] += mass_lumped[ie_node]

; ie_node ∈ [0..TDim]

ie_node = 0

while ie_node ≤ TDim do

gradient ≡ mDiagGradientMatrix[nodal_indices[ie_node]]

gradient[component] += dN_dx(ie_node, component) * weighted_volume

; component ∈ [0..TDim)

ie_node ++ endwhile

endfor

mInvertedMassMatrix[inode] = mLumpedMassMatrix[inode]

; inode ∈ [0..mLumpedMassMatrix.size()]

//perform MPI synchronization between the domains

mInvertedMassMatrix[inode] = \frac{1}{mInvertedMassMatrix[inode]}

; inode ∈ [0..mInvertedMassMatrix.size()]

//function to free dynamic memory
void Clear();

//functions to access database thought for parallel;
void FillCoordinatesFromDatabase(CalcVectorType& rDestination, ModelPart::NodesContainerType& rNodes);

for all inode in rNodes do

i_node = FastGetSolutionStepValue(AUX_INDEX) of the i node

(rDestination[i_node][component] = (node i)[component]; component ∈ [0..TDim])

endfor

void FillVectorFromDatabase(Variable<array_1d<double,3>> & rVariable, CalcVectorType& rDestination, ModelPart::NodesContainerType& rNodes);

for all inode in rNodes do

i_node = FastGetSolutionStepValue(AUX_INDEX) of the i node

vector ≡ FastGetSolutionStepValue(rVariable) of the i node

(rDestination[i_node][component] = vector[component]; component ∈ [0..TDim])

endfor

void FillOldVectorFromDatabase(Variable<array_1d<double,3>> & rVariable, CalcVectorType& rDestination, ModelPart::NodesContainerType& rNodes);

for all inode in rNodes do

i_node = FastGetSolutionStepValue(AUX_INDEX) of the i node

vector ≡ FastGetSolutionStepValue(rVariable, 1) of the i node
(rDestination[i_node])[component] = vector[component]; component ∈ [0..TDim)
endfor

void FillScalarFromDatabase(Variable<double>& rVariable, ValuesVectorType& rDestination,
ModelPart::NodesContainerType& rNodes);

for all i node in rNodes do
i_node = FastGetSolutionStepValue(AUX_INDEX) of the i node
scalar = FastGetSolutionStepValue(rVariable) of the i node
rDestination[i_node] = scalar
endfor

void FillOldScalarFromDatabase(Variable<double>& rVariable, ValuesVectorType& rDestination,
ModelPart::NodesContainerType& rNodes);

for all i node in rNodes do
i_node = FastGetSolutionStepValue(AUX_INDEX) of the i node
scalar = FastGetSolutionStepValue(rVariable, 1) of the i node
rDestination[i_node] = scalar
endfor

void WriteVectorToDatabase(Variable<array_1d<double,3>>& rVariable,
CalcVectorType& rOrigin,
ModelPart::NodesContainerType& rNodes);

for all i node in rNodes do
i_node = FastGetSolutionStepValue(AUX_INDEX) of the i node
vector = FastGetSolutionStepValue(rVariable) of the i node
vector[component] = (rOrigin[i_node])[component]; component ∈ [0..TDim)
endfor

void WriteScalarToDatabase(Variable<double>& rVariable, ValuesVectorType& rOrigin,
ModelPart::NodesContainerType& rNodes);

for all i node in rNodes do
i_node = FastGetSolutionStepValue(AUX_INDEX) of the i node
scalar = FastGetSolutionStepValue(rVariable) of the i node
scalar = rOrigin[i_node]
endfor

//destination = origin1 + value * Minv*origin
void Add_Minv_value(CalcVectorType& destination, const CalcVectorType& origin1, const double value,
const ValuesVectorType& Minv_vec, const CalcVectorType& origin);

#pragma omp parallel for
i_node = 0
while i_node < destination.size() do
dest = destination[i_node]
m_inv = Minv_vec[i_node]
void Add_Minv_value(ValuesVectorType& destination, const ValuesVectorType& origin1, const double value, const ValuesVectorType& Minv_vec, const ValuesVectorType& origin);

#pragma omp parallel for
i_node = 0

while i_node < destination.size() do
dest ≜ destination[i_node]
m_inv = Minv_vec[i_node]
origin_vec1 ≜ origin1[i_node]
origin_value ≜ origin[i_node]
temp = value * m_inv
dest[comp] = origin_vec1[comp] + temp * origin_value[comp]; comp ∈ [0..TDim]
i_node ++ endwhile

void SetToZero(CalcVectorType& data_vector);

#pragma omp parallel for
i_node = 0

while i_node < data_vector.size() do
aaa ≜ data_vector[i_node]
aaa[comp] = 0; comp ∈ [0..TDim]
i_node ++ endwhile

void SetToZero(ValuesVectorType& data_vector);

#pragma omp parallel for
i_node = 0

while i_node < data_vector.size() do
data_vector[i_node] = 0
i_node ++ endwhile

void AssignVectorToVector(const CalcVectorType& origin, CalcVectorType& destination);

#pragma omp parallel for
i_node = 0

while i_node < origin.size() do
orig ≜ origin[i_node]
dest ≜ destination[i_node]
dest[comp] = orig[comp]; comp ∈ [0..TDim]
i_node ++ endwhile

void AssignVectorToVector(const ValuesVectorType& origin, ValuesVectorType& destination);
#pragma omp parallel for
i_node = 0
while i_node < origin.size() do
destination[i_node] = origin[i_node]
i_node += + endwhile

Functions:

//functions to return private values
inline unsigned int GetNumberEdges(); return mNumberEdges
inline EdgesVectorType& GetEdgeValues(); return mNonzeroEdgeValues
inline IndicesVectorType& GetColumnIndex(); return mColumnIndex
inline IndicesVectorType& GetRowStartIndex(); return mRowStartIndex
inline ValuesVectorType& GetLumpedMass(); return mLumpedMassMatrix
inline ValuesVectorType& GetInvertedMass(); return mInvertedMassMatrix
inline CalcVectorType& GetDiagGradient(); return mDiagGradientMatrix
inline ValuesVectorType& GetHmin(); return mHmin

//function to calculate CSR index of edge ij
unsigned int GetCSRIndex(unsigned int NodeI, unsigned int NeighbourJ);
csr_index = mRowStartIndex[NodeI]
while csr_index != mRowStartIndex[NodeI + 1] do
    if mColumnIndex[csr_index] == NeighbourJ then
        return csr_index
    endif
csr_index += + endwhile
return csr_index

//function to get pointer to CSR tuple of edge ij
CSR_Tuple* GetTuplePointer(unsigned int NodeI, unsigned int NeighbourJ);
csr_index = mRowStartIndex[NodeI]
while csr_index != mRowStartIndex[NodeI + 1] do
    if mColumnIndex[csr_index] == NeighbourJ then
        return mNonzeroEdgeValues[csr_index]
    endif
csr_index += + endwhile
return mNonzeroEdgeValues[csr_index]

Private:

Variables:
unsigned int mNumberEdges; mNumberEdges = #edges

EdgesVectorType mNonzeroEdgeValues;
    mNonzeroEdgeValues = CSR vector to storage the G, L and M components of ij edges
IndicesVectorType mColumnIndex;
    mColumnIndex = vector to storage column indexes of values ≠ 0 of the matrix for each row
IndicesVectorType mRowStartIndex;
    mRowStartIndex = index vector to access to the beginning of each row in the column vector
ValuesVectorType mInvertedMassMatrix; mInvertedMassMatrix = inverse mass matrix
ValuesVectorType mHmin; mHmin = minimum height around a node
ValuesVectorType mLumpedMassMatrix;
    mLumpedMassMatrix = concentrated mass matrix
CalcVectorType mDiagGradientMatrix;
    mDiagGradientMatrix = diagonal of the matrix of gradients

Actions:

//functions to set up elemental mass matrices
void CalculateMassMatrix(boost::numeric::ublas::bounded_matrix<double,3,3>& mass_consistent, double volume);
i_node = 0
while i_node ≤ TDim do
    mass_consistent(i_node, i_node) = 1/6 * volume
    temp = 1/12 * volume
    j_neighbour = i_node + 1
    while j_neighbour ≤ TDim do
        mass_consistent(i_node, j_neighbour) = temp
        mass_consistent(j_neighbour, i_node) = temp
        j_neighbour += endwhile
    i_node += endwhile
void CalculateMassMatrix(boost::numeric::ublas::bounded_matrix<double,4,4>& mass_consistent, double volume);
i_node = 0
while i_node ≤ TDim do
    mass_consistent(i_node, i_node) = 1/10 * volume
    temp = 1/20 * volume
    j_neighbour = i_node + 1
    while j_neighbour ≤ TDim do
11.1.3. PureConvectionEdgeBased

Class redefinitions:

```cpp
// name for the self defined structure
typedef EdgesStructureType<TDim> CSR_Tuple;
typedef std::vector<CSR_Tuple> EdgesVectorType;

// name for row start and column index vectors
typedef std::vector<unsigned int> IndicesVectorType;

// defining matrix type for test calculations
typedef std::vector<array_1d<double, TDim>> CalcVectorType;

// defining type for local storage of nodal values
typedef std::vector<double> ValuesVectorType;

// defining types for matrix operations
typedef typename TSparseSpace::MatrixType TSystemMatrixType;
typedef typename TSparseSpace::VectorType TSystemVectorType;
```

Public:

Actions:

```cpp
// constructor
PureConvectionEdgeBased(MatrixContainer& mr_matrix_container, ModelPart& mr_model_part):
mr_matrix_container(mr_matrix_container), mr_model_part(mr_model_part) {};

// destructor
~PureConvectionEdgeBased() {};

// function to initialize fluid solver
void Initialize();
```

```cpp
n_nodes = mr_model_part.Nodes().size();
mWork.resize(n_nodes)
mPi.resize(n_nodes)
mUn.resize(n_nodes)
mUn1.resize(n_nodes)
mphi_n.resize(n_nodes)
```
mphi_n1.resize(n_nodes)
mA.resize(n_nodes)
mHmin.resize(n_nodes)
mtau.resize(n_nodes)

mr_matrix_container.FillVectorFromDatabase(VELOCITY, mUn1, mr_model_part.Nodes())

mr_matrix_container.FillOldVectorFromDatabase(VELOCITY, mUn, mr_model_part.Nodes())

mr_matrix_container.FillScalarFromDatabase(DISTANCE, mphi_n1, mr_model_part.Nodes())

mr_matrix_container.FillOldScalarFromDatabase(DISTANCE, mphi_n, mr_model_part.Nodes())

mFirstStep = true

aaa = mr_matrix_container.GetHmin()

mHmin[i_node] = aaa[i_node]; i ∈ [0..n_nodes]

//function to free dynamic memory
void Clear();

//function to set adequate time step size
void ComputeTimeStep(double CFLNumber);

delta_t = 1 * e^{10}

mr_matrix_container.FillVectorFromDatabase(VELOCITY, mUn1, mr_model_part.Nodes())

n_nodes = mUn1.size()

i_node = 0

while (i_node < n_nodes)

delta_t = CFLNumber * \frac{1}{\text{norm}_2(mUn1[i_node])/mHmin[i_node]}

delta_t = \text{MIN}(\text{SELF}, \delta t_{i, i})

i_node++

//perform MPI synchronization of the dt (minimum should be kept)

CurrentProcessInfo[DELTA_TIME] = delta_t

//function to solve fluid equations - fractional step 1: compute fractional momentum
void Solve();

rNodes = mr_model_part.Nodes()

n_nodes = rNodes.size()

rhs.resize(n_nodes)

mr_matrix_container.FillVectorFromDatabase(VELOCITY, mUn1, mr_model_part.Nodes())

mr_matrix_container.FillOldVectorFromDatabase(VELOCITY, mUn, mr_model_part.Nodes())

mr_matrix_container.FillScalarFromDatabase(DISTANCE, mphi_n1, mr_model_part.Nodes())

mr_matrix_container.FillOldScalarFromDatabase(DISTANCE, mphi_n, mr_model_part.Nodes())


delta_t = CurrentProcessInfo[DELTA_TIME]
coefficient = 1

CalculateAdvevtiveVelocity(mU0,mUn1,mA,coefficient)

time_inv = \frac{1}{\delta_t}

#pragma omp parallel for firstprivate(time_inv)

i_node = 0
while i_node < n_nodes do
h_i ≡ mHmin[i_node]
a_i ≡ mA[i_node]
vel_norm = norm_2(a_i)
mTau[i_node] = \frac{1}{2 \times \text{vel}_n / h_i + 0.01 \times \text{time}_n}
i_node += 1 endwhile

//mWork = mphi_n

mr_matrix_container.AssignVectorToVector(mphi_n,mWork)

//first step of Runge Kutta

mr_matrix_container.SetToZero(rhs)
CalculateRHS(mphi_n1,mA,rhs)
mr_matrix_container.Add_Minv_value(mWork,mWork,
\frac{\delta_t}{6}, mr_matrix_container.GetInvertedMass(), rhs)
mr_matrix_container.GetInvertedMass(), rhs)

//second step

mr_matrix_container.SetToZero(rhs)
CalculateRHS(mphi_n1,mA,rhs)
mr_matrix_container.Add_Minv_value(mWork,mWork,
\frac{\delta_t}{3}, mr_matrix_container.GetInvertedMass(), rhs)
mr_matrix_container.GetInvertedMass(), rhs)

//third step

CalculateAdvevtiveVelocity(mU0,mUn1,mA,coefficient)

mr_matrix_container.SetToZero(rhs)
CalculateRHS(mphi_n1,mA,rhs)
mr_matrix_container.Add_Minv_value(mWork,mWork,
\frac{\delta_t}{3}, mr_matrix_container.GetInvertedMass(), rhs)
mr_matrix_container.GetInvertedMass(), rhs)
// fourth step

CalculateAdvectiveVelocity(mUn, mUn1, mA, coefficient)
mr_matrix_container.SetToZero(rhs)
CalculateRHS(mphi_n1, mA, rhs)

mr_matrix_container.Add_Minv_value(mWork, mWork,
\[
\delta t
\]
6, mr_matrix_container.GetInvertedMass(), rhs)

// compute right-hand side

mr_matrix_container.AssignVectorToVector(mWork, mphi_n1)

mr_matrix_container.WriteScalarToDatabase(DISTANCE, mphi_n1, mr_model_part.Nodes())

// function to calculate right-hand side of fractional momentum equation
void CalculateRHS(const ValuesVectorType& mphi, const CalcVectorType& convective_velocity,
ValuesVectorType& rhs);

n_nodes = mphi.size()

// calculating the convective projection

#pragma omp parallel for
i_node = 0

while i_node < n_nodes do
    pi_i ≡ mPi[i_node]
    phi_i ≡ mphi[i_node]
    pi_i = 0
    a_i ≡ convective_velocity[i_node]
    csr_index = mr_matrix_container.GetRowIndex()[i_node]

    while csr_index ≠ mr_matrix_container.GetRowIndex()[i_node + 1] do
        j_neighbour = mr_matrix_container.GetColumnIndex()[csr_index]
        a_j ≡ convective_velocity[j_neighbour]
        phi_j ≡ mphi[j_neighbour]
        edge_i, j ≡ mr_matrix_container.GetEdgeValues()[csr_index]
        edge_ij.Add_ConvectiveContribution(pi_i, a_i, phi_i, a_j, phi_j)
        csr_index + + endwhile
    m_inv = mr_matrix_container.GetInvertedMass()[i_node]
    pi_i *= m_inv
    i_node + + endwhile

#pragma omp parallel for private(stab_low, stab_high)

i_node = 0

while i_node < n_nodes do
    rhs_i ≡ rhs[i_node]
    phi_i ≡ mphi[i_node]
a_i ≡ convective_velocity[i_node]
pi_i ≡ mPi[i_node]

//initializing with the external forces (e.g. gravity)

void CalculateAdvectiveVelocity(const CalcVectorType& mUn, const CalcVectorType& mUn1, CalcVectorType& mA, double coefficient);

// Parallel for loop to calculate advective velocity
#pragma omp parallel for
for (i_node = 0; i_node < n_nodes; i_node++)

Variables:

MatrixContainer& mr_matrix_container;
ModelPart& mr_model_part;
bool msmooth_convective_velocity;
bool minclude_shock_capturing;
// nodal values
CalcVectorType mUn1,mUn; \( \vec{U} = \) velocity vector; \( mU_{n1} = \vec{U} \) in \( t_{n+1} \); \( mUn = \vec{U} \) in \( t_n \)
ValuesVectorType mWork, mPi;
ValuesVectorType mphi_n, mphi_n1;

\[ \vec{P} = \text{pressure vector} \]
\[ mphi_n = \vec{P} \text{ in } t_n \]
\[ mphi_{n1} = \vec{P} \text{ in } t_{n+1} \]
CalcVectorType mA; \( mA = \) advective velocity vector
ValuesVectorType mHmin;

\( mHmin = \) minimum length of the edges around the edges which surround each node
bool mFirstStep; \( mFirstStep = \) flag for \( t_0 \)
ValuesVectorType mTau; \( mTau = \) temporal unit size
11.2. Laptop’s /proc/cpuinfo

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Core(TM)2 CPU T7200 @ 2.00GHz
stepping        : 6
cpu MHz         : 1994.996
cache size      : 4096 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
                  cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx lm
                  constant_tsc arch_perfmon pebs bts rep_good pni monitor ds_cpl vmx est tm2
                  ssse3 cx16 xtpr lahf_lm
bogomips        : 3989.99
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

```
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Core(TM)2 CPU T7200 @ 2.00GHz
stepping        : 6
cpu MHz         : 1994.996
cache size      : 4096 KB
physical id     : 0
siblings        : 2
core id         : 1
cpu cores       : 2
```
fpu : yes
fpu_exception : yes
cpuid_level : 10
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx lm
count_tsc arch_perfmon pebs bts rep_good pni monitor ds_cpl vmx est tm2
ssse3 cx16 xtpr lahf_lm
bogomips : 3990.06
clflush_size : 64
cache_alignment : 64
address_sizes : 36
11.3. env64.sh

This is the environment script {{env64.sh,env64-xl.sh}} created by the automatic installation script for installation 4.

```bash
export PATH=
/gpfs/projects/bsc15/kratos/full/migration/bin/:
/gpfs/apps/GCC/4.4.0/bin/:
/gpfs/apps/CMAKE/cmake-2.8.1/bin/:

$PATH
export LD_LIBRARY_PATH=
/gpfs/projects/bsc15/kratos/full/migration/lib/:
/gpfs/projects/bsc15/kratos/full/migration/kratos-svn/kratos/kratos/libs/:
/opt/ibmcmp/vacpp/10.1/lib64/:
/gpfs/apps/GCC/4.4.0/lib64/:
/go\ps/apps/CMAKE/cmake-2.8.1/lib/:
/opt/ibmcmp/xlf/12.1/lib64/:
/opt/ibmcmp/lib64/:
/opt/ibmcmp/mpich-mx/64/lib/shared/:

$LD_LIBRARY_PATH
export MP_CC={gcc,xlc}
export MP_F90={gfortran,xlf90}
export OBJECT_MODE=64
export MP_CXX={g++,xlc++}
export MP_FC={gfortran,xlf}
export PYTHONPATH=
/gpfs/projects/bsc15/kratos/full/migration/lib/:
/gpfs/projects/bsc15/kratos/full/migration/kratos-svn/kratos/kratos/libs/:
/opt/ibmcmp/vacpp/10.1/lib64/:
/gpfs/apps/GCC/4.4.0/lib64/:
/go\ps/apps/CMAKE/cmake-2.8.1/lib/:
/opt/ibmcmp/xlf/12.1/lib64/:
/opt/ibmcmp/lib64/:
/opt/ibmcmp/mpich-mx/64/lib/shared/:

$PYTHONPATH
```
11.4. do-configure

This is do-configure file created by the automatic installation script for installation 4.

```
TRILINOS_HOME="/gpfs/projects/bsc15/kratos/full/migration/trilinos-10.2.1-Source"
EXTRA_LINK_FLAGS="-lgfortran"
EXTRA_ARGS=@
cmake \
  -D CMAKE_INSTALL_PREFIX:PATH="/gpfs/projects/bsc15/kratos/full/migration/trilinos-10.2.1-Build" \
  -D CMAKE_BUILD_TYPE:STRING=RELEASE \n  -D TPL_ENABLE_MPI:BOOL=ON \n  -D BLAS_LIBRARY_DIRS:PATH="/gpfs/projects/bsc15/kratos/full/migration/lib" \n  -D LAPACK_LIBRARY_DIRS:PATH="/gpfs/projects/bsc15/kratos/full/migration/lib" \n  -D LAPACK_LIBRARY_NAMES:STRING="lapack" \n  -D MPI_BASE_DIR:PATH="/usr/bin" \n  -D MPI_INCLUDE_DIRS:PATH="/opt/osshpc/mpich/mx/64/include/" \n  -D BUILD_SHARED_LIBS:BOOL=ON \n  -D TPL_ENABLE_SuperLUDist:BOOL=ON \n  -D SuperLUDist_INCLUDE_DIRS:PATH="/gpfs/projects/bsc15/kratos/full/migration/include/" \n  -D SuperLUDist_LIBRARY_DIRS:PATH="/gpfs/projects/bsc15/kratos/full/migration/lib/" \n  -D SuperLUDist_LIBRARY_NAMES:STRING="superlu_dist" \n  -D TPL_METIS_INCLUDE_DIRS:PATH="/gpfs/projects/bsc15/kratos/full/migration/include/" \n  -D TPL_METIS_LIBRARY_DIRS:PATH="/gpfs/projects/bsc15/kratos/full/migration/lib/" \n  -D TPL_METIS_LIBRARY_NAMES:STRING="metis" \n  -D TPL_ENABLE_ParMETIS:BOOL=ON \n  -D ParMETIS_INCLUDE_DIRS:PATH="/gpfs/projects/bsc15/kratos/full/migration/include/" \n  -D ParMETIS_LIBRARY_DIRS:PATH="/gpfs/projects/bsc15/kratos/full/migration/lib/" \n  -D TPL_ENABLE_Amesos:BOOL=ON \n  -D Amosos_ENABLE_SuperLUDist:BOOL=ON \n  -D Amosos_ENABLE_Anasazi:BOOL=ON \n  -D Amosos_ENABLE_AztecOO:BOOL=ON \n  -D AztecOO_ENABLE_Teuchos:BOOL=ON \n  -D Amosos_ENABLE_Didaasko:BOOL=ON \n  -D Amosos_ENABLE_Epetra:BOOL=ON \n  -D Amosos_ENABLE_PyTrilinos:BOOL=OFF \n  -D Trilinos_ENABLE_Amesos:BOOL=ON \n  -D Trilinos_ENABLE_Amesos_ENABLE_SuperLUDist:BOOL=ON \n  -D Trilinos_ENABLE_Anasazi:BOOL=ON \n  -D Trilinos_ENABLE_AztecOO:BOOL=ON \n  -D Trilinos_ENABLE_Didaasko:BOOL=ON \n  -D Trilinos_ENABLE_Didaasko:BOOL=ON \n  -D Trilinos_ENABLE_Didaasko:BOOL=ON \n  -D Trilinos_ENABLE_Didaasko:BOOL=ON \n  -D Trilinos_ENABLE_Didaasko:BOOL=ON \n  -D Trilinos_ENABLE_Didaasko:BOOL=ON \n  -D Trilinos_ENABLE_Didaasko:BOOL=ON \n  -D Trilinos_ENABLE_Didaasko:BOOL=ON
```

23 / juny / 2011

Facultat d’Informàtica de Barcelona | Edgar Juanpere Cañameras
lxl -L/lib64 -lc-2.4 -I/opt/osshpc/mpich-mx/64/include/ -L/opt/osshpc/mpich-mx/64/lib/shared -lmpich " \
-D CMAKE_Fortran_FLAGS:STRING="-O5 -qunroll=yes -qmaxmem=-1 -q64 -qextname -qtune=ppc970 -qarch=ppc970 -L/opt/ibmcmp/xlf/12.1/lib64 -lxlopt" \
-D MPI_Fortran_FLAGS:STRING="-O5 -qunroll=yes -qmaxmem=-1 -q64 -qextname -qtune=ppc970 -qarch=ppc970 -L/opt/ibmcmp/xlf/12.1/lib64 -lxlopt" \\
$EXTRA_ARGS \n${TRILINOS_HOME}
11.5. Jamroot

For this project, I used Jamroot placed under these lines, green words are comments and yellow marks are for lines that must be customized (current text correspond to test installation). The arrows explain what every yellow marked line is:

- **GCC flags**
  - using gcc : : : `<cxxflags>` "-fopenmp -Ano-pragmaas" ;
  - using sun : : : `<cxxflags>` "-w -DBOOST UBLAS_UNSUPPORTED_COMPILER=0" "<cxxflags>" "-w -DBOOST UBLAS_UNSUPPORTED_COMPILER=0";

- **MPI wrapper for C/C++ path**
  - # uncomment and specify compiler
  - using mpi : /usr/bin/mpicc ;

- **Boost library path Boost include path**
  - # defining "common includes" and external libraries
  - path-constant TOP : . ;
  - path-constant BOOST_LIB_DIR : /usr/local/lib/ ;
  - path-constant BOOST_INCLUDE_DIR : /usr/local/include/boost-
    -1_38/ ;

- **BLAS library path LAPACK library path**
  - path-constant BLAS_LIB_DIR : /usr/lib/ ;
  - path-constant LAPACK_LIB_DIR : /usr/lib/ ;

- **MPI library path**
  - # Path to CUDA - required to compile and use GPU solvers - strictly optional
  - path-constant CUDA_LIB_DIR : /usr/local/cuda/lib ;
  - path-constant CUDA_INCLUDE_DIR : /usr/local/cuda/include ;

- **Python include path**
  - #uncomment and specify compiler
  - using python ;

- **Metis library path**
  - # Path to the mkl library (STRICTLY OPTIONAL)
  - path-constant MKL_INCLUDE_DIR : /opt/intel/mkl/10.0.1.014/include/ ;
  - path-constant MKL_LIB_DIR : /opt/intel/mkl/10.0.1.014/lib/ ;
# WARNING - `fPIC` is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the `kratos"static" library <warnings>

```bash
# gcc settings
<toolset>gcc:<cflags>"-fPIC -funroll-loops" #settings for external libraries
<toolset>gcc:<cxxflags>"-fPIC -ansi -funroll-loops -ffast-math -Wno-unknown-pragmas"
```

```bash
# msvc settings
<toolset>msvc:<linkflags>"NODEFAULTLIB:libcmt"
<toolset>msvc:<cxxflags>"/D_SCL_SECURE_NO_DEPRECATE /wd4335"
```

```bash
# intel settings
++ <toolset>intel:<cxxflags>"-fPIC -ansi -funroll-loops -ffast-math"
  default-build release;
```

```bash
#settings for external libraries <warnings>
<toolset>gcc:<cflags>
  -fPIC
  -funroll
  -loops
</toolset>
```

```bash
#kratos "static" library
<toolset>gcc:<cxxflags>
  -fPIC
  -ansi
  -funroll
  -loops
  -ffast
  -math
  -Wno
  -unknown
  -pragmas
</toolset>
```

```bash
#msvc settings
<toolset>msvc:<linkflags>"/NODEFAULTLIB:libcmt"
<toolset>msvc:<cxxflags>"/D_SCL_SECURE_NO_DEPRECATE /wd4335"
```

```bash
#intel settings
##settings for external libraries <warnings>
<toolset>intel:<cxxflags>
  -fPIC
  -ansi
  -funroll
  -loops
  -ffast
  -math
</toolset>
```

```bash
# default - build release;
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```

```bash
# WARNING - fPIC is NEEDED to compile on 64 bit systems - it has to be specified here in order to include it in the kratos"static" library
```
use-project /kratos-prj/MetisApplication : ${TOP}/applications/metis_application ;
## REQUIRES TRILINOS TO BE INSTALLED
use-project /kratos-prj/TrilinosApplication : ${TOP}/applications/trilinos_application ;

## MKL solvers: will not be compiled by default


# GPU Solvers: needs CUDA to be installed and appropriate series GPU card (GTX260, GTX 280, etc.) with the *latest* driver
# Before trying to compile this application, go to
"~/kratos/applications/gpu_solvers_application/custom_external_libraries" and issue a
use-project /kratos-prj/gpu_solvers_application : ${TOP}/applications/gpu_solvers_application ;

 Ubcommented lines activate MetisApplication and TrilinosApplication

Uncommented lines activate MetisApplication and TrilinosApplication solvers
11.6. kratos-submit.sh

This is the automatic submission script of Kratos jobs in Mare Nostrum. In order to work, in the directory where it is placed must exists:

- env64.sh file
- Subdirectory with the name of the problem (ahmed_25, TELESCOPIO and so on) and all the needed data in it (.mdpa, configuration variables...)
- A script called kratos-submit-{name_of_the_problem}.sh which casts python script of the problem.

Pitifully, wall time is not automatic so it is necessary to change in this script that variable manually.

#!/bin/bash

MAX=10000
WALL_TIME="7:59:59"

if [ "$1" = "-h" ]
then
echo "======HELP======"
echo "Usage:"
echo "kratos-submit.sh -h => help is displayed"
echo "kratos-submit.sh n => n=[2,$MAX] (number of processors)"
echo "ahmed_25 is launched with n processors"
echo "kratos-submit.sh n directory => n=[2,$MAX] (number of processors)"
echo "directory is launched with n processors"
echo "======HELP======"
exit
fi

if [ "$1" = "" ]
then
 echo "No arguments: try -h option"
 exit
else
 echo "An argument was found: "$1
fi

case "$1" in
 "-[0-9]*")
 echo "Numeric argument"
 if [ "$1" -le 1 ]
 then
  echo "Less than 2 processors is not allowed"
  exit
 fi
 if [ "$1" -gt "$MAX" ]
 then
  echo "More than $MAX processors is not allowed"
  exit
 fi
 ;;
*)
 echo "Non numeric argument"
 exit
 ;;
 esac
if [ "$2" = "" ]
then
  DIRECTORY="ahmed_25"
else
  DIRECTORY="$2"
fi

if [ -d "$DIRECTORY" ]
then
  echo "$DIRECTORY exists"
else
  echo "$DIRECTORY does not exist"
  exit
fi

if [ -a kratos-submit.q ]
then
  rm kratos-submit.q
fi

echo '#!/bin/bash
#
# @ initialdir = .
# @ job_name = kratos-submit
# @ total_tasks = '$1'
# @ tasks_per_node = 4
# @ nodeset = clos
# @ wall_clock_limit = '$WALL_TIME'
# @ queue

source 'pwd'/env64.sh
hostname
time
sl_get_machine_list
sl_get_machine_list > machinefile.$$

echo EXECUTING...
echo "###################################################################

cd 'pwd'/"$DIRECTORY"
mpirun -np '$1' -machinefile 'pwd'/machinefile.$$ 'pwd'/kratos-submit-"$DIRECTORY".sh

echo "###################################################################

hostname
time
' > kratos-submit.q

echo "Script created automatically!"
echo """
11.7. Compilers and their flags

F2C [55] (Fortran 2 C) and CBLAS were not finally used, because FBLAS includes all operations of CBLAS and adds many more. In addition, CBLAS compiled with IBM XL needs F2C in order to work, so as we used FBLAS it was not needed at all.

Also, we do not put all flags (e.g., there are not all library flags because it is only necessary to point libraries which are needed when using certain flags)

<table>
<thead>
<tr>
<th>Installation</th>
<th>F2C</th>
<th>CBLAS</th>
<th>FBLAS</th>
<th>METIS</th>
<th>LAPACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N/A</td>
<td>-O3 – m64</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
</tr>
<tr>
<td>2</td>
<td>N/A</td>
<td>N/A</td>
<td>-O5 –q64</td>
<td>-O5 –q64</td>
<td>-O5 –q64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-qextname</td>
<td>-qextname</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-qunroll=yes</td>
<td>-qunroll=yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-qhot=</td>
<td>-qhot=</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>noarraypad:level=1:nosi</td>
<td>noarraypad:level=1:nosi</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>md:novector</td>
<td>md:novector</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-L/opt/ibmcmp/xlf/12.1/lib64 – lxlopt -xl</td>
<td>-L/opt/ibmcmp/xlf/12.1/lib64 – lxlopt -xl</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
</tr>
<tr>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
<td>-O5 –q64</td>
<td>-O5 –q64</td>
<td>-O5 –q64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-qextname</td>
<td>-qextname</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-qunroll=yes</td>
<td>-qunroll=yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-L/opt/ibmcmp/xlf/12.1/lib64 – lxlopt -xl</td>
<td>-L/opt/ibmcmp/xlf/12.1/lib64 – lxlopt -xl</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>N/A</td>
<td>N/A</td>
<td>-O5 –q64</td>
<td>-O5 –q64</td>
<td>-O5 –q64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-qextname</td>
<td>-qextname</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-qunroll=yes</td>
<td>-qunroll=yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-qhot=</td>
<td>-qhot=</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>noarraypad:level=1:nosi</td>
<td>noarraypad:level=1:nosi</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>md:novector</td>
<td>md:novector</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-L/opt/ibmcmp/xlf/12.1/lib64 – lxlopt -xl</td>
<td>-L/opt/ibmcmp/xlf/12.1/lib64 – lxlopt -xl</td>
<td></td>
</tr>
<tr>
<td>Installation</td>
<td>SUPERLU</td>
<td>TRILINOS*</td>
<td>BOOST</td>
<td>KRATOS</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------</td>
<td>-------------------</td>
<td>----------------</td>
<td>-----------------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-O5 –q64</td>
<td>-O5 –q64</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-qextname</td>
<td>-qextname</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-qunroll=yes</td>
<td>-qunroll=yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-L/opt/xlf/12.1/lib64 –lxlopt –xl</td>
<td>-L/opt/xlf/12.1/lib64 –lxlopt –xl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-O5 –q64</td>
<td>-O5 –q64</td>
<td>-O3 –m64</td>
<td>-O3 –m64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-qextname</td>
<td>-qextname</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-qunroll=yes</td>
<td>-qunroll=yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-qhot=</td>
<td>-qhot=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>noarraypad:level=1:nosimd: novector</td>
<td>noarraypad:level=1:nosimd: novector</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-L/opt/xlf/12.1/lib64 –lxlopt –xl</td>
<td>-L/opt/xlf/12.1/lib64 –lxlopt –xl</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Trilinos compiles with C and FORTRAN compilers. In case of C compiler –O5 option has to be replaced by –O3 and –qhot option is removed.
If compiled with –O5 option the compilation fails because there is no enough space.
If compiled with –qhot option, the execution crashes with the message “Not enough memory allocated for overlapping Error: Out of space due to poor estimate of memory needed.”
As we can observe, the script is serial (installs a package when the previous has finished).

```bash
#!/bin/bash

echo "Initial current date: " `date`

#BEGIN OF USED VARIABLES
ADDITIONAL_TOOLS="/gpfs/apps/GCC/4.4.0/bin/:/gpfs/apps/CMAKE/cmake-2.8.1/bin/
ADDITIONAL_LIBRARIES="/opt/ibmcmp/vacpp/10.1/lib64/:/opt/ibmcmp/xlmp/1.8/lib64/:/gpfs/apps/CMAKE/cmake-2.8.1/lib/:/opt/ibmcmp/xlf/12.1/lib64/:/opt/ibmcmp/xlmp/1.8/lib64/:/opt/ibmcmp/lib64/"
F2C_DIR="f2c"
SUPERLU_DIR="SuperLU_DIST_2.3"
LAPACK_DIR="lapack-3.3.0"
METIS_DIR="ParMetis-3.1.1"
TRILINOS_S_DIR="trilinos-10.2.1-Source"
TRILINOS_B_DIR="trilinos-10.2.1-Build"
PYTHON_DIR="Python-2.5.5"
PYTHON_VERSION="2.5"
BOOST_DIR="boost_1_46_0"
KRATOS_DIR="kratos-svn"
#END OF USED VARIABLES

if [ "$1" = "-h" ]
then
  echo "====HELP===="
  echo "Usage:"
  echo "install             => needed files are deployed"
  echo "install -h          => help is displayed"
  echo "install clean       => unnecessary files for execution are deleted"
  echo "install clean all   => all installation is deleted"
  echo "====HELP===="
  exit
fi

DIR=$(pwd)

if [ "$1" = "clean" ]
then
  if [ -d "$DIR/$F2C_DIR" ]
  then
    echo "Deleting $DIR/$F2C_DIR..."
    rm -r "$DIR/$F2C_DIR"
  fi
  if [ -d "$DIR/$SUPERLU_DIR" ]
  then
    echo "Deleting $DIR/$SUPERLU_DIR..."
    rm -r "$DIR/$SUPERLU_DIR"
  fi
  if [ -d "$DIR/$LAPACK_DIR" ]
  then
    echo "Deleting $DIR/$LAPACK_DIR..."
    rm -r "$DIR/$LAPACK_DIR"
  fi
  if [ -d "$DIR/$METIS_DIR" ]
  then
    echo "Deleting $DIR/$METIS_DIR..."
    rm -r "$DIR/$METIS_DIR"
  fi
  if [ -d "$DIR/$TRILINOS_S_DIR" ]
  then
    echo "Deleting $DIR/$TRILINOS_S_DIR..."
    rm -r "$DIR/$TRILINOS_S_DIR"
  fi
  if [ -d "$DIR/$TRILINOS_B_DIR" ]
  then
    echo "Deleting $DIR/$TRILINOS_B_DIR..."
    rm -r "$DIR/$TRILINOS_B_DIR"
  fi
  if [ -d "$DIR/$KRATOS_DIR" ]
  then
    echo "Deleting $DIR/$KRATOS_DIR..."
    rm -r "$DIR/$KRATOS_DIR"
  fi
  if [ -d "$DIR/$PYTHON_DIR" ]
  then
    echo "Deleting $DIR/$PYTHON_DIR..."
    rm -r "$DIR/$PYTHON_DIR"
  fi
  if [ -d "$DIR/$BOOST_DIR" ]
  then
    echo "Deleting $DIR/$BOOST_DIR..."
    rm -r "$DIR/$BOOST_DIR"
  fi
fi
```
rm -r $DIR/TRILINOS_S_DIR
fi
if [ -d "$DIR/TRILINOS_B_DIR" ]
then
  echo "Deleting $DIR/TRILINOS_B_DIR..."
  rm -r $DIR/TRILINOS_B_DIR
fi
if [ -d "$DIR/PYTHON_DIR" ]
then
  echo "Deleting $DIR/PYTHON_DIR..."
  rm -r $DIR/PYTHON_DIR
fi
if [ -d "$DIR/BOOST_DIR" ]
then
  echo "Deleting $DIR/BOOST_DIR..."
  rm -r $DIR/BOOST_DIR
fi
if [ "$2" = "all" ]
then
  if [ -f "$DIR/env64-xl.sh" ]
  then
    echo "Deleting $DIR/env64-xl.sh..."
    rm $DIR/env64-xl.sh
  fi
  if [ -f "$DIR/env64.sh" ]
  then
    echo "Deleting $DIR/env64.sh..."
    rm $DIR/env64.sh
  fi
  if [ -d "$DIR/bin" ]
  then
    echo "Deleting $DIR/bin..."
    rm -r $DIR/bin
  fi
  if [ -d "$DIR/share" ]
  then
    echo "Deleting $DIR/share..."
    rm -r $DIR/share
  fi
  if [ -d "$DIR/include" ]
  then
    echo "Deleting $DIR/include..."
    rm -r $DIR/include
  fi
  if [ -d "$DIR/lib" ]
  then
    echo "Deleting $DIR/lib..."
    rm -rf $DIR/lib
  fi
  if [ -d "$DIR/KRATOS_DIR" ]
  then
    echo "Deleting $DIR/KRATOS_DIR..."
    rm -rf $DIR/KRATOS_DIR
  fi
else
  echo "Directory lib exists"
fi
if [ -d "$DIR/lib" ]
then
  echo "Directory lib does not exist, creating it..."
  mkdir $DIR/lib
fi
if [ -d "$DIR/include" ]
then

echo "Directory include exists"
else
echo "Directory include does not exist, creating it..."
mkdir $DIR/include
fi

echo "Current date: " `date`
echo "Creating env64-xl.sh..."
cd $DIR/files/env64/.
./ibm.sh $DIR $ADDITIONAL_TOOLS $ADDITIONAL_LIBRARIES $KRATOS_DIR
source $DIR/env64-xl.sh

echo "Current date: " `date`
PC=1
echo ">=[STEP "$PC"]=< Compiling F2C library..."
cd $DIR
if [ -d "$DIR/$F2C_DIR" ]
then
echo "$DIR/$F2C_DIR exists..."
else
echo "$DIR/$F2C_DIR does not exist..."
mkdir $F2C_DIR
unzip --v -d $DIR/$F2C_DIR $DIR/files/f2c/$F2C_DIR.zip
fi
cp $DIR/files/f2c/Makefile $DIR/$F2C_DIR/
cp $DIR/files/f2c/main.c $DIR/$F2C_DIR/
cd $DIR/$F2C_DIR/
make
make libf2c.so
cp $DIR/$F2C_DIR/libf2c.so $DIR/lib/

echo "Current date: " `date`
PC=2
echo ">=[STEP "$PC"]=< Compiling CBLAS library..."
cd $DIR
if [ -d "$DIR/$SUPERLU_DIR" ]
then
echo "$DIR/$SUPERLU_DIR exists..."
else
echo "$DIR/$SUPERLU_DIR does not exist..."
tar xf $DIR/files/superLU/$SUPERLU_DIR.tar.gz
fi
cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile.cblas $DIR/$SUPERLU_DIR/CBLAS/Makefile
cd $DIR/$SUPERLU_DIR/
echo $DIR > $DIR/$SUPERLU_DIR/dir.tmp
echo $SUPERLU_DIR > $DIR/$SUPERLU_DIR/superlu.tmp
echo $METIS_DIR > $DIR/$SUPERLU_DIR/metis.tmp
echo $DIR > $DIR/$SUPERLU_DIR/CBLAS/dir.tmp
echo $SUPERLU_DIR > $DIR/$SUPERLU_DIR/CBLAS/superlu.tmp
cp *.tmp EXAMPLE/
cp *.tmp FORTRAN/
make blaslib
cp $DIR/$SUPERLU_DIR/lib/libcblas.so $DIR/lib/

echo "Current date: " `date`
PC=3
echo ">=[STEP "$PC"]=< Compiling FBLAS library..."
cd $DIR
if [ -d "$DIR/$LAPACK_DIR" ]
then
echo "$DIR/$LAPACK_DIR exists..."
else
echo "$DIR/$LAPACK_DIR does not exist..."
tar xf $DIR/files/lapack/$LAPACK_DIR.tgz
fi
cp $DIR/files/lapack/make.inc $DIR/$LAPACK_DIR
cd $DIR/$LAPACK_DIR/

if [-d "$DIR/$METIS_DIR"]
then
  echo "$DIR/$METIS_DIR exists..."
else
  echo "$DIR/$METIS_DIR does not exist..."
tar xf $DIR/files/metis/$METIS_DIR.tar.gz
fi
cp $DIR/files/metis/Makefile.in $DIR/$METIS_DIR/
cp $DIR/files/metis/Makefile.metis $DIR/$METIS_DIR/METISLib/Makefile

if [ -d "$DIR/$SUPERLU_DIR" ]
then
  echo "$DIR/$SUPERLU_DIR exists..."
else
  echo "$DIR/$SUPERLU_DIR does not exist..."
tar xf $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR.tar.gz
fi
cp $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile.superlu $DIR/$SUPERLU_DIR/SRC/Makefile

then
echo "Current date: " `date`
else
echo "$DIR/$LAPACK_DIR exists..."
tar xf $DIR/files/lapack/$LAPACK_DIR.tgz
fi
cp $DIR/files/lapack/make.inc $DIR/$LAPACK_DIR
cd $DIR/$LAPACK_DIR/

cp $DIR/files/metis/Makefile.in $DIR/$METIS_DIR/
cp $DIR/files/metis/Makefile.metis $DIR/$METIS_DIR/METISLib/Makefile

cp $DIR/files/superLU/Makefile.superlu $DIR/$SUPERLU_DIR/SRC/Makefile

cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/

then
echo "$DIR/$METIS_DIR exists..."
else
echo "$DIR/$METIS_DIR does not exist..."
tar xf $DIR/files/metis/$METIS_DIR.tar.gz
fi
cp $DIR/files/metis/Makefile.in $DIR/$METIS_DIR/
cp $DIR/files/metis/Makefile.metis $DIR/$METIS_DIR/METISLib/Makefile

cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/

then
echo "$DIR/$SUPERLU_DIR exists..."
else
echo "$DIR/$SUPERLU_DIR does not exist..."
tar xf $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR.tar.gz
fi
cp $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile.superlu $DIR/$SUPERLU_DIR/SRC/Makefile

cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/

cp $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile.superlu $DIR/$SUPERLU_DIR/SRC/Makefile

cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/

then
echo "$DIR/$SUPERLU_DIR exists..."
else
echo "$DIR/$SUPERLU_DIR does not exist..."
tar xf $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR.tar.gz
fi
cp $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile.superlu $DIR/$SUPERLU_DIR/SRC/Makefile

cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/

then
echo "$DIR/$METIS_DIR exists..."
else
echo "$DIR/$METIS_DIR does not exist..."
tar xf $DIR/files/metis/$METIS_DIR.tar.gz
fi
cp $DIR/files/metis/Makefile.in $DIR/$METIS_DIR/
cp $DIR/files/metis/Makefile.metis $DIR/$METIS_DIR/METISLib/Makefile

cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/

then
echo "$DIR/$SUPERLU_DIR exists..."
else
echo "$DIR/$SUPERLU_DIR does not exist..."
tar xf $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR.tar.gz
fi
cp $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile.superlu $DIR/$SUPERLU_DIR/SRC/Makefile

cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/

then
echo "$DIR/$SUPERLU_DIR exists..."
else
echo "$DIR/$SUPERLU_DIR does not exist..."
tar xf $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR.tar.gz
fi
cp $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile.superlu $DIR/$SUPERLU_DIR/SRC/Makefile

cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/

then
echo "$DIR/$SUPERLU_DIR exists..."
else
echo "$DIR/$SUPERLU_DIR does not exist..."
tar xf $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR.tar.gz
fi
cp $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile.superlu $DIR/$SUPERLU_DIR/SRC/Makefile

cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/

then
echo "$DIR/$SUPERLU_DIR exists..."
else
echo "$DIR/$SUPERLU_DIR does not exist..."
tar xf $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR.tar.gz
fi
cp $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile.superlu $DIR/$SUPERLU_DIR/SRC/Makefile

cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/

then
echo "$DIR/$SUPERLU_DIR exists..."
else
echo "$DIR/$SUPERLU_DIR does not exist..."
tar xf $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR.tar.gz
fi
cp $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile.superlu $DIR/$SUPERLU_DIR/SRC/Makefile

cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/

then
echo "$DIR/$SUPERLU_DIR exists..."
else
echo "$DIR/$SUPERLU_DIR does not exist..."
tar xf $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR.tar.gz
fi
cp $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile.superlu $DIR/$SUPERLU_DIR/SRC/Makefile

cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/

then
echo "$DIR/$SUPERLU_DIR exists..."
else
echo "$DIR/$SUPERLU_DIR does not exist..."
tar xf $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR.tar.gz
fi
cp $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile.superlu $DIR/$SUPERLU_DIR/SRC/Makefile

cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/

then
echo "$DIR/$SUPERLU_DIR exists..."
else
echo "$DIR/$SUPERLU_DIR does not exist..."
tar xf $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR.tar.gz
fi
cp $DIR/files/superLU/make.inc $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
cp $DIR/files/superLU/Makefile.superlu $DIR/$SUPERLU_DIR/SRC/Makefile

cp $DIR/files/superLU/Makefile $DIR/$SUPERLU_DIR/
echo $SUPERLU_DIR > $DIR/$SUPERLU_DIR/SRC/superlu.tmp
make all

cp $DIR/$SUPERLU_DIR/lib/libsuperlu_dist.so $DIR/lib/
cp $DIR/$SUPERLU_DIR/SRC/*.h $DIR/include/

echo "Current date: " `date`
PC=7
echo ">={STEP "$PC"}={ Creating TRILINOS library..."
cd $DIR

if [ -d "$DIR/$TRILINOS_S_DIR" ]
then
    echo "$DIR/$TRILINOS_S_DIR exists..."
else
    echo "$DIR/$TRILINOS_S_DIR does not exist..."
tar xf $DIR/files/trilinos/$TRILINOS_S_DIR.tar.gz
fi

if [ -d "$DIR/$TRILINOS_B_DIR" ]
then
    echo "$DIR/$TRILINOS_B_DIR exists..."
else
    echo "$DIR/$TRILINOS_B_DIR does not exist..."
    mkdir $DIR/$TRILINOS_B_DIR
fi

echo "Creating $TRILINOS_B_DIR/do-configure..."
cd $DIR/files/trilinos/
./create_do-configure.sh $DIR $TRILINOS_S_DIR $TRILINOS_B_DIR

echo "Adding modified TRILINOS headers and source files..."
cd $DIR/files/trilinos/
./mod_files.sh $DIR $TRILINOS_S_DIR $TRILINOS_B_DIR

echo "Preparing TRILINOS compilation..."
cd $DIR/$TRILINOS_B_DIR
./do-configure
make VERBOSE=1
make install

echo "Installing TRILINOS headers and libraries..."
cp $DIR/$TRILINOS_B_DIR/lib/* $DIR/lib/
cp $DIR/$TRILINOS_B_DIR/packages/ml/src/libml.so $DIR/lib/
cp $DIR/$TRILINOS_B_DIR/include/* $DIR/include/

echo "Current date: " `date`
echo "Creating env64.sh..."
cd $DIR/files/env64/
./gnu.sh $DIR $ADDITIONAL_TOOLS $ADDITIONAL_LIBRARIES $KRATOS_DIR

source $DIR/env64.sh

echo "Current date: " `date`
PC=8
echo ">={STEP "$PC"}={ Creating PYTHON utility..."
cd $DIR

if [ -d "$DIR/$PYTHON_DIR" ]
then
    echo "$DIR/$PYTHON_DIR exists..."
else
    echo "$DIR/$PYTHON_DIR does not exist..."
tar xf $DIR/files/python/$PYTHON_DIR.tar.bz2
fi

cd $DIR/$PYTHON_DIR
./configure --enable-shared --prefix=$DIR/ CC="gcc -m64" CXX="g++ -m64" LDFLAGS="-m64"
make clean
make
make install

echo "Current date: " `date`
P=9

echo ">=STEP "$PC"=< Creating BOOST library...

if [ -d "$DIR/$BOOST_DIR" ]
then
 echo "$DIR/$BOOST_DIR exists..."
else
 echo "$DIR/$BOOST_DIR does not exist..."
tar xf $DIR/files/boost/$BOOST_DIR.tar.gz
fi

echo "Creating user-config.jam..."

if [ -d "$DIR/" ]
then
 echo "$DIR/ exists..."
else
 echo "$DIR/ does not exist..."
tar xf $DIR/files/boost/$BOOST_DIR.tar.gz
fi

echo "Creating user-config.jam..."

if [ -d "$DIR/$BOOST_DIR" ]
then
 echo "$DIR/$BOOST_DIR exists..."
else
 echo "$DIR/$BOOST_DIR does not exist..."
tar xf $DIR/files/boost/$BOOST_DIR.tar.gz
fi

echo "Creating user-config.jam..."

if [ -d "$DIR/$BOOST_DIR" ]
then
 echo "$DIR/$BOOST_DIR exists..."
else
 echo "$DIR/$BOOST_DIR does not exist..."
tar xf $DIR/files/boost/$BOOST_DIR.tar.gz
fi

echo "Creating user-config.jam..."

if [ -d "$DIR/$BOOST_DIR" ]
then
 echo "$DIR/$BOOST_DIR exists..."
else
 echo "$DIR/$BOOST_DIR does not exist..."
tar xf $DIR/files/boost/$BOOST_DIR.tar.gz
fi

echo "Creating user-config.jam..."

if [ -d "$DIR/$BOOST_DIR" ]
then
 echo "$DIR/$BOOST_DIR exists..."
else
 echo "$DIR/$BOOST_DIR does not exist..."
tar xf $DIR/files/boost/$BOOST_DIR.tar.gz
fi

echo "Creating user-config.jam..."
11.9. Snapshots of Ahmed_25

- **Partitions from bottom**
- **Triangles from bottom**
- **Partitions from top**
- **Triangles from top**
- **Pressure from bottom**
- **Turbulence from bottom**
- **Pressure from top**
- **Turbulence from top**
Partition in wheels detail

Pressure in wheels detail
11.10. Snapshots of TELESPIPIO

One piece of TELESPIPIO

Another view of drafts around the telescope

Drafts around the telescope

The real telescope
11.11. Snapshots of Ferrari

Pressure from top

Triangles from right side detail

Pressure from right side

Triangles in nose detail
Acknowledgements

We would like to thankfully acknowledge the researchers at the International Center for Numerical Methods in Engineering (CIMNE) for sharing their expertise in the Kratos framework, and providing the input data sets for our experiments; the Barcelona Supercomputing Center (BSC) for the use of the Marenostrum Supercomputer and the technical assistance; the Trilinos Project Team at Sandia National Laboratories for their technical assistance; and the Code Optimization and Quality Technologies group of the IBM Haifa Laboratory, for the insightful discussions on migration processes.

This work has been supported by the European Commission through the HiPEAC-2 Network of Excellence (FP7/ICT-217068), the Spanish Ministry of Education (TIN2007-60625, and CSD2007-00050), and the Generalitat de Catalunya (2009-SGR-980).
References


[27] GNU. GCC, the GNU Compiler Collection. http://gcc.gnu.org


[29] CMake. CMake FAQ. http://www.cmake.org/Wiki/CMake_FAQ


[34] Karypis Lab. Family of Graph and Hypergraph Partitioning Software | Karypis Lab. http://glaros.dtc.umn.edu/gkhome/views/metis


[CHARACTERIZATION OF APPLICATIONS IN NEW ARCHITECTURES]

[45] die.net. Time(1) – Linux man page. http://linux.die.net/man/1/time

[46] die.net. mpirun(1) – Linux man page. http://linux.die.net/man/1/mpirun


