Scalable System for Large Unstructured Mesh Simulation

Miguel A. Pasenau[*,**], P.Dadvand[*,**], R. Rossi[*,**], Jordi Cotela[*,**],
Abel Coll[*,**] and E. Oñate[*,**]
Corresponding author: pooyan@cimne.upc.edu

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

Abstract: Dealing with large simulation is a growing challenge. Ideally for the well-parallelized software prepared for high performance, the problem solving capability depends on the available hardware resources. But in practice there are several technical details which reduce the scalability of the system and prevent the effective use of such a software for large problems. In this work we describe solutions implemented in order to obtain a scalable system to solve and visualize large scale problems. The present work is based on Kratos MutliPhysics [1] framework in combination with GiD [2] pre and post processor. The applied techniques are verified by CFD simulation and visualization of a wind tunnel problem with more than 100 millions of elements in our in-hose cluster in CIMNE.

Keywords: Parallelization, Computational Fluid Dynamics, Domain Decomposition, Large Simulations.

1 Introduction

The present work is based on Kratos Multi-Physics [1] in combination with GiD [2] pre and post processor. Kratos Multi-Physics is a free, open source framework for the development of multi-disciplinary solvers. Kratos is parallelized for Shared Memory Machines (SMMs) and also Distributed Machines (DMMs). GiD is a universal pre- and post-processor. GiD is targeted for both small and large simulations on both small and large platforms. The applied techniques are verified by CFD simulation and visualization of a wind tunnel problem with more than 100 millions of elements in our in-hose cluster in CIMNE using 96 processes and visualized over desktop workstations.

Dealing with large and complex simulations provides the motivation of the present work. There are three main difficulties in dealing with large models: creation of the model, solution of problem and the visualization and data mining in the results. In this work we describe the methodology and changes made in both frameworks for dealing with large simulations.

2 Preparation and Simulation

Usually meshing a large model, apart from mesh generation problems, has the inconvenience that it can not be performed in normal workstations due to their limited memory. The memory limitation

also appears in time of partitioning and distribution of the work. These problems are addressed by two solutions: more efficient partitioning and parallel element splitting.

2.1 More Efficient Partitioning

The first approach used in Kratos was to read the model by first process it, partition it and distribute the work over the rest of the processes. This method shows to be very memory consuming because it has to hold the mesh and a large portion of model data in the memory.

This approach is changed to in place division of the files. The process is simple:

- Only the connectivity of the mesh is read and stored in a minimal structure.
- The connectivity is passed to the partitioning process, in this case METIS, to obtain the partitions
- Communication plan is generated by colouring the domains graph.
- The input data file is divided into partitions line by line.

The process of division consists of reading a line of the input file, and writing to the corresponding files of the partitions this line belongs to. For example if the line is the pressure of node 12 and this node is in the interface of partition 3 and 6, then this line will be copied to the input file 3 and 6. This approach drastically reduced the memory used by the partitioning process form 1.5 GB per million of elements to approximately 200 MB.

2.2 Parallel Element Splitting

We have used an additional tool in order to obtain a finer mesh without running into the memory limit of the machine where the pre-process was performed. We implemented a parallel mesh refinement tool in Kratos, intended to perform adaptive mesh refinement. We used it in the test case presented here to split all elements by the midpoint of their edges, generating eight tetrahedra from each original one. The basic outline of the partitioning algorithm is as follows:

- Generate a nodal connectivity matrix, and store the arbitrary value of `-1` in position i, j (for $i \leq j$, as the matrix is symmetric) if an edge between nodes i and j exists in the model. This matrix is built locally by each process and later assembled.
- Identify the elements that will be refined by marking their edges in the nodal connectivity matrix with the value `-2`. This information is then transferred to other processes when necessary. In this case, all edges were marked, as we wanted to perform a global refinement.
- Count the number of nodes to be added by each process and in total, and assign a range of node indices to each process.
- Each process marks the nodes it will create by assigning an index to the matrix position of the edge where it will be located. In case of edges shared between processes, the last one to assign an index keeps the ownership of the node. This allows us to assign node ownership in a way that is deterministic between processes. New elements are always owned by the same process as the element they originate from.
- Once all new elements and nodes have an index assigned, they are created by the process that owns them.
- To keep indices correlative, all elements and nodes are renumbered.

- Interpolate nodal data to obtain values in new nodes for all variables of interest.

The advantage of this approach is that it allowed us to refine the mesh using the same set-up as for the solution process, giving us the possibility to use all memory available on the cluster instead of being restricted to a single node.

3 Post Processing

The opportunity to visualize in GiD a unstructured mesh with more than one hundred million elements represents an ideal test-bed for the implemented mechanisms regarding not only the joining of the partitioned mesh and results, but also the visualization of such amount of information.

On one side GiD incorporates well known techniques in the game industry which take profit of installed graphics hardware in order to draw faster the mesh and results such as OpenGL(R)'s Vertex and Element Buffer objects [3] [4] and textures [5].

On the other hand some mechanisms and enhancements has been implemented in GiD so that it can handle, in a reasonable amount of time and memory, hundreds of gigabytes of results and hundreds, even thousand, mesh and result partitions.

3.1 Results Cache

This mechanism uses a memory pool, whose size can be defined by the user, to *cache* the results used to visualize, instead of maintain all the results in main memory. When this option is enabled GiD only verifies the results files and gets some information: the file and result's position(s) inside the file(s), its memory footprint and other information about the result (result type, location, etc.) except the result's values themselves [6] [8]. When a result file is read, only those results belonging to the last analysis step are held in memory, but can later be unloaded.

The results are loaded *on demand*, i.e. when the user selects a result to be visualized, like contour fill, deformation or graphs, or animated, then the mechanism checks if there is enough memory to hold the demanded result. If the user imposed limit has been reached, then the oldest(s) result(s) are *unloaded* from memory to make room to accommodate the demanded result [6] [7]. Every time the result is used, its time-stamp is actualized.

3.2 Merging Many Partitions

As massive parallelism grows, one big issue the user faces is to put together the many partitions the problem was divided into and that the solution has. What was acceptable when the problems were relatively small and the number of partitions were low, now is unacceptable.

The merging of the different partitions has been improved by disabling, during the process, any unnecessary calculations, such as finding the skin of the volume mesh, boundary edges, smoothed normals, vertex and elements buffer objects, and enabling then when all the partitions and results are joined.

Tables 1 and 2 show the considerably loading time reduction:

Before	32 partitions	24' 10"
After	32 partitions	4' 34"
	128 partitions	10' 43"
	Single file	2' 16"

Table 1: Loading times of a CFD simulation of a telescope with 23,870,544 tetrahedra on a Intel(R) Core(TM)2 Quad Q9550 @ 2.83 GHz

Before	96 partitions	> 5 hours
After	96 partitions	51' 21"
	Single file	13' 25"

Table 2: Loading times of a CFD simulation of a racing car with 103,671,344 tetrahedra on a Intel(R) Xeon(R) E5410 @ 2.33 GHz

In the last case, the increased loading time between the single model file and the files of the partitioned models is due to merging of mesh and results, and the overhead of the graphical interface on the remote display.

3.3 Memory usage

With the enhancements of the result's cache and adjustments of internal structures, GiD takes between 15 and 18 GB of memory to handle and visualize the simulation of the racing car with a mesh of more than hundred million elements. This memory includes a 2 GB results cache and the handling includes: contour fill (Figure 1 and Figure 2), making cuts (Figure 3), stream lines (Figure 4), iso-surfaces and creating an animation of the velocity field.

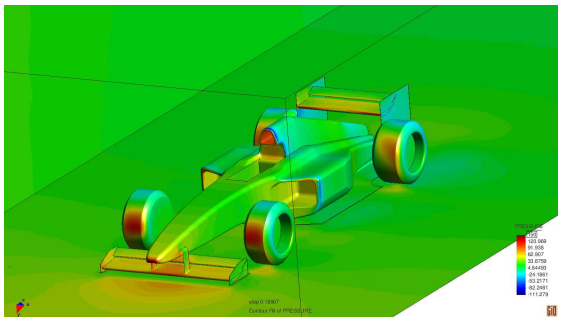

Figure 1: Pressure contour fill of the hundred million elements mesh

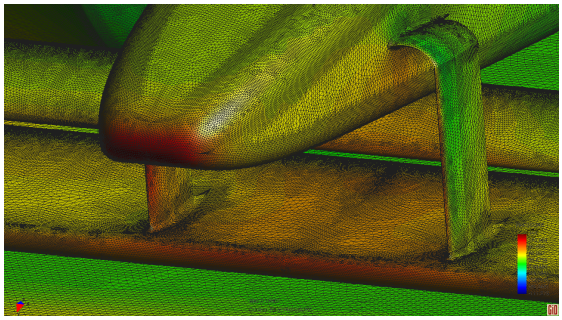

Figure 2: Detail of the front of the car showing the elements density

3.4 Off-screen mode

Another very interesting option which has also been implemented in GiD is the off-screen mode. In this mode, GiD does not need any graphical support nor opens any graphical window and allows the user to create pictures and animations of any visualization type in background or send this task as a job to a queue. All the graphical visualization is done by software, and so it takes longer than the interactive version, but it allows the user to automatize some processes and in any desired resolution too.

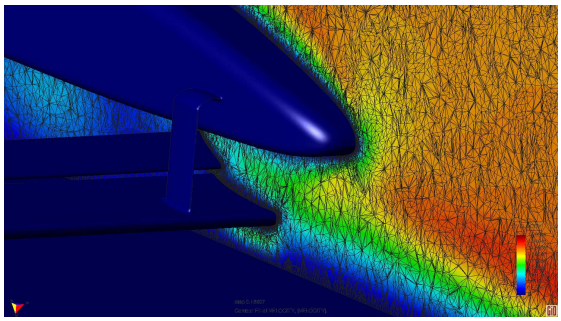

Figure 3: Cut of the volume mesh showing the velocity modulus

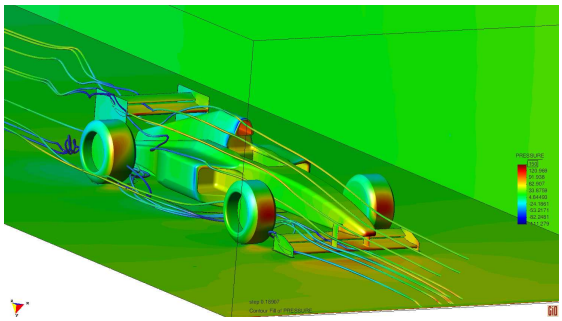

Figure 4: Stream lines coloured with the velocity modulus and contour fill of the pressure

4 Conclusions and future lines

The implemented improvements helped to achieve the hundred million elements milestone entirely with the resources of our Centre.

To speed the visualization process, one of the lines that the GiD team is working on is to provide several levels of detail by simplifying the mesh just for rendering purposes.

Another way to improve speed and responsiveness is to parallelize some algorithms such us the tree creation for stream lines, skin extraction and boundary edges and smooth normals calculation.

5 Acknowledgments

The authors wish to acknowledge the support of the Spanish *Ministerio de Ciencia e Innovación* through the E-DAMS project and the European Commission thorugh the Realtime project. The work of J. Cotela is funded by the Spanish *Ministerio de Educación* through a doctoral grant in the FPU program.

References

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

[2] Enrique Escolano, Abel Coll, Adrià Melendo, Ana Monròs and Miguel A. Pasenau. *GiD User Manual & GiD Reference Manual*, http://www.gidhome.com, CIMNE, 2010.

[3] NVIDIA. *White Paper: Using Vertex Buffer Objects (VBOs)*, http://developer.nvidia.com/object/using_VBOs.html, NVIDIA Corporation, October 2003.

[4] Kurt Akeley. *Buffer Objects*, http://developer.nvidia.com/docs/IO/8230/GDC2003_OGL_BufferObjects.ppt, NVIDIA Corporation, Game Developers Conference 2003.

[5] Miguel A. Pasenau de Riera. *Textures in Simulation*, 1st GiD Conference on Advances and Applications of GiD, http://www.gidhome.com/news/gid-conf/gid-conference-2002/presented-papers, February 2002.

[6] Maurice J. Bach. *The design of the Unix(R) operating system*, Prentice Hall International Editions, 1986.

[7] Theodore Johnson and Dennis Shasha. *2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm*, Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994.

[8] Xiaoning Ding, Song Jiang and Feng Chen. *A Buffer Cache Management Scheme Exploiting Both Temporal and Spatial Localities*, ACM Transactions on Storage, Vol. 3, No. 2, Article 5, June 2007.