ODAS 2022
22nd ONERA-DLR Aerospace Symposium
1st - 3rd June 2022, Hamburg, Germany

ParaDiGM: a library to handle *Para*llel *D*istributed *G*eneral *M*eshes

Eric Quémerais[a], Bastien Andrieu[a], Bruno Maugars[b], Julien Coulet[b], Stéphanie Lala[c], Julien Vanharen[a,*]

[a]*DMPE, ONERA, Université Paris Saclay, Châtillon, F-92322, France*
[b]*DAAA, ONERA, Université Paris Saclay, Châtillon, F-92322, France*
[c]*DTIS, ONERA, Université Paris Saclay, Palaiseau, F-91123, France*

Abstract

During the last decade, the Moore's law has proven to be outdated. The computational power does not double anymore every year for a given cost under the pursuit of the energy efficiency. To address this issue, the High Performance Computing (HPC) industry showed an incredible ingenuity by designing complex architectures. Today, the HPC tends to have more and more cores on each node, possibly with accelerators like General-Purpose Graphics Processing Unit (GPGPU) or Field-Programmable Gate Array (FPGA), and with less and less memory. As a corollary, the developers of scientific computing software need to pay attention and master numerous key concepts, which require great expertise, to hope to extract a reasonable performance. These key concepts include programming on heterogeneous architectures, multithreading programming, Message Passage Interface (MPI) programming, load balancing, asynchronous and non-blocking communications, cache blocking techniques, vectorization or even parallel Input/Output (I/O).

It is in this context that the ***ParaDiGM*** library was created to provide the developers a progressive framework, which consists of a set of low-, mid- and high-level services usable by many developers of scientific computing software that rely on a mesh, typically when solving partial differential equations on a given domain. The ***ParaDiGM*** kernel is based on the key concept of a distributed or partitioned view of data stored in an array. After introducing ***ParaDiGM*** and giving some general information in the first section, we shall attach great importance in the second section to describe in details the distributed and the partitioned approaches and their impact on parallel algorithms. For instance, we will show that the partitioned view is not always well suited for parallel algorithms and that the distributed view, even though less intuitive, is better suited for specific algorithms. New MPI communication protocols have been developed to switch from one view to the other. Then, we will present parallel dedicated algorithms for wall distance computation in the third section and isosurfaces computations in the fourth section. In each section, some applications will be shown with a special emphasis on the parallel computational efficiency on several thousand cores.

Keywords: High Performance Computing (HPC), Message Passing Interface (MPI), Massively Parallel, Distributed/Partitioned Approaches, Unstructured Meshes, Partial Differential Equation Simulation, Wall Distance, Isosurface.

1. Introduction

The development of scientific software to solve partial differential equations governing fluid mechanics, combustion, multiphase flow or even plasma physics require numerous skills in several different domains. For instance, implementing a parallel turbulent Navier-Stokes solver involves knowledge of HPC (MPI, cache blocking, vectorization, heterogeneous architectures), software architecture, numerical methods and even turbulence models, to mention just a few. Such knowledge cannot be held by a single person, not

*Corresponding author: julien.vanharen@onera.fr

even by a small team. In an attempt to solve this issue, ***ParaDiGM*** was developed to provide the developers a progressive framework, which consists of a set of low-, mid- and high-level services usable by many developers of scientific computing software. Such libraries already exists like *PUMI* [1] or *ZOLTAN* [2] but ***ParaDiGM*** tends to follow a different approach in the sense that it is truly a progressive library of services which can be called from C/C++, Fortran and Python without imposing complicated data structures. ***ParaDiGM*** is written in C99 and includes over 200,000 lines of code. Amongst others, ***ParaDiGM*** offers numerous parallel algorithms like octree, points clouds localization, simple mesh generation, wall distance computation, isosurface generation, mesh partitioning, mesh renumbering, or even geometric intersections. For obvious reasons, all of these algorithms cannot be detailed here. Hence, it was decided to highlight the wall distance and isosurface computations algorithms. Indeed, the wall distance computation can have a significant impact on the computational cost of a CFD computation, and in particular when dealing with moving meshes.

2. Distributed or partitioned?

ParaDiGM implements two different approaches to represent an array. An array is either distributed or partitioned. Typically, an array is stored in a distributed view after reading a mesh. Indeed, during a parallel reading, each MPI rank reads by blocks the mesh file as shown in Fig. 1a. Besides the array, it is sufficient to have a distribution to completely define the data. Notice at this stage that the arrays cannot be easily addressed because of the global numbering. However, each array element is unique and attached to a single MPI rank. To detail the example in Fig. 1a depicting what happens with two MPI ranks, one will obtain two arrays for the distributed view:

1. The array of triangles storing the global numbering of each triangle:
   - On MPI rank 1 in red, the array of triangles reads: $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]$.
   - On MPI rank 2 in blue, the array of triangles reads: $[14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]$.
2. The array of the triangles distribution, whose size is the number of MPI ranks plus one, shared by all the MPI ranks reads: $[0, 13, 26]$. When used in this manner, each MPI rank knows that the triangles 1 to 13 are on the first MPI rank and the triangles 14 to 26 are on the second MPI rank.

To represent the connectivity of each triangle, a connectivity array can obviously be added, which consists of an array, whose size is the number of triangles on each rank multiplied by three to store the three vertices with a global numbering. However, to keep it simple, only the array of triangles denoted by its global numbering is shown in this example.

Typically, an array is stored in a partitioned view after using a graph partitioning library such as ***ParMETIS*** [3] or ***PT-Scotch*** [4]. Besides the array, it is sufficient to have the link between the local and the global numbering. Notice at this stage that the arrays can be addressed since they store the local numbering. In addition to that, the same element can be present on several MPI ranks to deal with ghost cells for instance. To detail the example in Fig. 1b, one will also obtain two arrays for the partitioned view:

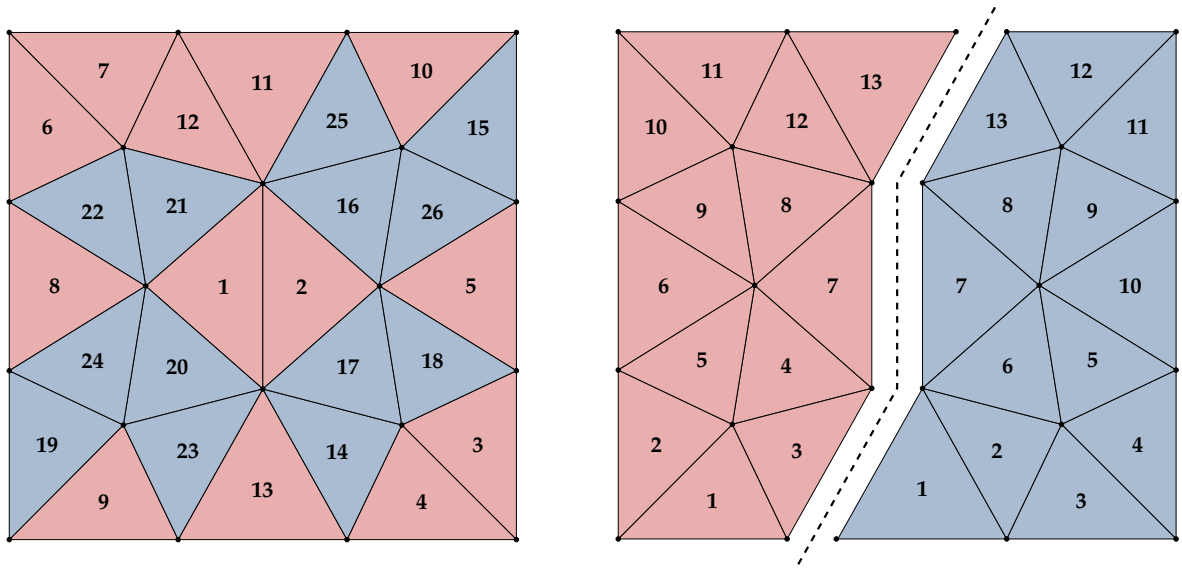
1. The array of triangles storing the local numbering of each triangle:
   - On MPI rank 1 in red, the array of triangles reads: $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]$.
   - On MPI rank 2 in blue, the array of triangles reads: $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]$.
2. The array of the link between the local and the global numbering called *ln_to_gn* in the ***ParaDiGM***'s jargon:
   - On MPI rank 1 in red, the *ln_to_gn* array reads: $[9, 19, 23, 20, 24, 8, 1, 21, 22, 6, 7, 12, 11]$.
   - On MPI rank 2 in blue, the *ln_to_gn* array reads: $[13, 14, 4, 3, 18, 17, 2, 16, 26, 5, 15, 10, 25]$.

Once these two views are defined, the main ***ParaDiGM***'s interest lies in the definition of simple functions to switch from one to another. For both cases, the implementation relies on the declaration and allocation of C structures and the exchange is performed by calling a simple function.

Performing the conversion from the partitioned view to the distributed one is carried out by the structure *PDM_part_to_block* as its name implies. As a first step, this structure is created by mainly giving the global numbering (*a.k.a* the *ln_to_gn* array) and the associated weight (elementary load) of each element, which allows the algorithm to deduce a load balanced distribution according to these weights thanks to the Newton's method. As a second step, the exchange is realized by informing the partitioned array with constant strides (for instance the partitioned triangle connectivity array) or variable strides (for instance a boundary condition value, some triangle having no boundary condition) and returns the distributed array with the associated strides.

Performing the conversion from the distributed view to the partitioned one is carried out by the structure *PDM_block_to_part* as its name implies. As a first step, this structure is created by mainly giving the global numbering of elements which the user requires on the partition after the exchange. As a second step, the exchange is realized by informing the distributed array with constant or variable strides and returns the partitioned array with the associated strides. These two functionalities are the keystones of the next presented algorithms.

All the exchanges described in the last two paragraphs are performed with non-blocking MPI collective communications.



(a) Distributed view.

(b) Partitioned view.

Figure 1: In ***ParaDiGM***, either an array is distributed or is partitioned.

3. Wall distance computation

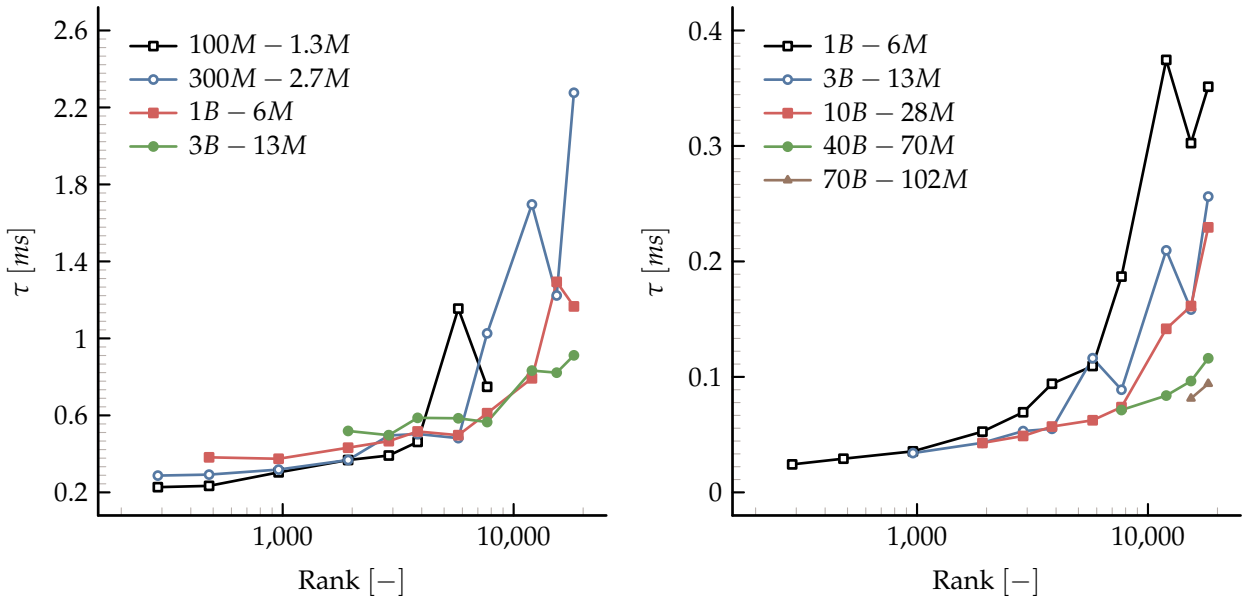
Computing the wall distance is of utmost importance for any turbulent flow computations in the context of Reynolds-Averaged Navier-Stokes (RANS) simulations. Indeed, a lot of turbulence models need the latter such as the Spalart-Allmaras [5] or the Menter $k-\omega$ SST [6] turbulence models for instance. Starting from a partitioned volume mesh, a naive implementation would be to copy the surface wall mesh on all processors and compute the wall distance between each cell center (resp. vertices) and the surface for a cell-center (resp. cell-vertex) solver. This naive approach is absolutely wrong for two main reasons:

1. Each volume partition rarely needs the whole surface mesh to find the wall distance. Hence no parallelism is exposed for the surface mesh.

2. Since ***ParaDiGM*** targets meshes with possibly several billion cells, as shown in Fig. 2, the associated surface mesh can contain several million faces and therefore cannot be stored on each processor due to the lack of memory.

It is against this background that a truly massively parallel algorithm was developed in ***ParaDiGM*** to estimate the wall distance. In fact, two algorithms were developed. The first one takes as inputs a cloud of points and a surface mesh. It then computes the distance between each point of the cloud and the surface mesh. It is important to highlight that each input has its own partitioning. The second one takes as inputs the output of the previous one, where the cloud of points is defined by the boundary cell centers (resp. the vertices) of each partition for a cell-center (resp. cell-vertex) solver, the cell-centers (resp. the vertices) and the surface mesh. It then computes the distance between each inner point and the surface mesh by applying a propagation algorithm. The first algorithm entails five main stages:

1. Find an upper bound wall distance, which is the distance between a point of the cloud and the nearest vertex of the surface mesh. It is implemented by stowing the surface mesh vertices into an octree and then by computing the desired distance.
2. Compute a bounding box structure around the previous found nearest vertex to obtain a list of candidate elements which contains the point of the wall distance.
3. Balance the load. Indeed, the problem is now divided into elementary tasks. Each task is composed of a point of the cloud and the list of candidate elements. The load balancing is greatly simplified by the MPI communication protocols defined in section 2.
4. Compute the projection to obtain the wall distance and find the point on the surface mesh. This stage can be entirely computed in serial: no communication is needed since all the previous stages acted like a preconditioning to expose parallelism as much as possible.
5. Finally send back to the initial partitions the wall distance and the point of the surface mesh which gives the minimum distance for each point of the cloud. Again, this stage is greatly simplified by the MPI communication protocols defined in section 2.



(a) Wall distance computation for a points cloud.

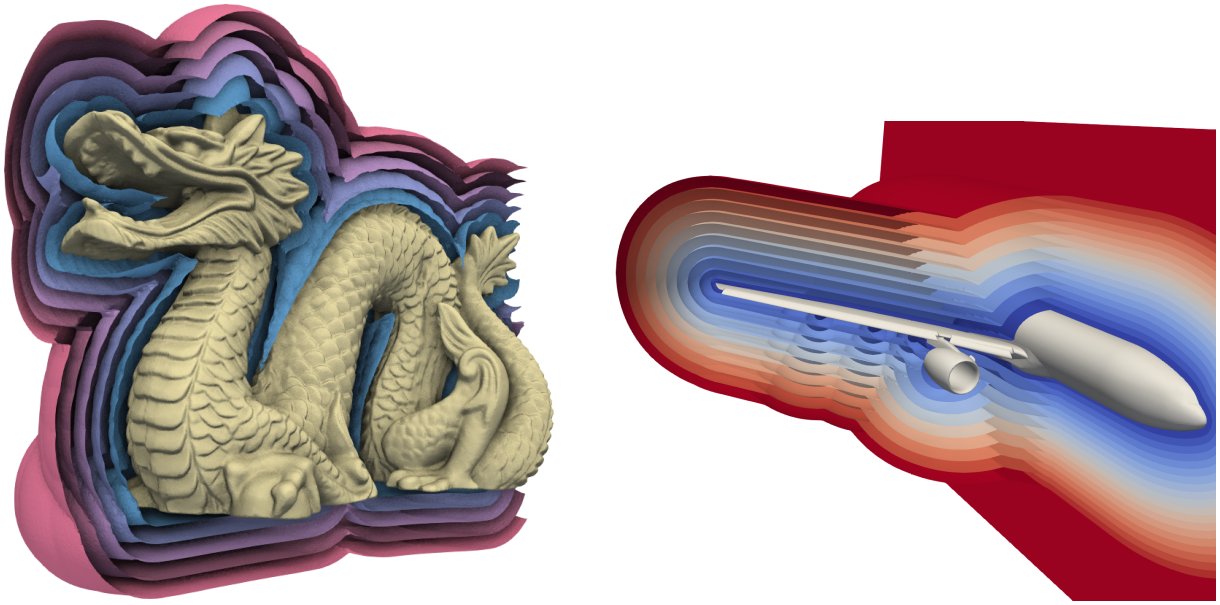
(b) Wall distance computation by propagation.

Figure 2: CPU time per cell τ to compute the wall distance versus the number of MPI ranks. The first number in the legends is the number of mesh cells whereas the second one is the number of faces from which the wall distance must be computed.

To evaluate the first algorithm's scalability, tests were carried out with cartesian meshes of a cube of successively $[100 \cdot 10^6, 300 \cdot 10^6, 1 \cdot 10^9, 3 \cdot 10^9]$ cells to find the distance with respect to the $[1.3 \cdot 10^6, 2.7 \cdot 10^6, 6 \cdot 10^6, 13 \cdot 10^6]$ faces which compose the six faces of the cube. For the second algorithm based on propagation, the meshes are composed of $[1 \cdot 10^9, 3 \cdot 10^9, 10 \cdot 10^9, 40 \cdot 10^9, 70 \cdot 10^9]$ cells and $[6 \cdot 10^6, 13 \cdot 10^6, 28 \cdot 10^6, 70 \cdot 10^6, 102 \cdot 10^6]$ faces. This configuration is quite challenging for wall distance computation. Indeed, points close to the many planes of symmetry planes may be almost equidistant from a very large amount of surface elements and thus require costly geometric computations. Both algorithms show very good results and make possible the wall distance computations with meshes of **several tens of billions** cells.

4. Isosurface computation

ParaDiGM also offers general algorithms for *post mortem* visualization and/or *in situ* processing since the library can be called from any solvers. In this section, an emphasis on isosurface computation is realized. Notice that ***ParaDiGM*** has the ability to compute either a geometric isosurface by specifying the equation of a plane, a sphere or any other surfaces which can be described by an equation or a solution isosurface. The implemented algorithm is based on dual contouring [7] and is limited to polyhedral cells which are convex and connected. As inputs, the algorithm needs a distributed or partitioned mesh with a face based connectivity and a solution and (optionally) its gradients at vertices. As outputs, the user obtains a coherent partitioned mesh of the isosurface.



(a) Several isosurface computation on the dragon from the Stanford University Computer Graphics Laboratory.

(b) Several isosurfaces of distance from the surface of an airplane. The geometry comes from the 4th AIAA CFD High Lift Predictions Workshop.

Figure 3: Examples of isosurface computation.

Furthermore, the user is provided with the location within the volume mesh of each element of the isosurface mesh. Any field defined on the volume mesh can then be easily interpolated on the isosurface mesh. The algorithm involves four main steps:

1. Loop over the edges of the volume mesh to select the cells which are intersected by the isosurface. Indeed, since the isosurface is defined by a function equal to zero, if the product of the function value at both vertices of an edge is negative, then the isosurface intersects this edge and possibly all the cells sharing the latter.

2. Balance the load to distribute all the edges and the neighboring cells. This partitioning is easily carried out thanks to a Hilbert renumbering [8, 9].
3. Apply the dual contouring algorithm to generate the isosurface mesh.
4. Build a coherent mesh by creating an absolute numbering. For each face of the isosurface, also return the parent cell numbering to be able to interpolate from the volume mesh to the isosurface mesh with a callback function if needed.

5. Conclusion and perspectives

ParaDiGM was developed with the aim of providing the developers a progressive framework and a set of core utilities to help them to develop complex parallel scientific applications. In this paper, it is clearly demonstrated the importance to be able to manipulate efficiently distributed and partitioned arrays. A particular focus is made on wall distance computations and isosurfaces extractions. ***ParaDiGM*** is under active development and it is planned to introduce many other functionalities such as high-order meshes, $k-d$ trees, ray tracing and even anisotropic mesh adaptation. Finally, particular attention will be paid to improve performance on heterogeneous machines (CPU/GPGPU) in a near future.

References

[1] D. A. Ibanez, E. S. Seol, C. W. Smith, M. S. Shephard, PUMI: Parallel unstructured mesh infrastructure, ACM Trans. Math. Softw. 42 (3) (5 2016). doi:10.1145/2814935.

[2] E. G. Boman, U. V. Catalyürek, C. Chevalier, K. D. Devine, The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: partitioning, ordering, and coloring, Sci. Program. 20 (2012) 129–150. doi:10.3233/SPR-2012-0342.

[3] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comp. 20 (1) (1998) 359–392. doi:10.1.1.106.4101.

[4] C. Chevalier, F. Pellegrini, PT-Scotch: A tool for efficient parallel graph ordering, Parallel Comput. 34 (6) (2008) 318–331. doi:10.1016/j.parco.2007.12.001.

[5] P. R. Spalart, S. R. Allmaras, A one-equation turbulence model for aerodynamic flows, Rech. Aerosp. (1994).

[6] F. R. Menter, Two-equation eddy-viscosity turbulence models for engineering applications, AIAA J. 32 (8) (1994) 1598–1605. doi:10.2514/3.12149.

[7] T. Ju, F. Losasso, S. Schaefer, J. Warren, Dual contouring of hermite data, in: SIGGRAPH '02: Proceedings of the 29$^{\text{th}}$ Annual Conference on Computer Graphics and Interactive Techniques, New York, NY, USA, 2002, pp. 339–346. doi:10.1145/566570.566586.

[8] D. Hilbert, Ueber die stetige abbildung einer linie auf ein flächenstück, Math. Ann. 38 (1891) 459–460. doi:10.1007/BF01199431.

[9] G. Peano, Sur une courbe, qui remplit toute une aire plane, Math. Ann. 36 (1890) 157–160. doi:10.1007/BF01199438.