Massively Parallel location and exchange tools for unstructured mesh-based CFD

Yvan Fournier

MASSIVELY PARALLEL LOCATION AND EXCHANGE TOOLS FOR UNSTRUCTURED MESH-BASED CFD

Yvan Fournier*

*EDF, R&D/MFEE, Chatou, France

ABSTRACT

Computational fluid dynamics (CFD) computations often require coupling, interpolation, or projection of some values on one computational domain or portion thereof to another. This is the case both for multi-physics couplings, and for the implementation of various features, such as restarting on a different mesh, boundary condition couplings, and possibly others.

In all cases, this requires a mapping, or location stage, where points or other element types are located relative to other points or elements, without a priori knowledge of any neighborhood or connectivity information. As simulations often require highly refined 3D meshes, with computations run on parallel distributed memory systems, using programming paradigms such as MPI, it is often more practical to build mappings in parallel, on the same computational resources as the computation itself. Providing these features as a library callable directly by the computational code further allows avoid costly extra file read/write operations.

In this article, we describe different algorithms used in the context of the code_saturne CFD tool, and the associated Parallel Location and Exchange (PLE) library, based on their design goals, performance, and API ease of use and stability. We also provide some elements on actual observed performance, and paths for improvements and extensions for increasingly massive parallelism.

KEYWORDS

Unstructured mesh
Parallel code coupling
High performance computing
code_saturne
MPI

1 INTRODUCTION

As in other simulation domains, computational fluid dynamics (CFD) computations often require coupling, interpolation, or projection of some values on one computational domain or portion thereof to another. This is the case both for multi-physics couplings, and for the implementation of various features, such as restarting on a different mesh, internal couplings, and possibly others, depending on the tool features.

In all cases, this requires a mapping, or location stage, where points or other element types are located relative to other points or elements, without a priori knowledge of any neighborhood or connectivity information.

In the case of CFD, simulation fidelity often requires highly refined 3D meshes, and computations are mostly run on parallel distributed memory systems, using programming paradigms such as MPI [1]. These systems allow for a very large aggregate memory, but most often limited local memory. Though some tools may compute mappings in a serial pre-processing stage on special compute nodes with large memory, it is more practical from a tool chain, portability, and deployment standpoint to handle those directly in parallel, on the same computational resources as the computation itself. Providing these features as a library callable directly by the computational code further avoids costly extra file read/write operations.

For the code_saturne software [2], [3] (https://code-saturne.org), continuously developed by EDF since 1997 and also released under the GPL free software license since 2007, we have developed a sub-library to simplify couplings with other codes, named PLE (Parallel Location and Exchange, https://github.com/code-saturne/code_saturne/tree/master/libple), specifically designed to simplify parallel *n* to *m* couplings, using a minimalist API and dependencies. This library allows location of points on meshes, and thus allows features complementary to the more recent MEDCoupling [4] library (mainly mesh to mesh intersection and overlay), MUI [5] (point cloud to point cloud location and interpolation), or more complex DTK [6] (DataTransferKit) and First preCICE [7].

2 HISTORY AND DESIGN GOALS

The PLE library is based on its predecessor FVM (Finite Volume Mesh), designed to provide various “generic” utility features, including coupling, IO, and post-processing export, to code_saturne, in a manner also usable by similar codes. FVM was released under the LGPL V2 license to allow unrestricted use by external tools, and is written in C for maximum portability. To reduce dependencies and especially avoid additional data copies, most parts of FVM were folded back into the main code_saturne source tree in 2010, while a streamlined and stable API was extracted as PLE 1.0, removing its dependency to a given mesh data model. In 2015, PLE 2.0 was released, with a few API changes further streamlining its use. The algorithm versioning scheme used and the lightweight API have allowed maintaining this API stable. In other organizations, the initial FVM library has been slightly modified by for use in the CWIPI library [8], [9] (https://w3.onera.fr/cwipi/bibliotheque-couplage-cwipi), which also adds interpolation operators, and various optimizations.

The initial design goals of a finite volume mesh utility library (named FVM) were to

- allow location of points from a given mesh onto another unstructured mesh, based on multiple cell types including generic (at least star-shaped) polyhedra;
- handle complex geometries, implying full automation, requiring minimal input parameters for ease of use, and robust location algorithms;
- separating location from interpolation, as co-located cell-centered finite volume schemes may use specific interpolation options rather than classical linear node-based interpolation;
    - the interpolation details are left to the calling code for better genericity;
- handle both 3D volume and surface point to mesh location;
- handle both “external coupling” between different codes and “internal coupling” between mesh sections of a single code.

This was needed among other purposes for coupling of code_saturne with itself using Chimera-like techniques to combine Reynolds Averages Navier Stokes (RANS) and Large Eddy Simulation (LES) turbulence models in each (partially overlapping) domain, proposed in the context of the DESider European project [10], [11] (2004-2007).

- contrary to true Chimera applications and techniques, a given point needs only to be located relative to a single mesh element, and not several overlapping elements in certain areas;
- must be usable in distributed parallel mode (for example with *n* processors using turbulence model A, *p* processors using turbulence model B);
    - since applications include LES, requiring large meshes, avoid approaches with local memory requirements that would not fit on a single node’s memory on a typical cluster or massively parallel computer : all stages of the coupling mechanism must be distributed

At the time, no freely available and distributable (LGPL license or less restrictive) library handled these requirements:

- MPCCI [12] is not freely available, and does not seem to handle polyhedra;

- DIRTlib [13] (Donor Interpolator Resource Transaction library) or CHIMPS [14], did do not seem freely available either; while the former seemed interesting, we also wanted to avoid the Python dependency of the latter, which could be a liability on machines with limited local memory such as the IBM BlueGene series.
- No available library handled polyhedral elements
  - though the FVM library would have allowed subdividing polyhedra into simpler elements to pass them to another library, this is best avoided if possible, as the memory requirements may become large.

3.1 REFACTORING OF THE PLE SUBSET

Coupling with the first parallel version of Syrthes, EDF's thermal conduction and radiation code used for conjugate heat transfer, which uses a tetrahedral mesh with non-interleaved connectivity arrays led us to 3 options:

- use a full copy of the code's mesh structure to allow its use with the existing FVM library;
- modify the FVM library so as to allow different interleaving strategies, which would have made it significantly more complex (especially a it is written in C, which does not enable programming techniques such as those used in VTK's recent "zero-copy" arrays)
- refactor the code so as to avoid copies

We chose the last option, which as lead to PLE's main distinguishing feature compared to libraries such as MEDCoupling or MPCCI; and allowed keeping the library lightweight by separating the data structure from the parallelization strategy. This leads to some limitations on the parallel algorithm, but as we will show later, still allows for good performance while keeping the library lightweight.

This library is now also used with non EDF codes such as ALYA, for example for FSI applications [15].

3 OUTLINE OF PARALLEL POINT LOCATION ALGORITHM

For many applications, we need to couple computations on different domains, using distinct programs or data sets, with different partitionings. In cases of partial overlap, different variable types may be exchanged both ways on some part of the computational domain, or one way only (depending on their nature); to account for this, in a distributed and partitioned domain paradigm, each process considers 2 sets:

- a set of local points at which "distant" variables will be received (referred to as "coupled" points); with co-located finite volumes, these points usually correspond to the cell or face centers; with finite elements, they would more often correspond to nodes or Gauss points;
- a set of local elements, in which target points may be located; and local variables will then be interpolated at these points, then sent to their owning processes).

We choose to make the API and algorithm symmetric (i.e. a locator defines and uses both "receiver" point and "donor" element sets simultaneously), so that when coupling 2 instances or more of a same code (using the same variables), the user need not specify which instance sends or receives first or use duplicate calls. This also avoids a risk of user or programmer error leading to deadlock in more complex cases.

Let us illustrate what happens in a simple case with partial mesh overlap, coupling a computation over a program and associated mesh A, shown in gray, and a program and associated mesh B, shown in green. Partial transparency is used, so that we may note the region of overlap;

Figure 1: example domain overlap

Now account for domain splitting: shades of gray are used for mesh A, colors for mesh B; different shades or colors indicate different domains (and MPI ranks); overlap is still visible;

Figure 2: example domain overlap with partitioning

Let us now zoom in on overlapping region.

On the left, part of mesh A is shown on top, mesh B on the bottom, (using the same x axis);On the right, the corresponding overlap is seen in more detail;

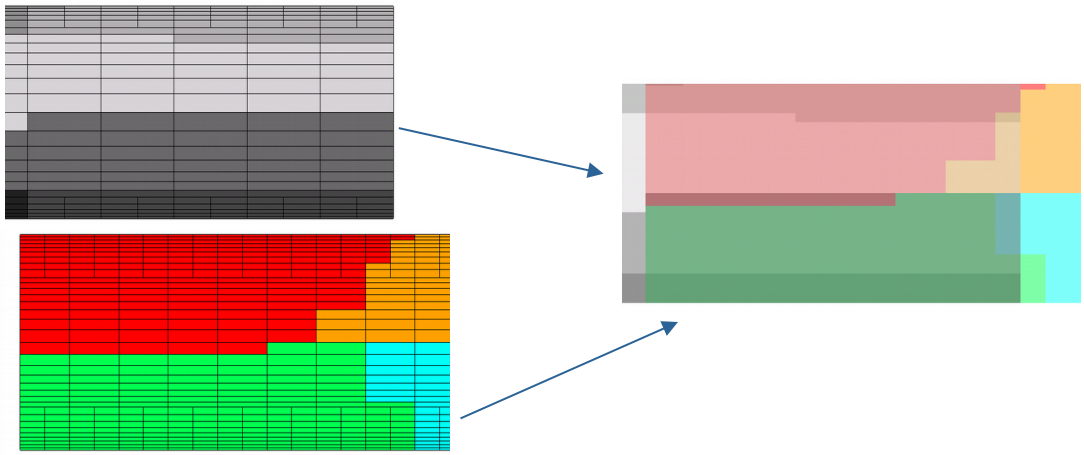

Figure 3: example overlap detail

Let us look at this example, as seen from one rank owning mesh A (the algorithm is symmetric, so this is “sufficient”).

- Select centers of coupled cells, and compute their bounding box (local step, executed by each rank independently);
- In a similar fashion, compute the bounding box of selected cells;

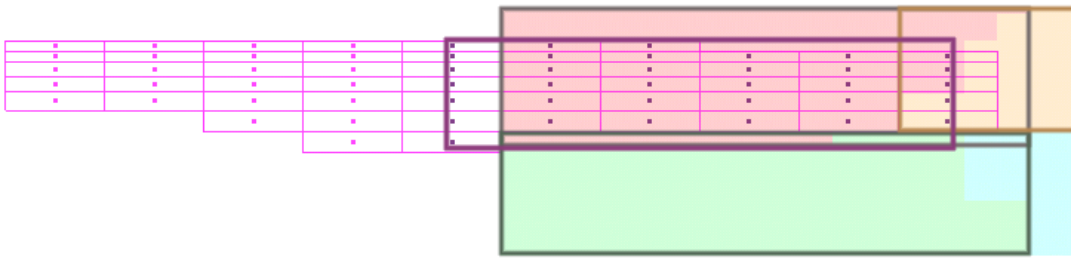

Figure 4: bounding boxes

- Exchange bounding boxes between all ranks (collective operation);
  The corresponding memory increases with processor count, but remains small (2 boxes x 6 coordinates x total ranks, so 12 floating numbers times communicator size)
  - Only those ranks whose selected point set and element set intersect will need to exchange data;
- Send coordinates of local points within distant element bounding boxes to the corresponding ranks; in a symmetric fashion, receive coordinates from other ranks:
  - For each corresponding rank, the points received are located regarding to the the (local) mesh, and a “best fitting element id” and corresponding distance (0 to 1 if truly within best fitting element, > 1 otherwise) is returned.
  - Some points will probably fall within the local interpolation sub-domain's bounding box, but not inside that sub-domain; if such a point is within the specified tolerance, we do not yet know if it lies fully within (or closer to) another processor’s sub-domain (hence the following step of this algorithm).

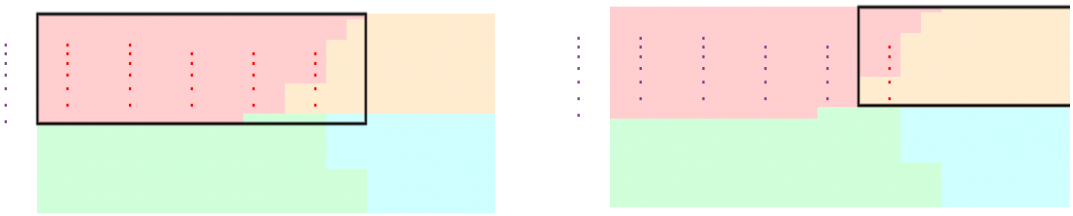

Figure 5: tested regions

Left and right: in red, points from distant rank whose bounding box is indicated by the black rectangle, for two distant ranks, tested for location on the light red subdomain

- Each point is assigned to the process which returned the smallest distance criteria so far for this point:
- Final updated information is exchanged;
  - Each rank/subdomain maintains coordinates of distant points it was assigned, as well as the corresponding local element id; distance is not needed anymore now that “best fit” has been determined, so it can be discarded.
  - The index of initial (local) points assigned to each processor is maintained locally on each rank for future re-assembly of incoming interpolated variables.

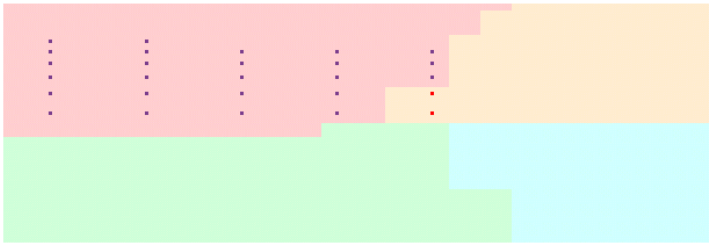

Figure 6: Final set of points from a given rank (different color indicates different donor mesh subdomain)

3.2 COMMUNICATION ORDERING

To ensure communication deadlock is avoided (where for example rank *i* waits for rank *j* while *j* waits for *i*, most often in an indirect manner), a simple solution is to execute the the `MPI_Sendreceive` series in increasing rank order.

This can lead to some serialization of operations, and is thus not optimal, but is simple ans robust. For example if based on the bounding box intersections rank 0 of 10 needed to communicate with ranks 1 and 9, rank 1 with 0 and 2, … and rank 9 with ranks 8 and 0, than rank 2 would need to wait for rank 1 to finish exchanging with rank 0 before any other

exchange, and rank 9 wait until rank 8 had finished communicating with previous ranks, and wait for intermediate steps to finish. In a 1D case, with ranks mapped to increasing x coordinates in a linear manner, this could actually lead to a full serialization of exchanges.

To improve this situation, ordering by recursive subdivision of communicating ranks can be done: in the previous example with 10 ranks, all communications between ranks 0-4 and between ranks 5-9 can be done first, and this logic applied in a recursive manner (ranks 0-2 and ranks 3-4, …). When communicating between ranks from sets 0-4 and 5-9, the recursive ordering can also be applied to subsets.

4 THE PARALLEL LOCATION AND EXCHANGE LIBRARY

The PLE library is designed as a lightweight and minimally intrusive helper library for parallel *n* to *p* code coupling. It is written in standard C (C 99), and is parallelized using the MPI paradigm.

As “simple” or “non-intrusive” may mean different, usually incompatible things to different people in the context of parallel coupling, it may be useful to explain our priorities and target software:

We assume the following, which is usually true for our target codes:

1. source code is available, and is actively maintained, or is at least comprehensible
2. codes using distributed parallelism are parallelized using MPI
3. we can use the same MPI library for our codes
4. minor modification of a parallel code’s source, and linking with a lightweight library is less costly (and frustrating) than writing complex workflow and data mapping descriptions just to avoid touching the code

In our R&D environment, where codes are developed, and funding is related to various industrial studies including flows in nuclear reactors, (1) can be naturally expected, and the long lifecycle of most tools leads to (2), as the alternatives to MPI are either not mature or less well established. From (1) and (2), (3) is almost automatic as long a we do not require MPI library extensions available only in one library in the target codes.

The main assumption with which not everyone may agree here is (4). In our own experience, which both includes developing PLE and plugging various third-party libraries or implementing couplings using other paradigms imposed by other (usually less HPC-oriented) codes, we strongly defend this, as total solution cost is not limited to source code development, but to understanding documentation adapting scripts, build systems, debugging, and support.

Those assumptions could actually be relaxed in case of need, but doing so has not been a priority:

- Though PLE mainly targets codes running in a shared MPI environment, the main operators could be adapted in a few days work to a socket-based communication, at least for the basic algorithms, but this has never been required so far.
- not having access to source code at all would require additional wrappers and overhead; so the difficulty would depend mostly on the target codes interoperability features
- using a layered approach, PLE could be used as an “engine” inside a framework or library having a different approach or constraints. Though it is currently based on an older fork of FVM, the CWIPI library is a good example of layering additional tools and features over “PLE-like” code to serve the needs of a wider community.

3.3 PLE COUPLING HELPER FUNCTIONS

When running coupled computations with multiple codes, we mostly use the following approach:

- assign a different working directory to each code

- start each code in its own working directory using `mpiexec` (or equivalent) to start the programs together in MPMD (Multiple Program Multiple Data) mode
  - codes than share an `MPI_COMM_WORLD` communicator
- use `MPI_Comm_split` to assign each code its “main” communicator

In “multiple codes”, we include coupling the same code with a different computational domain and mesh and possibly different model options, exchanging boundary conditions or source terms to be equivalent to running a different code, for all practical purposes. This is actually quite often used, a simple example being a heat exchanger where 2 disjoint fluid domains can be coupled with a conjugate heat transfer code, and not directly to each other.

The simplest features of PLE are a few helper functions designed to make things simpler when setting up this type of communication with MPI codes. The **ple_coupling** API includes:

- a standalone function allowing processes in an MPI communicator to convert a character string (a name) to an integer. This allows running each of the coupled codes (or computational domains) and their own subdirectory, and converting the directory name to an integer so as to use it in a robust manner with `MPI_Comm_split`, simply by first calling `ple_coupling_name_to_id(<mpiexec_comm>, <working_directory_path>);`
- a utility function building an MPI intra-communicator from a local and distant communicator within a base communicator; for example, when 3 codes A, B, and C are launched in a single “mpiexec” command and each one builds its own communicator *comm_a*, *comm_b*, *comm_c* from `MPI_COMM_WORLD` using `MPI_Comm_split`, the `ple_coupling_mpi_intracomm_create` function can be used to build intra-communicators *comm_ab* and *comm_ac* for example (depending on which domains need to communicate)
- functions managing a `ple_coupling_mpi_set` object, which allows coupled codes to discover each other easily, and collectively synchronize and exchange status and time step information at chosen call points. Even for coupling of multiple (more than 2) code instances, this mechanism removes the need for a specific supervisor/controller process by handling this supervision role directly inside the codes (usually requiring only a few 10s of extra lines of code);

Note that these functions are not incompatible with MPI 2 dynamic process creation and management functions, as the top communicator provided to the API could very well be `MPI_COMM_UNIVERSE`. So in an environment using theses functions , although `MPI_Comm_split` would not be of use, the other features could still be useful.

Comparison with other tools

At least the MUI library uses a similar MPI_Comm_split, approach to program initialization.

The MEDCoupling library builds and used “Data Exchange Channel” objects, which do not require intra-communicators but are built using low-level MPI group management functions. At least within EDF, this low-level initialization approach seems to have slowed adoption of the MPI parallel coupling features of that library, although it provides many useful features. In code_saturne, we have used PLE utility functions to make this step easier, even when using MEDCoupling instead of PLE for its different set of mapping and interpolation features.

Remarks

We do require that codes not directly use `MPI_COMM_WORLD`, but any code following the MPI standard recommendations should already do that, and otherwise, `sed` or any other search and replace tool can help. An alternative approach, used by the Commsplitter library [16] from the Trilinos Trios package is to use MPI’s PMPI interposition layer to use a split communicator without modifying the source code, but this seems a rather complex solution (and the Trios package has been deprecated and removed from Trilinos recently).

In practice, we do not use MPI-2 process creation and management functions with code_saturne, because the actual availablilty of these function is usually dependent on the MPI runtime and network type, and in most HPC environments we have used, these functions have not been available. An additional difficulty is that when launching

MPI codes separately in a batch environment, whereas programs launched together are usually distributed automatically through the cooperation between the resource manager and the MPI library, when multiple `mpiexec` commands are used, the usual solution is to query the resource manager or check its environment variables to determine which resources are available, then build "hostfiles" to pass to each launch command.

Since resource managers are not standardized, and even a given resource manager's environment may change over time, we prefer to leave the maintenance of MPI/resource manager interaction to the developers of those libraries. We choose this based on experience: for several years, in the code_saturne un scripts, we parsed the environment variables for various resource managers, so as to be able to determine how many processes were available to assign rank ranges automatically. Now, when we need this information, we simply run a helper MPI program returning the number of ranks, which is less elegant but easier to maintain.

To run MPMD programs, all the current systems we know of can start programs in MPMD mode using the standard "`mpiexec -n <n1> -wd <d> prog1 : -n <N2> -wd <d2> prog2`" syntax or MPMD configuration files such as those used by SLURM's "`srun –multi-prog`", MPICH's "`mpiexec -configfile`", or Open MPI's "`mpirun -app`" options.

In the few cases we have encountered where this was not available, running a wrapper script under MPI usually allows determining a program's rank before starting a program by checking environment variables (the usual way libraries pass the rank information used by `MPI_Init*`), and even on the IBM Blue Gene/L, where a program had to be run directly and where MPMD was not supported in theory, we could emulate this using the `execve` system call in a compiled launcher. We have not needed to resort to such tricks since circa. 2013.

3.4 PLE LOCATOR FUNCTIONS

The main functionality of the PLE library is its mesh mapping and location functionality. To adapt to different mesh data structures and remain lightweight, the data model is kept to a bare minimum, consisting of:

- a cloud of "target", or receiver 3D points where data will be sent
  - points are defined by an array of interleaved (double precision) coordinates, and for cases where parts of and optional integer "exclusion tag"
- a "source", or donor opaque mesh object, whose structure is unknown to PLE.

The base locator object is defined relative to a given MPI intra-communicator, a number of ranks associated with the distant (coupled) domain, and a start rank for the current domain in the communicator. This allows handling tow cases in a nearly identical manner:

- When locating points on a mesh inside the same program and computational and domain (such as for internal couplings, or for interpolation from a previous to a modified mesh), the "coupled" domain is actually the same as the "local" domain, so the number of ranks is equal to the communicator size and the start rank is 0.
- when coupling between two programs A and B, we assume that A is assigned the first n ranks and B the last p ranks of a communicator of size $n+p$. So seen from A, the number of ranks and start rank of the distant location (B) are p and n respectively, while seen from B, the distant location (A) has n ranks starting at 0.

PLE does not handle the case where ranges for each program partially overlap: either they overlap completely (internal mapping) or not at all (coupled programs running on a separate set of MPI ranks). The algorithms do not depend on this hypothesis, which mostly allows a simple understanding of the MPI rank placement. computational domain placement.

To avoid requiring a more complex data model, PLE delegates all rank-local point location operations to the caller program. This is done by requiring that the calling code provide two functions:

- a function returning the “extents” associated with a mesh.
  The minimum required functionality for this function is to compute whole mesh extents, but to provide for future higher performance algorithms, it could also return extents of individual or subsets of elements
- a function locating a given set of points on a local mesh

Given these two functions, PLE can determine which ranks contain possibly matching elements and points (comparing the extents intersections to send data only where it is useful), and send copies of subsets of a domain’s points to all the ranks whose elements may contain those points, obtaining the best match after comparing values returned from different ranks. This is done using the simple `ple_locator_set_mesh` function, which locates a given point set on a mesh. If some points are not located, a call to `ple_locator_extend_search`, may be used to try to update the location only of those points not yet located (usually by increasing the tolerance, but using other search functions is also possible).

Extents computation function

The function computing extents should match the `ple_mesh_extents_t` typedef, which has the following definition (where `int` is replaced by a `ple_lnum_t` typedef for future-proofing in the actual code):

```
typedef ple_lnum_t
(ple_mesh_extents_t) (const void  *mesh,
                      int          n_max_extents,
                      double       tolerance,
                      double       extents[]);
```

Currently, only the extents of the full (local) mesh are used, but to allow for future algorithmic improvements without requiring changes in the API, it is recommended that it be able to compute local element or element subset extents. As such, it takes an argument indicating the maximum local number of extents it should compute, but returns the number of extents really computed, which may be lower (usually 1 for mesh extents, possibly 0 if the mesh is locally empty). If `n_max_extents` = 1, the whole mesh extents should be computed.

If `n_max_extents` is set to a negative value (-1), no extents are computed, but the function returns the maximum number of extents it may compute. This query mode allows for the caller to allocate the correct amount of memory for a subsequent call.

Point location function

The function computing extents should match the `ple_mesh_extents_t` typedef, which has the following definition (where `int` and `double` are replaced by `ple_lnum_t` and `ple_coord_t` typedefs for future-proofing in the actual code):

```
typedef void
(ple_mesh_elements_locate_t) (const void    *mesh,
                              float          tolerance_base,
                              float          tolerance_fraction,
                              int            n_points,
                              const double   point_coords[],
                              const int      point_tag[],
                              int            location[],
                              float          distance[]);
```

The tolerance arguments simply define a absolute and relative tolerance, so that bounding box actually used for point in element tests is expanded by the given tolerance. For example, for the x-component of a bounding box of width w_x the actual width considered will be: $w_x.(1+tolerance_{fraction}) + 2.tolerance_{base}$.

The number of points and their coordinates define the target points which we try to locate on the given mesh. Those points may be related to the mesh or not. In our co-located cell-centered Finite Volume schemes, points used with this

function are usually the cell or face centers of a mesh or portion thereof we are mapping to another. They could also be vertices or Gauss points in vertex-based or Finite-Element schemes, or represent independent particles.

The point tag is an optional argument which allows specifying that those points will not be located on mesh elements with the same tag. This is useful when mapping parts of a mesh to another in the same computational domain, for example vertices on each side of a sliding interface. Since we do not provide connectivity information on the existing relation of the points the mesh, using the point tag for points and the elements they belong to ensure we can map points the to the elements on the other side of the interface, ignoring the elements they already belong to.

The location and distance arrays are associated with the points and updated by the location function, and are the key to the parallel logic used here.

- the `location` array returns the mesh element id containing a given point, or -1 when the point has not been located
- the `distance` array returns the distance associated with the current location element:
  - `distance` < 0: the point has not been located
  - `distance` > 0: the point has been located

When possible, is is recommended to use a distance between 0 and 1 when a point is inside and element (in which case there is no being “closer” to that and a distance greater than 1 when a point is within tolerance but not quite inside an element.

This can be better understood if we assume the distance is based on the maximum value of a point’s barycentric coordinates in a given element, such that $d = max_{i,i=1,n}(2.|\lambda_i - 0.5|)$, and updated only if that point is indeed inside or just outside that element. In the following figure, where shades of gray represent different ranks, the blue point is well inside the highlighted element, while both the green and purple points are just outside. The distance returned for the blue point should be between 0 and 1, while that for the green and white points should be slightly above 1, if within the chosen tolerance ratio. No element is closer to the green point, so it will remain located there, but when we test for the location of the white point in the neighboring element (on another subdomain and MPI rank in this case), we will obtain a distance less than 1, so will choose that element as the one the point is finally located in.

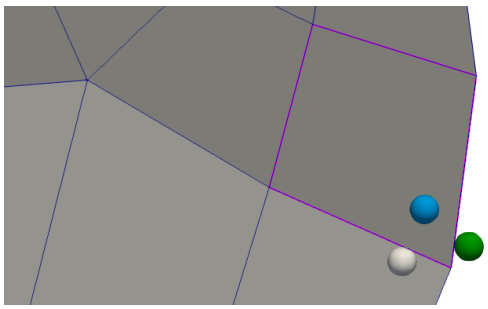

Figure 7: different cases of location inside or just outside a cell

For 3D surface meshes, the absolute distance to the surface may be a more sensible choice.

Interpolation or projection method

Interpolation is deliberately not a feature of PLE, as the choice of interpolation depends heavily on the type of discretization used. The API simply allows determining the number of target points located relative to the associated local “source” mesh, and reading arrays containing a copy of their coordinates and the matching element id.

In a co-located Finite Volume tool such as code_saturne, a simple volume interpolation scheme, given a point P's coordinates, a variable field A's cellwise gradient, and the associated cell center, is to use a first-order Taylor expansion:

$$A_p = A_{center} + \vec{\nabla}\left(A_{center}\right) \cdot \left(\vec{X}_p - \vec{X}_{center}\right)$$

For boundary variables, other interpolation schemes are possible.

Once values are assigned to the target points using the caller's preferred interpolation scheme, values can be scattered to their target points on their assigned MPI ranks using the `ple_locator_exchange_point_var` function. A "reverse" mode fo this function also allows passing extra information on the target points (in addition to their coordinates) to the donor elements.

Other functions

The PLE locator API also has a few simple functions to determine how many target points were not located (as seen from the rank owning these points), and list of located and unlocated points. Including these and a few timing and minor optional setting functions, the total number of functions in this API doe not exceed 20.

Point location function used by code_saturne

Whereas code_saturne's main mesh representation is face-based with ascending connectivity (based on face→ vertex and face→ cell adjacency), we chose to implement the location/interpolation functions using the the more common "node-based" connectivity, estimating that this would allow simpler and faster "point in element" tests for all simple element types, using barycentric coordinates or similar mappings.

Building a node-based connectivity can be done in quite efficient manner by the FVM library, and can optionally be restricted to subsets of the volume or surface mesh, as it was first designed to allow conversion and output to external mesh formats. This requires that it allows mesh extract representations using structures easily mappable to those of the unstructred mesh data models used by EnSight Gold, MED, or CGNS [17] (to which it has an export capability), or VTK [18] or Exodus II [19] (which could be considered).

To optimize the local search, it is necessary that at least of the the point sets or element sets can be queried rapidly, to avoid quadratic point over elements loops. In the code_saturne location function (similar to that provided in the PLE example code), the query loops over all elements, using an octree to located the vertices lying inside each element's bounding box. Since the octree only references vertex coordinates, if is simpler that an tree that would reference elements which can have a more complex overlap behavior. Another advantage to this method is also that it allows subdividing polyhedral elements "on the fly" (reducing memory usage), and only per location call if their bounding boxes indicate potential points inside.

Combining more advanced local search structures for elements with those used for vertices could certainly improve performance in a significant manner, especially, in cases where there are very few points to locate and many elements to located them in.

Use with other types of meshes.

An interesting aspect of the approach used here is that the point location function used by the caller code can be optimized relative to its own mesh structure. For example, a structured or octree-based code could certainly implement much faster location functions than the more or less generic ones used for unstructured meshes.In the same manner, though the location functions used by code_saturne assume linear element types, an tool using high order curvilinear meshes could adapt the location function relative to the mesh features.

It is assumed that many mesh-based scientific simulation tools already implement algorithms for the location of points relative to a mesh. When this is the case, adapting those functions to the PLE specifications (rather than copying or mapping the full data structure to a less familiar one can be the more efficient solution).

Data movement

Since the PLE approach avoids moving mesh data, exchanging only arrays of point values (coordinates and possibly tags) it may lose some optimization algorithms compared to a rendezvous approach in which mesh elements are exchanged [20], but even if it leads to a few additional data exchange sequences or distributing data over more MPI ranks, the fact that the data moved is “lighter” may keep this approach competitive.

Symmetric exchange

A specific aspect of the PLE library is to possibility to make exchanges symmetric. Both when locating points on a mesh and exchanging data, each task can simultaneously request location and data on a set of points, and provide a (local) mesh on which points from multiple tasks will be located and data provided.

An example application is boundary condition based coupling for turbomachinery applications. In code_saturne, two rotor-stator models are available. The first implies full mesh joining, and is recommended when meshes are sufficiently regular. An alternative model is based on boundary coupling using the pLE library. Using this model, points (face centers) from one mesh may be located on opposing faces from the same mesh. An illustration of the coupling surface is given below (with the coupling surface in red).

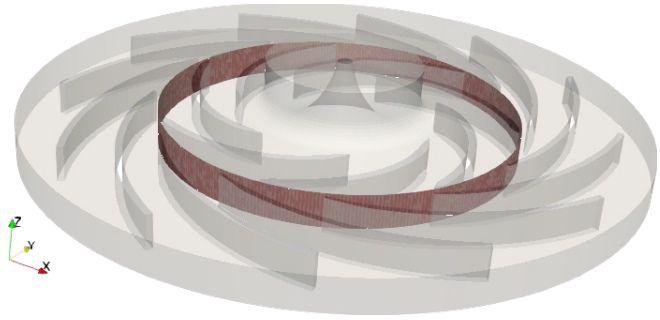

Figure 8: rotor stator (pump= example geometry

To avoid locating cell faces on the faces they already belong to, we use an optional “exclusion tag” mechanism. Associating a tag to both points and elements, elements with the same tag as a given point are ignored when locatin that point. Defining tags as exclusive rather than inclusive was chosen as it allows simpler definitions in multiple rotor configurations; we do not need to keep track of which rotor zone is matched with a given stator (or other rotor) zone, only to ensure the points have the same tag as their owning face, which can easily be determined based on their adjacency.

In the following view, we show points and faces from both matching sizes on a rotor-stator section. Shades of green indicate MPI ranks for tag 1, shades of gray ranks for tag 2.

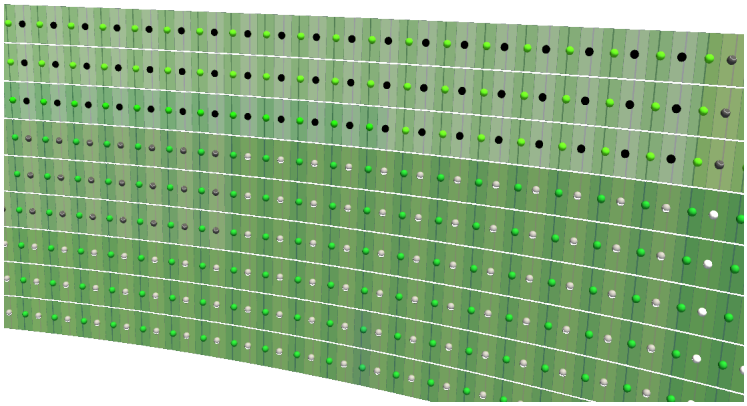

Figure 9: points to locate and location faces on both sides of interface

Showing only points with tag 2 and faces with tag 1, we see which location can actually occur.

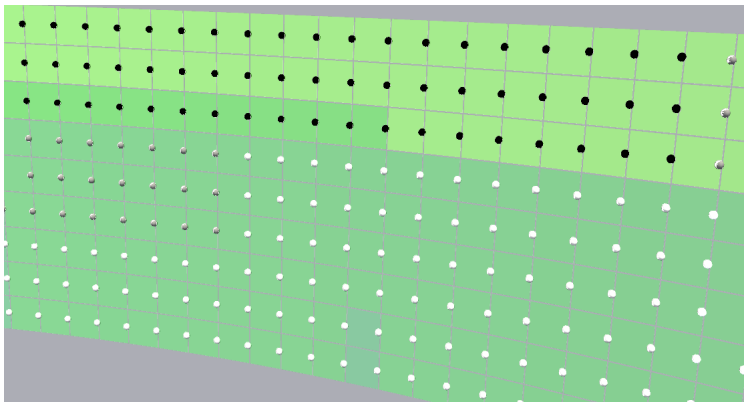

Figure 10: points to locate and location faces filtered by tag

5 POINT LOCATION PERFORMANCE BEHAVIOR

Performance was measured on a test case commonly used for weak scaling studies, representing a tube bundle. This case's mesh is quite regular, with small variations in cell size and small aspect ratios, so performance measured should be better than on some more complex or irregular cases. Based on the number of tube repetitions, mesh sizes used include 12.4, 51, and 204 million cells.

To measure location performance on a significant number of points and cells, we force the location and interpolation used when restarting a computation on a different mesh, even though in this case the initial and restart meshes are identical. To ensure the case is not too simple, we use a different partitioning scheme for the computation and restart mesh, as shown on figures 14 and 14 for the 204 million-cell mesh. In this configuration, we have a non-symmetric exchange, where points (cell centers) from the current mesh and partition are located relative to the old mesh and partition.

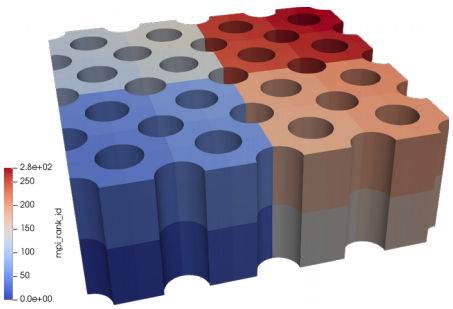

Figure 11: BUNDLE: Morton SFC partition

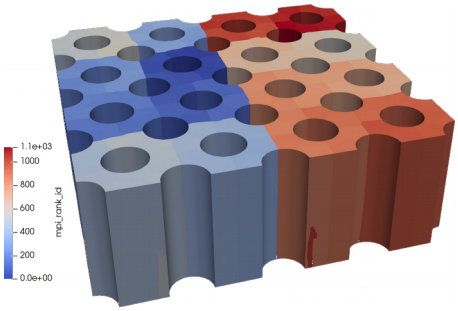

Figure 12: BUNDLE: PT-Scotch partition

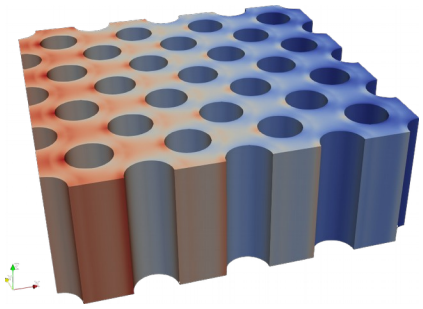

Figure 13: BUNDLE: pressure field

The number of “neighbors” should not increase much with the processor count (given a “good” domain splitting).

Tests were run on EDF’s Gaia cluster, whose compute nodes are based on Intel Xeon Gold 6140 18C Processors connected by and Intel Omni Path fabric. Several variants were tested:

- Handling of periodicity in the graph provided to thr PT-Scotch partitioning was deactivated in some cases (“np” in the graph legend), as periodicity may cause subdomains to “wrap” around the whole domain, resulting in larger bounding boxes. Since this only affects the main mesh but not the initial mesh used for point location).
- Forcing the older “unordered” PLE algorithm, where the `MPI_Sendreceive` series are done in a rank-sequential manner instead of first exchanging with neighbor ranks (recursively considering half of the ranks in the top-level MPI communicator and first exchanging within that subset)

For best results, 35 of the 36 cores per node are used for the computation, so the number of MPI ranks uses is a multiple of 35. In figure 15, we see that ignoring the (y-axis wise) periodicity in the partition does not provide any benefit, but actually leads to lower performance. We would expect the opposite, unless the partitioning quality is significantly modified, so we can consider that the performance difference illustrates variability in results based on the partitioning rather than a deterministic tendency.

We see that performance certainly dos not scale linearly, but for a given mesh size, adding more ranks does improve performance. While adding more ranks leads to more communication steps and (in the case of the local point location function used here) additional loops over all local elements to locate potential points, the number of elements per rank decreases, and the net gain is positive.

In this test case, the location cost is about 1.5 to 4 times the partitioning time, and comparable to the time needed for a few time steps, but does not scale as well as the computation. So the algorithm is usable to high rank counts for pre-processing type operations, but becomes costly if it needs to be repealed frequently (note that we are doing 3D volume location here: many usages of PLE imply boundary points and elements so are significantly less costly).

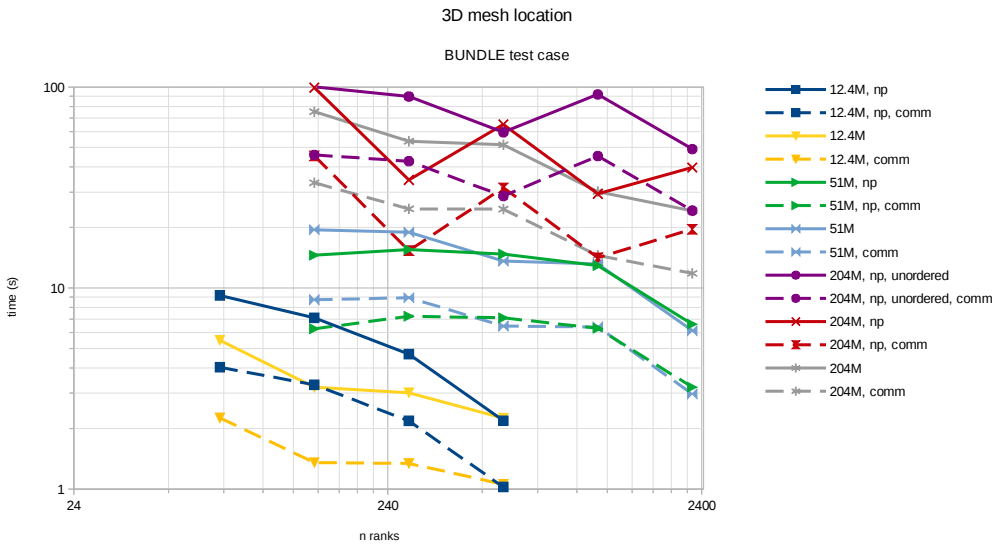

Figure 14: mesh location times (subdomain bounding boxes)

Once the mesh location is done, values exchange is much faster. Though to ensure we avoid deadlocking due to an excessive number of messages received or send on a given rank, we can use a similar `MPI_Sendrecv` sequence as for mesh location, we can define a threshold (by default 100) such that when the maximum number of communicating ranks does not exceed this threshold, asynchronous `MPI_Isend`/`MPI_Irecv` are used instead.

So as seen in figure 15, the performance of values exchange scales better, and the associated exchanges mush faster than for the location stage. For the smaller mesh, the degradation of performance between 280 and 560 MPI ranks is not restricted to this operation. For the full computation, performance still increases, but the communication/computation times ratio begins to degrade, and latency becomes an issue.

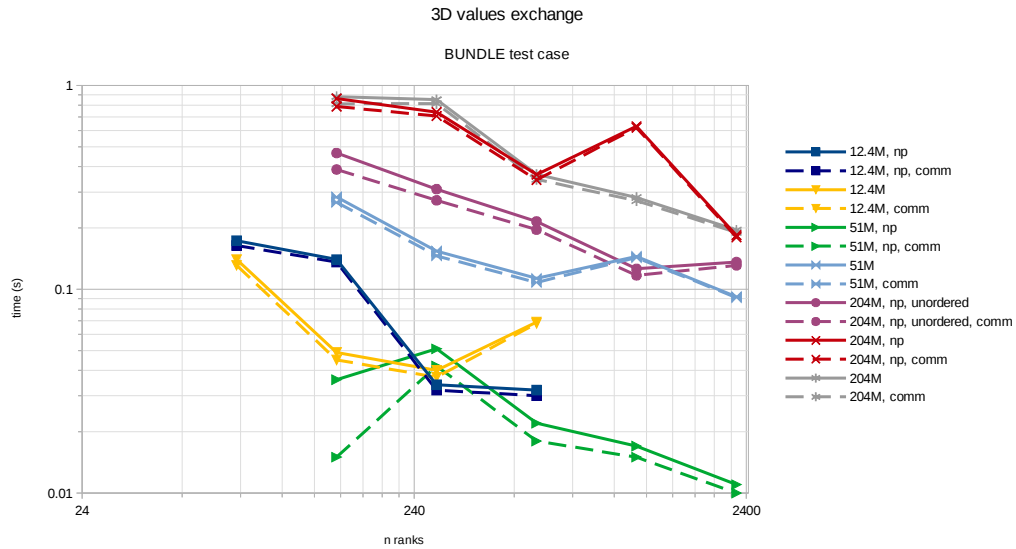

Figure 15: values exchange (MPI_Isend/MPI_Ireceive)

Possible optimizations

While the current algorithms have worked well so far for the intended uses, moving to higher rank counts will require some improvements.

A simple approach would be to use asynchronous communications and group point sets from different ranks in the location step, but in complex geometries or partitions where the bounding box volume/partition volume ratio is high, this could lead to high memory usage requirements, so we avoid this approach.

In other parts of code_saturne, a parallel “box-tree” is used for mesh element neighborhood detection, especially in the mesh joining stages. The box-tree is similar to an octree, but contains bounding boxes instead of points, and when boxes overlap multiple octree quadrants, they are copied to each quadrant. This leads to additional memory usage, and the tree depth needs to be controlled not only based on a maximum number of elements per leaf, but also based on the size ratio relative to the top-level data due to bounding box duplication. The tree is built bottom-up, so is always partitioning, and a temporary,,coarser version of the tree is used by the algorith to estimate load balance.

The advantage of using a box tree or similar structure is that since bounding boxes are used not only at the partition level but at the element level, the possible element point/element matching is done at a finer grained level, reducing the size of the point sets to send to each rank for the local point location test, and making it possible to run this step in a single pass.

To check how this would behave on the above test case, we measured the performance of this box tree to locate neighbors in this context, using boxes 1/10 or the matching element bounds to represent points (a structure separating points and element bounding boxes would allow further improvements). This was done for various settings of the box-tree memory usage vs. performance trade-off settings, with results shown in figure 16 for the 204 million cell case.

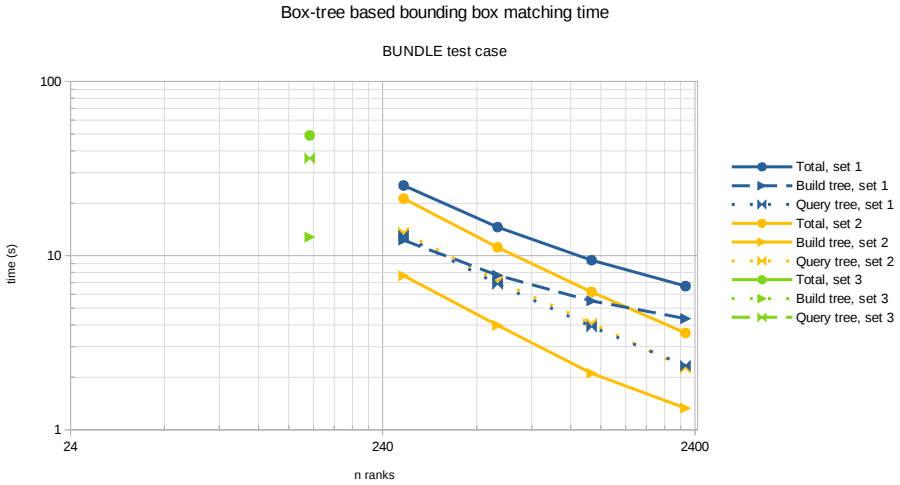

Figure 16: parallel box tree query time

The additional memory usage required for these steps is shown figure 17. Note that more aggressive settings (leading to a slower query) needed to be used when running on only 140 ranks, as the memory overhead was otherwise too high to run the computation.

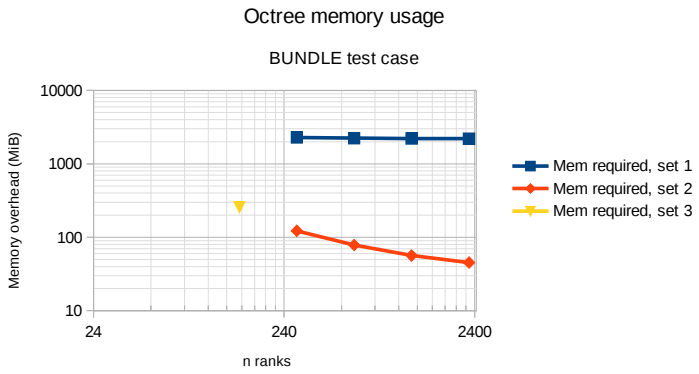

Figure 17: parallel box tree memory overhead

We confirm that the box-tree would a a good candidate for an improved algorithm, but its memory requirements must be constrained by adequate settings, as the defaults used for surface mesh matching (with much less elements) lead to excessive memory usage. Also, though only 4 values are required to control this performance/memory usage trade-off, their combinations can be complex, and are already uncomfortable to use whe we strive for a “black box” library.

Another alternative would be to use a Kd-tree type algorithm, which might be slightly slower than the ideal box-tree case, but would allow (at least according to preliminary design) handling load balancing in a simpler manner.

6 CONCLUSIONS

The approach and algorithms used in the PLE library have allowed handling parallel n to p couplings in a simple manner, targeting development teams who have good access to their code’s features and prefer to keep cascading external library dependencies to a minimum, especially when other code features already imply quite a few such dependencies. It deliberately avoids using a data model, allowing its adaptation to many different types of codes, though abandoning some advanced features which are associated with such models.

In actual industrial computation environments, this approach can be combined with those of more complex libraries to allow for a larger feature set, and having access both to complete but often changing APIs and a more minimalist but stable API. Through built-in algorithm versioning, it should be possible to extend this approach to very high rank counts without any external API changes, and experiments with other low-level algorithms such as one based on octree type structures show that scaling could be dramatically improved while maintaining this stability and simplicity.

REFERENCES

[1] Message Passing Forum, “MPI: A Message-Passing Interface Standard”, https://dl.acm.org/doi/book/10.5555/898758 (1994)

[2] F. Archambeau, N. Mechitoua and M. Sakiz, “Code_Saturne: a Finite Volume Code for the Computation of Turbulent Incompressible Flows”, International Journal on Finite Volumes 1 https://hal.archives-ouvertes.fr/hal-01115371/document (2004)

[3] Y. Fournier, J. Bonelle, C. Moulinec, Z. Shang,A.G. Sunderland and J.C. Uribe, “Optimizing Code_Saturne computations on Petascale systems”, Computers & Fluids 45:1 103 - 108 ISSN 0045-7930 https://doi.org/10.1016/j.compfluid.2011.01.028 (2011)

[4] SALOME platform documentation, “MEDCoupling user’s guide”, https://docs.salome-platform.org/latest/dev/MEDCoupling/index.html

[5] Y.-H. Tang, S. Kudo, X. Bian, Z. Li, and G. E. Karniadakis, “Multiscale Universal Interface: A Concurrent Framework for Coupling Heterogeneous Solvers”, Journal of Computational Physics 297.15 13-31 https://doi.org/10.1016/j.jcp.2015.05.004 (2015)

[6] S. R. Slattery , P. P. H. Wilson and R. P. Pawlowski, “The Data Transfer Kit: A geometric rendezvous-based tool for multiphysics data transfer”, International Conference on Mathematics and Computational Methods Applied

to Nuclear Science & Engineering (M&C 2013) Sun Valley, ID, May 5-9, 2013, American Nuclear Society https://www.casl.gov/sites/default/files/docs/DTK_slattery_wilson_pawlowski_MC_Proceedings_2013.pdf (2013)
[7] H.-J. Bungartz, F. Lindner, B. Gatzhammer, M. Mehl, K. Scheufele, A. Shukaev, B. Uekermann, “preCICE – A fully parallel library for multi-physics surface coupling”, Computers & Fluids 141: 250-258 ISSN 0045-7930 https://doi.org/10.1016/j.compfluid.2016.04.003 (2016)
[8] A. Refloch, B. Courbet, A. Murrone, P. Villedieu, C. Laurent, P. Gilbank, J. Troyes, L. Tesse, G. Chaineray, J.B. Dargaud, E. Quemerais, and F. Vuillot, “CEDRE Software”, Aerospace Lab Journal http://aerospacelab.onera.fr/sites/www.aerospacelab-journal.org/files/AL2-11.pdf (2011)
[9] F. Duchaine, T. Morel and A. Piacentini, “On a first use of CWIPI at CERFACS”, Report TR-CMGC-11-3,CERFACS ,http://pantar.cerfacs.fr/globc/publication/technicalreport/2011/TR-CMGC-11-3.pdf (2011)
[10] W. Haase, M. Braza, A. Revell, “DESider – A European Effort on Hybrid RANS-LES Modelling”, Results of the European-Union Funded Project, 2004–2007, https://doi.org/10.1007/978-3-540-92773-0 (2009)
[11] S. Benhamadouche., Y. Fournier, F. Billard, N. Jarrin, R. Prosser, “RANS/LES coupling in the industrial CFD tool Code_Saturne: implementation and first results”, Proceedings of international symposium on turbulence, heat and mass transfer , (2008)
[12] W. Joppich and M. Kürschner, “Mpcci - a tool for the simulation of coupled applications”, Concurrency Computation Practice and Experience. 18.2 :183–192 (2006)
[13] R. Noack, “DiRTlib: A Library to Add an Overset Capability to Your Flow Solver”, 17th AIAA Computational Fluid Dynamics Conference , https://arc.aiaa.org/doi/abs/10.2514/6.2005-5116 (2005)
[14] X. Wu, S. Hahn, M. Herrmann, J. J. Alonso and H. Pitsch, “A python approach to multi-code simulations: CHIMPS”, , (2005)
[15] J.C. Cajas, M. Zavala, G. Houzeaux, E. Casoni, M. Vázquez, C. Moulinec, and Y. Fournier, “Fluid Structure Interaction in HPC Multi-Code coupling”, In The Fourth International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering (PARENG15) Dubrovnik (Croatia), 24-27 March 2015, (2015)
[16] R. Oldfield, G. Sjaardema, J. Lofstead, and T. Kordenbrock, “Trilinos I/O Support Trios”, Scientific Programming 20: 181:196 https://doi.org/10.1155/2012/842791 (2012)
[17] CGNS Web-site, “CFD General Notation System”, http://www.cgns.org/
[18] W. Schroeder, K. Martin, B. Lorensen, “The Visualization Toolkit”, , (2006)
[19] Exodus II documentation, “EXODUS:A Finite Element Data Model”, http://endo.sandia.gov/SEACAS/Documentation/exodusII.pdf
[20] S. J. Plimpton, B. Hendrickson and J. R. Stewart, “A parallel rendezvous algorithm for interpolation between multiple grids”, Journal of Parallel and Distributed Computing 64.2: 66 – 276 ISSN 0743-7315 https://doi.org/10.1016/j.jpdc.2003.11.006 (2004)