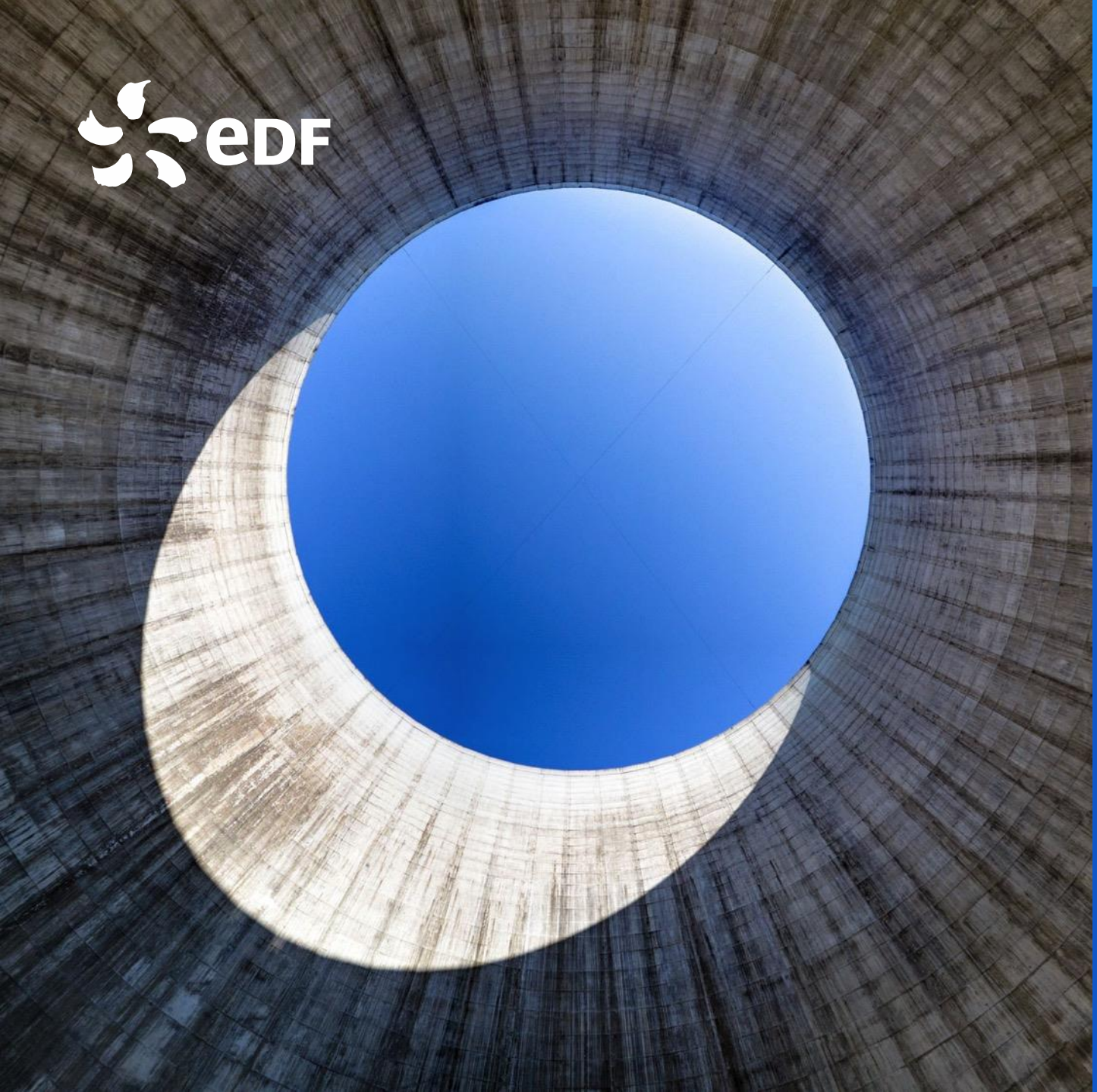


Tools for code coupling

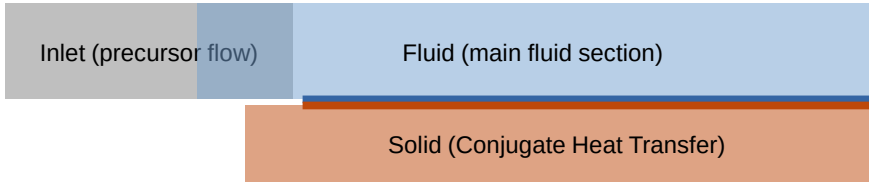
PLE : Parallel Location and Exchange

1 Introduction

Coupling Paradigm

The PLE (Parallel Location and Exchange) library is designed to enable couplings between mesh-based simulation tools using MPI-based parallelism.

- Both volume (with overlap) and surface-based couplings are possible
- Multiple couplings must be possible, as shown in the following illustration:



- For each coupling, an MPI intracommunicator is constructed and used.
- Most function calls involving communication appear as a form of high-level MPI collective operations.
- PLE currently requires MPI but could easily be extended to other communication libraries with only minor changes.

2

Origins and History

Explaining the design choices

Origins of the PLE library (context and team)

The parallel Location and Exchange Library is a subset of the code_saturne CFD tool

- Code_saturne (https://code-saturne.org) is EDF's main CFD flow solver
  - Developed mostly by EDF since 1997
    - With collaborations with academia or other research labs, especially *University of Manchester* (originally UMIST) and *UKRI-STFC* in the UK.
  - Mostly based on a co-located Finite Volume approach
    - Handles unstructured meshes with any type of cell (tetrahedral, hexahedral, prismatic, pyramidal, polyhedral…)
  - Parallelized using a classical domain partitioning approach (using MPI)
    - Good weak scaling to 10 000's of MPI processes
    - Can handle multi-billion cell meshes
    - With some hybrid parallelism (out of the scope of this presentation)
  - Released under the GPL licence since 2007
  - 5 to 10 "core"" developers, plus several contributors on specific models

- PLE developers and development effort are part of the code_saturne development pool
  - PLE was only recently moved to its own Git repository : https://github.com/code-saturne/libple

Origins of the PLE library (requirements)

Defining use case

- Couple 2 instances (or more) of code_saturne in partially overlapping domains with different turbulence models (RANS/LES)
  - each domain using its own mesh;
  - partial overlapping of communication domains possible, so coupling of different variables may be of the surface ↔ surface, surface ↔ volume, or volume ↔ volume type.
- Initially developed in the context of the DESider European project (2004-2006).

Requirements for location and interpolation :

- Coupling must be usable large LES meshes, which do not fit in a single node's memory but need to be distributed
  - This implies it must be usable in distributed parallel mode (i.e. using MPI)
    - For example, n processors for zone and associated code instance with turbulence model A, p processors for instance using turbulence model B;

  To avoid memory bottlenecks, all stages of the coupling mechanism must remain distributed
    - Preprocessing on a large shared-memory front-end node to be avoided, as it would complexify the workflow and might still ne be enough
- Must handle all (linear) element types (especially hexahedra and polyhedral)
  - localization of points in cells, or on faces, allowing for some means of geometric tolerance;
- After location, use the code's own interpolation (greater flexibility and consistency with numerical scheme)

Origins of the PLE library (code base)

To our knowledge, no freely available and distributable library handled these requirements at the time

- MPCCI was not freely available, and did not handle polyhedra
- DIRTlib (Donor Interpolator Resource Transaction library) seemed nice, but did not seem to be freely available.
- EDF and CEA's MEDCoupling or CASCL's Data Transfer Kit did not exist yet
- ANL's MOAB appeared at a similar time.

In the context of the code_saturne parallelization effort, a **Finite Volume Mesh (FVM)** library was developed to handle various mesh import/export aspects, including postprocessing output.

- As FVM handles representation of cells in classical nodal or descending connectivity (elements→ vertices), it is a good place to handle point-in-cell tests using barycentric or parametric coordinates .
  - Most parts of code_saturne use a faces → cells connectivity, not a good fit for this
  - So the parallel coupling features were developed in the FVM library
- When code_saturne was released under the GPL licence in 2007, FVM was licenced under the LGPL, as it was also used by other non-GPL projects

From FVM to PLE : separating the mesh model

When EDF's heat transfer simulation code (Syrthes) was parallelized, updating the existing conjugate heat transfer with code_saturne presented 2 main options if we were to reuse the work done on FVM

- Use FVM, with mesh conversions to and from Syrthes's non-interleaved (SOA) structures to code_saturne's interleaved (AOS-like) structures.
  - Would require extra copies
- Separate the mesh model from the parallel scheme
  - The resulting **Parallel Location and Exchange** library only handles pointers to each code's mesh structure, and a callback logic using local location functions provided by each code
    - A simplified version to the mesh structure and associated location functions is provided as an example, but is not built into the library
    - The rest of the FVM library was folded back into code_saturne.
      - Note that the original FVM library (before PLE extraction) was the basis for the **CWIPI** library.

Thanks to this minimalistic approach of the parallel (MPI) aspects, the common library that needs to be used by coupled simulation codes is very stable, requiring very little maintenance on the application side.

3

Example Workflow

Where and how PLE is used

Basic operations implied

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, each process defines 2 sets:

- A set of local points at which “distant” variables will be received and projected
  - 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.
- A set of local elements, relative to which distant points may be located, and local variables interpolated at these points, then sent to their owning processes (referred to as “interpolation basis” , or “donor” elements).
  - “Distant” refers to mesh connectivity: geometrically near but topologically non-adjacent points on the same MPI rank are considered distant for all purposes.

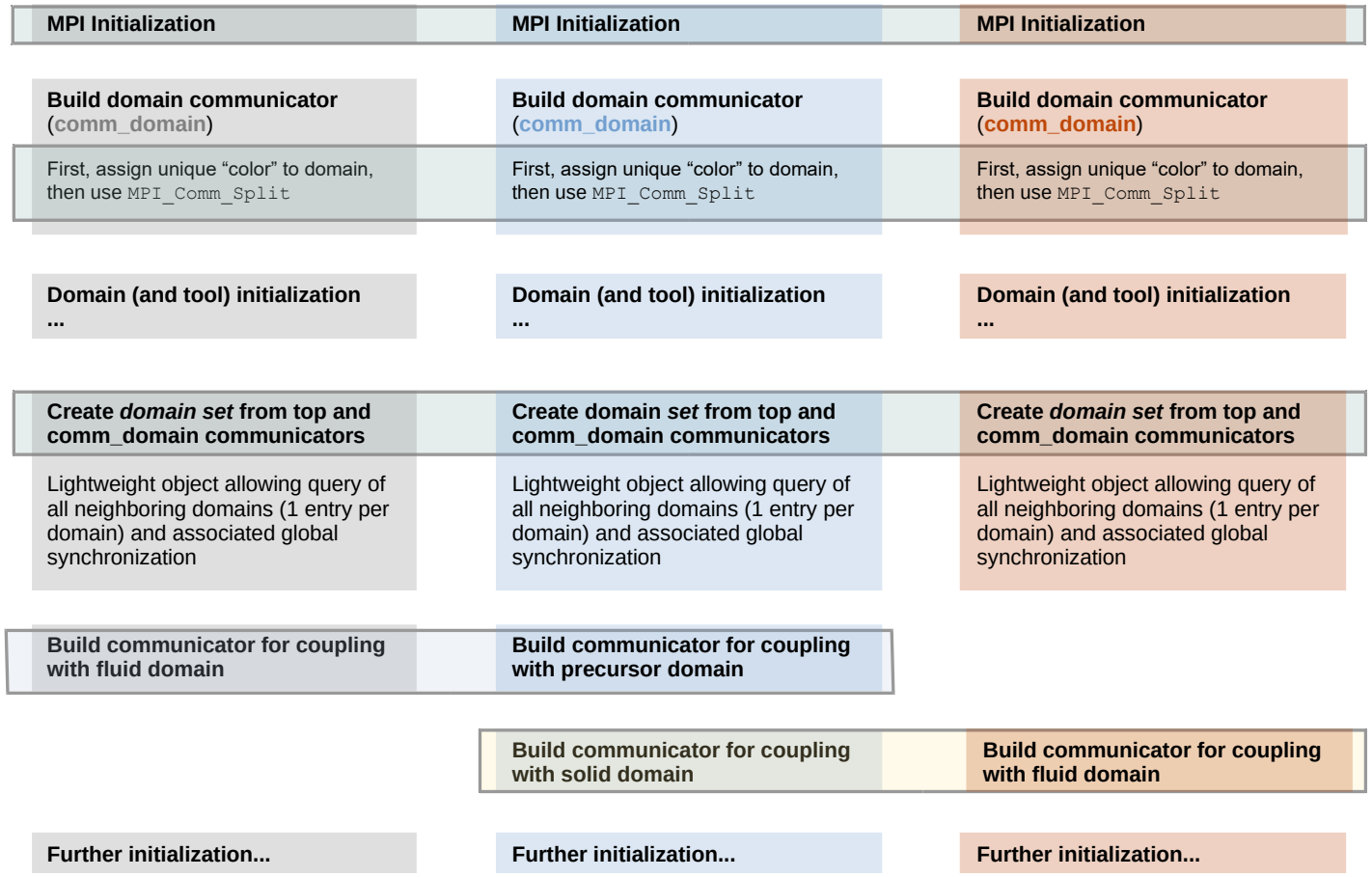
We choose to make the **API and algorithm symmetric** (i.e. a locator defines and uses both “receiver” point and “donor” element sets simultaneously), so that when coupling 2 instances or more of a same code (using the same variables), **the user need not specify which instance sends or receives first**.

- Avoiding deadlock is thus handled by the library, and not the user’s problem.

MPI communicator manipulation

To enable its main Parallel Location and Exchange role, PLE provides a first **`ple_coupling`** API, allowing simple communicator manipulation:

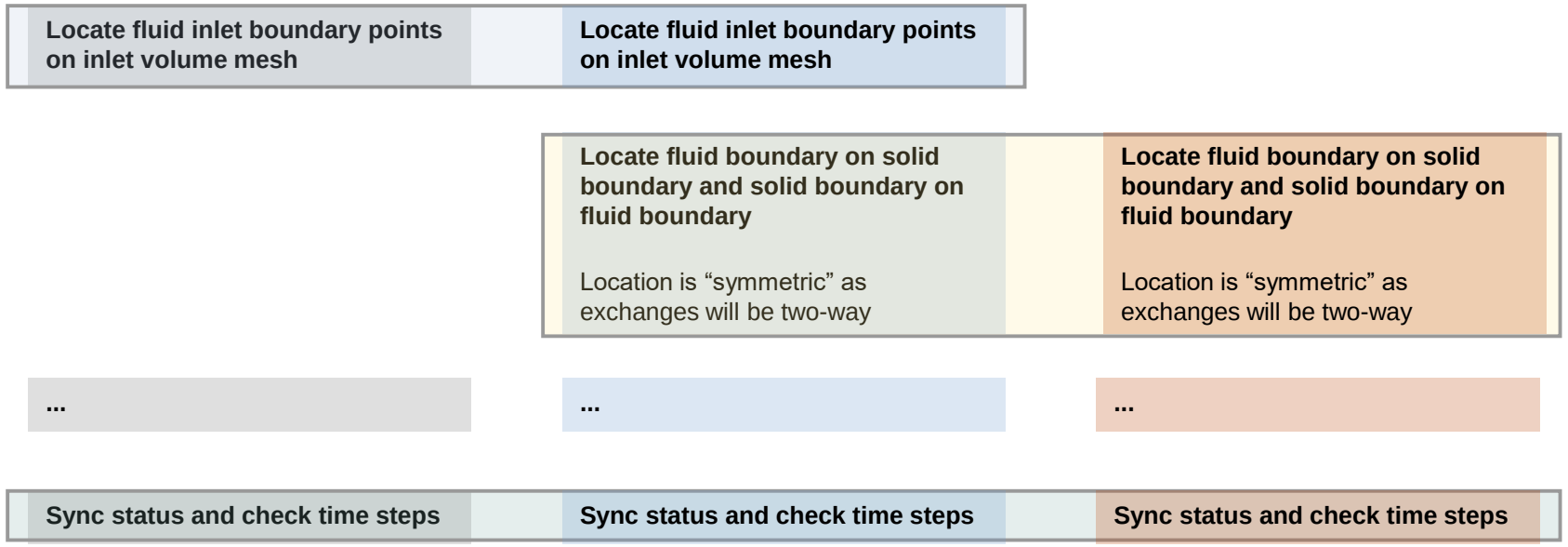
- MPI communicator splitting, based on a character string (name) instead of an integer key
- Build MPI intracommunicators based on MPI rank ranges.
  - Assume ranks [0, n1[for first domain, [n1, n2[for second domain, ...
- Provide a lightweight "coupling set" object to synchronize time-stepping status across codes.



Location and Exchange

The essential role of PLE is to assist in mapping from one mesh to another

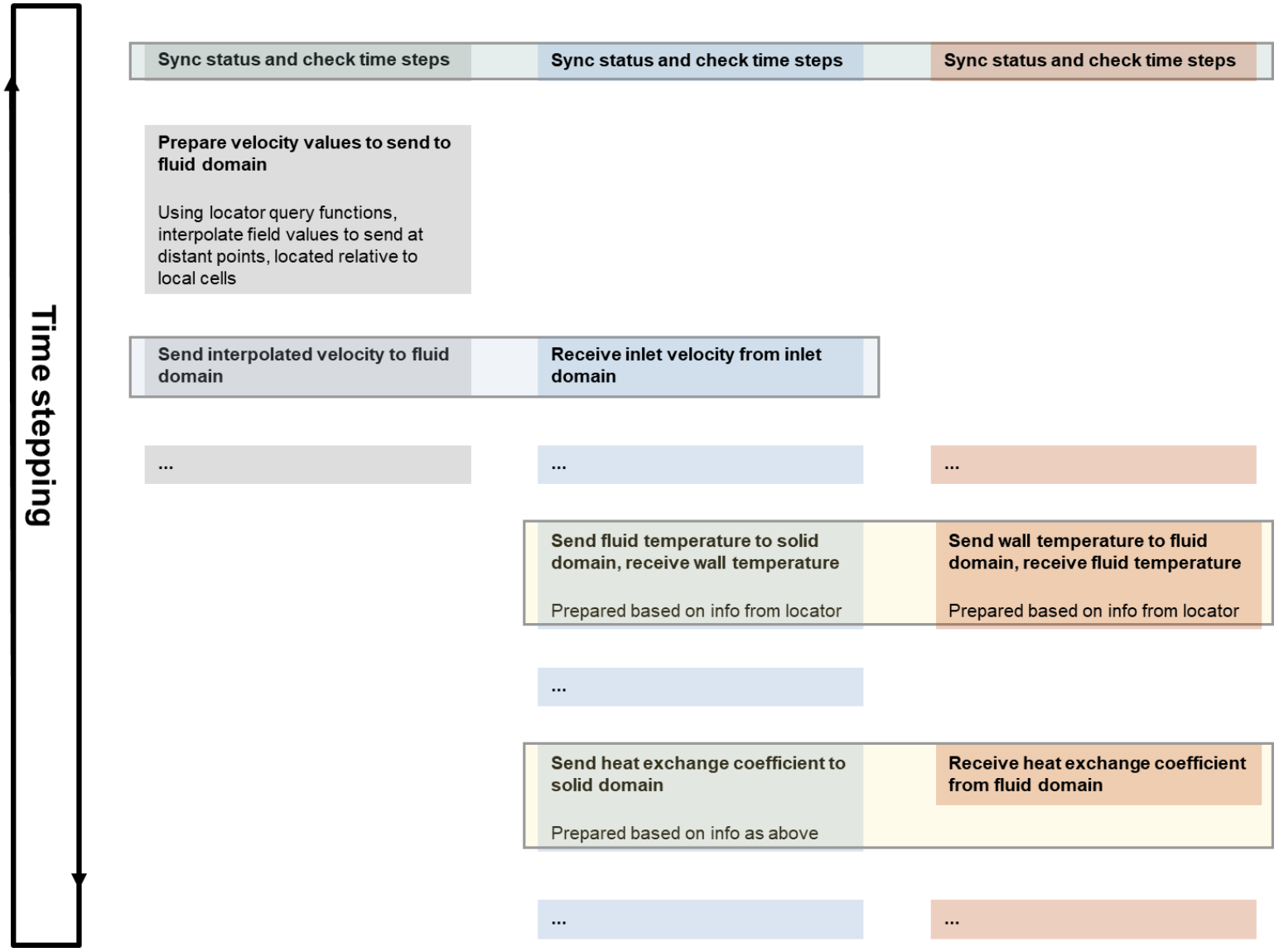
- This is done independently for each coupling, within the associated MPI intra-communicator.
  - A **locator** object keeps track of this.
- Depending on whether meshes are static or mobile, this can be done once or repeated during time stepping.



Time Stepping

Time stepping can involve the 2 PLE object types:

- locators used for field date exchange
- Coupling used for global synchronization and status checking
- Using only one of the 2 for a given coupling (and external libraries for the rest) is possible.



4

Location Principle

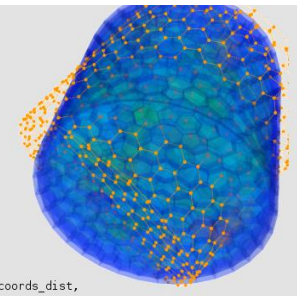
Priorizing robustness and memory usage

```
for (j = 0 ; j < n_coords_loc ; j++) {

  coord_idx = send_index[location_shift[i] + j];
  this_locator->local_point_index[start_idx + j] = coord_idx;
  send_location[j] = location[coord_idx];
  if (point_index != NULL) {
    for (k = 0 ; k < dim ; k++)
      send_coords[j*dim + k]
        = point_coords[dim*(point_index[coord_idx] - 1) + k];
  }
  else {
    for (k = 0 ; k < dim ; k++)
      send_coords[j*dim + k]
        = point_coords[dim*coord_idx + k];
  }
}

MPI_Sendrecv(send_location, (int)n_coords_loc,
             FVM_MPI_LNUM, dist_rank, FVM_MPI_TAG,
             (this_locator->distant_point_location
              + this_locator->distant_points_idx[i]), (int)n_coords_dist,
             FVM_MPI_LNUM, dist_rank, FVM_MPI_TAG,
             this_locator->comm, &status);

MPI_Sendrecv(send_coords, (int)(n_coords_loc*dim),
             FVM_MPI_COORD, dist_rank, FVM_MPI_TAG,
             (this_locator->distant_point_coords
              + (this_locator->distant_points_idx[i]*dim)),
             (int)(n_coords_dist*dim),
             FVM_MPI_COORD, dist_rank, FVM_MPI_TAG,
             this_locator->comm, &status);
```



Point location algorithm on a simple example

Let us illustrate what happens in a simple example, with 2 overlapping domains :

- RANS inflow is shown in gray, LES outflow in green; partial transparency is used, so that we may note the region of overlap;

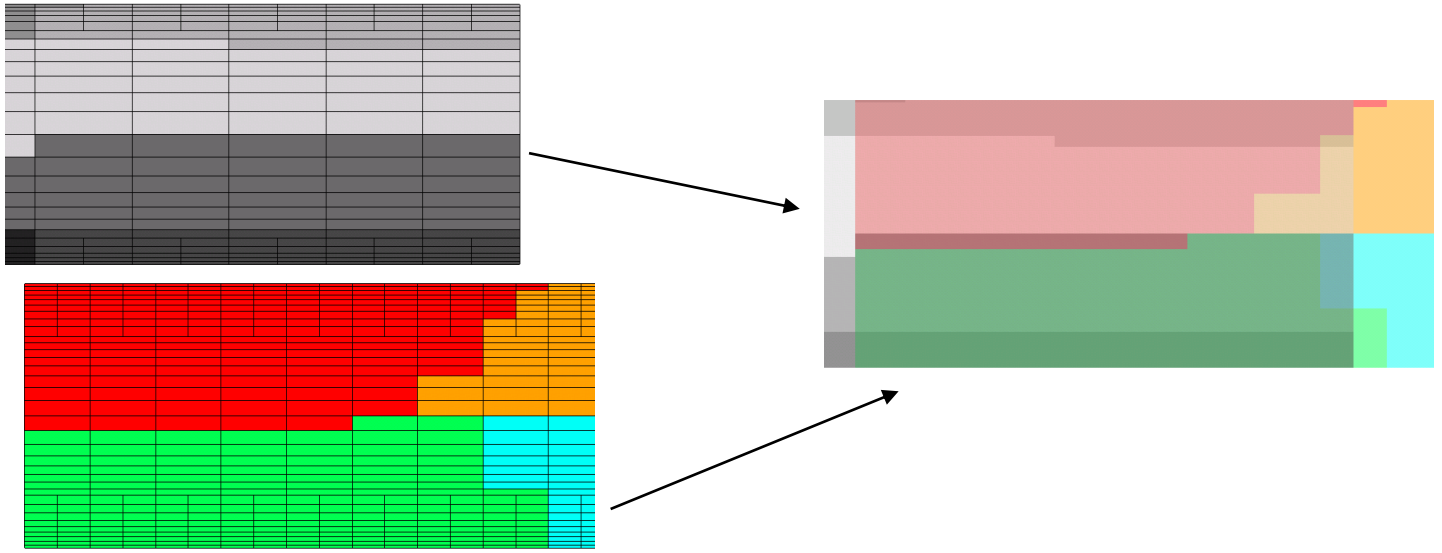
Now account for domain partitioning:

- Shades of gray are used for RANS inflow, colors for LES outflow; different shades or colors indicate different domains (and processors); overlap is still visible;

Point location algorithm on a simple example

Zoom in on overlapping region

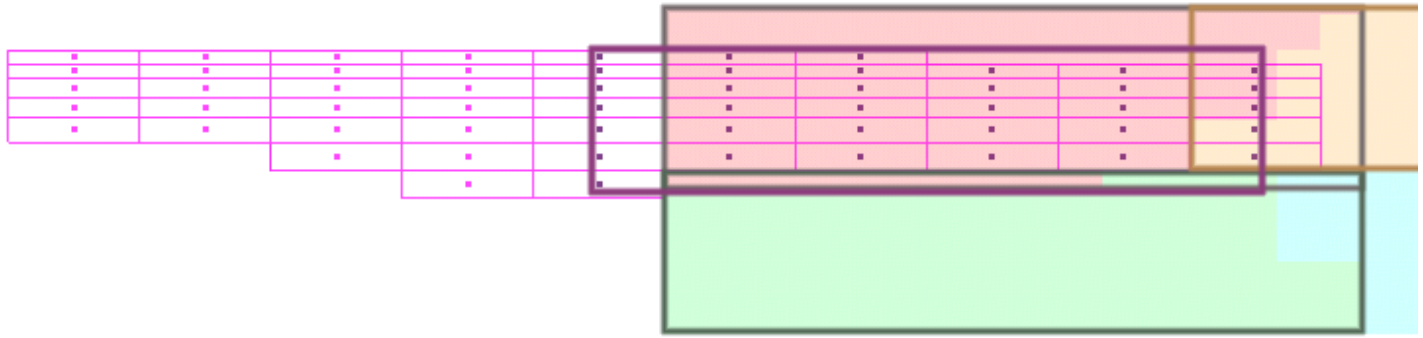
- On the left, part of the RANS domain is shown on top, LES domain on bottom, (using the same x axis);
- On the right, the corresponding overlap is seen in more detail;



Point location algorithm on a simple example

Same example, as seen from one RANS processor

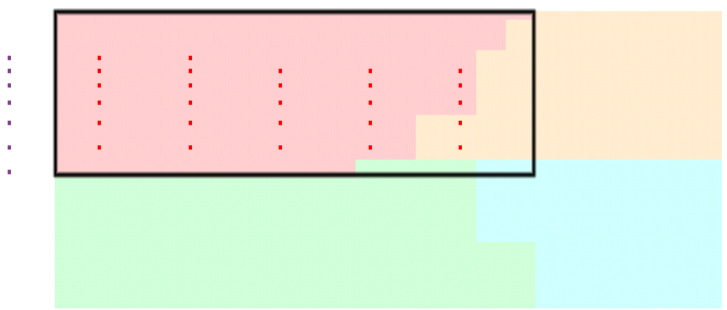
- Algorithm is symmetric, so this example is “sufficient”
- Select centers of coupled cells, and compute their bounding box (local step);
- In a similar fashion, compute the bounding box of selected “interpolation basis” cells;



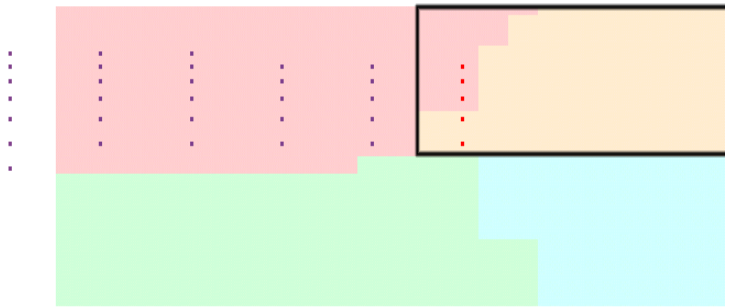
- Exchange bounding boxes between all processors (PLE);
  - The corresponding memory increases with MPI process count, but remains small
    - (2 boxes x 6 coordinates x total_ranks,
  - Only those processors whose selected point set and element set intersect will need to exchange data;
    - Single-level, could be extended to multiple levels with an octree-like structure to increase performance

Point location algorithm on a simple example

- 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 MPI rank, points received are located regarding to the (local) mesh
  - Return “best fitting element id” and corresponding distance (0 to 1 if inside element, > 1 otherwise)
  - 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 rank’s sub-domain (hence the following step of this algorithm).



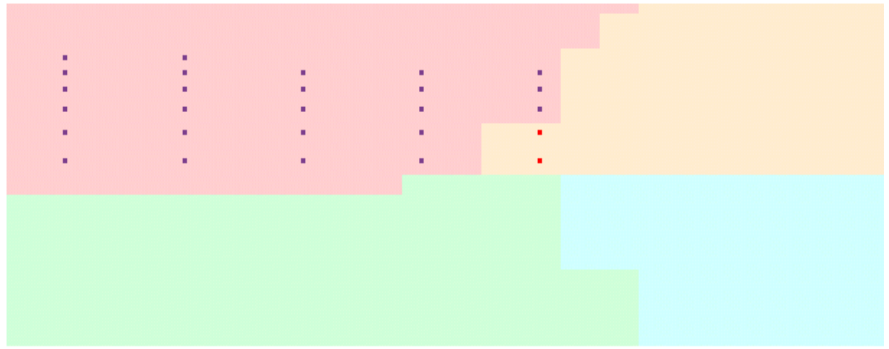
In red, set of points fitting
one domain’s bounding box



In red, set of points fitting
another domain’s bounding box

Point location algorithm on a simple example

- Each point is assigned to the rank which returned the smallest distance criteria for this point:
  - Exchange this final informationb between communicating neighbors;
- Each rank maintains coordinates of distant points it was assigned, as well as the ids of local elements containing them
  - 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 rank is maintained by the **locator** structure for future re-assembly of incoming interpolated variables.
  - It is hidden from the user, providing a high level “MPI collective” rather than “point to point” API.



In purple, set of points assigned to one domain;

in red, set of points assigned to another domain

Remarks on point set location algorithm

Possible optimizations

- We could use a few levels of “hierarchic” bounding boxes to obtain a finer representation of domain shapes and thus reduce the number of points sent to ranks whose sub-domains do not contain them.
  - The number of levels should remain small, as the bounding box data of all processors are known by all others at one point
- The number of “neighbors” should not increase much with the processor count (given a “good” domain splitting).
  - To be safe, we currently use ordered blocking communications (MPI_Sendrecv) to reuse the same buffers and limit memory use.

Local search

- Each coupled code should provide its own local point in mesh location algorithm
  - Example code can be adapted for linear unstructured elements
  - Structured and AMR codes can probably provide an optimal search
  - Unstructured codes with curved elements need to provide their own implementation
    - In return, they are not hampered by an inadequate model

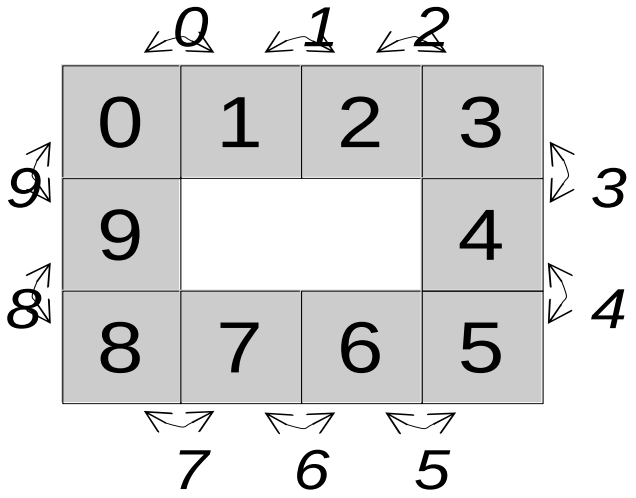
Communication Ordering

As some mesh partitions may represent a smaller volumethan their bounding box, more point coordinates can be received by a given rank than finally located.

- If all those coordinates were received simultaneously (using non-blocking communication), memory requirements could become excessive.
- If on the other hand we locate points then discard excess points in sequence, memory use should be better balanced (assuming a balanced partitioning).

To avoid communication deadlock,
a simple solution is to execute the `MPI_Sendreceive` series in increasing rank order.

- This can lead to some serialization of operations, and is thus not optimal, but is simple and robust.
- The worst case is illustrated here, but in practice things are not so bad.

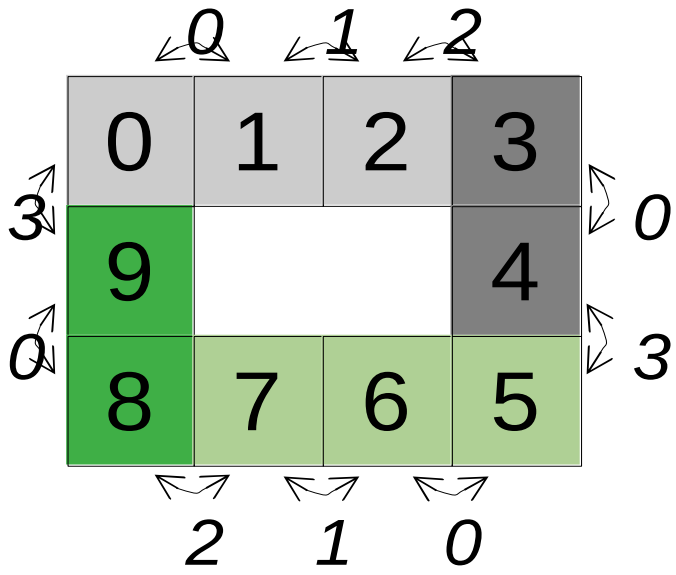


Communication Ordering Optimization

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.

On the figure, communication first occurs inside the gray and green groups; within each group, first within the light or dark group. We see that this reduces serialization.



Additional Features

When exchanging metadata for mesh location, PLE exchanges minimum and maximum, and preferred available algorithm versions.

- This should allow codes using different releases of PLE to agree on a compatible algorithm.
  - For example, when using the latest release for a given code and an older release for a coupled code, PLE will automatically switch to unoptimized ordering if it is not available in the older release.

Mesh location can be extended

- We can use this locate points not located on a mesh to “nearest” elements by extending the search with a larger tolerance, without increasing the search cost too much compared to using a larger tolerance for all points.
- PLE also allows querying which points are located or not, letting the user control the required behavior.

5

API Essentials

Callbacks and basic use

Callback Mechanism

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 values will be projected or interpolated.
  - Points are defined by an array of interleaved (double precision) coordinates, and if zones of a same mesh must be located relative to each other, an optional integer "exclusion tag"
- A "source", or donor opaque mesh object, whose structure is unknown to PLE.

PLE delegates all rank-local point location operations to the caller program. This is done by requiring that the calling code provide two functions:

1. A function returning the "extents" associated with local mesh elements.
   - The minimum required of this function is to compute the local mesh extents, but to provide for future higher performance algorithms, it could also return extents of individual or subsets of elements.
   - Extents of the point set to locate can be computed directly based on the point coordinates, so do not need to be considered here.
2. A function locating a given set of points on a local mesh.

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 locating points should match the `ple_mesh_elements_locate_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: $w_x.(1+tolerance_{fraction}) + 2.tolerance_{base}$

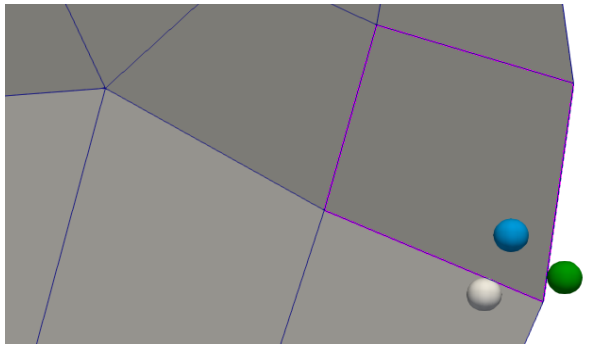
Location Tolerance

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 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, it is recommended to use a distance between 0 and 1 when a point is inside and element and a distance greater than 1 when a point is within tolerance but not quite inside an element.

- In the figure here:
  - The green point is inside no element, but close to one, so mapped to it if within the selected tolerance
  - The white point might be within tolerance and located on the nearby dark gray cell if tested first, but its distance to the containing (light grey) cell should be lower, so it will finally be assigned to that cell.



Interpolation and exchange

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 cell-centered Finite Volume tool such as code_saturne, a simple volume interpolation is to use a first-order Taylor expansion based on the cell center, value and gradient. For boundary variables, other interpolation schemes are possible.
- A code using shape functions or other quadratures should apply its favored method here.

Once values are assigned to the target points using the caller’s preferred interpolation scheme, they can be scattered to their target points on their originating MPI ranks using the collective `ple_locator_exchange_point_var` function.

- A “reverse” mode also allows passing extra information on the target points (in addition to their coordinates) to the donor elements.
- The exchange may be symmetric, when a domain has both donor and receiver zones.

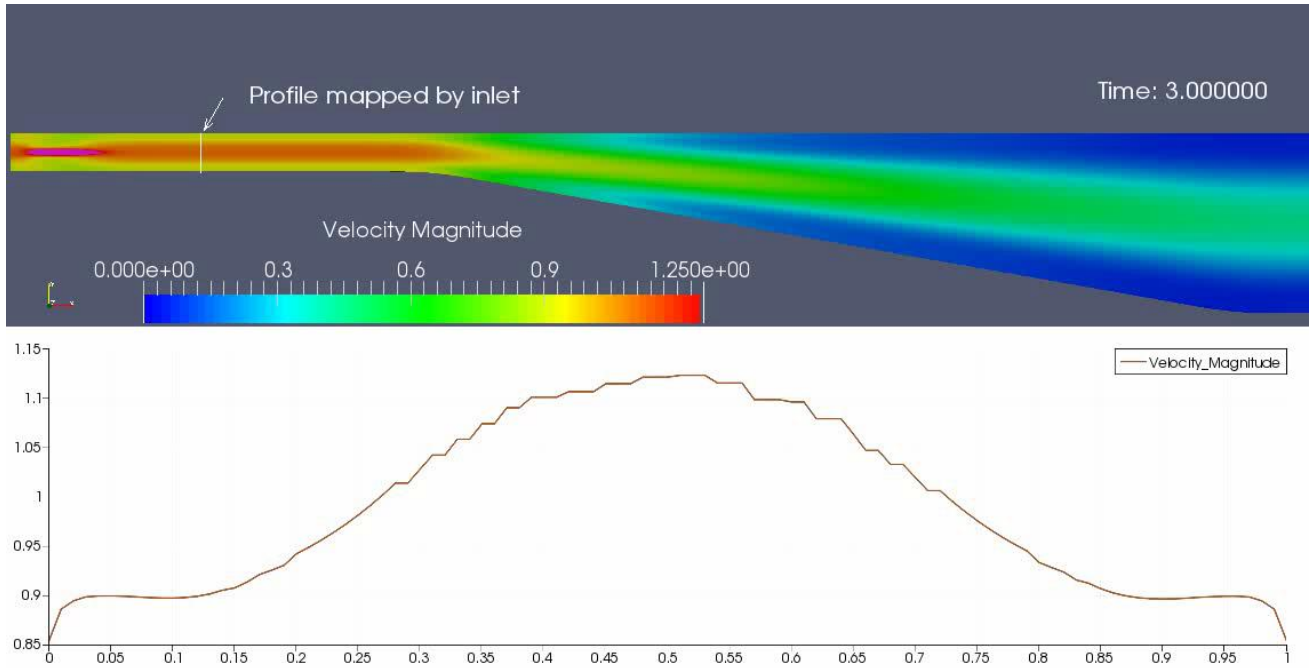
5

Applications

A few examples

Coupling a domain with itself

- Coupling with self allows “mapped inlet boundary conditions”
  - point values at inlet mapped to cells inside domain
  - allows good profiles even with short inlets
- Also used as an option for turbomachinery applications and for internal
- fluid/solid coupling

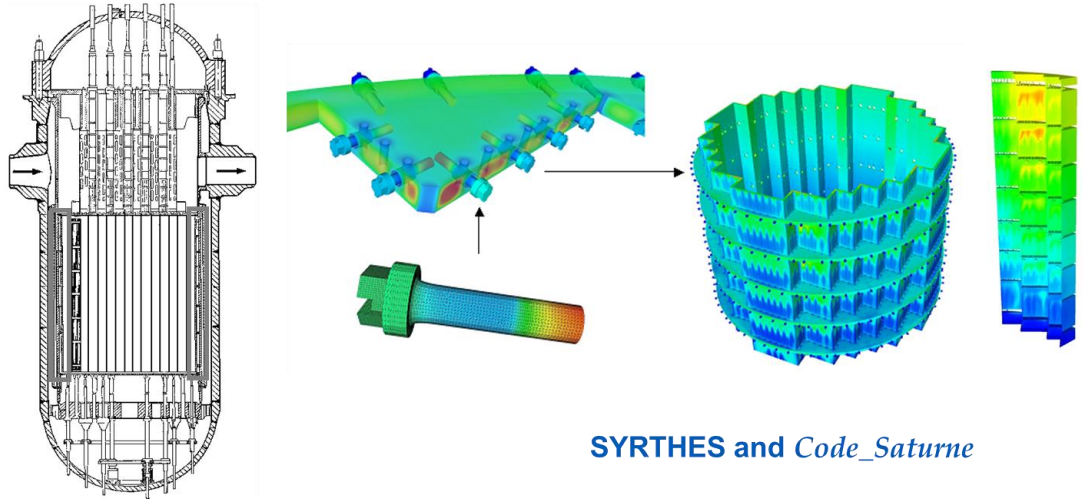


Conjugate Heat Transfer

Coupling between code_saturne or neptune_cfd and EDF's Syrthes thermal solver routinely used for large studies.

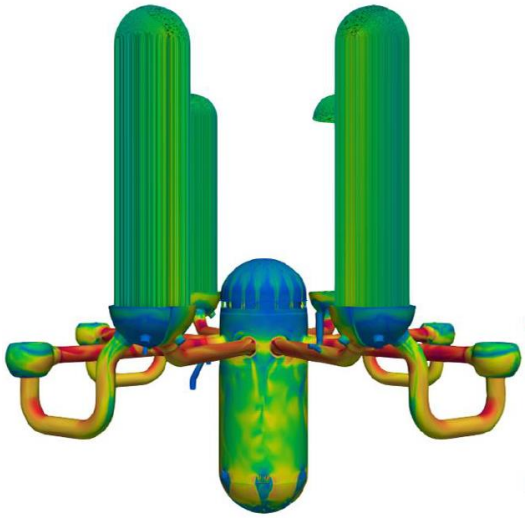
- Mostly surface coupling, though porous volume coupling possible also

1h30 on 2048 BG cores
1 billion tet mesh



SYRTHES and *Code_Saturne*

Industrial application:
IBLOCA scenario of a full primary circuit



6

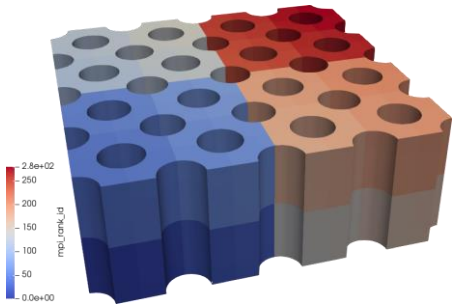
Performance Results

Current and prospective

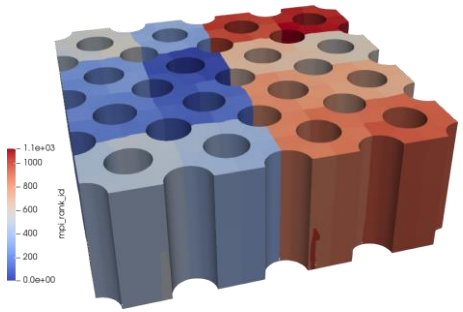
Performance Test Case

On a test case commonly used for weak scaling studies, representing a tube bundle.

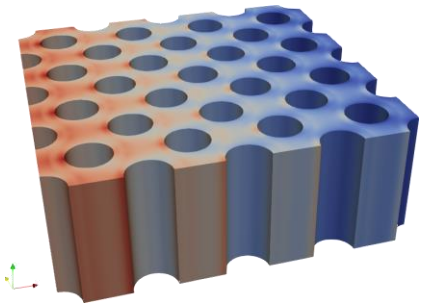
- Mesh is quite regular, with small variations in cell size and small aspect ratios, so performance measured should be better than on some more complex or irregular cases. Based on the number of tube repetitions, mesh sizes used include 12.4, 51, and 204 million cells.
- We force the location and interpolation used when restarting a computation on a different mesh, though the meshes are actually identical.
  - To ensure the case is not too simple, we use a different partitioning scheme for the computation and restart meshes, as shown here 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.



Morton SFC



PT-Scotch

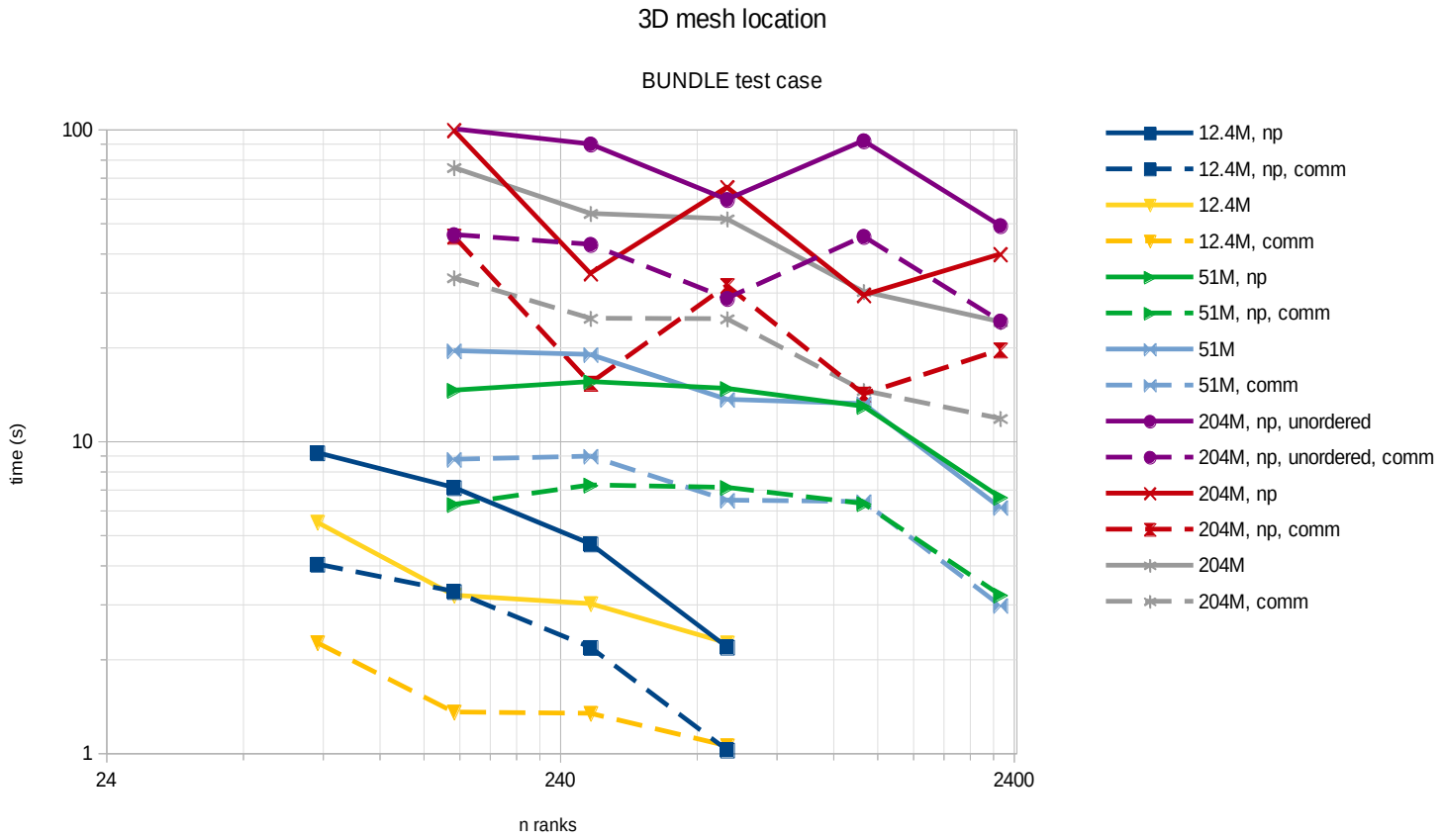


Pressure Field

Mesh Location Performance

We compare location results using different mesh sizes, rank counts, and with optimized or non-optimized ("unordered") communication order.

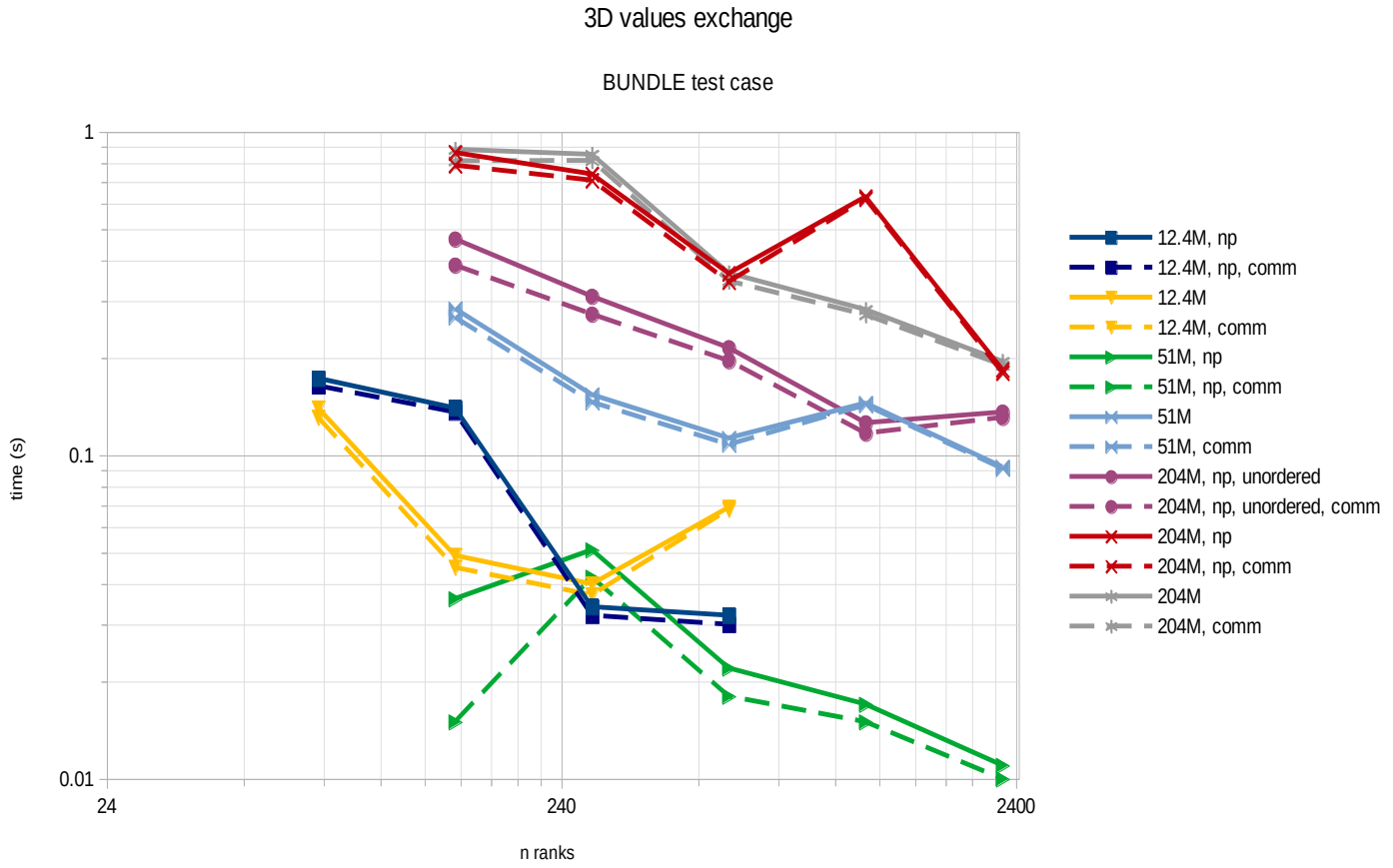
- The dashed lines represent communication times, the full lines total time.
- "np" means non-periodic, improving the partitioning a bit.
- We see that optimized communication ordering usually improves performance, though not spectacularly.
- Scaling is far from linear, but not so bad for such a simple algorithm.



Value Exchange Performance

Value exchange is much faster and scales better than mesh location

- When the maximum number of communicating ranks is below a chose threshold (100 by default), asynchronous MPI calls are used.
  - For safety, revert to blocking communication if too many neighbors.
- For the smaller case, performance degradation is not specific to the PLE part of code_saturne



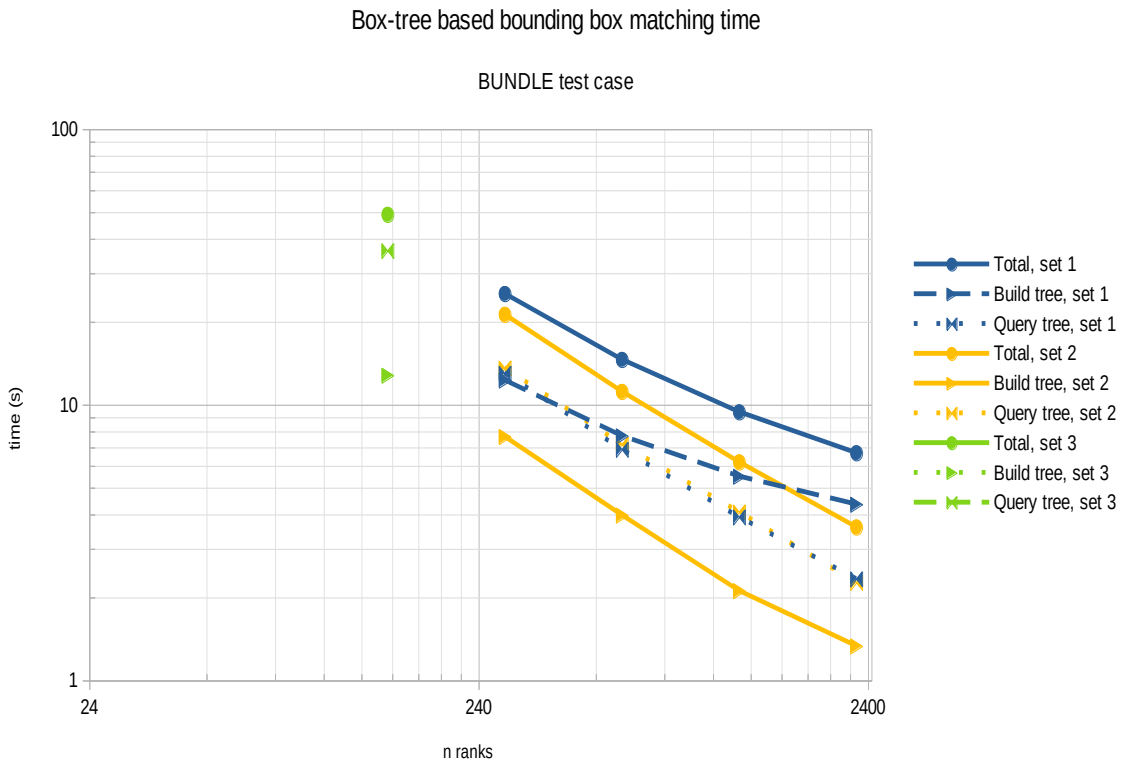
Alternative Location Evaluation: Scaling

A parallel “box-tree” is used outside of PLE
for code_saturne’s parallel mesh joining algorithm

- Similar to an octree, for bounding boxes instead of points.
- When boxes overlap multiple quadrants, they are copied to each quadrant, leading to additional memory usage.
- Built bottom-up, so associated data is always distributed, though it involves data movement from the mesh partition to the implicit octree partition
- A shallower tree is used to estimate load balance and repartition before building it to its full depth.

Using such a tree, the possible element point/element matching is done at a finer grained level, making it possible to run this step in a single pass.

To check how this would behave, we measured the performance of this box tree to locate neighbors for the 204 million cell case, using boxes 1/10 or the element bounds to represent points.



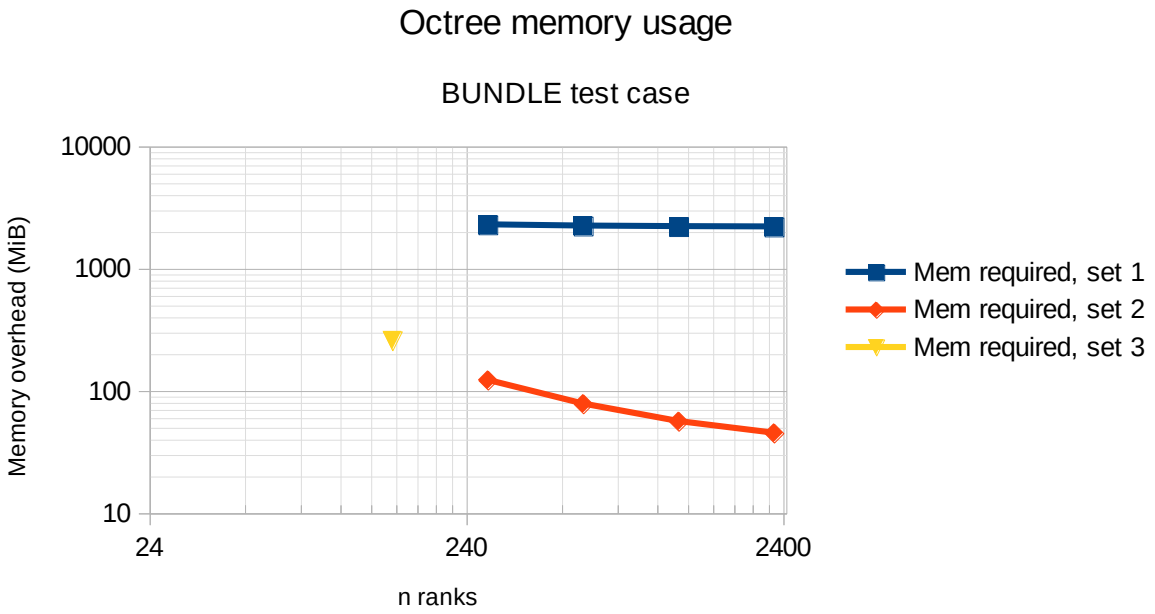
Alternative Location Evaluation: Memory Usage

The additional memory usage required for these steps is shown here. 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.

The box-tree would be a good candidate for an improved algorithm, but its must be tuned, as the de matching (with much less elements) lead to excessive memory usage.

- Only 4 parameters are needed to control this, but this still leads to a large search space.

Another alternative would be to use a Kd-tree type algorithm, which might allow simpler load balancing.



5

Final Remarks

PLE in today's environment

Final Remarks (Thanks for your Attention)

PLE's minimalism is both a weakness :

- Does not offer nearly as many optimized algorithms as modern alternatives such as DTK (Data Transfer Kit)
- No element intersection (useful for quantity-conserving interpolation) such as that of MEDCoupling.
- Not easily compatible with true Chimera-type meshes, as a point belongs to a single element layer.
- Requires providing a local search function.
- Reasonable scaling for 1000's of processes, probably not for 10 000's.

And a strength :

- Low memory usage
- Low induced maintenance (stable API, no dependency other than MPI)

PLE could constitute a good component in a more complete Exascale toolbox.