Chapter 1

Parallel solution of the Boussinesq convection problem

Part I of this document provides an overview of how to distribute the Boussinesq convection problem, using the single-domain and multi-domain approaches. It is part of a series of tutorials that discuss how to modify existing serial driver codes so that the Problem object can be distributed across multiple processors. Part II provides a more detailed discussion of the serial and parallel implementation of the various helper functions that may be used to set up multi-domain interactions.

1.1 Part I: Distributing the Boussinesq convection problem

For the single-domain discretisation of the Boussinesq convection problem, in which the Navier–Stokes and advection-diffusion equations are combined using a single element, the procedure required to distribute the problem is exactly the same as that described in the adaptive driven cavity tutorial. We therefore omit a detailed discussion of the changes to the driver code but encourage you to compare the serial driver codes,

demo_drivers/multi_physics/boussinesq_convection/multi_domain_boussinesq_convection.cc

and

demo_drivers/multi_physics/boussinesq_convection/multi_domain_ref_b_convection.cc

with their distributed counterparts,

demo_drivers/mpi/multi_domain/boussinesq_convection/multi_domain_boussinesq_convection.cc

and

demo_drivers/mpi/multi_domain/boussinesq_convection/multi_domain_ref_convection.cc.

The parallelisation of the multi-domain driver code is also quite straightforward. The key point is that the interaction between the domains must be set up again after the distribution of the problem because, on each processor, some of the "external elements" will have been deleted during the distribution. The required changes to the serial driver code are described below.

1.1.1 The main function

The main function begins and ends with the usual calls to `MPI_Helpers::init()` and `MPI_Helpers::finalize()`; the problem is distributed with a simple call to `Problem::distribute()`.

1.1.2 The problem class

The problem class is identical to its serial counterpart, apart from the addition of the `actions_after_distribute()` function, discussed below.

1.1.3 Actions after adaptation

The boundary conditions are such that a single pressure degree of freedom must be pinned after the adaptation. Once the problem has been distributed, pinning the first pressure freedom in the first element no longer works because the first element will be different on each processor. The required modifications mirror those in the adaptive driven cavity problem and ensure that the pressure is only pinned in the first element in the mesh by testing the Eulerian position of the element. The remainder of the function is the same as in the serial driver code.

```
/// Actions after adaptation, re-set all interactions, then
/// re-pin a single pressure degree of freedom
void actions_after_adapt()
 {
  //Unpin all the pressures in NST mesh to avoid pinning two pressures
  RefineableNavierStokesEquations<2>::
   unpin_all_pressure_dofs(nst_mesh_pt()->element_pt());

  //Pin the zero-th pressure dof in the zero-th element and set
  // its value to zero
  unsigned nnod=nst_mesh_pt()->nnode();
  for (unsigned j=0; j<nnod; j++)
   {
    if (mesh_pt()->node_pt(j)->x(0)==0.0 &&
        mesh_pt()->node_pt(j)->x(1)==0.0) // 2d problem only
     {
      fix_pressure(0,0,0.0);
      break;
     }
   }

  // Set external elements for the multi-domain solution.
  Multi_domain_functions::
   setup_multi_domain_interactions<NST_ELEMENT,AD_ELEMENT>
   (this,nst_mesh_pt(),adv_diff_mesh_pt());
 } //end_of_actions_after_adapt
```

1.1.4 Actions after distribution

After the problem distribution, the multi-domain interactions must be set up again because some of the "external elements" may now only exist on other processors. Such elements are re-created locally as "external halo elements" when `Multi_domain_functions::setup_multi_domain_interactions(...)` is called.

```
/// Actions after distribute: Re-setup multi-domain interaction
void actions_after_distribute()
 {
  // Re-set all the interactions in the problem
  Multi_domain_functions::setup_multi_domain_interactions
   <NST_ELEMENT,AD_ELEMENT>(this,nst_mesh_pt(),adv_diff_mesh_pt());
 }
```

1.1.5 The doc_solution() routine

As usual, the `doc_solution()` function is modified by adding the processor ID to each output file, ensuring that output from one processor does not overwrite that from another.

The remainder of the driver code is identical to the serial version.

1.2 Part II: Setting up multi-domain interactions

The namespace `Multi_domain_functions` provides several helper functions that facilitate the location of "external elements" in multi-domain problems, in which the two interacting domains occupy the same physical space. The namespace also provides functions that can be used for problems in which the interacting domains meet at a lower-dimensional interface, *e.g.* in fluid-structure interaction problems. The functions have a sensible default behaviour and can be used as "block-box" routines. We provide a brief overview of their (serial and parallel)

implementation, providing enough detail to allow users to appreciate the role of the parameters that can be adjusted in order to optimise the functions' performance in specific applications.

The function

```
template<class ELEMENT_0,class ELEMENT_1>
void Multi_domain_functions::setup_multi_domain_interactions(
                                    Problem* problem_pt,
                                    Mesh* const &first_mesh_pt,
                                    Mesh* const &second_mesh_pt);
```

sets up a two-way interaction between two domains of the same spatial dimension represented by `first_mesh_pt`, a mesh of elements of type `ELEMENT_0`, and `second_mesh_pt`, a mesh of elements of type `ELEMENT_1`. Both `ELEMENT_0` and `ELEMENT_1` must inherit from `ElementWithExternalElements`. The function loops over the integration points of all the elements in `first_mesh_pt` and establishes which "external element" in `second_mesh_pt` covers the same spatial position as the integration point. A pointer to this "external element" and the appropriate local coordinate are stored in the element in the `first_mesh_pt`, using the storage provided by the `ElementWithExternalElement` base class. Once the "external elements" have been found for all elements in `first_mesh_pt`, the procedure is repeated with the roles of the two meshes interchanged.

The corresponding function

```
template<class EXT_ELEMENT>
void Multi_domain_functions::setup_multi_domain_interaction(
                                    Problem* problem_pt,
                                    Mesh* const &mesh_pt,
                                    Mesh* const &external_mesh_pt);
```

(note the singular) sets up a one-way interaction in which the mesh pointed to by `external_mesh_pt` provides the "external elements" (of type `EXT_ELEMENT`) for the `ElementWithExternalElements` stored in `mesh_pt`, but not vice versa. In fact, the two-way interaction described above is performed by two successive calls to this function

```
setup_multi_domain_interaction<ELEMENT_1>
     (problem_pt,first_mesh_pt,second_mesh_pt);
setup_multi_domain_interaction<ELEMENT_0>
     (problem_pt,second_mesh_pt,first_mesh_pt);
```

Finally, the function

```
template<class EXT_ELEMENT, class FACE_ELEMENT_GEOM_OBJECT>
void Multi_domain_functions:: setup_multi_domain_interaction(
                                    Problem* problem_pt,
                                    Mesh* const &mesh_pt,
                                    Mesh* const &external_mesh_pt,
                                    Mesh* const &external_face_mesh_pt);
```

may be used to set up multi-domain interactions for problems in which the interaction occurs across the boundaries of adjacent domains (*e.g.* in FSI problems where the fluid and solid domains meet along a lower-dimensional interface). We do not anticipate that users will have to call this function directly; it is called internally by the function `FSI_functions::setup_fluid_load_info_for_solid_elements(...)`.

1.2.1 Overview of the implementation

The setup of multi-domain interactions requires the identification of local coordinates within "external elements" that occupy the same Eulerian position as the integration points of the `ElementWithExternalElements` with which they interact. Initially, the helper functions convert the mesh containing the "external elements" into a `MeshAsGeomObject` — a compound `GeomObject` in which the constituent finite elements act as sub-`GeomObjects`. Given the Eulerian coordinates of a "target point", the function `MeshAsGeomObject::locate_zeta(...)` is used to locate the "external element" (and the local coordinate within it) that contains the "target point". The simplest implementation of this function is to loop over all the elements in the mesh, calling `FiniteElement::locate_zeta(...)` until the required point is located. The function `FiniteElement::locate_zeta(...)` can be overloaded for specific elements to take into account the interpolation method used within the element. By default, a Newton method is employed to determine the local coordinates of the "target point" within an element. If the Newton method fails to converge or if the "target point" is found at a local coordinate that is outside the element, then the functions return values are set to indicate that the "target point" has not been found within the element.

One problem with the naive "loop over all the elements" approach is that in most cases, the candidate element will not contain the "target point", so there will be a lot of wasted work. In addition, there is no guarantee that the (default) Newton iteration will converge, even when the element does contain the required point. A further complication in a distributed problem is that the element that contains the "target point" may not be located on the current processor. For these reasons, `MeshAsGeomObject::locate_zeta(...)` uses a bin-based search procedure described below. Conceptually, the algorithm still loops over all elements until it finds the one containing the "target point"; the efficiency gains are achieved by choosing a sensible search order so that the element containing the point is found quickly.

1.2.2 Basic (serial) implementation

To make the search process efficient, the constructor of the `MeshAsGeomObject` automatically creates a bin structure that aids the identification of elements that are close to a given "target point". The setup of the bin structure is performed as follows:

Setting up the bin structure

1. We start by sampling the position of the elements in the `MeshAsGeomObject` to determine the mesh's maximum and minimum Eulerian coordinates.

2. Next we create a rectangular (or cubic) bin structure that spans the entire mesh (using the extremal coordinates determined in the previous step, plus a small margin for safety), using `Nx_bin` $\times$ `Ny_bin` ($\times$ `Nz_bin`) equal-sized rectangular (or cubic) bins.

3. Finally, we populate the bin structure by evaluating the position of a number of sampling points within each element. For each sampling point we determine in which bin the point is located and associate a pair, comprising the pointer to the element and appropriate local coordinate, with that bin.

Once the bin structure has been set up, a given "target point" can be located very quickly within the `MeshAs↩GeomObject` by the following procedure:

Locating points within the `MeshAsGeomObject`

1. Given the coordinates of the "target point", we identify the bin in which it is located. This is a cheap operation because all bins have the same size and are aligned with the coordinate axes.

2. Next we visit the element/local coordinate pairs associated with that bin. For each pair, we use the `Finite↩Element::locate_zeta(...)` function to (attempt to!) find the the local coordinate within the element that corresponds to the "target point". The local coordinate stored in the pair is passed to the function to be used as the initial guess for the (default) Newton iteration. If the "target point" is not found, the procedure is repeated with next pair of element and local coordinates.

3. If the relevant point is not located within the initial bin (*e.g.* because the bin is empty) the procedure is repeated in adjacent bins, spiralling outwards through the bin structure.

4. If the "target point" cannot be found in any of the bins, the search fails, indicating/suggesting that the "target point" is not contained in the mesh. (The search may also fail when the Newton method is used if none of the element/local coordinate pairs associated with the bins were close enough to the "target point". If this happens, we recommend increasing the number of sampling points by increasing the value of `Multi_↩domain_functions::Nsample_points` before re-running the code. Note, however, that we have never encountered this problem in any of our test cases.)

A tacit assumption in the above procedure is that, since the two interacting meshes represent the same physical domain, any point in one mesh can also be found in the other, regardless of possible differences in the meshes' refinement patterns. While this is true for meshes that discretise domains with polygonal boundaries, problems may arise in domains whose boundaries are curvilinear because `oomph-lib's MacroElement/Domain`-based representation of such domains ensures that during successive mesh refinements, any newly-created nodes are placed exactly on the curvilinear boundary. This procedure is necessary to guarantee convergence to the exact solution under mesh refinement, but it creates the problem that a strongly refined mesh (whose boundaries provide a

better approximation to the exact curvilinear boundary) may contain points that are not contained within the coarser mesh with which it interacts. To avoid this problem, we determine the location of "target points" within each element using the `MacroElement` (exact curvilinear) representation, if there is one. An important consequence of this approach is that points in the two interacting meshes deemed to be located at the same spatial position (same position in the exact representation) may actually have slightly different positions. Nonetheless, the difference between their positions decreases under mesh refinement at the same rate at which the finite-element representation of the curvilinear domain approaches the domain itself.

1.2.3 Parallel implementation

When setting up multi-domain interactions for distributed meshes we face the additional challenge that the "external element" that contains a "target point" may not be located on the same processor as the `ElementWith↩ ExternalElement` with which it interacts. When dealing with distributed meshes, the search procedure described above is therefore modified as follows:

1. When creating the `MeshAsGeomObject` representation of the distributed mesh that contains the "external elements", each processor sets up a bin structure for its own part of that mesh. Some communication takes place at this point so that every processor holds the "global" extrema of the mesh.

2. When setting up the multi-domain interaction, each processor only determines the "external elements" for the locally-stored `ElementWithExternalElements` Initially, the required "external elements" are sought among the locally-stored "external elements" and the search is restricted to the bin that contains the "target point". Each processor then creates a list of the "external elements" that have not been found locally.

3. Each processor communicates its list of missing "external elements" to the "next" processor, using a ring-like communication pattern.

4. All processors search for the yet-to-be-found "external elements" amongst the "external elements" stored in their part of the distributed mesh. Again the search is restricted to the "external elements" that are associated with the bin that contains the "target point". If an "external element" is found, a halo copy of it is created on the processor that contains the `ElementWithExternalElement`.

5. The list of any remaining missing "external elements" is then forwarded to the next processor in the communication ring where the search process is repeated.

6. If, after a complete sweep through all processors, some "external elements" have still not been located, we initiate another search loop, re-commencing the search at the next level of the search spiral through the bins.

7. The setup of the multi-domain interaction is complete when all "external elements" have been located and halo copies have been made where required. The setup fails (for the same possible reasons listed in the discussion of the serial implementation) if none of the processors are able to locate one or more of the required "external elements".

The algorithm is based on the assumption that the majority of the pairs of interacting elements will be stored on the same processor, which is typically the case when `METIS` is used to determine the distribution. The algorithm also assumes that the external elements can usually be found by searching only in the bin containing the "target point", which is true provided the bin structure is not too fine. The algorithm should always find the external elements even if these assumptions are not satisfied, but it may not be optimal in these situations.

1.3 Source files for this tutorial

The full parallel driver code for the distributed, multi-domain based solution of the Boussinesq convection problem can be found at

demo_drivers/mpi/multi_domain/boussinesq_convection/multi_domain_ref_b_↩ convection.cc

Additional parallel driver codes for the distributed Boussinesq problem using both single- and multi-domain methods are located in

demo_drivers/mpi/multi_domain/boussinesq_convection

1.4 PDF file

A pdf version of this document is available. \