MASTER IN HIGH PERFORMANCE COMPUTING

Core Building Blocks for Massively Parellel Multi-Physics Applications

Supervisor:
Luca HELTAI

Candidate:
Giovanni ALZETTA

3rd EDITION
2016–2017

Contents

1 Multiphysics Coupling **3**
1.1 Generalized Coupling 4
1.2 Existing Solutions 8

2 Serial Case **10**

3 Parallel Case **13**
3.1 Bounding Boxes 15

4 Benchmarks **20**
4.1 Serial Baseline 21
4.2 Improvements for the Serial Code 22
4.3 Parallel Results 30

5 Conclusions **33**

1 Multiphysics Coupling

Many scientists have already adventured into making far better introduction than I possibly could. I shall only quote three articles ([11], [4], and [16] respectively):

- David E Keyes, Lois C McInnes, Carol Woodward, William Gropp, Eric Myra, Michael Pernice, John Bell, Jed Brown, Alain Clo, Jeffrey Connors, et al. Multiphysics simulations: Challenges and opportunities. *The International Journal of High Performance Computing Applications*, 27(1):4–83, 2013

A multiphysics system consists of more than one component governed by its own principle(s) for evolution or equilibrium, typically conservation or constitutive laws. A major classification in such systems is whether the coupling occurs in the bulk (e.g., through source terms or constitutive relations that are active in the overlapping domains of the individual components) or whether it occurs over an idealized interface that is lower dimensional or a narrow buffer zone (e.g., through boundary conditions that transmit fluxes, pressures, or displace- ments). Typical examples of bulk-coupled multiphysics systems with their own extensively developed literature include radiation with hydrodynamics in astrophysics (radiation-hydrodynamics, or "rad-hydro"), electricity and magnetism with hydrodynamics in plasma physics (magnetohydrodynamics), and chemical reaction with trans- port in combustion or subsurface flows (reactive transport). Typical examples of interface-coupled multiphysics systems are ocean-atmosphere dynamics in geophysics, fluid-structure dynamics in aeroelasticity, and core-edge coupling in tokamaks. Beyond these classic multiphysics systems are many others that share important structural features.

- Wolfgang Bangerth, Carsten Burstedde, Timo Heister, and Martin Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Trans. Math. Softw.*, 38(2):14:1–14:28, January 2012

Computer clusters with tens of thousands and more processor cores are becoming more and more common and are going to form the backbone of most scientific computing for the currently foreseeable future. When using thousands of processors in parallel, two basic tenets need to be followed in algorithm and data structure design: (i) no sizable amount of data can be replicated across all processors, and (ii) all-to-all communications between

processors have to be avoided in favor of point-to-point communication where at all possible. These two points will inform to a large part what can and cannot work as we scale finite element computations to larger and larger processor counts. For example, even if it would make many operations simpler, no processor will be able to hold all of the possibly billions of cells contained in the global mesh in its local memory, or even be able to compute a threshold for which cells exceed a certain error indicator and should therefore be refined.

- S Slattery, P Wilson, and RP Pawlowski. The data transfer kit: a geometric rendezvous-based tool for multiphysics data transfer

For massively parallel simulations, it is typical that geometric domains not only do not conform spatially, but also that their parallel decompositions do not correlate and are independent of one another due to physics-based partitioning and discretization requirements.

Data distribution makes it possible to fit the simulation in the system, but it inherently complicates the process of coupling in both bulk-coupled and interface-coupled problems, making it necessary to resort to complex communication patterns and strategies. After formalizing a general coupling problem and showing other interesting study-cases for mesh-coupling, we report on the strategies found in literature. We propose solutions for a serial approach to the problem, and show how they fall short in a distributed setting, making it necessary to develop new strategies. A distributed algorithm is proposed and analyzed, and possible future developments are discussed.

1.1 Generalized Coupling

The algorithms reported in this work are quite generic, and can be applied to a number of problems in finite elements, finite volumes or finite differences, whenever a form of coupling is required.

Multi-physics problems require the exchange of information between two (or more) problems, defined on possibly different domains, either overlapping, both partially or totally, or neighbouring.

To settle the ideas, consider two physical problems defined on (possibly different) domains $A, B \subseteq \mathbb{R}^n$. A generic definition of coupling between fields defined on A and fields defined on B can be regarded as a bi-linear operator between pairs of elements in the respective spaces, i.e., assuming that $V(A)$ and $V(B)$ are some functional spaces on A and B respectively, the coupling

$$C : V(A) \times W(B) \mapsto \mathbb{R}, \tag{1}$$

is a bi-linear operator.

Among all possible couplings, we restrict our study to a specific class of couplings, depending on a coupling *kernel* K, i.e., we assume that $C(v, w)$ takes the form

$$C(v, w) := \int_A \int_B v(x) K(x, y) w(y) \mathrm{d}A_x \, \mathrm{d}B_y, \tag{2}$$

and that $V(A)$ and $W(B)$ are some standard functional spaces where the above integral makes sense.

As an example, lets consider the case where $B \subset A$, and fix $V(A)$ to be the space of continuous functions on A, i.e., $V(A) := C^0(A)$, and $W(B)$ the space of distributions on B, i.e., $W(B) := \mathcal{D}(B)$, and let $K(x, y) := \delta(x - y)$ where δ is the n-dimensional Dirac delta distribution defined by its action on continuous functions as

$$v(x) := \int_{\mathbb{R}^n} \delta(x - y) v(y) \mathrm{d}y, \qquad \forall v \in C^0(\mathbb{R}^n). \tag{3}$$

In this very particular case, if we pick a subset of points y_i contained in the domain $B \subset A$, and associate a Dirac delta to each of these points as an element of $W(B)$, say

$$w_i(y) := \delta(y - y_i), \tag{4}$$

the coupling defined above between an arbitrary function $v \in V(A)$ and w_i reduces simply to its point evaluation on y_i, i.e.:

$$C(v, w_i) := \int_A \int_B v(x) K(x, y) w_i(y) \mathrm{d}A_x \, \mathrm{d}B_y = \int_B v(y) w_i(y) \mathrm{d}B_y = v(y_i), \tag{5}$$

where the Kernel K removes the first integral in $\mathrm{d}A_x$ by the definition in (3), and the definition of the distributions w_i remove the second integral in $\mathrm{d}B_y$ using the same principle.

When the spaces and the Kernel are chosen as above, the coupling is simply called *interpolation* on the points y_i of the function v. The typical application of this *interpolation* operator is when the field $V(A)$ represents some physical quantity, for example a continuous temperature field, and y_i are the vertices of an embedded triangulation, not aligned with A, where we would like to evaluate the temperature.

Different definitions of the spaces $V(A)$ and $W(B)$, and of the coupling kernel K lead to different coupling operators. In this work we will restrict our examples to the *interpolation* coupling above, since most of the other couplings can be reinterpreted in terms of the above one.

In general, the domains A and B are discretised using (possibly independent) triangulations, and are split on non-intersecting *cells*.

When physical fields are defined on one of these triangulations, say for example on A, they use the seperation on M cells to restrict the possible

choice of functions to a *finite dimensional space* of dimension N, constructed using a linear combination of some *basis functions*, defined through the triangulation itself.

In general this construction is done in four steps:

- triangulate the domain A into a collection of M cells $A_h := \cup_{i=1}^{M}(K_i = F_i(\hat{K}))$, images under M (possibly non linear) iso-morphism of a referential cell $\hat{K}$;
- define a set of *nloc* basis functions on the referential element $\hat{K}$, $\{\hat{v}_i\}_{i=1}^{nloc}$;
- define some global basis functions on A_h, as the push forward of $\hat{v}_j$ under F_i on $K = F_i(\hat{K})$, i.e., $v_l(F_i(\hat{x}))|_K = \hat{v}_j(\hat{x})$, where l is an appropriate global numbering depending on j and i
- enumerate the global basis functions so that we have $V_h(A) \subset V(A)$ and $V_h(A) := \text{span}\{v_i\}_{i=1}^{N}$.

The construction above guarantees that any function $v_h \in V_h(A_h)$ can be expressed as

$$v_h(x) := \sum_{i=1}^{N} v^i v_i(x), \tag{6}$$

where the functions $v_i(x)$ are different from zero only on a limited number of cells K of A_h, namely $\text{supp}(v_i) := \{K \in A_h \text{ s.t. } v_i|_K \neq 0\}$. Notice that, on any K, only *nloc* global basis functions are different from zero.

For the space $V_h(A_h)$, the *interpolation* coupling on a collection of points $y_\alpha \in B$ can be expressed by the interpolation matrix C:

$$C_{\alpha i} := C(v_i, w_\alpha) := v_i(y_\alpha), \tag{7}$$

such that, when it is multiplied with the vector of coefficients v^i of a generic function v_h, we obtain the *interpolation* of the function v_h on the points y_α:

$$\sum_{i=1}^{N} C_{\alpha i} v^i = C(v_h, w_\alpha) := \sum_{i=1}^{N} v^i v_i(y_\alpha) = v_h(y_\alpha). \tag{8}$$

Numerically only $\hat{v}_i$, F_i and F_i^{-1} can be computed directly; thus the numerical solution to equation 8 becomes:

$$v_i(y_\alpha) = \hat{v}_i\left(F_i^{-1}(y_\alpha)\right). \tag{9}$$

This translates in the following algorithm, which is illustrated in figure 1

i. Use the triangulation of B to identify y_α in the real space.

ii. Find in which cell K_i of A y_α lies.

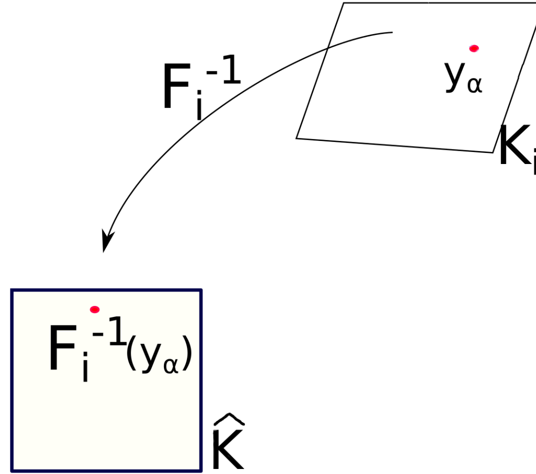


Figure 1: Interpolation Problem

iii. Use the triangulation of A to obtain F_i^{-1} and compute $F_i^{-1}(y_\alpha)$.

iv. Evaluate the result on the basis functions v_i.

DAG Structure The structure of modern general purpose numerical libraries is organized to offer an abstraction to the user while optimizing the object structure and performance. This nowadays can achieved organizing the library following a dataflow graph; the central idea being the use of a Directed Acyclic Graph (DAG) as structure of the underlying software [6].

A DAG is a directed graph with topological ordering: each node structurally represents an object and one or more directed edges represent how it can be used to generate new objects or data. This structure is efficient and has many benefits for a parallel code, but creates "asymmetries" in the operations which can be done with the code, favouring one direction while making the inverse operation slower.

This means, for instance, it is fast in deal.II find the vertices of a cell while it is inefficient, given a vertex, to find its neighbouring cells. This can have huge impact on the code's performance and the only possible solution is to build, with a new function, an edge with the inverse direction in the DAG and store it with a caching mechanism. If the library used is well structured it should be seldom necessary to pay this overhead, but for the particular case of the vertex point relation it proved important for our code.

1.2 Existing Solutions

Coupling is difficult, but often necessary; a number of related coupling problems is already studied and known in literature; here we shall briefly sketch the various solutions used.

To fix the terminology: in a *distributed mesh* each process [1] "owns" a number of cells, of which it knowns all the variables and on which it runs computations. We shall call these cells *locally owned*. Often each process maintains also the information of one or more layers of cells that belong to another process, often termed *ghost cells*. Each process knowns all variables relative to ghost cells but can't modify their value. We assume that the rest of the triangulation is unknown and any cell laying on the area we do not know is called an *artificial cell*, in the sense that its existence is purely instrumental to the chosen data structure, but has no participation in the solution of the actual physical problem

1.2.1 Particle-in-Cell Methods

The particle-in-cell (PIC) method are used to solve a certain class of partial differential equations(PDEs); the technique consist in tracking individual particles, which have a position and a number of individual properties, in a Lagrangian frame; at the same time moments of a distribution such as densities are computed on a stationary mesh. PIC methods have a long history, which sparked with applications in plasma physics [13].

Only recently PIC have been implemented in a fully distributed setting; Rene Gassmoeller et al. published a paper [9] describing how to implement PIC methods in a state-of-the art fluid dynamic solver and implemented it for the ASPECT library [10] [2].

The problem here described has similar aspects to the ones studied by Gassmoeller and, in fact, we benefited greatly from their work; but there are some important differences.

In a PIC method each process owns a number of cells and, following some algorithm (for example randomly, or following particular patterns) generates particles inside it. After generating the particles each process owns both cells and the particles inside it; when working with meshes this is no longer the case (see next section, 1.2.2).

The method described by Gassmoeller particle's speed is, because of the simulation parameters, small enough that if a particle exists the locally owned part of the mesh, it will most likely end up on a ghost cell i.e. the owner shall be likely known if the particle exists the locally owned part of the mesh. At the same time, if a particle ends up on an artificial cell, it can

[1] Since this work is algorithmic, we use "process" as a generic term not making distinctions between processors, processor cores or MPI processes

be deleted. This is possible because, as reported in [9] the cases in which a particle is removed are small with numerical effects.

While the strategy of communicating points lying on ghost cells clearly adapts to our cases, in a general coupling problem eliminating a point evaluation is not an option. In general coupling problems we often have little control on the topological distribution of the part of the domain B over A, and we may have significant portions of B that we own, laying on part of A that we do not own.

Developing a strategy to communicate points in all possible settings is thus fundamental; this would add the possibility to to change the meshes' distribution arbitrarily during simulation, allowing to re-initialize the mesh distribution even at running time, with a different number of processes and/or a different distribution scheme.

1.2.2 Data Transfer Kit

Many modern physical simulations make use of a partitioned approach: different kernels handling different parts of the problem e.g. in a FSI simulation, one handling fluid equations and one simulating the structure. In this scenario accuracy and speed at data transfer is fundamental.

Another issue when working with multiple meshes is the initial distribution: because of different computational and physical needs it is complicated to create a balanced configuration where meshes' parts which needs to be coupled are locally owned by the same processes. To tackle this sort of problems multi-constraint partitioning techniques [15] are one possibility. Another consists in rendezvous algorithms, developed by Plimpton et. al. [12], which allows to generate balanced meshes, at the cost of communication. These have been implemented in the Data Transfer Library (DTK) [17].

This is how the algorithm is briefly described in [16]:

> for mesh-based data transfer generates the rendezvous decomposition which behaves as a hierarchical parallel and geometric search tree. Using this algorithm, a secondary decomposition of a subset of the source mesh that will participate in data transfer is generated, forming the rendezvous decomposition as described in the example above. The rendezvous decomposition is encapsulated as a separate entity from the original geometric description of the domain. It can be viewed as a temporary copy of the source mesh subset that intersects the target geometry. With the rendezvous decomposition, we effectively have a search structure that spans both parallel and physical space.

This "search structure that spans both parallel and physical space" is the missing step in the solution of our interpolation problem which has is not

present in the PIC solution (section 1.2.1). The structure is implemented in DTK using trees of bounding boxes which are needed for the initial partition. In this work we shall create a similar structure at any time deemed necessary and without constraints on the methods used for mesh partitioning.

1.2.3 deal.II Utilities

The coding part of this project was made using and developing the deal.II library [1] [3], which is a modern example of *state of the art* numerical library; as stated in the deal.II website (http://www.dealii.org/):

> deal.II is a C++ program library targeted at the computational solution of partial differential equations using adaptive finite elements. It uses state-of-the-art programming techniques to offer a modern interface to the complex data structures and algorithms required.
>
> The main aim of deal.II is to enable rapid development of modern finite element codes, using among other aspects adaptive meshes and a wide array of tools classes often used in finite element program. Writing such programs is a non-trivial task, and successful programs tend to become very large and complex. We believe that this is best done using a program library that takes care of the details of grid handling and refinement, handling of degrees of freedom, input of meshes and output of results in graphics formats, and the like. Likewise, support for several space dimensions at once is included in a way such that programs can be written independent of the space dimension without unreasonable penalties on run-time and memory consumption.

In particular deal.II contains a class, **FEFieldFunction**, which has interesting interpolating capabilities and can solve the coupling problem proposed in section 1.1; though it currently suffers from limitations on parallel distributed triangulation i.e. if a point isn't found inside a locally owned or ghost cell an exception is thrown: while with deal.II the coupling problem 6 can be easily solved in a serial setting, in a distributed one it simply isn't currently possible without the code we implemented.

2 Serial Case

Before attempting a distributed solution of equation 9, the serial case must be solved; in figure 2 we can see a generic coupling example in two dimensions: two meshes intersect and the point $y_i \in B$ needs to be known by A.

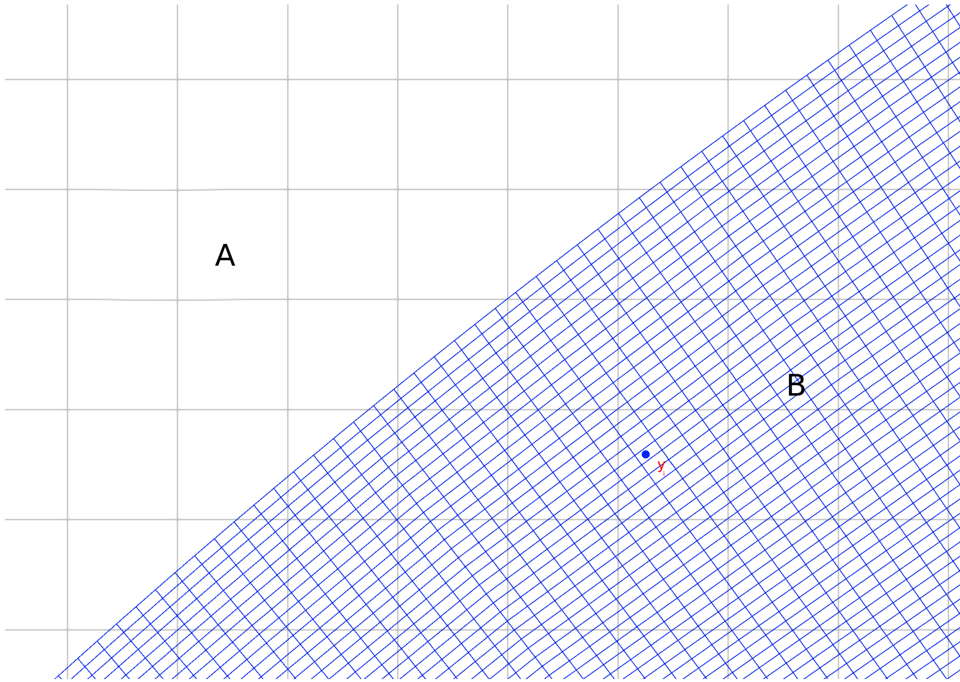


Figure 2: Coupling of two meshes

2.0.1 Compute Point Locations

Recall, from iv, that the main problem in solving 9 are the following steps:

- Find in which cell K_i of A y_α lies.
- Use the triangulation of A to obtain F_i^{-1} and compute $F_i^{-1}(y_\alpha)$.

In a fully serial environment all the variables needed are present and, in the deal.II library, these are solved by a function called *compute point*

locations which is implemented in the following manner:

Input: A triangulation, a list of points

Output: cells containing points and their transformed

Initialization:
$p =$ first point of the list;
Find the cell K surrounding p and compute $q = F_K^{-1}(p)$;
set p as found;
while *there are points not found* **do**
  for p *in points left* **do**
    compute $q = F_K^{-1}(p)$;
    if q *inside unit cell* **then**
      add p and q to the cell's list;
      set p as found;
    end
  end
  if *there are points not found* **then**
    $p =$ first point not found;
    Find the cell K surrounding p and compute $q = F_K^{-1}(p)$;
    set p as found;
  end
end

Algorithm 1: Compute Point Locations in deal.II 8.5.1

The following should be noted on algorithm 1:

- If points are pre-ordered by cell the computation is faster[2]; this assumption is often satisfied when working with meshes and their vertices.
- Function F_K^{-1} has a high cost, but it's always used to check if the points are inside K.
- Finding the cell surrounding p is achieved by another function, *find active cell around point*, which has a high computational cost as explained in 1.1.

This is how *find active cell around point* is currently implemented[3]

[2]Notice that, if the cells are ordered by cell, at every step the number of points on which the loop runs decreases by the amount of points inside the last cell

[3]in deal.II 8.5.1.

Input: A triangulation, a point p

Output: cell containing the point and it's transformed

initialize the vector **distances**;
for *v in triangulation's vector* **do**
  compute the distance $||p - v||$;
  add it to **distances**;
end
Find the minimum in **distances**;
Save the corresponding vector v;
Initialize the cell's vector **neighbours**;
for *K in triangulation's cell* **do**
  if *v is a vertex of K* **then**
    Add K to **neighbours**;
  end
end
for *K in **neighbours*** **do**
  compute $q = F_K^{-1}(p)$;
  if *q inside unit cell* **then**
    return q and K;
  end
end

Algorithm 2: Find active cell around point in deal.II 8.5.1

Two main problems affect algorithm 2:

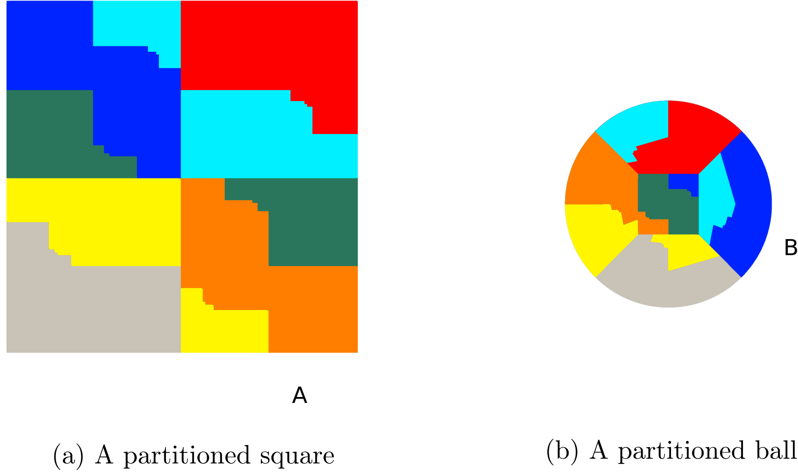
- Because of deal.II's DAG structure relation from *vertex* to *neighbour cells* has a high cost.
- $F_K^{-1}(p)$ has a high cost and no attempt is made to use a simpler method to guess which cell is most likely to be the one containing p.

As shown in section 4.1.1 this results in poor performance.

3 Parallel Case

In addition to what stated in section 1.2, we shall make the following general assumptions on distributed meshes (as reported in [4]):

- *Common coarse mesh*: all cells are derived by refinement from a common coarse mesh which needs to capture the topology of the computational domain. Each cell is hierarchically refined into four (2d) or 8 (3d) children which may be further refined, forming a forest.



(a) A partitioned square

(b) A partitioned ball

Figure 3: Partitioned meshes

- *Distributed storage*: Each processor in a parallel program may only store a part of the entire forest, the *locally owned* part.

Figure 3 shows an example obtained with deal.II: each color represents a different process. On the left there's an example of mesh A, a unitary square $[0, 1] \times [0, 1]$, on right an example of mesh B: a ball of radius 0.35, centered in $(0.4, 0.45)$. Images are separated for better visualization; figure 4 shows a similar situation in three dimensions.

These simple images show how coupling becomes extremely difficult in a distributed environment; looking at the algorithm steps iv we notice the following: to solve part *ii* and *iii* the process owning $y_\alpha \in B$ must also own the cell K of B containing $y_\alpha \in B$. In a distributed setting this condition is most likely **not** satisfied[4] and ghost cells are not enough to solve the problem.

The problem arises also in the other way round: in order to do the coupling properly a processor which locally owns a part of mesh A needs to know about all parts of B occupying the same space, or interfacing with it: with algorithm 1 this is simply not possible.

The problem which needs to be solve has become an *ownership* problem: "which process owns the part of A in which p lies?" or "Is a part of B inside this portion of space?".

[4]In fact, in version 8.5.1. of deal.II, compute point locations throws an assert error if a point is found lying on artificial cells

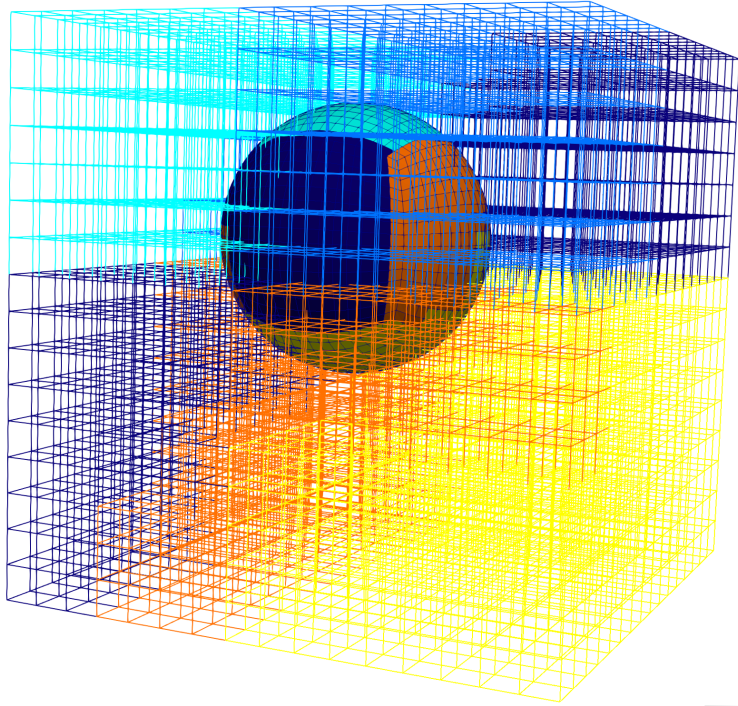


Figure 4: Distributed cube and sphere

3.1 Bounding Boxes

While an accurate description of the boundaries of each locally owned domain would, theoretically, answer to the problem, this is not a feasible solution: too much information would need to be shared, defeating the purpose of distributing the mesh. The solution we need is an approximation with which we can return the process owning a point, possibly among few "guesses".

A new data structure has to be introduced; ideally this structure should be efficient and reliable while using as little memory and communication as possible; the natural solution is using simple containers, such as bounding spheres (BS) or axis-aligned bounding boxes (AABB). Both solutions are covered by a wide literature because of their applications ranging in the most diverse areas: from robotics to computer graphics.[5].

Our choice was AABB, as done in the DTK library, because of their best fitting abilities.

[5]A detailed study of this and other related matters can be found in [8]

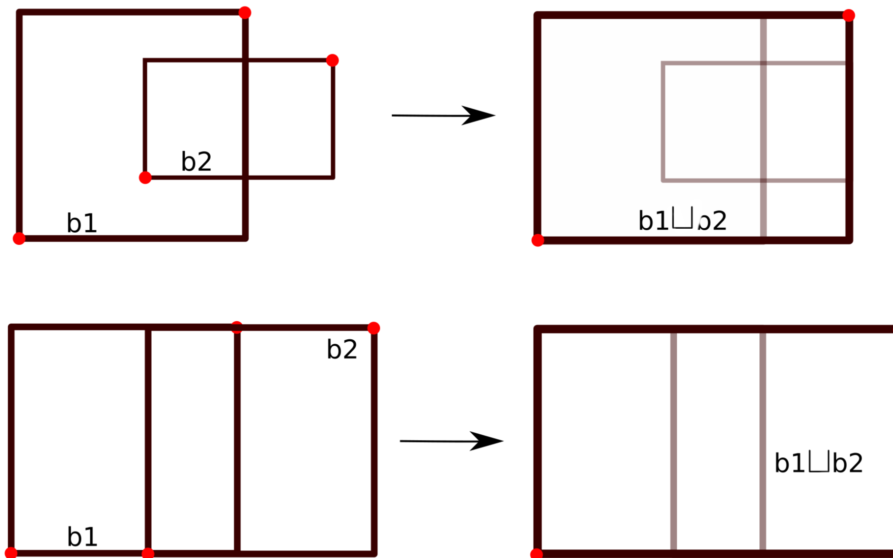


Figure 5: Non-mergeable and mergeable bounding boxes

3.1.1 Bounding Boxes Computations

An AABB is described using a pair of points; usually describing, in order, the bottom-left and top-right corners of the box. After the definition of a bounding box many operations can be defined; there are standard methods to quickly check if a point is inside/outside an AABB and, given two bounding boxes $b1$ and $b2$, to compute intersection and union (as bounding boxes i.e. the smallest bounding box containing both $b1$ and $b2$ which we shall indicate with $\sqcup$) of AABBs (see article [8]).

Thinking of bounding boxes as closed sets result in their intersection coinciding with the set intersection; though in general $b1 \sqcup b2 \subseteq b1 \cup b2$ (see figure 5). In the second example we see a case in which $b1 \sqcup b2 = b1 \cup b2$; we shall call such cases "mergeable". For the algorithm implemented it is of importance to understand if two bounding boxes are mergeable or not; this is computed with the following algorithm 3.

[6] Given a refinement level which has enough coarse cells to describe the space it is possible to create a description of the space occupied by the locally owned cells. This is a sketch of the algorithm used:

[6]For the reader's sake algorithms are sketched without any technical subtlety.

Input: Two bounding boxes $b1$ **and** $b2$ **in the space of dimension** ***spacedim***

Output: True if the bounding boxes are mergeable

compute $b3 = b1 \cap b2$;
if $b3$ *is empty or has dimension* $\leq$ spacedim-*2* **then**
    return false;
else if $b1$ *and* $b2$ *are aligned or* $b1 \subset b2$ *or* $b2 \subseteq b1$ **then**
    return true;
end
return false;

Algorithm 3: Checking if bounding boxes are mergeable

Input: A triangulation, a refinement level

Output: A list of bounding boxes

initialize the list of bounding boxes b_list;
for K *in triangulation's cell of refinement level* **do**
    if *children of* K *are locally owned* **then**
        Create a bounding box bb surrounding all locally owned cells inside K;
        Add bb to b_list;
    end
end
Merging section:;
for $b1$ *in b_list;*
do
    merge_happened = false;
    for $b2$ *in b_list;*
    do
        if $b1! = b2$ *and* $b1, b2$ *are mergeable* **then**
            Add $b1 \sqcup b2$ to b_list;
            Remove $b1$ and $b2$ from b_list;
            merge_happened = true;
        end
        if *merge_happened* $==$ *false* **then**
            return b_list;
        end
    end
end

Algorithm 4: Creating a description of the locally owned meshes

The algorithm 4 is guaranteed to return a collection of bounding boxes which contains all locally owned cells [7]; but, because of the quadratic time required in the merging section of algorithm 4, it is suitable only for meshes obtained from a relatively low number of coarse cells. Another flaw is the need to choose a refinement level which works well with the current mesh. While these problems shall be corrected in the future (see section 5), coupling complexity is so high that algorithm 4 suits our needs without using any relevant computing time.

Guessing Ownership Once every process has a global description of the mesh, in terms of bounding boxes, the following algorithm can be used to effectively guess the "owner" of each point:

```
Input: A list of points
Output: A list of points for which the owner was found;
A list of points for which there is a number of probable
  owners;
initialize the output lists;
for p in points do
    for rank in processes do
        for box in rank bounding boxes do
            if p inside box then
                add rank to owners of p;
                go to next rank;
            end
        end
    end
    if p has one owner then
        Add p to points for which the owner was found;
    else
        Add p to points with multiple owners;
    end
end
```

Algorithm 5: Guessing Owner Algorithm

[7]The current implementation in deal.II works only with non-curved geometry because of technical limitations.

3.1.2 Bounding Boxes Exchange

It is important for the output of 4 to contain few bounding boxes per process: once the locally owned part of the mesh is described with a list of bounding boxes, a collective "allgather" is used to distribute the information about locally owned cells to all processes and storing it into a container, *global bounding boxes*; this is the only necessary collective operation used in our algorithm.

With *global bounding boxes* it is now possible to identify the owners of an arbitrary point inside the mesh with a simple search, which we implemented in the function *guess point owner*. Once the owner(s) of a point are found the coupling problem can be solved with point-to-point communication (see section 3.1.3).

Object Communication Sending different objects using MPI is often difficult, complicated and impractical; a key ingredient for our implementation was implementing two interfaces for an "all gather" and a "some to some" function which can send arbitrary objects, as long as a serialize function is implemented for them [7]. Avoiding the manual de-construction and reconstruction of objects or the use of MPI Types results in simpler and cleaner code and saves time in both the writing and debugging part of code implementation.

3.1.3 Distributed Compute Point Locations

The structure built now allows for a distributed solution of steps *ii*) and *iii*) in the algorithm steps iv; a parallel version of *compute point locations* was written.

The function needs to return the same output in case of a serial and, when called on a distributed mesh from all processes, it needs to return the output of *compute point location* as if it was called on the points geometrically inside the locally owned domain.[8].

For clarity look at figure 3: what we aim at is, given the domain of each process which is defined in the square picture, return *compute point location* for the points of the circle lying over it. This must happen independently from the process owning each part of the circle.

The idea behind a "distributed compute point locations" is thus the following

- Use the bounding boxes to guess where each point lies

[8]Because some of these points might not be owned initially, the output is the same of *compute point location* plus a list of the points for which the output is presented and an id for each point

- Use send and receive for points lying on parts of the domain not locally owned
- Use compute locations on all the points which lie on the locally owned part of the domain

To implement this algorithm it needs to be articulated in more parts, a sketch is presented in algorithm 6

Input: A triangulation, global bounding box description, a list of points

Output: cells containing points located inside the locally owned part of the mesh, their transformed, and unique point ids

Use *global bounding boxes* to guess the owners of each point;
Call *compute point location* on the points which are probably local;
Send and receive points which have a single owner;
Send the output of *compute point location* and relative points for what resulted to be in ghost cells;
Call *compute point location* on the points which received and owned;
Send and receive points with multiple possible owners;
Call *compute point location* on points which might be owned;
Build output from all computed data;

Algorithm 6: A scheme for distributed compute point locations

The practical implementation has many complications:

- The tasks described are either communication intensive or computation intensive; task spawning was used in an attempt to minimize communication overhead
- The scheme is articulated in more, shorter, sections and sub-functions.
- The output of *compute point location* is articulated and merging multiple outputs has a non trivial computational cost

Final results for this function are presented in section 4.3.

4 Benchmarks

After a quick analysis of the current deal.II performance, this final section presents the code implemented for this thesis project and shows the results obtained with it both in serial and in parallel.

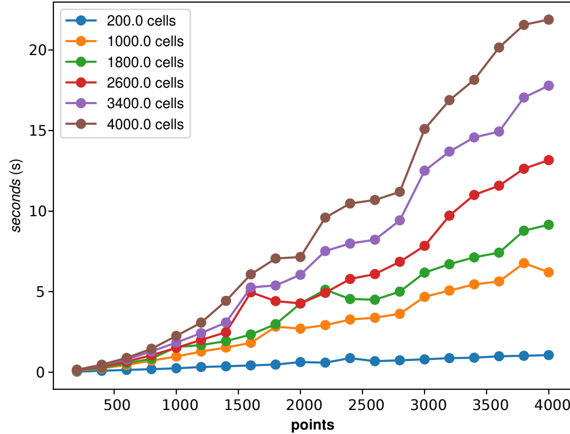


Figure 6: *compute_point_locations* v1: varying number of points

4.1 Serial Baseline

We shall call the version of *compute_point_locations* present in deal.II version 8.5.1 "version 1" (from now on: v1). The initial benchmark 6 was run with random points on a square grid.[9]

Notice how, even if the growth is linear, computational cost is extremely high; taking more than 20 seconds to find 4000 points inside a 4000 cells. For a deeper analysis *callgrind* and *kcachegrind* were used; results are reported in 2.

This preliminary analysis results can be summarized in the following:

- Function's performance is mediocre
- Its biggest computational burden lies in *tranform_real_to_unit_cell* i.e. $F_K^{-1}(p)$, which is called 10176 to classify 200 points: this can probably be avoided
- The second big cost lies on *find_active_cell_around_point*

4.1.1 Find active cell around point

As pointed out in 2 the deal.II implementation of *find active cell around point* is quite slow but, because of its relevance to our problem we quickly

[9]This preliminary result was obtained on a laptop running ArchLinux on an Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz; afterwards the benchmarks were run on SISSA's cluster Ulysses, which runs on Intel(R) Xeon(R) CPU E5-2640 0 @ 2.50GHz.

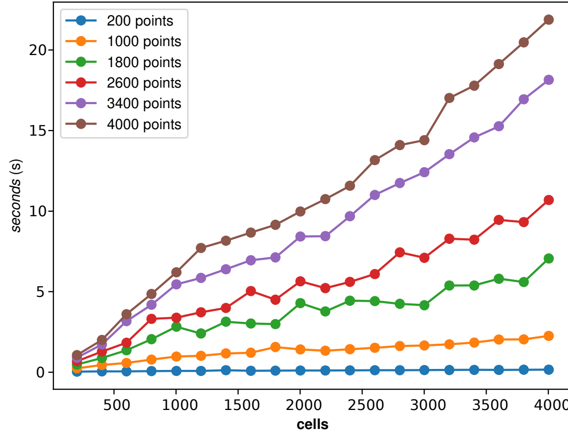


Figure 7: *compute_point_locations* v1: varying number of cells

report on the improvements brought by Gassmoeller and Heltai.

The initial improvements are due to Gassmoeller et al. [9], who implemented them in the ASPECT library [10] [2]. Afterwards, together with Luca Heltai, the following enhancements were brought to the deal.II library:

- Adding a storing system, **GridTools::Cache**, which allows to compute only once non DAG information
- Implementing an algorithm to guess which cell among the neighbours of v is most likely to be the one containing p.

There are many simple methods to guess which cell might contain a point, such as checking the distance from the point to the center; but a better algorithm, based on the angle between the vector $p-v$ and the vectors going from each cell's center to v, was developed by Rene Gassmoeller et al. (see [9]).

These improvements radically changed the performance of *find active cell around point* with a great benefit for *compute point locations*.

4.2 Improvements for the Serial Code

As pointed out in section 2.0.1 a number of things can be improved in the current algorithm.

Calls to *tranform_real_to_unit_cell* The first problem is that *tranform_real_to_unit_cell* is **always** used to identify whether or not a point is inside a cell, even if it could be discarded using simpler methods, such as

Table 1: Kcachegrind analysis for v1 with 200 cells, 200 random points

CEst	CEst per call	Count	Callee
87.68	25372	10176	*tranform_real_to_unit_cell*
11.73	300388	115	*find_active_cell_around_point*
0.15	146937	3	dl runtime resolve avx
0.02	560	115	std::allocator<std::vector<Point<2> > >

Table 2: Kcachegrind analysis for v1 with 400 cells, 200 random points

CEst	CEst per call	Count	Callee
82.08	25499	13657	*tranform_real_to_unit_cell*
17.29	513026	143	*find_active_cell_around_point*
0.17	247442	3	dl runtime resolve avx
0.07	2135	143	std::allocator<std::vector<Point<2> > >

the distance from the cell's center.

In particular, given a cell K, let p_K be its center, and d_K the cell's diameter; then the following "distance check" can be used to avoid many calls to *tranform_real_to_unit_cell* when checking if the point p is inside [10]. The *tranform_real_to_unit_cell* function is called only if:

$$||p - p_K|| < \frac{1}{2} d_K$$

This method reduces the number of calls to *tranform_real_to_unit_cell* cell by more than an order of magnitude, with a great impact on performance. We shall call the function with this improvement "version 2".

Looping on all the points The second bottleneck comes from the fact that the algorithm always loops on all remaining points: this number is potentially huge and, thus, even the method proposed in paragraph 4.2, which discards them quickly, is not completely satisfying.

Thanks to section 4.1.1 the now improved speed of *find active cell around point* makes it possible to use a completely different approach:

[10] For curved elements $\frac{1}{2}d_K$ is sometimes too low, resulting in a cell being repeated in the output; to solve this possible problem a parameter controlling this cut-off value was added

Data: A triangulation, a list of points
Result: cells containing points and their transformed

Initialize the vector with cells, points,qpoints;
for *p in points* **do**
  Find the active cell K around p and $q = F_K^{-1}(p)$;
  if K *in cells* **then**
    add p and q to the points in K;
  else
    Add K to cells;
    add p and q to the points in K;
  end
end
return cells, points, qpoints;

Algorithm 7: Compute Point Locations, version 3

This algorithm is much more elegant than the previous ones and its cost lies almost entirely on *find active cell around point*, making it benefit greatly from the improvements of 4.1.1. Moreover its performance is quite consistent and independent of the number of the point's configuration.

Searching for K in the cell list has also a high cost but, in many in many practical problems coming from mesh coupling, points are clustered making the number of cell's they occupy low compared to the number of points.

Using different Containers In order to reduce the searching cost different containers were used instead of vectors; unordered maps and unordered multimaps were used, because of their constant $O(1)$ access time[18], depending on the hashing function; in this case the unique *active cell index* was used. The versions compared are:

- version 2
- version 3
- version 3 modified to check if the point is inside the last found cell (to take advantage of point's order, see paragraph 4.2).
- version 3 which uses an unordered multimap and merges the different mapped values.
- version 3 which uses and unordered map and then creates an output vector.

As shown in 10 this results in a speed up of approximately 5%.

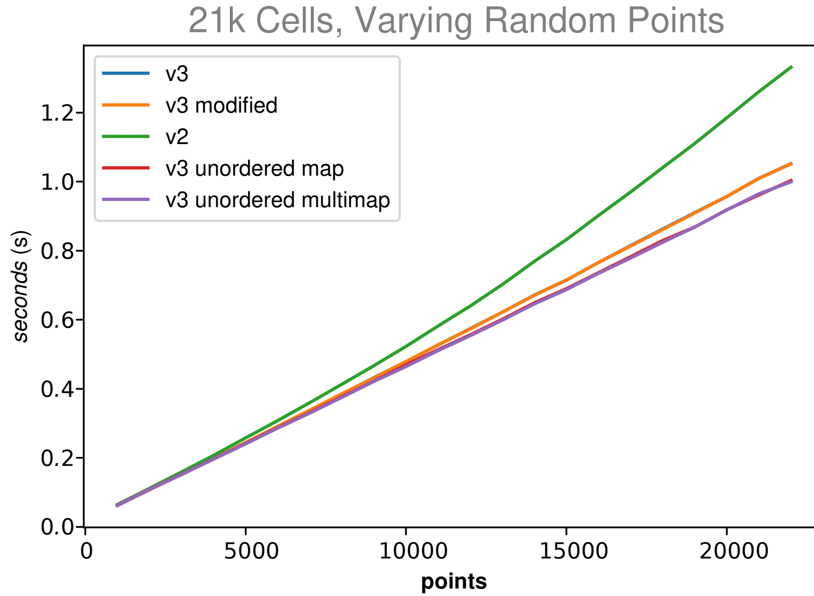


Figure 8: Container Comparison

Mixing Both Approaches The remaining way to improve the function's performance is to take point's order into account which leads to assuming that p is probably in the last found cell.

This leads to two algorithm changes:

i. Before calling the search function, check if the current cell coincides with the last one found.

ii. Use the distance test exposed in paragraph 4.2 to evaluate if the last cell is a good hint cell for *find active cell around point*.

Passing a wrong hint cell to *find active cell around point* crushes performance because it results in testing an extra set of wrong neighbour cells, before looking for a better option [11]. At the same time a simple test such as the one proposed in paragraph 4.2 represents a small overhead with a random set of points, while it allows to safely pass hint cells when points are ordered making it much faster than v3 on such sets.

[11] More precisely: the function looks for v, the closest vertex of the hint cell to p and checks if p is inside one v's neighbours. If this test fails it uses the standard algorithm: look for the closest vertex v in the whole mesh.

Data: A triangulation, a list of points
Result: cells containing points and their transformed

Initialize the vector with cells, points,qpoints;
Find the active cell K around p and $q = F_K^{-1}(p)$;
Compute the K's center p_K and diameter d_K;
for *p in points* **do**
  if $||p - p_K|| < d_K$ **then**
    Find the active cell with hint cell K;
  else
    Find the active cell;
  end
  if $K ==$ *last cell* **then**
    add p and q to the points in K;
  else
    Look for K in cells;
    Same as version 3;
  end
end
return cells, points, qpoints;

Algorithm 8: Compute Point Locations, version 4

4.2.1 2d Serial Benchmarks

Test in two dimensions were run with the following settings:

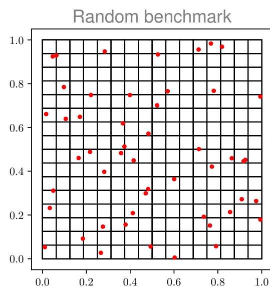
- a "Random Benchmark": random points (see figure 10)
- a "Clusterized Benchmark": points formin a spiral, see figure 9b[12]

Figures 9b represent examples with 50 points and a uniformly refined grid. If N_{cells} is the target number of cells and N_{child} the number of children obtained from the refinement of a single cell the cells were refined uniformly **floor** $\left(\log_{N_{child}}(N_{cells})\right)$ times[13], and then further cells were refined reaching a total number of cells between N_{cells} and $N_{cells} + N_{child} - 1$.

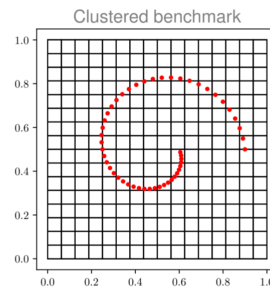
To time the code the the timing tools offered by deal.II and deal2lkit [14] were used.

Here the profiling for clustered simulation on 100000 points and 50000 cells is reported:

[12]Obtained from $r_p\,(\sin(p), \cos(p))$ with $r_p \in]0, 1[$ decreasing towards 0.0, $0 < p < 2\pi$
[13]It's the greatest number of possible uniform refinements without exceeding N_{cells}



(a) Random Benchmark



(b) Clustered Benchmark

```
Version 2
-------------------------------------------------------------
Timer Name                   Global time (num calls)
-------------------------------------------------------------
Add new cell                 0.04804 (399)
add point to existing cell 0.5034 (9.96e+04)
Bench v2                     114.1 (1)
Loop on points of cell       106.9 (2.1e+07)
Transform point              0.7632 (1.259e+05)
find_active_cell             0.08292 (400)
=============================================================
Version 3
-------------------------------------------------------------
Timer Name                   Global time (num calls)
-------------------------------------------------------------
Bench v3                     12.56 (1)
add new cell                 0.002193 (400)
add point to existing cell 0.5158 (9.96e+04)
find if cell present         0.609 (1e+05)
find_active_cell             11.31 (1e+05)
=============================================================
Version 4
-------------------------------------------------------------
Timer Name                   Global time (num calls)
-------------------------------------------------------------
Bench v4                     1.185 (1)
add new cell                 0.002221 (399)
add point to existing cell 0.4949 (9.96e+04)
find if cell is present      0.002243 (399)
find_active_cell             0.03758 (4)
find_active_cell hint        0.5732 (1e+05)
=============================================================
```

As expected the time needed to add new cells and points it's the same

Table 3: Clustered Benchmark 100k points, 50k cells: timings

Points	Cells	v1	v2	v3	v4
10^5	510^4	-	114.1	12.56	1.185
510^4	210^4	137.9	43.35	3.169	0.584
10^4	210^4	26.96	7.668	0.6298	0.1233
10^4	510^3	11.81	3.999	0.2818	0.1144

Table 4: Random Benchmark 100k points, 50k cells: timings

Points	Cells	v1	v2	v3	v4
10^5	510^4	-	8227	13.86	14.04
510^4	210^4	-	1879	3.451	3.58
10^4	210^4	-	171.6	0.6645	0.6654
10^4	510^3	231.6	86.06	0.2979	0.297

among all versions[14]; this time clearly can't be improved much.

Remaining observations endorse what has been written in subsection 4.2: a comparison between v3 and v4 shows that, using the distance to evaluate if the old cell is a reasonable hint, makes the finding process much faster while for v2 the cost of *find active cell around point* is low, but the time needed to loop on all points slows down the process.

15

Finally the speed ups are reported in 5: for the clustered case it was of **236**, for the random case it was of **780**.

[14]In the parenthesis there is the number of calls to a function; the difference 399 and 400 it's because the first call to the function was unintentionally not instrumented

[15]v1 was not tested in all cases because of its slow performance

Table 5: Speed up w.r. of v1 in 2d

	v2	v3	v4
Random (10^4, 510^3)	2.69	777	**780**
Clustered (10^4, 210^4)	2.95	41.9	103
Clustered (510^4, 210^4)	3.18	43.5	**236**

Table 6: Benchmark 100k points, 50k cells: timings

Type	v2	v3	v4
Clustered	12.27	29.56	2.091
Random	8434	30.84	30.98

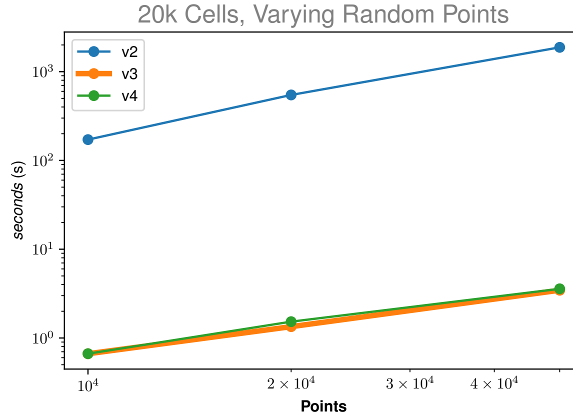


Figure 10: 3d Benchmark With Random Points

4.2.2 3d Serial Benchmarks

For 3D simulations the "Clusterized Benchmark" a spiral with parametrization $t \mapsto (0.4\cos(t) + 0.5, 0.4\sin(t) + 0.5, ht + 0.1)$ was used. Table 6 shows how v4 keeps being the best function even in three dimensions, while v2 now becomes extremely slow with random points.

In both clustered tand random tests scaling results to be linear but, as shown in figure 10, v3 is slow in comparison to the others. The speed ups reported in table 7 show number reduced in comparison with the 2d case: this is because in three dimensions the algorithm used in *find active cell around point* is much slower, having to work with an average of 8 neighbouring cells instead of 4.

Table 7: Speed up w.r. of v1 in 3D

Type (points, cells)	v2	v3	v4
Random (10^4, 510^3)	-	1.77	**5.27**
Clustered (10^4, 510^3)	1.98	395	385

Table 8: v5 with unordered map comparison

(Points,Cells)	v5	v5 with unordered map	Speed Up
(3431, 4096)	0.1797	0.1427	1.26
(27967, 32768)	4.082	3.927	1.04
(226415, 32768)	33.53	32.27	1.04

Final 3d Benchmarks In paragraph 4.2 a benchmark with different containers shown a speed improvement of about 5%. For this reason a new implementation of version 4 was added which uses unordered map as container. The tests were done in a 3D setting: the grid being a cube and the points being arranged in a shape form (see figure 4, which is the distributed case).

Table 8 shows how, consistently with what found before, using unordered maps as containers gives a boost of about 5%.

4.3 Parallel Results

In our initial test we used a cube and a sphere, see figure 4, partitioned among a varying number of processes. In table 4.3 the profiling with 6 processes is shown.

Notice how *Compute and merge other points* takes most of the time: this is the section of the code were compute point locations is run on points received from other processes and then added to the current output. This is process is slow because after running *compute point locations* for each process, we also need to search on cells in order to merge them; as reported in 4.2, this has a high cost.

```
==================================================

      TimeMonitor results over 12 processors

Timer Name                         MeanOverProcs
--------------------------------------------------
Compute and merge other points     0.1 (1)
Compute mesh predicate box         0.001423 (1)
Constructing points to be sent     0.0001599 (1)
Dcploc, cube 4 sphere: 3           0.2365 (1)
Merge ghost                        0.07784 (1)
Using BBoxes to guess owner        0.001361 (1)
all gather for bboxes              0.001553 (1)
some_to_some ghost part            0.01146 (1)
some_to_some other points          0.005509 (1)
some_to_some owned points          0.01564 (1)
==================================================
```

Making a scaling test for this problems is complicated: using a distributed mesh the number of points which has to be computed, computed and communicated or simply communicated varies. A preliminary scaling result is shown in 11: in this case *compute point locations* was run on a total of $10k$ points. With a limited number of processes, up to 16, the scaling is reasonable. Afterwards the function *Compute and merge other points* dominates the computational time. A first attempt to solve this was done by merging the point vector received from other processes and calling compute point locations only once, but this didn't improve performance.

Using unordered maps After the results of paragraph 4.2.2 we tried to improve scaling by the use of another container, an unordered map, and of version 6 of *compute point locations.*

The test was run using a cube and and a sphere inside it, as shown in figure 4; table 4.3 shows the profiling: where the algorithm is now well-balanced. Figure 12 shows the preliminary scaling results: clearly, for this particular use-case, unordered multimaps outperform vectors.

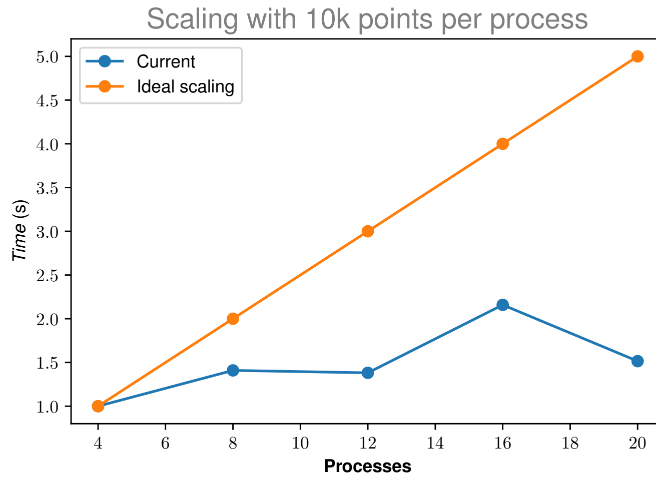


Figure 11: Strong Scaling test

```
TimeMonitor results over 20 processors

Timer Name                         MinOverProcs      MeanOverProcs
-------------------------------------------------------------------
Compute and merge other points     0.03702 (1)       0.03702 (1)
Compute mesh predicate box         0.00342 (1)       0.003455 (1)
Constructing points to be sent     0.0001049 (1)     0.0001074 (1)
Dcploc, cube 5 sphere: 4           0.07959 (1)       0.07962 (1)
Merge ghost                        0.000176 (1)      0.003586 (1)
Using BBoxes to guess owner        0.00106 (1)       0.001089 (1)
all gather for bboxes              0.002365 (1)      0.002371 (1)
some_to_some ghost part            0.000526 (1)      0.009646 (1)
some_to_some other points          0.001664 (1)      0.001669 (1)
some_to_some owned points          0.0008979 (1)     0.0009018 (1)
```

Notice the distribution of the sphere, and the point's distribution, changes with every new number of processes uses: this affects the performance of *compute point locations* and it's behind the apparent super-linear scaling.

While a more reliable test should be devised, the plot clearly shows that the algorithm is scaling well at least with up to a few dozens of cores.

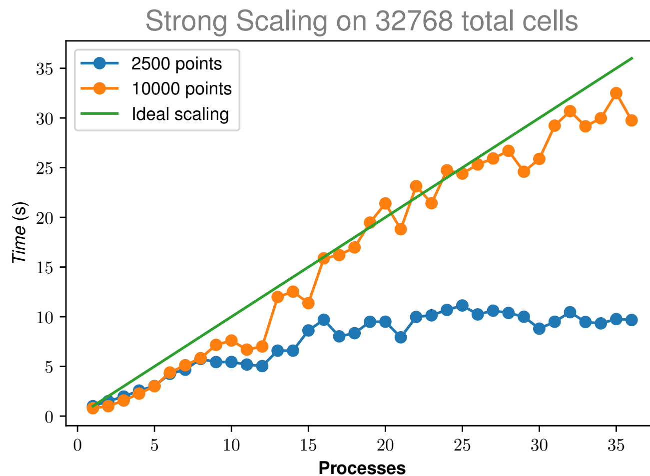


Figure 12: Strong Scaling test

5 Conclusions

Results obtained in serial are extremely good: we obtained a reliable function which performs greatly with any sort of points and in both two and three dimensions. The next step is probably the use of kd-trees to speed up the search for the closest vertex in the *find active cell around point* function.

Considering the intrinsic difficulty in this parallelization problem, the results obtained in the distributed version of compute point locations are extremely promising. Better tests with different settings and more cores should be done.

When the number of processes, and thus of bounding boxes, becomes extremely high the search on bounding boxes shall probably become a bottleneck; a possible solution is a better research algorithm: research trees. This sort of solution is known in literature [5] and implemented also in the DTK library.

Finally a better algorithm to compute the bounding boxes describing the locally owned part of the mesh shall be implemented: possibly one which doesn't need calibration.

References

[1] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells. The `deal.II` library, version 8.5. *Journal of Numerical Mathematics*, 2017.

[2] W. Bangerth, J. Dannberg, R. Gassmöller, T. Heister, et al. ASPECT: Advanced Solver for Problems in Earth's ConvecTion, UserManual. 4 2017. doi:10.6084/m9.figshare.4865333.

[3] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

[4] Wolfgang Bangerth, Carsten Burstedde, Timo Heister, and Martin Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Trans. Math. Softw.*, 38(2):14:1–14:28, January 2012.

[5] Gino van den Bergen. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools*, 2(4):1–13, 1997.

[6] Martin Berzins, Qingyu Meng, John Schmidt, and James C Sutherland. Dag-based software frameworks for pdes. In *European Conference on Parallel Processing*, pages 324–333. Springer, 2011.

[7] Boost. Boost C++ Libraries. `http://www.boost.org/`, 2015. Last accessed 2015-06-30.

[8] Christer Ericson. *Real-Time Collision Detection*. The Morgan Kaufmann Series in Interactive 3d Technology. CRC Press, 2004.

[9] Rene Gassmöller, Eric Heien, Elbridge Gerry Puckett, and Wolfgang Bangerth. Flexible and scalable particle-in-cell methods for massively parallel computations. *arXiv preprint arXiv:1612.03369*, 2016.

[10] Timo Heister, Juliane Dannberg, Rene Gassmöller, and Wolfgang Bangerth. High accuracy mantle convection simulation through modern numerical methods. II: Realistic models and problems. *Geophysical Journal International*, 210(2):833–851, 2017.

[11] David E Keyes, Lois C McInnes, Carol Woodward, William Gropp, Eric Myra, Michael Pernice, John Bell, Jed Brown, Alain Clo, Jeffrey Connors, et al. Multiphysics simulations: Challenges and opportunities. *The International Journal of High Performance Computing Applications*, 27(1):4–83, 2013.

[12] Steven J Plimpton, Bruce Hendrickson, and James R Stewart. A parallel rendezvous algorithm for interpolation between multiple grids. *Journal of Parallel and Distributed Computing*, 64(2):266–276, 2004.

[13] J.W Eastwood R.W Hockney. *Computer Simulation Using Particles.* CRC Press, 1988.

[14] Alberto Sartori, Nicola Giuliani, Mauro Bardelloni, and Luca Heltai. deal2lkit: a toolkit library for deal.ii. Technical Report 57/2015/MATE, SISSA, 2015.

[15] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[16] S Slattery, P Wilson, and RP Pawlowski. The data transfer kit: a geometric rendezvous-based tool for multiphysics data transfer.

[17] Stuart R. Slattery. Mesh-free data transfer algorithms for partitioned multiphysics problems: Conservation, accuracy, and parallelism. *Journal of Computational Physics*, 307(Supplement C):164 – 188, 2016.

[18] Bjarne Stroustrup. *The C++ Programming Language, 4th edition.* Addison-Wesley, 2013.