Algorithms and Data Structures for Massively Parallel Generic Adaptive Finite Element Codes

WOLFGANG BANGERTH
Texas A&M University
and
CARSTEN BURSTEDDE
The University of Texas at Austin
and
TIMO HEISTER
University of Göttingen
and
MARTIN KRONBICHLER
Uppsala University

Author's addresses: W. Bangerth: Department of Mathematics, Texas A&M University, College Station, TX 77843, USA. bangerth@math.tamu.edu — C. Burstedde: Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712, USA. carsten@ices.utexas.edu — T. Heister: Institute for Numerical and Applied Mathematics, University of Göttingen, Lotzestr. 16-18, D-37083 Göttingen, Germany. heister@math.uni-goettingen.de — M. Kronbichler: Department of Information Technology, Uppsala University, Box 337, SE-75105 Uppsala, Sweden. martin.kronbichler@it.uu.se

W. Bangerth was partially supported by Award No. KUS-C1-016-04 made by King Abdullah University of Science and Technology (KAUST), by a grant from the NSF-funded Computational Infrastructure in Geodynamics initiative, and by an Alfred P. Sloan Research Fellowship. C. Burstedde was partially supported by NSF grants OPP-0941678, OCI-0749334, DMS-0724746, AFOSR grant FA9550-09-1-0608, and DOE grants DE-SC0002710 and DEFC02-06ER25782. C. Burstedde would like to thank Lucas C. Wilcox, Tobin Isaac, Georg Stadler, and Omar Ghattas for extended discussion. T. Heister was partially supported by the German Research Foundation (DFG) through Research Training Group GK 1023. M. Kronbichler was supported by the Graduate School in Mathematics and Computation (FMB).
Most of the work was performed while T. Heister and M. Kronbichler were visitors at Texas A&M University.
Some computations for this paper were performed on the "Ranger" cluster at the Texas Advanced Computing Center (TACC), and the "Brazos" and "Hurr" clusters at the Institute for Applied Mathematics and Computational Science (IAMCS) at Texas A&M University. Ranger was funded by NSF award OCI-0622780, and we used an allocation obtained under NSF award TG-MCA04N026. Part of Brazos was supported by NSF award DMS-0922866. Hurr is supported by Award No. KUS-C1-016-04 made by King Abdullah University of Science and Technology (KAUST).

Today's largest supercomputers have 100,000s of processor cores and offer the potential to solve partial differential equations discretized by billions of unknowns. However, the complexity of scaling to such large machines and problem sizes has so far prevented the emergence of generic software libraries that support such computations, although these would lower the threshold of entry and enable many more applications to benefit from large-scale computing.

We are concerned with providing this functionality for mesh-adaptive finite element computations. We assume the existence of an "oracle" that implements the generation and modification of an adaptive mesh distributed across many processors, and that responds to queries about its structure. Based on querying the oracle, we develop scalable algorithms and data structures for generic finite element methods. Specifically, we consider the parallel distribution of mesh data, global enumeration of degrees of freedom, constraints, and postprocessing. Our algorithms remove the bottlenecks that typically limit large-scale adaptive finite element analyses.

We demonstrate scalability of complete finite element workflows on up to 16,384 processors. An implementation of the proposed algorithms, based on the open source software `p4est` as mesh oracle, is provided under an open source license through the widely used `deal.II` finite element software library.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Finite element software—*mesh handling*; *linear algebra*; G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*finite element method.*

General Terms: Adaptive mesh refinement, Parallel algorithms, Software design, Object orientation

---

1. INTRODUCTION

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. At the same time, the number of codes that efficiently utilize these machines is relatively limited, due in large part to two reasons: (i) existing legacy codes are difficult to parallelize to these massive numbers of processors since data structures have to be conceived entirely differently; (ii) algorithm and data structure design is not trivial when the goal is to exploit machines of this size in a scalable way.

In the past, continuum mechanics codes—for example based on the finite element method—have been among the largest consumers of supercomputing resources. The desire to run such codes on "massively" parallel machines dates back to at least the early 1990s [Mathur et al. 1993; Devine et al. 1993; Tezduyar et al. 1994], although the notion of what constitutes a large machine has evolved since then. On the other hand, codes that scale to the largest available machines were then, and are now, almost exclusively purpose-built for individual applications. For example, the codes SPECFEM3D [Carrington et al. 2008], CitcomS [Tan et al. 2008], and Rhea [Burstedde et al. 2008] have been written for particular geophysical applications and are not based on general-purpose finite element libraries. The reason, of course, is that none of the libraries widely used in academic and applied finite element simulations—such as PLTMG [Bank 1998], DiffPack [Bruaset and Langtangen 1997; Langtangen 2003], libMesh [Kirk et al. 2006], Getfem++ [Renard and Pommier 2006], OOFEM [Patzák and Bittnar 2001], FEniCS/DOLFIN [Logg 2007; Logg and Wells 2010], or `deal.II` up to version 6.x [Bangerth et al. 2007; Bangerth and Kanschat 2011]—support massively parallel computations that will run on thousands of processors and routinely solve problems with hundreds of millions or billions of cells and several billion unknowns.

This notwithstanding, there clearly is a demand for general-purpose finite element libraries supporting such computations through a relatively simple, generic interface. In this article, we will outline the algorithms that we have implemented in version 7.0 of the open

source library `deal.II`, offering the ability to solve finite element problems on fully adaptive meshes of the sizes mentioned above to a wider community. While `deal.II` provides a reference implementation of both the mentioned algorithms as well as of tutorial programs that showcase their use in actual applications, our goal is to be generic and our methods would certainly apply to other finite element libraries as well. In particular, we will not require specific aspects of the type of finite element, nor will the algorithms be restricted to quadrilaterals (2d) and hexahedra (3d) that are used exclusively in `deal.II`.

When using thousands of processors[1] 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.

In this paper, we will not be concerned with the question of how to efficiently generate and partition hierarchically refined meshes on large numbers of processors, which presents a major challenge on its own. Rather, we will assign this task to an "oracle" that allows `deal.II` to obtain information on the distributed nature of the mesh through a well-defined set of queries. These include for example whether a certain cell exists on the current processor, or whether it is a "ghost" cell that is owned by another processor. Using queries to the oracle, each processor can then rebuild the rich data structures necessary for finite element computations for the "locally owned" part of the global mesh and perform the necessary computations on them. In our implementation, we use the `p4est` algorithms for 2d and 3d parallel mesh topology [Burstedde et al. 2010] as the oracle; however, it is entirely conceivable to connect to different oracle implementations—for example packages that support the ITAPS iMesh interface [Ollivier-Gooch et al. 2010]—, provided they adhere to the query structure detailed in this article, and can respond to certain mesh modification directives discussed below.

We will detail the requirements `deal.II` has of the mesh oracle, a description of the way `p4est` works, and the algorithm that builds the processor-local meshes in Section 2. In Section 3, we will discuss dealing with the degrees of freedom defined on a distributed mesh. Section 4 will then be concerned with setting up, assembling and solving the linear systems that result from the application of the finite element method; Section 5 discusses the parallel computation of thresholds for a-posteriori error indicators and postprocessing of the solution. In reality, today's supercomputers do not consist of a set of equal, interconnected single-processor machines; rather, they have multicore chips and may have general-purpose graphics processing units (GPUs). Section 6 reviews our designs in light of these considerations on processor architecture and argues that, at least given the current state of parallel linear algebra and solver software, using a "flat" MPI space consisting of single-threaded processes is still the most practical choice. We provide numerical results that support the scalability of all proposed algorithms in Section 7 and conclude in Section 8.

[1] Since this paper is purely algorithmic, we will not distinguish between *processors*, *processor cores*, and *MPI processes* [Message Passing Interface Forum 2009]. We will here use the terms interchangeably.

2. PARALLEL CONSTRUCTION OF DISTRIBUTED MESHES

Finite element methods decompose the computational domain into a collection of cells, called a *mesh* or *grid*. We understand the term *mesh* primarily as information on the topological relationships between cells—i.e., how cells are connected by neighborship relations—rather than on the geometric location of cells, although the latter also needs to be stored for finite element applications.

The parallel scalability of previous versions of `deal.II` was restricted by a bottleneck present in many generic finite element libraries, namely the requirement to replicate the global mesh structure on each processor. As the number of mesh cells is increased, memory requirements increase in proportion. It is thus obvious that replicating the mesh storage on each processor limits the total mesh size by the amount of *local* processor memory, which is not likely to grow in the future. We resolve this limitation by *distributed* mesh storage with *coarsened overlap*: Each processor still stores a local mesh that covers the whole domain, but this mesh is now different on each processor. It is identical to the global mesh only in the part that is identified by the oracle as "locally owned" by the current processor, whereas the remaining and much larger non-owned part of the local mesh is coarsened as much as possible, rendering its memory footprint insignificant. With this approach the global mesh is not replicated anymore but understood implicitly as the disjoint union of the locally owned parts on each processor. To achieve parallel scalability of the complete finite element pipeline, the storage of degrees of freedom and matrices arising from a finite element discretization must be fully distributed as well, which can be achieved by querying the oracle about ghost cells and creating efficient communication patterns and data structures for index sets as we will explain below.

We encode the distributed mesh in a two-layered approach. The inner layer, which we refer to as the "oracle", provides rudimentary information on the owned part of the mesh and the parallel neighborhood, and executes directives to coarsen, refine, and re-partition the mesh. The outer layer interacts with the oracle through a well-defined set of queries and builds a representation of the mesh that includes the refinement hierarchy and some overlap with neighboring mesh parts, and is rich enough to provide all information needed for finite element operations. This two-layered approach effectively separates a large part of the parallel management of mesh topology in the oracle from the locally stored representation retaining the existing infrastructure in `deal.II`.

There is a significant amount of literature on how to generate and modify distributed adaptive meshes in parallel. For example, [Burri et al. 2005; Tu et al. 2005; Chand et al. 2008; Sundar et al. 2008; Knepley and Karpeev 2009] discuss related data structures and algorithms. In the current contribution, we base our work on the open source software library `p4est`, which realizes the oracle functionality in the sense outlined above, and has been shown to scale to hundreds of thousands of processors [Burstedde et al. 2010]. However, any other software that allows the well-defined and small list of queries detailed below may equally well be used in place of `p4est`. For example, this could include the packages that support the ITAPS iMesh interface [Ollivier-Gooch et al. 2010].

In this section, we define the general characteristics of the mesh, propose an algorithm to construct the local mesh representation based on querying the oracle, and document mesh modification capabilities required from the oracle.

2.1 Assumptions on parallel distributed meshes

We will here not be concerned with the technical details of the parallel storage of meshes or the algorithms hidden within the oracle. In particular, for our purposes we only need to be able to infer what cells exist, how they relate to each other via neighborship, and how they have been derived by hierarchic refinement from a small to moderate set of initial coarse mesh cells. We will make the following general assumptions that are respected by both the inner layer or oracle (`p4est`) and the outer layer (implemented within `deal.II`):

—*Common coarse mesh:* All cells are derived by refinement from a common coarse mesh that can be held completely on each of the processors and should therefore not exceed a few 100,000 to a million cells. In general, the common coarse mesh only needs to provide a sufficient number of cells to capture the topology of the computational domain, which is often below 100 or even as small as 1, while mesh refinement takes care of geometric details. Only in rare cases does geometric complexity require 100,000s or more coarse mesh cells; consequently, we reserve the dynamic partitioning of coarse cells, which is certainly feasible, for a future extension. Because `deal.II` exclusively supports quadrilaterals and hexahedra, we will henceforth assume that the common coarse mesh only consists of such cells, though this is immaterial for almost all that follows, and our algorithms are equally valid when applied to meshes consisting of triangles or tetrahedra.

—*Hierarchic refinement:* Each of the common coarse mesh cells may be hierarchically refined into four (2d) or eight (3d) children which may in turn be further refined themselves. This naturally gives rise to a quad- or oc*tree* rooted in each common coarse cell, and an appropriate data structure for the entire mesh then is a quad- or oct*forest*. Therefore each cell can be uniquely identified by an index into the common coarse mesh (i.e., its tree number) and an identifier that describes how to walk through the corresponding tree to the (refined) cell.

—*2:1 mesh balance:* We demand that geometrically neighboring cells may differ by only a single refinement level, thereby enforcing that only a single *hanging node* can exist per face or edge. This condition is mostly for convenience, since it simplifies the creation of interpolation operators on interfaces between cells.

—*Distributed storage:* Each processor in a parallel program may only store a part of the entire forest that is not much larger than the total number of cells divided by the number of processors. This may include a fixed number of ghost cell layers, but it cannot be a fraction of the entire mesh that is independent of the number of processors. We explicitly permit that each processor may store parts of more than one tree, and that parts of a given tree may be stored on multiple processors.

Note that a mesh is independent of its use; in particular, it has no knowledge of finite element spaces defined on it, or values of nodal vectors associated with such a space. It is, thus, a rather minimal data structure to simplify parallel distributed storage. Furthermore, the separation of mesh and finite element data structures establishes a clean modularization of the respective algorithms and implementations.

2.2 A mesh oracle and interface to `deal.II`

`deal.II` needs to keep rich data structures for the mesh and derived objects. For example, it needs to know the actual geometric location of vertices, boundary indicators, material

properties, etc. It also stores the complete mesh hierarchy and data structures of surfaces, lines and points and their neighborship information for traversal, all of which are required for the rest of the library and to support algorithms built on it.

On the other hand, `p4est` only stores the terminal nodes (i.e., the leaves) of the parallel forest explicitly. By itself, this is not enough for all but the most basic finite element algorithms. However, we can resolve this apparent conflict if `deal.II` builds its own local mesh on the current processor, using the locally stored portion of the parallel distributed mesh stored by `p4est` as the template, and augmenting it with the information needed for more complex algorithms. In a sense, this approach forms a synthesis of the completely distributed and lean data structures of `p4est` and the rich structures of `deal.II`. Designing this synthesis in a practical and scalable way is one of the innovations of this paper; we will demonstrate its efficiency in Section 7.

In order to explain the algorithm that reconstructs the local part of a mesh on one processor, let us assume that both `deal.II` and `p4est` already share knowledge about the set of common coarse cells. Then, `deal.II` uses `p4est` as an oracle for the following rather minimal set of queries:

—Does a given terminal `deal.II` cell exist in the portion of the `p4est` mesh stored on the current processor?

—Does a given `deal.II` cell (terminal or not) overlap with any of the terminal `p4est` cells stored on the current processor?

—Does a given `deal.II` cell (terminal or not) overlap with any of the terminal `p4est` ghost cells (defined as a foreign cell sharing at least one corner with a cell owned by the current processor)?

—Is a given `p4est` cell a ghost cell and if yes, which processor owns it?

The algorithm for mesh reconstruction based on only these queries is shown in Fig. 1. It is essential that all queries are executed fast, i.e., in constant time or at most $\mathcal{O}(\log N)$, where N is the number of local cells, to ensure overall optimal complexity. Furthermore, no query may entail communication between processors. Note that the reconstruction algorithm makes no assumptions on the prior state of the `deal.II` mesh, allowing for its coarsening and refinement as the oracle may have moved cells to a different processor during adaptive mesh refinement and re-partitioning. In a `deal.II` mesh so constructed, different kinds of cells exist on any particular processor:

—*Active cells* are cells without children. Active cells cover the entire domain. If an active cell belongs to a part of the global mesh that is owned by the current processor, then it corresponds to a leaf of the global distributed forest that forms the mesh. In that case, we call it a *locally owned* active cell.

—*Ghost cells* are active cells that correspond to leaves of the distributed forest that are not locally owned but are adjacent to locally owned active cells.

—*Artificial cells* are active cells that are neither locally owned nor ghost cells. They are stored to satisfy `deal.II`'s invariants of never having more than one hanging node per face or edge, and of storing all common coarse mesh cells. Artificial cells can, but need not correspond to leaves of the distributed forest, and are skipped in every algorithm inside `deal.II`.

—*Non-active cells* are cells that have children. `deal.II` stores all intermediate cells that form the hierarchy between coarse mesh cells (the roots of the trees) and active cells.

```
copy_to_deal:
do
  for all coarse mesh cells K:
    match_tree_recursively(K)
  refine and coarsen all cells previously marked
while (the mesh changed in the last iteration)

match_tree_recursively(K):
if (mesh oracle: does K overlap with a cell in the locally owned or ghost parts of the mesh?)
  if (K has children)
    for each child K_c of K
      match_tree_recursively(K_c)
  else
    if (not (mesh oracle: does K exist in the locally owned or ghost parts of the mesh?))
      mark K for refinement
else
  mark the most refined descendents of K, or K itself, for coarsening
```

Fig. 1. *Pseudo-code for reconstructing the local part of a mesh in deal.II, based on querying the mesh oracle provided by* `p4est`. *The algorithm starts with an arbitrary mesh and terminates once the mesh contains all cells that the oracle indicates as either locally owned or ghost cells.*

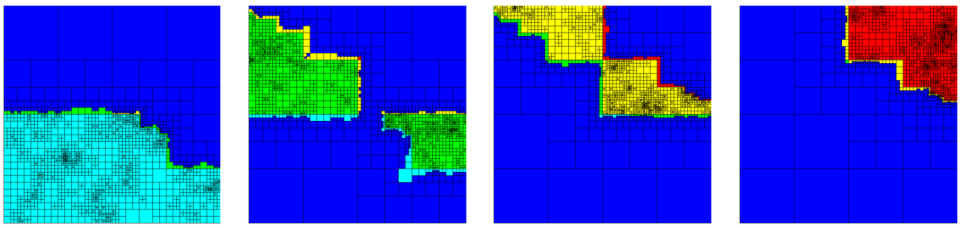

Fig. 2. *Example of an adaptively refined mesh distributed across four processors. The cyan, green, yellow, and red colors indicate which processor owns any given cell. The four panels depict the views each of the four processors has of the mesh. Note that each processor knows only (i) the global cells it owns, and (ii) one layer of ghost cells in the global mesh and their owner processor identifiers. The artificial cells (indicated in dark blue) carry no information. The effective mesh used for computation is the union of the four locally owned parts.*

Fig. 2 shows the result of executing `copy_to_deal` (Fig. 1) on an example mesh distributed among four processors. Note that no processor has knowledge of the entire global mesh—each processor only matches its own cells as well as one layer of ghost cells. Here, the parallel partition and identification of ghost cells is computed by `p4est`, which orders all cells according to a space-filling z-curve [Morton 1966]. Therefore, the part of the global mesh owned by a processor may not be contiguous. This can be seen in the second panel of the figure.

REMARK 1. *Storing artificial cells that do not belong to the coarse mesh appears wasteful since these cells are indeed unnecessary for almost all computations. As pointed out above we only store them to maintain the invariants for which the base library,* `deal.II`, *has been extensively validated. Clearly, the fraction of artificial cells decreases as the number of cells stored locally increases. For the 2d example discussed in Section 7.1, which has 1 coarse cell, our numerical experiments suggest that the ratio* $N_{artificial}/(N_{active}+N_{ghost})$ *is only very weakly dependent on the number of processors, and*

decreases as $\mathcal{O}((N_{active} + N_{ghost})^{-0.55})$. On a fine mesh with 4,096 processors and a total of almost 600 million cells, on average only 3% of the cells stored locally are artificial and on no processor does this number exceed 5%.

We reflect the different types of cells using the following notation: Let $\mathbb{T}$ denote the set of all terminal cells that exist in the distributed mesh. Furthermore, let $\mathbb{T}^p_{\text{loc}} \subset \mathbb{T}$ be the subset of cells that processor p owns; obviously, $\bigcup_p \mathbb{T}^p_{\text{loc}} = \mathbb{T}$, and we will require that $\mathbb{T}^p_{\text{loc}} \cap \mathbb{T}^q_{\text{loc}} = \emptyset$ for all $p \neq q$. Finally, let $\mathbb{T}^p_{\text{ghost}} \subset \mathbb{T}$ be the set of ghost cells that processor p knows about; we have that $\mathbb{T}^p_{\text{ghost}} \cap \mathbb{T}^p_{\text{loc}} = \emptyset$ and we will assume that each ghost cell $K \subset \mathbb{T}^p_{\text{ghost}}$ has at least one neighbor in $\mathbb{T}^p_{\text{loc}}$ where neighborship is via faces, lines, or vertices. In addition to $\mathbb{T}^p_{\text{loc}}$ and $\mathbb{T}^p_{\text{ghost}}$, each processor stores additional terminal cells that may or may not be terminal cells in $\mathbb{T}$, for example some coarse mesh cells. We will call these *artificial* cells and denote them by $\mathbb{T}^p_{\text{artificial}}$; they are shown in dark blue in Fig. 2.

2.3 Directives for mesh modification

We will require that the oracle not only responds to the queries listed above, but also performs several operations that modify the distributed global mesh. Such mesh modifications are often used during the startup phase of a simulation, or repeatedly to adapt according to error indicators or to track dynamical features of a simulation that evolves over time.

—Refine and/or coarsen the mesh based on flags set by `deal.II`. Refinement and coarsening shall be executed locally without communication between processors.

—Enforce 2:1 mesh balance by additional refinement where necessary, limiting the level difference between neighboring cells to one. This is done as a postprocessing step to local refinement and coarsening which involves communication with processors that own nearby parts of the mesh.

—Re-partition the cells of the global mesh in parallel to ensure load balance (the most commonly used criterion being to equalize the number of cells among processors). This operation involves mostly point-to-point communication. During the re-partitioning step, additional information shall be attached to the cells. When a cell is migrated from one processor to another, this data is automatically migrated with the cell.

While this functionality can entail considerable complexity, it is likely to be available from implementations of parallel mesh data bases. Thus, we do not consider the above specifications unnecessarily restrictive. In the case of `p4est` we refer the reader to the algorithms presented in [Burstedde et al. 2010]. `deal.II` makes use of these capabilities to efficiently implement a number of operations typical of finite element codes; see Section 5.

3. DEALING WITH GLOBAL INDICES OF DEGREES OF FREEDOM

Once we have a local representation of a distributed mesh, the next step in any finite element program is to connect the finite element space to be used with the triangulation. In `deal.II`, this tasks falls to the `DoFHandler` class [Bangerth et al. 2007] that inspects a `FiniteElement` object for the number of degrees of freedom that are required per vertex, line, face, and cell. For example, for a Taylor-Hood ($Q_2^d \times Q_1$) element used for the Stokes equations in d space dimensions, we need $d+1$ degrees of freedom per vertex, and d for each line, quad and hex (if in 3d). The `DoFHandler` will then allocate global numbers for each of the degrees of freedom located on the vertices, lines, quads and hexes

that exist in the triangulation. A variant of this class, `hp::DoFHandler`, is able to do the same task if different finite elements are to be used on different cells such as in hp-adaptive computations [Bangerth and Kayser-Herold 2009].

In the current context, we will have to assign global indices for degrees of freedom defined on a mesh of which we only know a certain part on each processor. In the following subsections, we will discuss the algorithms that achieve this task, followed by strategies to deal with the constraints that result from hanging nodes. Together, indices of degrees of freedom and constraints will completely describe a basis of the finite element space we want to use.

3.1 Enumerating degrees of freedom

The simplest way to distribute global indices of degrees of freedom on the distributed mesh would be to first let processor 0 enumerate the degrees of freedom on the cells it owns, then communicate the next unused index to processor 1 that will then enumerate those degrees of freedom on its own cells that have not been enumerated yet, pass the next unused index to processor 2, and so on. Obviously, this strategy does not scale beyond a small number of processors.

Rather, we use the following algorithm to achieve the same end result in a parallel fashion where all processors $p = 0, \ldots, P-1$ work independently unless noted otherwise. This algorithm also determines the ownership of degrees of freedom on the interface between cells belonging to different processors. The rule for decision of ownership is arbitrary but needs to be consistent and must not require communication. The number of processors involved is typically up to eight for a degree of freedom on a vertex in 3d, but can be even higher for a coarse mesh with complicated topology. We resolve to assign each degree of freedom on an interface between processors to the processor with the smallest processor identifier (the "rank" in MPI terminology).

(0) On all active cells (locally owned or not), initialize all indices of degrees of freedom with an invalid value, for example -1.

(1) Flag the indices of all degrees of freedom defined on all cells $K \in \mathbb{T}^p_{\text{loc}}$ by assigning to them a valid value, for example 0. At the end of this step, all degrees of freedom on the locally owned cells have been flagged, including those that are located on interfaces between cells in $\mathbb{T}^p_{\text{loc}}$ and $\mathbb{T}^p_{\text{ghost}}$.

(2) Loop over all ghost cells $K \in \mathbb{T}^p_{\text{ghost}}$; if the owner of K is processor q and $q < p$ then reset indices of the degrees of freedom located on this cell to the invalid value. After this step, all flagged degrees of freedom are the ones we own locally.

(3) Loop over all cells $K \in \mathbb{T}^p_{\text{loc}}$ and assign indices in ascending order to all degrees of freedom marked as valid. Start at zero, and let n_p be the number of indices assigned. Note that this step cannot be incorporated into step (2) because degrees of freedom may be located on interfaces between more than two processors and a cell in $\mathbb{T}^p_{\text{loc}}$ may not be able to easily determine whether cells that are not locally owned share such an interface.

(4) Let all processors communicate the number n_p of locally owned degrees of freedom to all others. In MPI terminology, this amounts to calling `MPI_Allgather`. Shift the indices of all enumerated degrees of freedom by $\sum_{q=0}^{p-1} n_q$. At the end of this step, all degrees of freedom on the entire distributed mesh have been assigned globally unique

indices between 0 and $N = \sum_{q=0}^{P-1} n_q$, and every processor knows the correct indices of all degrees of freedom it owns. However, processor p may not know the correct indices of degrees of freedom on the interface between its cells and those owned by other processors, as well as the indices on ghost cells that we need for some algorithms. These remaining indices will be obtained in the next two steps.

(5) Communicate indices of degrees of freedom on cells in $\mathbb{T}^p_{\text{loc}}$ to other processors according to the following algorithm:
   (a) Flag all vertices of cells in $\mathbb{T}^p_{\text{loc}}$.
   (b) Loop over vertices of cells in $\mathbb{T}^p_{\text{ghost}}$ and populate a map that stores for each of the vertices flagged in step (a) the owning processor identifier(s) of adjacent ghost cells.
   (c) Loop over all cells in $\mathbb{T}^p_{\text{loc}}$. If according to the previous step one of its vertices is adjacent to a ghost cell owned by processor q, then add the pair *[cell_id, indices of degrees of freedom on this cell]* to a list of such pairs to be sent to processor q. Note that the same pair can be added to multiple such lists if the current cell is adjacent to several other processors' cells. Note also that every cell we add on processor p to the list for processor q is in $\mathbb{T}^q_{\text{ghost}}$. This communication pattern is symmetric, i.e., processor p receives a message from q if and only if it sends to q; this symmetry avoids the need to negotiate communications.
   (d) Send the contents of each of these lists to their respective destination processor q using non-blocking point-to-point communication.
   (e) From all processors that the current one borders to (i.e., the owners of any of the cells in $\mathbb{T}^p_{\text{ghost}}$), receive a list as created above. Each of the cells in this list refer to a ghost cell; for each of these cells, set the indices of the degrees of freedom on this cell to the ones given by the list unless the index in the list is invalid.

   Note that while the lists created in step (c) contain only cells owned by the current processor, not all indices in them are known as they may lie on an interface to another processor. These will then be the invalid index, prompting the need for the conditional set in step (e). On the other hand, it is easy to see that if an index located on the interface between two ghost cells is set more than once, then the value so set is always either the same (if the ghost cells belong to the same processor) or the invalid marker (in which case we ignore it).

(6) At the end of the previous step, all cells in $\mathbb{T}^p_{\text{loc}}$ have their final, correct indices set. However, some ghost cells may still have invalid markers since their indices were sent by processors that at the time did not know all correct indices yet. They do now, however. Consequently, the last step is to repeat the actions of step (5). We can optimize this step by only adding cells to the send lists that prior to step (5e) had invalid index markers.

At the end of this algorithm, each processor knows the correct global indices of degrees of freedom on all of the cells it locally owns as well as on all the ghost cells. We note that this algorithm is not restricted to h-refined meshes but is equally applicable to hp-adaptivity.

A similar algorithm that makes the same decision for degrees of freedom on the interface is detailed in [Logg 2009], but there are a few crucial differences to our approach: First, their algorithm contains a sequential part to compute the indices of shared degrees of freedom (Stage 2), while ours does that computation in parallel. Second, our approach lends

itself to non-blocking communication (see step 5d above). Third, we decided to realize the communication over shared vertices instead of facets, which simplifies the calculation and enables us to send data directly to the destination (instead of sending it indirectly via other processors when only a vertex is shared). Fourth, instead of implementing a more complicated logic for transferring individual degrees of freedom we opted to always send all degrees of freedom belonging to a cell and even to accept sending a cell twice, which can only happen for some cells that touch more than one other processor (see step 6). Because we transfer the data of the whole cell, we ensure knowledge of *all* degrees of freedom on ghost cells, not only those on the interface to locally owned cells as described in [Logg 2009] or [Burstedde et al. 2010]; this is necessary for a number of algorithms that need to, for example, evaluate the gradient of the solution on both sides of an interface. While our approach requires sending slightly larger messages, it is overall more efficient because that data does not need to be sent later in an additional communication step. The rationale here is that since the amount of data exchanged is modest in either case, communication cost is dominated by latency rather than message size.

REMARK 2. *Our algorithm always assigns degrees of freedom on the interface between processors to the one with the smallest processor identifier. This results in a slight imbalance: processors with identifiers close to zero tend to own more degrees of freedom than the average, and processors with ranks close to the parallel job size own less, while most processors in the bulk own roughly the average number.*

One may therefore think about constructing a better tie breaker for ownership of degrees of freedom on processor interfaces. `deal.II` *implements such a fairer scheme in a mode where each processor stores the entire mesh, as does the current code of FEniCS/DOLFIN. However, our experiments indicate that at least relatively simple schemes do not pay off, for several reasons. First, when different degrees of freedom on the same edge or face are assigned to two different processors A and B, matrix-vector multiplications require roughly twice the amount of data transfer because the connectivity graph between degrees of freedom is partitioned by cutting more edges than when assigning all degrees of freedom on a complete face to one side alone. Second, determining ownership is easily done without communication in our algorithm above. Third, the workload in downstream parts of the finite element code is typically quite well balanced, as the cost for many operations is proportional to the number of local cells—which* `p4est` *balances perfectly—and not to the number of degrees of freedom. Finally, by enumerating the degrees of freedom on at least one of the cells adjacent to an interface in a natural ordering, we improve cache locality and thus the performance when accessing corresponding data. To evaluate these arguments, we carefully analyzed the distribution of degrees of freedom and observed only a small imbalance in memory consumption in our numerical tests, while we found excellent scalability of our matrix-vector product implementation.*

3.2 Subsets of degrees of freedom

In the following sections, we will frequently need to identify certain subsets of degrees of freedom (by convention by identifying their respective global indices). To this end, let us define the following subsets of the complete set of indices $\mathcal{I} = [0, N)$:

—$\mathcal{I}^p_{l.o.}$ denotes the set of degrees of freedom *locally owned* by processor p. These are all defined on cells in $\mathbb{T}^p_{\text{loc}}$, though some of the degrees of freedom located on the interfaces of these cells with other processors may be owned by the neighboring processor. We

have $n_p = \#\mathcal{I}^p_{l.o.}$, $\bigcup_q \mathcal{I}^q_{l.o.} = \mathcal{I}$, and $\mathcal{I}^p_{l.o.} \cap \mathcal{I}^q_{l.o.} = \emptyset$ for $p \neq q$. Note that following the algorithm described in the previous section, the set of indices in $\mathcal{I}^p_{l.o.}$ is contiguous. However, this is no longer true when degrees of freedom are renumbered later.

—$\mathcal{I}^p_{l.a.}$ denotes the set of degrees of freedom that are *locally active* for processor p. This set contains all degrees of freedom defined on $\mathbb{T}^p_{\text{loc}}$, and $\mathcal{I}^p_{l.a.} \cap \mathcal{I}^q_{l.a.}$ identifies all those degrees of freedom that live on the interface between the subdomains owned by processors p and q if these are neighbors connected by at least one vertex of the mesh.

—$\mathcal{I}^p_{l.r.}$ denotes the set of degrees of freedom that are *locally relevant* for processor p. We define these to be the degrees of freedom that are located on all cells in $\mathbb{T}^p_{\text{loc}} \cup \mathbb{T}^p_{\text{ghost}}$.

These index sets $\mathcal{I}^p_{l.o.} \subset \mathcal{I}^p_{l.a.} \subset \mathcal{I}^p_{l.r.}$ need to be represented in a computer program for the algorithms discussed below. Maybe surprisingly, we have found that the data structures chosen for this have an enormous impact on the efficiency of our programs as the number of queries into these index sets is very large. In particular, we will frequently have to test whether a given index is in an index set, and if it is we will have to determine the position of an index within this set. The latter is important to achieve our goal that no processor should ever hold arrays on all elements of $\mathcal{I}$: rather, we would like to compress these arrays by only storing data for all elements of a set $\tilde{\mathcal{I}} \subset \mathcal{I}$, but for this we need to map global indices into positions in index sets and vice versa. The efficient implementation of such operations is therefore an important aspect, in particular if the index set is not simply a single contiguous range of indices.

In `deal.II`, the `IndexSet` class implements all such queries. It stores an index set as the union $\tilde{\mathcal{I}} = \bigcup_{k=0}^{K} [b_k, e_k)$ of K half open, disjoint, contiguous intervals that we store sorted by their first indices b_k. Here, we denote by $\tilde{\mathcal{I}}$ a generic index set that could, for example, be any of the sets defined above. For isolated indices, we have $e_k = b_k + 1$. This data structure allows to test whether an index is in the set in $\mathcal{O}(\log_2 K)$ operations. However, the determination of the position of a given index i in the set would require $\mathcal{O}(K)$ operations: if k' is the interval in which i is located, i.e. $b_{k'} \leq i < e_{k'}$, then

$$\text{pos}(i, \tilde{\mathcal{I}}) = \sum_{k=0}^{k'-1} (e_k - b_k) + (i - b_{k'}),$$

where the determination of $k' = \min\{k : i < e_k\}$ can be done in parallel to summing over the sizes of intervals. Similarly, computing the value of the mth index in a set $\tilde{\mathcal{I}}$ would require $\mathcal{O}(K)$ operations on average.

We can remove both these bottlenecks by storing with each interval $[b_k, e_k)$ the number $p_k = \sum_{\kappa=0}^{k-1} (e_\kappa - b_\kappa) = p_{k-1} + (e_{k-1} - b_{k-1})$ of indices in previous intervals. We update these numbers at the end of generating an index set, or whenever they have become outdated but a query requires them. Finding the position of index i then only requires finding which interval k' it lies in, i.e. an $\mathcal{O}(\log_2 K)$ operation, and then computing $p_{k'} + i - b_{k'}$. Likewise finding the value of the mth index requires finding the largest $p_k < m$, which can also be implemented in $\mathcal{O}(\log_2 K)$ operations. In summary, storing an index set as a sorted collection $\tilde{\mathcal{I}} \sim \{(b_k, e_k, p_k)_{k=0}^{K}\}$ of triplets allows for efficient implementation of all operations that we will need below.

3.3 Constraints on degrees of freedom

The algorithms described above provide for a complete characterization of the basis of the finite element space on each cell. However, since we allow hanging nodes in our mesh, not every local degree of freedom is actually a global degree of freedom: some are in fact constrained by adjacent degrees of freedom. In general, the construction of such constraints for hanging nodes is not overly complicated and can be found in [Rheinboldt and Mesztenyi 1980; Carey 1997; Šolín et al. 2003; Šolín et al. 2008]; the algorithms used in `deal.II` are described in [Bangerth et al. 2007; Bangerth and Kayser-Herold 2009]. We will here focus on those aspects particular to distributed computations.

Constraints on degrees of freedom typically have the form

$$x_i = \sum_{j=0}^{N-1} c_{ij} x_j + b_i, \qquad i \in \mathcal{I}_c \subset \mathcal{I},$$

where $\mathcal{I}_c$ is the set of constrained degrees of freedom, and the *constraint matrix* c_{ij} is typically very sparse. For hanging nodes, the inhomogeneities b_i are zero; as an example, for lowest order elements the constraints on edge mid-nodes have the form $x_2 = \frac{1}{2}x_0 + \frac{1}{2}x_1$. Constraints may also originate from strongly imposed Dirichlet-type boundary values in the form $x_0 = 42$, for example.

3.3.1 *Which constraints need to be stored.* As in other parts of this paper, it is clear that not every processor will be able to store the data that describes *all* the constraints that may exist on the distributed finite element space. In fact, each processor can only construct constraints for a subset of $\mathcal{I}^p_{l.r.} \cap \mathcal{I}_c$ since it has no knowledge of any of the other degrees of freedom. Consequently, the question here is rather which subset $\mathcal{I}^p_c$ of constraints each processor could in principle construct, and which it needs to construct and store locally for the algorithms described below to work.

For sequential computations, one can first assemble the linear system from all cell contributions irrespective of constraints and in a second step "eliminate" constrained degrees of freedom in an in-place procedure (see, for example, [Bangerth and Kayser-Herold 2009, Section 5.2]). On the other hand, in distributed parallel computations, no processor has access to a sufficient number of matrix rows to eliminate constrained degrees of freedom after the linear system has already been assembled from its cell-wise contributions. Consequently, we have to eliminate constrained degrees of freedom already when copying local contributions into the global linear system. While this may not be quite as elegant, it has the benefit that we know exactly what degrees of freedom we may have to resolve constraints for. Namely, exactly those that may appear in local contributions to the global linear system: if processor p has a contribution to global entry (i, j) of the matrix, then it needs to know about constraints on degrees of freedom i and j. Which these are depends on both the finite element as well as the bilinear form in use.

Local contributions to the global linear system are computed by each processor for all cells $\mathbb{T}^p_{\text{loc}}$ (i.e. for all degrees of freedom in $\mathcal{I}^p_{l.a.}$). For most finite elements and bilinear forms, the local contribution consists of integrals only over each cell $K \in \mathbb{T}^p_{\text{loc}}$ and consequently every processor will only need to know constraints on all degrees of freedom in $\mathcal{I}^p_c = \mathcal{I}_c \cap \mathcal{I}^p_{l.a.}$. Discontinuous Galerkin methods also have jump terms between cells, and consequently need to also know about constraints on degrees of freedom on cells neighboring those that are locally owned; in that case, we need to know about all constraints in

$\mathcal{I}_c^p = \mathcal{I}_c \cap \mathcal{I}_{l.r.}^p$.

3.3.2 *Dealing with chains of constraints.* The considerations above are of only theoretical interest if constraints can be against degrees of freedom that are themselves constrained, i.e. if constraints form chains. This frequently happens in at least two situations. First, it is common in hp-adaptive methods, see for example [Bangerth and Kayser-Herold 2009]. In that case, it is even conceivable that chains of constraints extend to the boundary between ghost and artificial cells. Then, the depth of the ghost layer would need to be extended to more than one layer of cells, thereby also expanding the set $\mathcal{I}_{l.r.}^p$. We will not consider such cases here.

The second, more common situation is if we have Dirichlet boundary conditions on degrees of freedom, i.e. constraints of the form $x_0 = 42$. If another constraint, e.g. $x_2 = \frac{1}{2}x_0 + \frac{1}{2}x_1$, references such a degree of freedom x_0 and if the latter is located in the ghost layer, then we need to know about the constraint on x_0. For this reason, in deal.II each processor always stores all those constraints in $\mathcal{I}_c^p = \mathcal{I}_c \cap \mathcal{I}_{l.r.}^p$ that can be computed on locally owned and ghost cells.

3.3.3 *Computing constraints for hanging nodes.* Of equal importance to the question of which constraints we need to store is the question how we can compute the necessary constraints that result from hanging nodes. Let us first consider the case of continuous elements with only cell integration, i.e. $\mathcal{I}_c^p = \mathcal{I}_c \cap \mathcal{I}_{l.a.}^p$. Since all of these degrees of freedom are adjacent to locally owned cells, it may appear that it is sufficient to compute constraints by only considering hanging nodes at faces between two locally owned cells, or between a locally owned cell and a ghost neighbor. While we believe that this true in two space dimensions, this is not so in 3d. For example, consider the situation depicted in Fig. 3, assuming trilinear finite elements. The degree of freedom indicated by the blue dot is locally active both on processor 0 (white cells) and processor 1 (yellow cells in front). However, since hanging node constraints are computed based on the face between coarse and fine cells, not solely on edges, processor 1 can only know about the constraint on this degree of freedom by computing the constraint on the interface between the white cells, all of which are ghost cells for processor 1.

Since the structure and size of the set $\mathcal{I}_c^p$ depends also on the bilinear form, one can imagine situations in which computing it is even more involved than described in the previous paragraph. For example, if the bilinear form calls for face integrals involving all shape functions from both sides of the face, we would need to have constraints also on all degrees in $\mathcal{I}_{l.r.}^p$ which we may not be able to compute only from a single layer of ghost cells. Fortunately, most discretizations that require such terms have discontinuous shape functions that do not carry constraints on hanging nodes; for a counter example see [Kanschat and Rivière 2010].

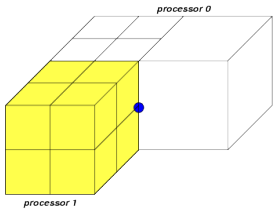

Fig. 3. *Illustration of a situation where constraints need to be computed between two ghost cells.*

3.3.4 *Evaluating constraints.* When copying local contributions into the global matrix and right hand side vector objects during finite element assembly of linear systems, we have to determine for each involved degree of freedom i whether it is constrained or not, and if it is what the coefficients c_{ij}, b_i of its constraint

are. For the sequential case, the `deal.II` class `ConstraintMatrix` stores an array of integers for all degrees of freedom. These integers contain the position of the constraint in the list of all constraints, or -1 for $i \notin \mathcal{I}_c$. This guarantees that the query whether i is constrained can be performed in $\mathcal{O}(1)$, as is actually accessing the constraints.

On the other hand, in the parallel distributed case under consideration here, this strategy is not compatible with our desire to never store arrays on all degrees of freedom on a single processor. Rather, we are presented with two options:

—On each processor, the `ConstraintMatrix` stores a sorted container of $\#\mathcal{I}_c^p$ elements each of which contains the index of the constrained degree of freedom and its constraints. Finding whether index i is constrained and if so accessing its constraints can then be done using $\mathcal{O}\left(\log_2(\#\mathcal{I}_c^p)\right)$ operations.

—On each processor, this class stores an array of $\#\mathcal{I}_{l.r.}^p$ integers. Finding whether an index $i \in \mathcal{I}_{l.r.}^p$ is constrained then requires finding the position r_i of i within $\mathcal{I}_{l.r.}^p$ and testing position r_i in the array whether the integer stored there is -1 (indicating that there are no constraints on i) or otherwise is an index into an array describing the constraints on i. As explained in Section 3.2, finding r_i can be done in $\mathcal{O}(\log_2 K^p)$ operations where K^p is the number of half-open intervals that are needed to describe $\mathcal{I}_{l.r.}^p$.

Here, the second strategy requires a factor of $\#\mathcal{I}_{l.r.}^p/\#\mathcal{I}_c^p$ more memory, but it is cheaper in terms of run time if $K^p \ll \#\mathcal{I}_c^p$. The former is not a significant problem, since storing a single integer for every locally active degree of freedom is not a noticeable expense overall. Whether the latter condition is true depends on a number of application dependent factors: (i) how much local refinement is required to resolve the solution, as this influences $\#\mathcal{I}_c^p$; (ii) the ratio of the number of ghost cells (which roughly determines K^p) to the number of cells (which roughly determines $\#\mathcal{I}_c^p$ up to a factor). The ratio in the second point also depends on the number of refinement steps as well as the number of processors available.

In a number of numerical experiments, we have not been able to conclusively determine which of the two strategies above would be more efficient since the ratio of K^p to $\#\mathcal{I}_c^p$ is highly variable. In particular, neither of these numbers are uniformly much smaller than the other. `deal.II` currently implements the second strategy.

As a final note in this section, let us remark that the strategies described above turn out to be as conservative as one can be with only one layer of ghost cells: we compute even constraints for degrees of freedom located between ghost cells, and we also store the maximal set of constraints available. Coming to the conclusion that both is necessary is the result of many long debugging sessions since forgetting to compute or store constraints does not typically result in failing assertions or other easy to find errors. Rather, it simply leads to the wrong linear system with generally unpredictable, though always wrong, solutions.

4. ALGORITHMS FOR SETTING UP AND SOLVING LINEAR SYSTEMS

After creating the mesh and the index sets for degrees of freedom as discussed above, we can turn to the core objective of finite element codes, namely assembling and solving linear systems. We note that for parallel linear algebra, `deal.II` makes use of `PETSc` [Balay et al. 2008; Balay et al. 2010] and `Trilinos` [Heroux et al. 2005; Heroux et al. 2011], rather than implementing this functionality directly. We will therefore not elaborate on algorithms and data structures for these linear algebra operations, but rather show how distributed finite element programs can interface with such packages to be correct and efficient.

4.1 Setting up sparsity patterns

Finite element discretizations lead to sparse matrices that are most efficiently stored in compressed row format. Both `PETSc` and `Trilinos` allow application programs to preset the sparsity pattern of matrices to avoid re-allocating memory over and over during matrix assembly. `deal.II` makes use of this by first building objects with specialized data structures that allow the efficient build-up of column indices for each row in a matrix, and then bulk-copying all the indices in one row into the respective `PETSc` or `Trilinos` matrix classes. Both of these libraries can then store the actual matrix entries in a contiguous array in memory. We note here that each processor will only store matrix and vector rows indexed by $\mathcal{I}^p_{l.o.}$ when using either `PETSc` or `Trilinos` objects. Since $\mathcal{I} = \bigcup_p \mathcal{I}^p_{l.o.}$ and the sets $\mathcal{I}^p_{l.o.}$ are mutually disjoint, we achieve a non-overlapping distribution of rows between the available processors.

In the current context, we are interested in how pre-computing sparsity patterns can be achieved in a parallel distributed program. We can build the sparsity pattern if every processor loops over its own cells in $\mathbb{T}^p_{\text{loc}}$ and simulates which elements of the matrix would be written to if we were assembling the global matrix from local contributions. It is immediately clear that we will not only write into rows r that belong to the current processor (i.e., $r \in \mathcal{I}^p_{l.o.}$), but also into rows r that correspond to degrees of freedom owned by a neighboring processor q but located at the boundary (i.e., $r \in \mathcal{I}^p_{l.a.} \cap \mathcal{I}^q_{l.o.}$), and last but not least into rows which the degree of freedom r may be constrained to (these rows may lie in $\mathcal{I}^p_{l.r.} \cap \mathcal{I}^q_{l.o.}$).

It would therefore seem that processor p needs to communicate to processor q the elements it will write to in these rows in order for processor q to complete the sparsity pattern of those rows that it locally stores. One may now ask whether it is possible for processor q to determine which entries in rows corresponding to $\mathcal{I}^p_{l.a.} \cap \mathcal{I}^q_{l.o.}$ will be written to by processor p, thereby avoiding communication. This is, in fact, possible as long as there are no constrained degrees of freedom: each processor will simply have to loop over all cells $\mathbb{T}^p_{\text{loc}} \cup \mathbb{T}^p_{\text{ghost}}$, simulate assembly of the matrix, and only record which elements in rows $\mathcal{I}^p_{l.o.}$ will be written to, ignoring all writes to other rows.

Unfortunately, this process does no longer work once constraints are involved, since processors cannot always know all involved constraints. This is illustrated in Fig. 4. Consider the situation that the bottom three cells are owned by processor 0, and the rest by processor 1. Then $\mathcal{I}^0_{l.o.} = [0, 8], \mathcal{I}^1_{l.o.} = [9, 20]$, and these two processors will store constraints for $\mathcal{I}^0_c = \{6, 17, 19\}, \mathcal{I}^1_c = \{6, 17, 19, 20\}$ as explained in Section 3.3.2.

Consider now the matrix entries that processor 1 will have to write to when assembling on cell B (shaded yellow). Since degrees of freedom 17 and 20 are constrained to 5,10 and 10,11, respectively, after resolution of constraints we will have matrix entries $(5, 10)$ and $(5, 11)$, among others. But because $5 \in \mathcal{I}^0_{l.o.}$, these entries need to be stored on processor 0. The question now is whether processor 0 could know about this without communicating with processor 1. The answer is no: we could have known about entry $(5, 10)$ by simulating assembly on cell A, which is a ghost cell on processor 0. However, processor 0 cannot possibly know about the matrix entry $(5, 11)$: the cells B and C are not in $\mathbb{T}^0_{\text{ghost}}$, and so processor 0 does not know anything about degree of freedom 20 in the first place, and certainly not that it is constrained to degrees of freedom 10 and 11.

In summary, we cannot avoid communicating entries into the sparsity pattern between processors, though at least this communication can be implemented point-to-point. We

note that in the case of `Trilinos`, the `Epetra_FECrsGraph` class (implementing sparsity patterns) can take care of this kind of communication: if we add elements to rows of the sparsity pattern that are not stored on the current processor, then these will automatically be transferred to the owning processor upon calling `Epetra_FECrsGraph`'s `GlobalAssemble` function. A similar statement holds for `PETSc` objects though there does not seem to be a way to communicate entries of sparsity patterns between processors. Consequently, when interfacing with `PETSc`, we send the entries generated in rows that are not locally owned to the corresponding processor after concluding creation of the sparsity pattern. This way each processor sends one data packet with indices to each of its neighboring processors. The process is fast because each processor only has to look at the rows with indices $\mathcal{I}_{l.r.} \setminus \mathcal{I}_{l.o.}$ and all communication can be done point-to-point in a non-blocking fashion. Received indices are then inserted into the local rows of the sparsity pattern.

4.2 Assembling the linear system

After pre-setting the sparsity pattern of the matrix, assembling the linear system happens in the usual way by computing contributions from all cells in $\mathbb{T}^p_{\text{loc}}$, resolving constraints, and transferring the entries into the global matrix and vector objects. For the same reasons as discussed above, some communication cannot be avoided, but both `PETSc` and `Trilinos` automatically take care of communicating matrix entries to the correct processors at the end of the assembly process.

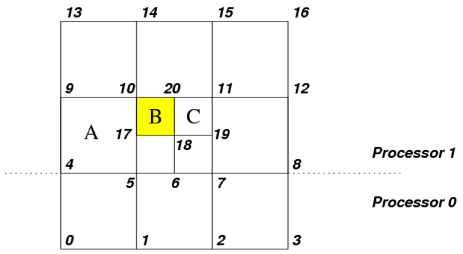

Fig. 4. *Illustrating why sparsity patterns cannot be built up without communication: Degrees of freedom associated with a Q_1 finite element on a mesh split between two processors. Processor 0 owns degrees of freedom $0 \ldots 8$, processor 1 owns $9 \ldots 20$.*

4.3 Solving the linear system

Once assembled, we need to solve the resulting linear system that can contain billions of unknowns. Both `PETSc` and `Trilinos` offer a large variety of solvers, including Krylov-space methods and all of the commonly used preconditioners, including highly effective algebraic multigrid preconditioners available through the packages `hypre` [Falgout et al. 2005; 2006] and `ML` [Gee et al. 2006].

5. POSTPROCESSING

Once a solution to the linear system has been computed, finite element applications typically perform a number of postprocessing steps such as generating graphical output, estimating errors, adaptively refining the mesh, and interpolating the solution from the old to the new mesh. In the following, we will briefly comment on the latter three of these points. We will not discuss generating graphical output—storing and visualizing tens or hundreds of gigabytes of data resulting from massively parallel computations is nontrivial and the realm of specialized tools not under consideration here.

5.1 Adaptive refinement of meshes

Once a solution has been computed, we frequently want to adjust the mesh to better resolve the solution. In order to drive this adaptation, we need to (i) compute error indicators for each of the cells in the global mesh, and (ii) determine which cells to refine, for example by setting a threshold on the error indicators above which a cell should be refined (and similarly for coarsening).

The literature contains a large number of methods to estimate the error in finite element solutions, see for example [Verfürth 1994; Ainsworth and Oden 2000; Bangerth and Rannacher 2003] and the references cited in these publications. Without going into detail, it is natural to let every processor p compute error indicators for the cells $\mathbb{T}^p_{\text{loc}}$ it owns. The primary complication from the perspective of parallelization is that in order to compute these indicators, we not only have to have access to all degrees of freedom located on cells in $\mathbb{T}^p_{\text{loc}}$, i.e. to the elements of the solution vector indexed by $\mathcal{I}^p_{l.a.}$, but frequently also solution values on all neighboring cells in order to compute jump residuals at the interfaces between cells. In other words, we need access to solution vector elements indexed by $\mathcal{I}^p_{l.r.}$ while by default every processor only stores solution vector elements it owns, i.e. $\mathcal{I}^p_{l.o.}$. Before computing error indicators, we therefore have to import the missing elements (and preferably only those since, in particular, we cannot expect to store the entire solution vector on each processor). Both `PETSc` and `Trilinos` support this kind of operation.

Once error indicators $e_i \geq 0, i \in [0, N_{\text{cells}})$, where $N_{\text{cells}} = \#\bigcup_p \mathbb{T}^p_{\text{loc}}$, have been computed, we have to decide which cells to refine and coarsen. A typical strategy is to refine and coarsen certain fractions $\alpha_r, \alpha_c \in [0, 1]$ of all cells. To do that, we need to compute thresholds θ_r, θ_c so that, for example, $\#\{i : e_i \geq \theta_r\} \approx \alpha_r N_{\text{cells}}$. On a single processor, this is easily achieved by sorting the e_i according to their size and choosing that error indicator as the threshold θ_r corresponding to position $\alpha_r N_{\text{cells}}$, though it is also possible to find this threshold without completely sorting the set of indicators e_i. This task can be performed using the algorithm commonly referred to as `nth_element`, which can be implemented with average linear complexity, and is, for example, part of the C++ standard library [Stroustrup 1997]. On the other hand, `nth_element` does more than we need since it also shuffles the elements of the input sequence so that they are ordered relative to the n-th element we are seeking.

In distributed parallel computations, no single processor has access to all error indicators. Consequently, we could use a parallel `nth_element` algorithm, see for example [Tikhonova et al. 2005]. We can, however, avoid the partial sorting step by using the distributed algorithm outlined in Figure 5. The algorithm computes the threshold θ to an accuracy ϵ. For practical reasons, we are not usually interested in very high accuracy for these thresholds and typically set ϵ so that the while-loop terminates after, for example, at most 25 iterations. Since the interval in which θ must lie is halved in each iteration, this corresponds to a relative accuracy of $\frac{1}{2^{25}} \approx 3 \times 10^{-8}$. The compute time for the algorithm with a fixed maximal number of iterations is then $\mathcal{O}(\frac{N_{\text{cells}}}{P} \log_2 P)$, where the logarithmic factor results from the global reduce and broadcast operations. Furthermore, the constant in this complexity can be improved by letting each processor not only compute the number of cells $n_t^{1/2} = \#\{i : e_i^0 \geq m = \frac{1}{2}(b+e)\}$, but also $n_t^{1/4} = \#\{i : e_i^0 \geq \frac{1}{4}(b+e)\}$ and $n_t^{3/4} = \#\{i : e_i^0 \geq \frac{3}{4}(b+e)\}$, thereby obviating the need for any communication in the next iteration (because the data needed in the next iteration is already available) and cutting the number of communication steps in half. This procedure can of course be re-

$b^p = \min e^p, e^p = \max e^p$	compute local min and max
$\{b, -e\}$ = MPI_Reduce ($\{b^p, -e^p\}$, MPI_MIN)	compute global min and max on processor 0
while ($\|e - b\| \geq \epsilon$)	
if (p==0) then	master processor
MPI_Bcast ($\{b, e\}$, $0 \to$ all)	broadcast current interval
$m = \frac{1}{2}(b + e)$	compute interval split point
$n_t = \#\{i : e_i^0 \geq m\}$	count local elements greater than m
n_t =MPI_Reduce (n_t, MPI_SUM)	count total number of elements
if ($n_t > \alpha N$) then $\{b, e\} = \{b, m\}$	adjust interval
else $\{b, e\} = \{m, e\}$	
else	worker processor
MPI_Bcast ($\{b, e\}$, $0 \to$ all)	receive current interval
$m = \frac{1}{2}(b + e)$	compute interval split point
$n_t = \#\{i : e_i^p \geq m\}$	count local elements greater than m
MPI_Reduce (n_t, MPI_SUM)	accumulate total number of elements on proc. 0
endif	
endwhile	
return $\theta = m$	return threshold

Fig. 5. *Pseudo-code for determining a threshold θ so that approximately αN elements of a vector $(e_i)_{i=0}^{N-1}$ satisfy $e_i \geq \theta$. Each processor only stores a part e^p of n^p elements of the input vector. The algorithm runs on each processor $p, 0 \leq p < P$. This algorithm is a variant of the parallel binary search described in [Burstedde et al. 2008].*

peated to reduce the number of communication steps even further, at the expense of larger numbers of variables n_t sent to processor 0 in the reduction step. Note that the combination of `MPI_Reduce` and `MPI_Bcast` could be replaced by `MPI_Allreduce` as done in [Burstedde et al. 2008, Section 3.1].

In actual finite element computations, the algorithm as stated turns out to not be very efficient. The reason for this is that for practical problems, error indicators e_i are often scattered across many orders of magnitude, with only large e_i. Consequently, reducing the interval to $\frac{1}{2^{25}}$ of its original size does not accurately determine a useful threshold value θ. This problem can be avoided by using a larger number of iterations. A better alternative is to exploit the fact that the numbers $\log e_i$ are much more uniformly distributed than e_i; one can then choose $m = \exp\left[\frac{1}{2}(\log b + \log e)\right] = \sqrt{be}$. We use this modification in our code if $b > 0$, with at most 25 iterations.

The algorithm outlined above computes a threshold so that a certain fraction of the cells are refined. A different strategy often used in finite element codes is to refine those cells with the largest indicators that together make up a certain fraction αe of the total error $e = \sum_i e_i$. This is easily achieved with minor modifications when determining n_t. In either of these two cases, once thresholds θ_c, θ_r have been computed in this way, each processor can flag those among its cells $\mathbb{T}^p_{\text{loc}}$ whose error indicators are larger than θ_r or smaller than θ_c for refinement or coarsening, respectively.

5.2 Transferring solutions between meshes

In time dependent or nonlinear problems, it is important that we can carry the solution of one time step or nonlinear iteration from one mesh over to the next mesh that we obtain by refining or coarsening the previous one. This functionality is implemented in the `deal.II` class `SolutionTransfer`. It relies on the fact that after setting refinement and coarsening flags, we can determine exactly which cells will be refined and which will

be coarsened (even though these sets of cells may not coincide with the ones actually flagged, for example because the triangulation has to respect the 2:1 mesh balance invariant).

Since the solution transfer is relatively trivial if all necessary information is available locally, we describe the sequential algorithm first before discussing the modifications necessary for a scalable parallel implementation. To this end, let $x^i, i = 1 \dots I$ be the vectors that we want to transfer to the new mesh. Then the sequential algorithm begins as follows:

—On every terminal (active) cell K that will not be coarsened, collect the values $x^i|_K$ of all degrees of freedom located on K. Add the tuple $(K, \{x^i|_K\}_{i=1}^I)$ to a list of such tuples.

—On every non-terminal cell K that has 2^d terminal children $K_c, c = 1 \dots 2^d$ that will be coarsened, interpolate or project the values from the children onto K and call the result $x^i|_K$. Add the tuple $(K, \{x^i|_K\}_{i=1}^I)$ to a list of such tuples.

Next refine and coarsen the triangulation, enumerate all degrees of freedom on the new mesh, resize the vectors x^i to their correct new sizes and perform the following actions:

—On every terminal cell K, see if an entry for this cell exists in the list of tuples. If so, which will be the case for all cells that have not been changed at all and those whose children have been deleted in the previous coarsening step, extract the values of the solution $x^i|_K$ on the current cell and copy them into the global solution vectors x^i.

—On all non-terminal cells K for which an entry exists in the list of tuples, i.e. those that have been refined exactly once, extract the local values $x^i|_K$, interpolate them to the children $x^i|_{K_c}, c = 1 \dots 2^d$, and copy the results into the global solution vectors x^i.

By ordering the list of tuples in the same way as we traverse cells in the second half of the algorithm, we can make both adding an element to the list and finding tuples in the list an $\mathcal{O}(1)$ operation.

This algorithm does not immediately work for parallel distributed meshes, first because we will not be able to tell exactly which cells will be refined and coarsened without communication (a precondition for the first part of the algorithm), and second because, due to repartitioning, the cells we have after refinement on processor p may not be those for which we stored tuples in the list before refinement on the same processor.

In our parallel distributed re-implementation of the `SolutionTransfer` class, we make use of the fact that the master version of the mesh is maintained by `p4est` and stored independently of the `deal.II` object that represents the mesh including all auxiliary information. Consequently, after `deal.II` notifies `p4est` of which cells to refine and coarsen, and `p4est` performs the necessary mesh modification including the 2:1 mesh balance (see Section 2.3), we have the opportunity to determine which cells have been refined and coarsened by comparing the modified, `p4est`-maintained master version of the mesh and the still unchanged mesh data in `deal.II`. This allows us to create the list of tuples in the first part of the algorithm outlined above. In a second step, `deal.II` calls `p4est` to repartition the mesh to ensure a load-balanced distribution of terminal cells; in this step, `p4est` allows attaching additional data to cells that are transferred point-to-point from one processor to another. In our case, we attach the values $x^i|_K$. After partitioning the mesh and re-building the deal.II triangulation, we query `p4est` for the stored values on the machine the cell now belongs to. This allows us to perform the second part of the algorithm like in the serial case, without adding communication to `deal.II` itself.

6. MACHINE ARCHITECTURE CONSIDERATIONS

In the description of algorithms and data structures above, we have assumed a rather abstract hardware implementation of a parallel machine whose processor cores are coupled through MPI. In particular, we have not taken into consideration that some MPI processes may be "closer" than others (for example because they run on different cores of the same processor). Neither did we consider that actual program speed depends significantly on data layout, for example so that we can utilize on-chip vector instructions, or employ off-chip support through general-purpose graphics processing units (GPUs) that is present in a number of high-end machines today.

While our numerical results in Section 7 below show that this omission does not affect the *scalability* of our approach, it may affect the *performance*. Consequently, it would be naive to ignore these considerations given that the "flat" MPI model cannot be expected to work on future machines with millions of processor cores. This notwithstanding, we believe that the approach we have chosen is appropriate for the vast majority of large machines available today and in the near future. We will discuss this in the following subsections for hybrid computing, GPU acceleration, and vectorization—today's three primary directions pointing beyond the "flat", homogeneous MPI model.

Hybrid computing. Hybrid computing refers to the use of multiple processor cores with shared memory that form the nodes of a distributed memory machine. In the rest of this paper, we have assumed that each core hosts a separate, single-threaded MPI process. One might imagine that it is more efficient to have a single, multithreaded MPI process per machine. In fact, our implementation in `deal.II` can be used in this way since many time-consuming operations (such as assembly and error estimation) are already parallelized using the Threading Building Blocks [Reinders 2007], or could relatively easily be parallelized (e.g. the creation of sparsity patterns or of constraints). However, there are practical obstacles to this approach. For example, MPI implementations tie processes to individual cores, ensuring fast access to data that is stored in processor-adjacent memory whenever possible. On the other hand, multithreaded processes have to deal with the non-uniform memory access latency if data is stored in the same address space but in memory chips adjacent to a different processor; starting tasks on new threads almost always also leads to higher cache miss rates. The biggest obstacle, however, is that while our methods scale very well in the "flat" MPI model, they would only scale well in a hybrid model if *all* parts support multithreaded operation. Unfortunately, this is not the case: in our numerical experiments, we spend 80-90% of the compute time in solvers provided by either PETSc or Trilinos, and while those scale well through MPI, neither supports hybrid models. In other words, utilizing `deal.II`'s multithreading capabilities will only make sense once linear solver packages can also do so.

GPU support. While graphics processing units are poorly suited to accelerate the complex and if-then-else laden integer algorithms that form the focus of this paper, they are ideally suited to accelerate the floating-point-focused linear solvers that consume the bulk of the compute time in our numerical examples. Again, we have not discussed this interaction in more detail since we use external packages as black box solvers to solve our linear systems. Once these packages learn to use GPUs, our programs will benefit as well.

Vectorization and streaming. Most CPUs today also have vector operations, though on a finer scale than GPUs. Their use is, again, primarily confined to the external solver

packages. At the same time, vectorization is only possible if data can be streamed into processors, i.e., if data is arranged in memory in a linear fashion. `deal.II` goes to great lengths to arrange "like" data in linear arrays rather than scattered data structures, and in the same order in which cells are traversed in most operations; consequently, it has been shown to have a relatively low cache miss rate compared to other scientific computing applications (see, for example, the comparison of SPEC CPU 2006 programs—including `447.dealII`—in [Henning 2007]). Similarly, enumerating degrees of freedom in such a way that vector entries corresponding to neighboring degrees of freedom are adjacent in memory ensures low cache miss rates; not by coincidence, p4est's use of a space filling curve to enumerate cells ensures this property very well, as does our algorithm to assign degrees of freedom to individual processors (see Remark 2).

In summary, the current lack of support for hybrid and GPU-accelerated programming models in widely used external solver packages prevents us from using such approaches in our implementation. At the same time, several of the algorithms discussed here can efficiently be parallelized using multiple threads once this becomes necessary. Finally, our numerical results in Section 7 show that our methods scale well on a contemporary supercomputer and that the limits of the "flat" MPI model have not yet been reached.

7. NUMERICAL RESULTS

In the following, we will present two test cases that are intended to demonstrate the scalability of the algorithms and data structures discussed above. The first test case solves a 2d Laplace equation on an sequence of adaptively refined meshes. The relative simplicity of this example implies that the solver and preconditioner for the linear system—while still the most expensive part of the program—are not completely dominating. Consequently, we will be able to better demonstrate the scalability of the remaining parts of the program, namely the algorithms discussed in this paper. The second test case investigates the solution of a viscous thermal convection problem under the Boussinesq approximation.

The programs that implement these test cases will be made available as the step-40 and step-32 tutorial programs of `deal.II`, respectively. Tutorial programs are extensively documented to demonstrate both the computational techniques used to solve a problem as well as their implementation using `deal.II`'s classes and functions. They are licensed in the same way as the library and serve well as starting points for new programs.

The computational results shown in the following subsections were obtained on the Ranger supercomputer at the Texas Advanced Computing Center (TACC) at The University of Texas at Austin. Some computations and the majority of code testing were done on the Brazos and Hurr clusters at the Institute for Applied Mathematics and Computational Science at Texas A&M University.

7.1 A simple 2d Laplace test case

The first test case solves the scalar Laplace equation, $-\Delta u = f$ on the unit square $\Omega = [0,1]^2$. We choose homogeneous boundary values and

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } x_2 > \frac{1}{2} + \frac{1}{4}\sin(4\pi x_1), \\ -1 & \text{otherwise.} \end{cases}$$

The discontinuity in the right hand side leads to a sinusoidal line through the domain along which the solution $u(\mathbf{x})$ is non-smooth, resulting in very localized adaptive mesh refinement. The equations are discretized using biquadratic finite elements and solved using

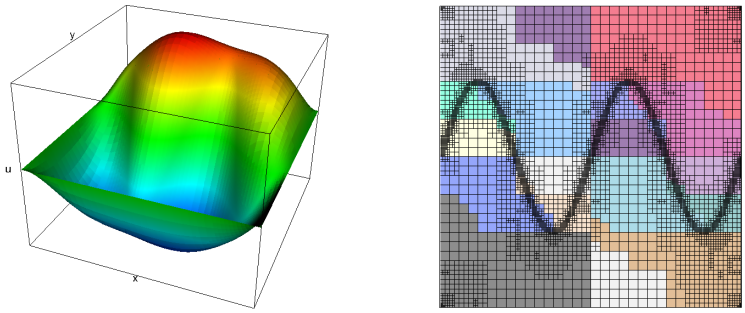

Fig. 6. *Two-dimensional scalar Laplace example. Left: Solution u on the unit square. Right: Adapted mesh at an early stage with 7,069 cells. The partition between 16 processors is indicated by colors.*

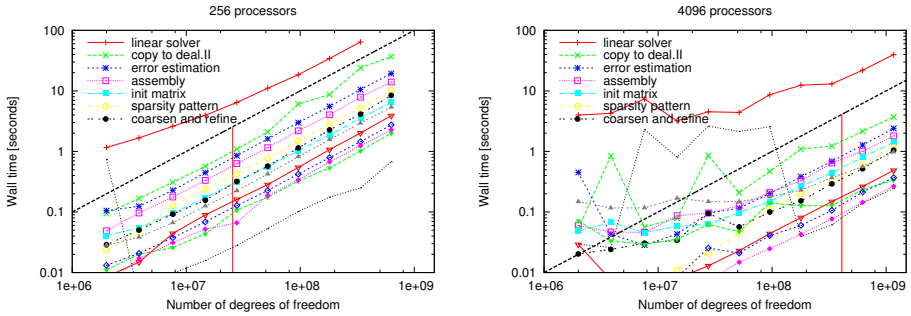

Fig. 7. *Two-dimensional scalar Laplace example. Scaling results on 256 (left) and 4,096 processors (right) for a sequence of successively refined grids. The various categories of wall clock times are explained in the text. The labeled categories together account for more then 90% of the total wall clock time of each cycle. In both graphs, the thick, dashed line indicates linear scaling with the number of degrees of freedom. Each processor has more than 10^5 degrees of freedom only to the right of the vertical red line. Both the small number of elements per processor left of the vertical line and small absolute run times of a few seconds make the timings prone to jitter.*

the conjugate gradient method preconditioned by the `BoomerAMG` implementation of the algebraic multigrid method in the `hypre` package [Falgout et al. 2005; 2006]. We call `hypre` through its interface to `PETSc`. Fig. 6 shows the solution along with an adaptive mesh at an early stage of the refinement process containing 7,069 cells and a partition onto 16 processors.

To demonstrate the scalability of the algorithms and data structures discussed in this paper, we solve the Laplace equation on a sequence of meshes each of which is derived from the previous one using adaptive mesh refinement and coarsening (the mesh in Fig. 6 results from three cycles of adaptation). For a given number of processors, we can then show the wall clock time required by the various operations in our program as a function of the number of degrees of freedom on each mesh in this sequence. Fig. 7 shows this for 256 and 4,096 processors and up to around 1.2×10^9 degrees of freedom.[2] While we have

[2]Note that the next refinement would yield a number of degrees of freedom that exceeds the range of the 32-bit signed integers used by `hypre` for indexing (`PETSc` can use 64-bit integers for this purpose). Unfortunately, `Trilinos`' `Epetra` package that we use in our second numerical test case suffers from the same limitation.

measured wall clock times for a large number of parts of the program, the graph only labels those seven most expensive ones that together account for more than 90% of the overall time. However, as can be seen, even the remaining parts of the program scale linearly. The dominant parts of the program in terms of their wall clock time are:

—*Linear solver:* Setting up the algebraic multigrid preconditioner from the distributed finite element system matrix, and solving the linear system with the conjugate gradient method including the application of the AMG preconditioner.

—*Copy to* `deal.II`*:* This is the operation that recreates the mesh in `deal.II`'s own data structures from the more compressed representation in `p4est`. The algorithm is shown in Fig. 1.

—*Error estimation:* Given the solution of the linear system, compute and communicate error indicators for each locally owned cell, compute global thresholds for refinement and coarsening, and flag cells accordingly (see the algorithm in Fig. 5).

—*Assembly:* Assembling the contributions of locally owned cells to the global system matrix and right hand side vector. This includes the transfer of matrix and vector elements locally computed but stored on other processors.

—*Sparsity pattern:* Determine the locations of non-zero matrix entries as described in 4.1.

—*Init matrix:* Exchange between processors which non-locally owned matrix entries they will write to in order to populate the necessary sparsity pattern for the global matrix. Copy intermediate data structures used to collect these entries into a more compact one and allocate memory for the system matrix.

—*Coarsen and refine:* Coarsen and refine marked cells, and enforce the 2:1 cell balance across all cell interfaces (this includes the largest volume of communication within `p4est`; see Section 2.3).

The results presented in Fig. 7 show that all operations appear to scale linearly (or better) with the number of degrees of freedom whenever the number of elements per processor exceeds 10^5. For smaller element counts per processor, and run times of under a few seconds, most operations behave somewhat irregularly—in particular in the graph with 4,096 processors—, which can be attributed to the fact that in this situation there is simply not enough numerical work to hide the overhead and inherent randomness caused by communication. This behavior is most marked in the Copy-to-`deal.II` and `p4est` repartitioning operations. (The scalability of the latter has been independently demonstrated in [Burstedde et al. 2010].)

While the results discussed above show that a fixed number of processors can solve larger and larger problems in a time proportional to the problem's size, Fig 8 shows the results of a "strong" scaling experiment. Here we select two refinement levels that result in roughly 52 and 335 million unknowns, respectively, and compare run times for different numbers of processors. Again, above roughly 10^5 elements per processor we observe nearly ideal scalability of all algorithms discussed in this paper.

7.2 A thermal convection test case

The second test case considers solving the equations that describe convection driven by buoyancy due to temperature variations. We model this phenomenon involving velocity, pressure, and temperature variables $\mathbf{u}, p, T$ using the Boussinesq approximation [McKen-

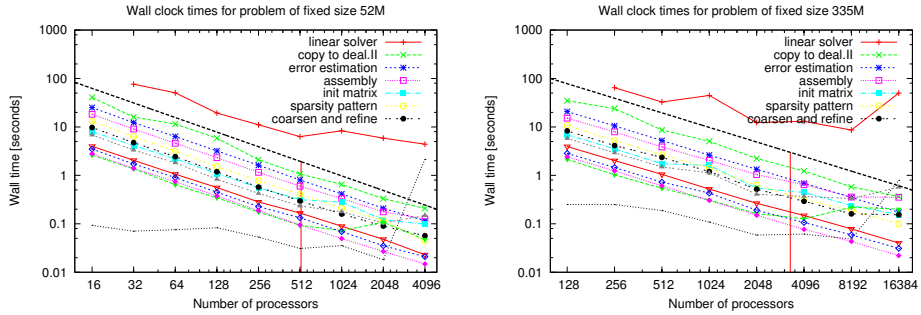

Fig. 8. *Two-dimensional scalar Laplace example. Strong scaling results for a refinement level at which meshes have approximately 52 million (left) and 335 million unknowns (right), for up to 16,384 processors. The thick, dashed line indicates linear scaling with the number of processors. Each processor has more than 10^5 degrees of freedom only to the left of the vertical red line.*

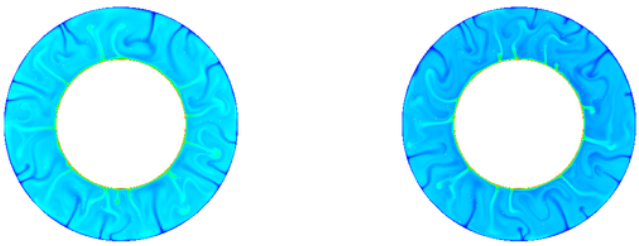

Fig. 9. *Solution of the mantle convection test case at two instants in time during the simulation. Mesh adaptation ensures that the plumes are adequately resolved.*

zie et al. 1974; Schubert et al. 2001],

$$
\begin{aligned}
-\nabla \cdot (2\eta\varepsilon(\mathbf{u})) + \nabla p &= -\rho\beta T\mathbf{g}, \\
\nabla \cdot \mathbf{u} &= 0, \\
\frac{\partial T}{\partial t} + \mathbf{u}\cdot\nabla T - \nabla\cdot\kappa\nabla T &= \gamma.
\end{aligned}
$$

Here, $\varepsilon(\mathbf{u}) = \frac{1}{2}[(\nabla\mathbf{u}) + (\nabla\mathbf{u})^T]$ is the symmetric gradient of the velocity, η and κ denote the viscosity and diffusivity coefficients, respectively, which we assume to be constant in space, ρ is the density of the fluid, β is the thermal expansion coefficient, γ represents internal heat sources, and $\mathbf{g}$ is the gravity vector, which may be spatially variable. These equations are posed on a spherical shell mimicking the earth mantle, i.e., the region above the liquid iron outer core and below the solid earth crust. Dimensions of the domain, boundary and initial conditions, and values for the physical constants mentioned above can be found in the description of the step-32 tutorial program that implements this test case. Typical solutions at two time steps during the simulation are shown in Fig. 9.

We spatially discretize this system using $Q_2^d \times Q_1 \times Q_2$ elements for velocity, pressure and temperature elements, respectively, and use a nonlinear artificial viscosity scheme to stabilize the advection equation for the temperature. We solve the resulting system in time

step n by first solving the Stokes part,

$$\begin{pmatrix} A_U & B \\ B^T & 0 \end{pmatrix} \begin{pmatrix} U^n \\ P^n \end{pmatrix} = \begin{pmatrix} F_U^n \\ F_P^n \end{pmatrix}, \tag{1}$$

and then using an explicit BDF-2 time stepping scheme to obtain the discretized temperature equation at time step n:

$$(M + \alpha^n A_T)T^n = F_T^n. \tag{2}$$

Here, F_U^n, F_P^n, F_T^n are right hand side vectors that depend on previously computed solutions. α^n is a coefficient that depends on the time step length. A_T is a matrix that results from natural and artificial diffusion of the temperature and M is the mass matrix on the temperature space.

We solve the Stokes system (1) using Flexible-GMRES and the Silvester-Wathen preconditioner [Silvester and Wathen 1994]

$$\begin{pmatrix} \tilde{A}_U^{-1} & B \\ 0 & \tilde{S}_P^{-1} \end{pmatrix}$$

where $\tilde{A}_U^{-1}, \tilde{S}_P^{-1}$ are approximations of the inverse of the elliptic stress operator A_U in the Stokes system and the pressure Schur complement $S = B^T A_U^{-1} B$, respectively. We implement $\tilde{A}_U^{-1}$ by solving the corresponding linear system using BiCGStab and the ML implementation [Gee et al. 2006] of the algebraic multigrid method as preconditioner. $\tilde{S}_P^{-1}$ is obtained by solving a linear system with the pressure mass matrix, using an ILU decomposition of this matrix as a preconditioner. This scheme resembles the one also chosen in [Geenen et al. 2009].

The temperature system (2) is solved using the CG method, preconditioned by an incomplete Cholesky (IC) decomposition of the temperature system matrix. Note that the ILU and IC preconditioners are implemented in block Jacobi fashion across the range of different processors, i.e. all coupling between different processors is neglected.

As expected for simulations of reasonably realistic physics, the resulting scheme is heavily dominated by the linear solver, which has to be invoked in every time step whereas the mesh and DoF handling algorithms are only called every tenth time step when the mesh is changed. On the other hand, the highly unstructured mesh and the much larger number of couplings between degrees of freedom for this vector-valued problem impose additional stress on many parts of our implementation.

Fig. 10 shows scaling results for this test case. There, we time the first time step with $t_n \geq t^* = 10^5$ years for a number of different computations with a variable number of cells (and consequently a variable number of time steps before we reach t^*). The "weak" scaling shown in the left panel indicates that all operations scale linearly with the overall size of the problem, at least if the problem is sufficiently large. The right panel demonstrates strong scalability. Here, scalability is lost once the number of degrees of freedom per processor becomes too small; this happens relatively soon due to the small size of the problem shown here (22 million unknowns overall).

8. CONCLUSIONS

In this paper, we present a set of algorithms and data structures that enable us to parallelize all computations associated with adaptive finite element methods. The design of

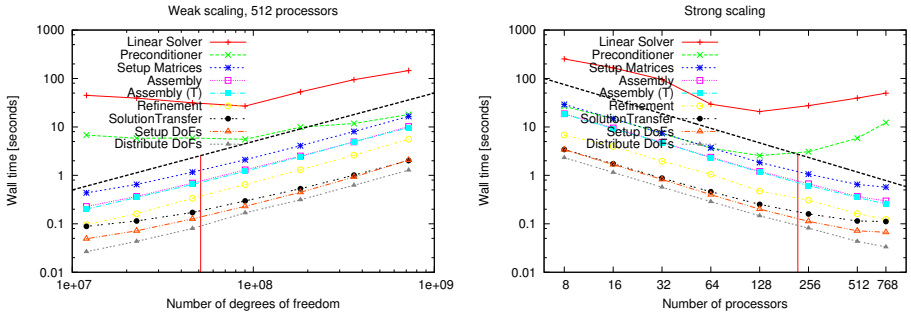

Fig. 10. *Thermal convection example. Weak scaling with 512 processors (left) and strong scaling with roughly 22 million unknowns (right). In both graphs, the thick, dashed line indicates optimal scaling. In the left graph, processors have more than* 10^5 *degrees of freedom to the right of the vertical red line; in the right graph to the left of the vertical red line.*

our methods is based on a complete distribution of all data structures and avoiding global communication wherever possible in favor of scatter/gather and point-to-point operations. However, the key to making this work in practice is not only to distribute storage but also to think about the details of finite element applications—such as constraints on degrees of freedom or sparsity patterns—and in particular what kind of information each processor needs. The question of what processors need to know about what happens on ghost cells turns out to be crucial to the correctness of resulting programs.

The numerical results presented in Section 7 demonstrate that we can achieve optimal scalability for those components of the two finite element applications that are within the scope of this paper. In particular, we could show that all operations scale linearly with the overall problem size (with fixed number of processors) and with the number of processors (with fixed problem size, at least for large enough problems). While we have only shown results for up to 16,384 processors, these scaling results indicate that the methods presented here are likely to scale significantly further once the required technology for the solution of even larger problems becomes available in linear solver packages.

Given the complexity and size of typical finite element libraries, it must necessarily be a significant concern to convert such codes to support massively parallel computations. In Section 2 we lay out one avenue to perform such a conversion. There, we show how an existing finite element library can be extended to support fully distributed meshes by using an external scalable adaptive-mesh provider which we refer to as an "oracle". This oracle has to answer a small number of relatively simple questions, and to encapsulate mesh modification directives for refinement and coarsening, 2:1 mesh balance, and re-partitioning. In our case, the interface between the `deal.II` library and the oracle `p4est` has only around 600 lines of code (lines of C++ code with a semicolon on it), and the entire extension of `deal.II` to support distributed parallel computations required us to write only approximately 10,000 lines of code (some 2,000 of which have a semicolon, the majority of the rest being comments and class or member documentation). The external `p4est` implementation contains about 25,000 lines of code. These numbers have to be compared with the overall size of the `deal.II` library of currently approximately 540,000 lines and its average growth per month of 4,000 lines. A major benefit of this approach is that it allows us to re-use almost all of the existing code that has been validated for years.

Generally, we believe that our implementation of the methods introduced here can be used with relative ease by applications developers. For example, the step-40 tutorial program used in Section 7.1 has only around 150 lines of code, which can be compared to the 125 lines in its sequential predecessor step-6 upon which it is based. As a consequence, we believe that our work not only shows the efficiency of our approach with respect to scaling to very large problems, but also the possibility of implementing these methods efficiently with respect to code complexity. We also hope that this article may serve as a guideline to realize dynamic mesh parallelization within other numerical software packages, and thus can help making true scalability available to an even broader range of scientists.

The results of our work are available under an open source license starting with release 7.0 of the `deal.II` library, as well as through two extensively documented tutorial programs—step-32 and step-40—that explain the use of these techniques and their implementation.

REFERENCES

AINSWORTH, M. AND ODEN, J. T. 2000. *A Posteriori Error Estimation in Finite Element Analysis*. John Wiley and Sons.

BALAY, S., BUSCHELMAN, K., EIJKHOUT, V., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., SMITH, B. F., AND ZHANG, H. 2008. PETSc users manual. Technical Report ANL-95/11 - Revision 3.0.0, Argonne National Laboratory.

BALAY, S., BUSCHELMAN, K., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., SMITH, B. F., AND ZHANG, H. 2010. PETSc Web page. http://www.mcs.anl.gov/petsc.

BANGERTH, W., HARTMANN, R., AND KANSCHAT, G. 2007. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw. 33,* 4, 24/1–24/27.

BANGERTH, W. AND KANSCHAT, G. 2011. `deal.II` *Differential Equations Analysis Library, Technical Reference*. `http://www.dealii.org/`.

BANGERTH, W. AND KAYSER-HEROLD, O. 2009. Data structures and requirements for *hp* finite element software. *ACM Trans. Math. Softw. 36,* 1, 4/1–4/31.

BANGERTH, W. AND RANNACHER, R. 2003. *Adaptive Finite Element Methods for Differential Equations*. Birkhäuser Verlag.

BANK, R. E. 1998. *PLTMG: a software package for solving elliptic partial differential equations*. SIAM, Philadelphia. Users' guide 8.0.

BRUASET, A. M. AND LANGTANGEN, H. P. 1997. A comprehensive set of tools for solving partial differential equations; DiffPack. In M. DÆHLEN AND A. TVEITO (Eds.), *Numerical Methods and Software Tools in Industrial Mathematics*, pp. 61–90. Birkhäuser, Boston.

BURRI, A., DEDNER, A., KLÖFKORN, R., AND OHLBERGER, M. 2005. An efficient implementation of an adaptive and parallel grid in DUNE. In *Computational Science and High Performance Computing II. Proceedings of the 2nd Russian-German Advanced Research Workshop, Stuttgart, Germany, March 14 to 16, 2005*, pp. 67–82. Springer.

BURSTEDDE, C., GHATTAS, O., GURNIS, M., ISAAC, T., STADLER, G., WARBURTON, T., AND WILCOX, L. C. 2010. Extreme-scale AMR. In *SC '10: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE.

BURSTEDDE, C., GHATTAS, O., GURNIS, M., TAN, E., TU, T., STADLER, G., WILCOX, L. C., AND ZHONG, S. 2008. Scalable adaptive mantle convection simulation on petascale supercomputers. In *SC '08: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE.

BURSTEDDE, C., GHATTAS, O., STADLER, G., TU, T., AND WILCOX, L. C. 2008. Towards adaptive mesh PDE simulations on petascale computers. In *Proceedings of Teragrid '08*.

BURSTEDDE, C., WILCOX, L. C., AND GHATTAS, O. 2010. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM J. Sci. Comput., submitted*.

CAREY, G. F. 1997. *Computational Grids: Generation, Adaptation and Solution Strategies*. Taylor & Francis.

CARRINGTON, L., KOMATITSCH, D., LAURENZANO, M., TIKIR, M. M., MICHÉA, D., GOFF, N. L., SNAVELY, A., AND TROMP, J. 2008. High-frequency simulations of global seismic wave propagation using SPECFEM3D GLOBE on 62K processors. In *SC '08: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE.

CHAND, K. K., DIACHIN, L. F., LI, X., OLLIVIER-GOOCH, C., SEOL, E. S., SHEPHARD, M. S., TAUTGES, T., AND TREASE, H. 2008. Toward interoperable mesh, geometry and field components for PDE simulation development. *Engineering with Computers 24*, 165–182.

DEVINE, K. D., FLAHERTY, J. E., WHEAT, S. R., AND MACCABE, A. B. 1993. A massively parallel adaptive finite element method with dynamic load balancing. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pp. 2–11. ACM.

FALGOUT, R. D., JONES, J. E., AND YANG, U. M. 2005. Pursuing scalability for hypre's conceptual interfaces. *ACM Trans. Math. Softw. 31*, 326–350.

FALGOUT, R. D., JONES, J. E., AND YANG, U. M. 2006. The design and implementation of hypre, a library of parallel high performance preconditioners. In T. J. BARTH, M. GRIEBEL, D. E. KEYES, R. M. NIEMINEN, D. ROOSE, T. SCHLICK, A. M. BRUASET, AND A. TVEITO (Eds.), *Numerical Solution of Partial Differential Equations on Parallel Computers*, Volume 51 of *Lecture Notes in Computational Science and Engineering*, pp. 267–294. Springer.

GEE, M. W., SIEFERT, C. M., HU, J. J., TUMINARO, R. S., AND SALA, M. G. 2006. ML 5.0 smoothed aggregation user's guide. Technical Report SAND2006-2649, Sandia National Laboratories.

GEENEN, T., UR REHMAN, M., MACLACHLAN, S. P., SEGAL, G., VUIK, C., VAN DEN BERG, A. P., AND SPAKMAN, W. 2009. Scalable robust solvers for unstructured fe geodynamic modeling applications: Solving the Stokes equation for models with large localized viscosity contrasts. *Geoch. Geoph. Geosyst. 10*, Q09002/1–12.

HENNING, J. L. 2007. Performance counters and development of spec cpu2006. *ACM SIGARCH Computer Architecture News 35*, 118–121.

HEROUX, M. A. ET AL. 2011. Trilinos web page. http://trilinos.sandia.gov.

HEROUX, M. A., BARTLETT, R. A., HOWLE, V. E., HOEKSTRA, R. J., HU, J. J., KOLDA, T. G., LEHOUCQ, R. B., LONG, K. R., PAWLOWSKI, R. P., PHIPPS, E. T., SALINGER, A. G., THORNQUIST, H. K., TUMINARO, R. S., WILLENBRING, J. M., WILLIAMS, A., AND STANLEY, K. S. 2005. An overview of the Trilinos project. *ACM Trans. Math. Softw. 31*, 397–423.

KANSCHAT, G. AND RIVIÈRE, B. 2010. A strongly conservative finite element method for the coupling of Stokes and Darcy flow. *J. Comp. Phys. 229*, 5933–5943.

KIRK, B., PETERSON, J. W., STOGNER, R. H., AND CAREY, G. F. 2006. `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers 22,* 3–4, 237–254.

KNEPLEY, M. G. AND KARPEEV, D. A. 2009. Mesh algorithms for PDE with Sieve I: Mesh distribution. *Scientific Programming 17*, 215–230.

LANGTANGEN, H. P. 2003. *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*. Texts in Computational Science and Engineering. Springer Verlag.

LOGG, A. 2007. Automating the finite element method. *Arch. Comput. Meth. Eng. 14,* 2, 93–138.

LOGG, A. 2009. Efficient representation of computational meshes. *Int. J. Comput. Sc. Engrg. 4,* 4, 283–295.

LOGG, A. AND WELLS, G. 2010. DOLFIN: Automated finite element computing. *ACM Transactions on Mathematical Software (TOMS) 37,* 2, 1–28.

MATHUR, K. K., JOHAN, Z., JOHNSSON, S. L., AND HUGHES, T. J. R. 1993. Massively parallel computing: Unstructures finite element simulations. Technical Report TR-08-93, Center for Research in Computing Technology, Harvard University.

MCKENZIE, D. P., ROBERTS, J. M., AND WEISS, N. O. 1974. Convection in the Earth's mantle: Towards a numerical solution. *J. Fluid Mech. 62*, 465–538.

MESSAGE PASSING INTERFACE FORUM 2009. MPI: A message-passing interface standard (version 2.2). Technical report, `http://www.mpi-forum.org/`.

MORTON, G. M. 1966. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM Ltd.

OLLIVIER-GOOCH, C., DIACHIN, L., SHEPHARD, M. S., TAUTGES, T., KRAFTCHECK, J., LEUNG, V., LUO, X., AND MILLER, M. 2010. An interoperable, data-structure-neutral component for mesh query and manipulation. *ACM Trans. Math. Software 37*, 29/1–28.

PATZÁK, B. AND BITTNAR, Z. 2001. Design of object oriented finite element code. *Adv. Eng. Software 32,* 10–11, 759–767.

REINDERS, J. 2007. *Intel Threading Building Blocks*. O'Reilly.

RENARD, Y. AND POMMIER, J. 2006. Getfem++. Technical report, INSA Toulouse, available from `http://www-gmm.insa-toulouse.fr/getfem/`.

RHEINBOLDT, W. C. AND MESZTENYI, C. K. 1980. On a data structure for adaptive finite element mesh refinements. *ACM Trans. Math. Softw. 6*, 166–187.

SCHUBERT, G., TURCOTTE, D. L., AND OLSON, P. 2001. *Mantle Convection in the Earth and Planets, Part 1*. Cambridge.

SILVESTER, D. AND WATHEN, A. 1994. Fast iterative solution of stabilised Stokes systems. Part II: Using general block preconditioners. *SIAM J. Numer. Anal. 31*, 1352–1367.

ŠOLÍN, P., ČERVENÝ, J., AND DOLEŽEL, I. 2008. Arbitrary-level hanging nodes and automatic adaptivity in the hp-FEM. *Math. Comput. Sim. 77*, 117–132.

ŠOLÍN, P., SEGETH, K., AND DOLEŽEL, I. 2003. *Higher-Order Finite Element Methods*. Chapman & Hall/CRC.

STROUSTRUP, B. 1997. *The C++ Programming Language* (3 ed.). Addison-Wesley.

SUNDAR, H., SAMPATH, R., AND BIROS, G. 2008. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM J. Sci. Comput. 30,* 5, 2675–2708.

TAN, E., GURNIS, M., ARMENDARIZ, L., STRAND, L., AND KIENTZ, S. 2008. Citcoms user manual version 3.0.1.

TEZDUYAR, T. E., ALIABADI, S. K., BEHR, M., AND MITTAL, S. 1994. Massively parallel finite element simulation of compressible and incompressible flows. *Comp. Meth. Appl. Mech. Engrg. 119*, 157–177.

TIKHONOVA, A., TANASE, G., TKACHYSHYN, O., AMATO, N. M., AND RAUCHWERGER, L. 2005. Parallel algorithms in STAPL: Sorting and the selection problem. Technical Report TR05-005, Parasol Lab, Department of Computer Science, Texas A&M University.

TU, T., O'HALLARON, D. R., AND GHATTAS, O. 2005. Scalable parallel octree meshing for terascale applications. In *SC '05: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE.

VERFÜRTH, R. 1994. A posteriori error estimation and adaptive mesh-refinement techniques. *J. Comput. Appl. Math. 50*, 67–83.

Received Month Year; revised Month Year; accepted Month Year