Massively Parallel Finite Element Programming

Timo Heister[1], Martin Kronbichler[2], and Wolfgang Bangerth[3]

[1] NAM, University of Göttingen, Germany
heister@math.uni-goettingen.de
[2] Department of Information Technology, Uppsala University, Sweden
martin.kronbichler@it.uu.se
[3] Department of Mathematics, Texas A&M University
bangerth@math.tamu.edu

Abstract. Today's large finite element simulations require parallel algorithms to scale on clusters with thousands or tens of thousands of processor cores. We present data structures and algorithms to take advantage of the power of high performance computers in generic finite element codes.

Existing generic finite element libraries often restrict the parallelization to parallel linear algebra routines. This is a limiting factor when solving on more than a few hundreds of cores. We describe routines for distributed storage of all major components coupled with efficient, scalable algorithms. We give an overview of our effort to enable the modern and generic finite element library `deal.II` to take advantage of the power of large clusters. In particular, we describe the construction of a distributed mesh and develop algorithms to fully parallelize the finite element calculation. Numerical results demonstrate good scalability.

Keywords: Finite Element Software, Parallel Algorithms, Massively Parallel Scalability.

1 Introduction

Modern computer clusters have up to tens of thousands of cores and are the foundation to deal with large numerical problems in finite element calculations. The hardware architecture requires software libraries to be specifically designed.

This has led to a significant disparity between the capabilities of current hardware and the software infrastructure that underlies many finite element codes for the numerical simulation of partial differential equations: there is a large gap in parallel scalability between the specialized codes designed to run on those large clusters and general libraries. The former are hand-tailored to the numerical problem to be solved and often only feature basic numerical algorithms such as low order time and spatial discretizations on uniform meshes. On the other hand, most general purpose finite element libraries like `deal.II` [4,5] presently do not scale to large clusters but provide more features, such as higher order finite elements, mesh adaptivity, and flexible coupling of different elements and equations, and more.

R. Keller et al. (Eds.): EuroMPI 2010, LNCS 6305, pp. 122–131, 2010.

By developing parallel data structures and algorithms we enable `deal.II` to perform numerical simulations on massively parallel machines with a distributed memory architecture. At the same time, by implementing these algorithms at a generic level, we maintain the advanced features of `deal.II`.

While we describe our progress with `deal.II` in [3], the generic algorithms developed here are applicable to nearly any finite element code. The modifications to `deal.II` discussed in this paper are a work in progress but will become available soon as open source. In Section 3 we describe the data structures and algorithms for the parallelization of finite element software. We conclude with numerical results in Section 4. The parallel scalability is shown using a Poisson problem and we present results for a more involved mantle convection problem.

2 Related Work

Finite element software has long been packaged in the form of libraries. A cursory search of the internet will yield several dozen libraries that support the creation of finite element applications to various degrees. While most of these are poorly documented, poorly maintained, or both, there are several widely used and professionally developed libraries. Most of these, such as DiffPack [6,14], libMesh [12], Getfem++ [17], OOFEM [16], or FEniCS [15] are smaller than the `deal.II` library but have similar approaches and features.

To the best of our knowledge, none of these libraries currently support massive parallel computations. What parallel computation they support is similar to what is available in publicly available releases of `deal.II`: meshes are either statically partitioned or need to be replicated on every processor, only linear solvers are fully distributed. While this allows for good scaling of solvers, the replication of meshes on all processors is a bottleneck that limits overall scalability of parallel adaptive codes to a few dozen processors. The reason for this lack of functionality is the generally acknowledged difficulty of fully distributing the dynamically changing, complex data structures used to describe adaptive finite element meshes. A particular complication is the fact that all of the widely used libraries originate from software that predates massively parallel computations, and retrofitting the basic data structures in existing software to new requirements is nontrivial in all areas of software design.

The only general framework for unstructured, fully parallel adaptive finite element codes we are aware of that scales to massive numbers of processors is ALPS, see [7]. Like the work described here, ALPS is based on the `p4est` library [8]. On the other hand, ALPS lacks the extensive support infrastructure of `deal.II` and is not publicly available.

3 Massively Parallel Finite Element Software Design

In order for finite element simulations to scale to a large number of processors, the compute time must scale linearly with respect to the number of processors

and the problem size. Additionally, local memory consumption should only depend on the local, not the global, problem size. The former requires minimizing global communication. The latter requires distributed data structures, where only necessary data is stored locally.

Thus, our focus is on the primary bottlenecks to parallel scalability: the mesh handling, the distribution and global numbering of the degrees of freedom, and the numerical linear algebra. See [3] for the technical details concerning the implementation with `deal.II`.

3.1 Distributed Mesh Handling

The computational mesh is duplicated on each processor in most generic finite element libraries, if they support distributed parallel computing at all. This is not feasible for massively parallel computations because the generic description of a mesh involves a significant amount of data for each cell which is then replicated on each processor, resulting in huge memory overhead.

Each processor only needs to access a small subset of cells. Most parts of the replicated global mesh are unnecessary locally. We will say that the processor "owns" this required subset of cells. In addition, a processor also needs to store cells touching the cells it owns, but that are in fact owned by neighboring machines. These neighboring cells are called *ghost cells*. Information about ghost cells is needed for several reasons, most obviously because continuous finite elements share degrees of freedom on the lines and vertices connecting cells. Ghost cells are also needed for adaptive refinement, error estimation, and more.

Since `deal.II` only supports hexahedral cells, we restrict the discussion to that type of meshes. The active cells in h-adaptive finite element methods are attained from recursively refining a given "coarse" mesh. Thus, one also stores the hierarchy of refinement steps. We distinguish between three different kinds of information for mesh storage:

1. The *coarse mesh* consists of a number of coarse cells describing the domain. We assume that the coarse mesh only consists of a relatively small number of cells compared to the number of active cells in the parallel computation, say a few tens of thousands; it can be stored on each processor. The refinement process starts with the coarse mesh.
2. *Refinement information* can be stored in a (sparse) octree (a quadtree in two spatial dimensions) of refinement flags for each coarse cell. Each flag either states that this cells has been refined into eight (four in two dimensions) children or that it is an active cell if not.
3. *Active cells* are those on which the finite element calculation is done. Typical finite element programs attach a significant amount of information to each cell: vertex coordinates, connectivity information to faces, lines, corners, and neighboring cells, material indicators, boundary indicators, etc. We include the ghost cells here, but we set a flag to indicate that they to belong to a different machine.

We can store the coarse mesh (1.) on each machine without a problem. For the refinement information (2.) we interface to an external library called `p4est`, see

[8]. This library handles the abstract collection of octrees describing refinement from the coarse mesh and handles coarsening, refinement, and distribution of cells. Internally `p4est` indexes all terminal cells in a collection of octrees with a space filling curve. This allows rapid operations and scalability to billions of cells. For partitioning the space filling curve is cut into equally sized subsets, which ensures a well balanced workload even during adaptive refinement and coarsening. `p4est` allows queries about the local mesh and the ghost layer. With this information `deal.II` recreates only the active cells and the ghost layer. Because we chose to recreate a local triangulation on each machine, most of the finite element library works without modification, e.g. implementation of finite element spaces. However, we need a completely new method for creating a global enumeration of degrees of freedom. This is discussed next.

3.2 Handling of Degrees of Freedom

The finite element calculation requires a global enumeration of the degrees of freedom. The difficulty lies in the fact that every machine only knows about a small part of the mesh. The calculation of this numbering involves communication between processors. Additionally, the numbering on the ghost layer and the interface must be available on each machine. This is done in a second step.

Like all cells, each degree of freedom is owned by a single processor. All degrees of freedom inside a cell belong to the machine that owns the cell. The ownership of degrees of freedom on the interface between cells belonging to different machines is arbitrary, but processors that share such an interface need to deterministically agree who owns them. We assign such degrees of freedom to the processor with the smaller index.[1] The following algorithm describes the calculation and communication to acquire a global enumeration on the machines $p = 0, \ldots, P-1$:

1. Mark all degrees of freedom as invalid (e.g. -1).
2. Loop over the locally owned cells and mark all degrees of freedom as valid (e.g. 0).
3. Loop over the ghost cells and reset the indices back to invalid if the cell is owned by a processor $q < p$. Now only indices that are owned locally are marked as valid.
4. Assign indices starting from 0 to all valid DoFs. This is done separately from the previous steps, because otherwise all neighbors sharing a degree of freedom would have to be checked for ownership. We denote the number of distributed DoFs on machine p with n_p.
5. Communicate the numbers n_p to all machines and shift the local indices by $\sum_{q=0}^{p-1} n_q$

Now all degrees of freedom are uniquely numbered with indices between 0 and $N = \sum_{q=0}^{P-1} n_q$. Next we must communicate the indices of degrees of freedom

[1] This rule is evaluated without communication. Assigning all degrees of freedom on one interface to the same processor also minimizes the coupling in the system matrix between the processors, see [3].

on the interface to the ghost layer and on the ghost layer itself. Each machine collects a packet of indices to send to its neighbors. Indices of a cell are sent to a neighbor if a ghost cell owned by that neighbor touches the cell. Indices on the interface may not be known at this point because they might belong to a third machine. As these communications are done concurrently there is no way to incorporate this information in this step. So we do this communication step twice: in the first round every machine receives all indices on its own cells, and after the second round every machine knows every index on the own cells and the ghost cells. There is no global communication required in these two steps.

3.3 Efficient Indexing

A subset $\mathcal{I}$ of the indices $\{0, \ldots, N\}$ is managed on each processor p. Each processor also needs to have the indices of degrees of freedom on the interface owned by another machine and indices on the ghost layer. Algebraic constraints induced by hanging nodes, solution vectors and other data structures need to access or store information for those indices. We typically look at three different subsets of indices:

1. The *locally owned* indices $\mathcal{I}_{l.o.}$ as described earlier. Following the algorithm outlined in the previous step, this is initially a contiguous range of n_p indices. However, we may later renumber indices, for example to in a block-wise way to reflect the structure of a partial differential equation in the linear system.
2. The *locally active* indices $\mathcal{I}_{l.a.}$ defined as the locally owned indices as well as the other indices on the interface. This is no longer a contiguous range.
3. The *locally relevant* indices $\mathcal{I}_{l.r.}$, which also includes the indices on the ghost cells.

We need an efficient data structure to define these subsets. If we store some information for each index in $\mathcal{I}_{l.r.}$, we would like to put that information into a contiguous memory location of $\#\mathcal{I}_{l.r.}$ elements. To access the information of an index $i \in \mathcal{I}_{l.r.}$, we need to find its position in the list $\mathcal{I}_{l.r.}$ (in other words, the number of indices $j \in \mathcal{I}_{l.r.}$ with $j < i$). This query is performed repeatedly, and thus it should be optimized.

We create a data structure of K sorted, disjoint, contiguous, intervals $[b_k, e_k)$ for defining the subset $\mathcal{I} \subset \{0, \ldots, N\}$ as $\tilde{\mathcal{I}} = \bigcup_{k=0}^{K} [b_k, e_k)$. Other libraries often go for a simpler description of this subset as a list of numbers, but this means more entries are stored because of the large contiguous subranges. Thus the important queries are slower. We also store 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 with each interval $[b_k, e_k)$. This allows us to do queries like the one above in $\mathcal{O}(\log_2 K)$ operations.

3.4 Numerical Linear Algebra

The linear system can be stored with the global numbering of the degrees of freedom. There are existing, extensively tested, and widely used parallel libraries like PETSc, see [1,2] and Trilinos, see [11,10] to handle the linear system. They supply row-wise distributed matrices, vectors and algorithms, like iterative Krylov

solvers, and preconditioners. `deal.II`, like most other finite element libraries, has interfaces to these libraries. We have tested both PETSc and Trilinos solvers up to many thousand cores and obtain excellent scaling results (see below).

The linear system is assembled on the local cells. Matrix and vector values for rows on different machines are sent to the owner using point-to-point communication. The linear system is subsequently solved in parallel.

3.5 Summary of Finite Element Algorithms

The building blocks of a distributed finite element calculation are described above. In an actual implementation, additional technical details must be addressed. To perform adaptive mesh refinement, an error estimate is needed to decide which cells to refine, solutions must be transferred between meshes, and hanging nodes must be handled. Hanging nodes come from degrees of freedom on interfaces between two cells on different refinement levels. For finite element systems with several components, like velocity, pressure, and temperature with different elements as used in Section 4, the global indices must be sorted by vector components. See [3] for details.

3.6 Communication Patterns

Because of the complex nature and algorithmic diversity of the operations outlined above it is difficult to analyze the MPI communication patterns appearing in the code.

The distribution of the mesh given by `p4est` has an interesting property: the number of *neighbors* for each processor (number of owners of the ghost cells) is bounded by a small number independent of the total problem size and the number of processors[2]. Most communication necessary in our code is therefore in the form of point-to-point messages between processors and a relatively small number of their neighbors, which results in optimal scaling.

Further efficiency is gained by *hiding* the latency of communication where possible: MPI communication uses non-blocking transfers and computations proceed while waiting for completion instead of leaving processors idle.

The amount of MPI communication within `deal.II` (outside that handled by `p4est`) consists of the parts described in section 3 and was created with the massive parallel implementation in mind. Exchanging degrees of freedom on the ghost cells is consequently done with neighbors only and is effectively hidden using non-blocking transfers. Other algorithms using communication, like error estimation and solution transfer, behave similarly. All other communication is done inside the linear algebra package.

As in all good finite element codes, the majority of compute time in our applications is spent in the linear solvers. For massively parallel applications, either PETSc or Trilinos provides this functionality. Limiting factors are then the speed of scalar product evaluations and matrix-vector products. The former

[2] The number of neighbors in all experiments is always smaller than fifty, but much lower for typical meshes. This seems to be a property of the space filling curve.

is a global reduction operation, it only consists of scalar data. On the other hand, matrix-vector products require this sort of communication but do not require the global reduction step. Furthermore, matrix-vector products are often not the most difficult bottleneck because communication can be hidden behind expensive local parts of the product. In summary, good scalability in the linear algebra is achieved if a low latency network like InfiniBand is available, as it is on most current high performance clusters.

4 Numerical Results

4.1 Scalability Test

We start by testing a Poisson equation in 2 and 3 dimensions with adaptive and global refinement, respectively. In Figure 1 we show the weak scalability from 8 to 1000 processors with about 500,000 degrees of freedom per processor on the three dimensional unit cube. We measure computation times for different parts of the program and average memory consumption on each machine. All parts but the solver (BiCGStab preconditioned with an algebraic multigrid) scale linearly. Figure 2 shows individual iterations in a fully adaptive refinement loop for a two dimensional Poisson equation on 1024 processors (left). The problem size increases over several refinement cycles from 1.5 million to 1.5 billion degrees of freedom. The right panel shows strong scalability starting from 256 and going to 4096 processors for the same cycle with a fixed problem size within the adaptive iteration loop. We have excellent scalability with respect to problem size and the number of processors. The memory consumption is nearly constant even when the problem size increases by over a factor of one hundred.

4.2 Results for a Mantle Convection Problem

Our second test case is a more complicated problem modeling thermal convection in the Earth's mantle. Details and motivation for the discretization and solver choices are given in [13] and [9].

In the Earth's mantle, fluid flow is strongly dominated by viscous stresses and is driven (among other factors) by temperature differences in the material, while

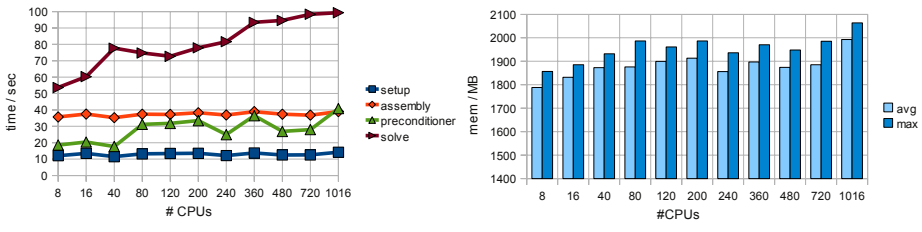

Fig. 1. 3D Poisson Problem, regular refinement, 500,000 degrees of freedom per processor. Left: Weak scaling up to 1016 processors. Right: Average peak memory for the same data.

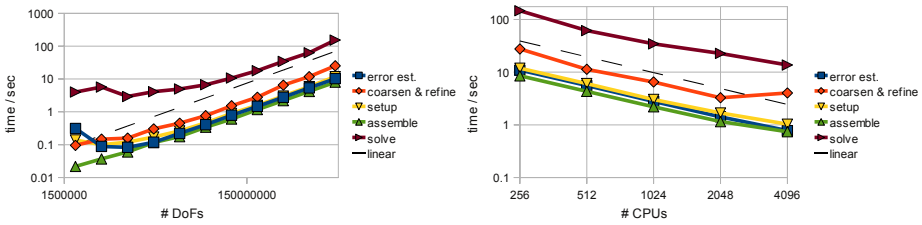

Fig. 2. 2D Poisson Problem, fully adaptive. Left: Weak scaling on 1024 processors. Right: Strong scaling, problem size of around 400 million DoFs.

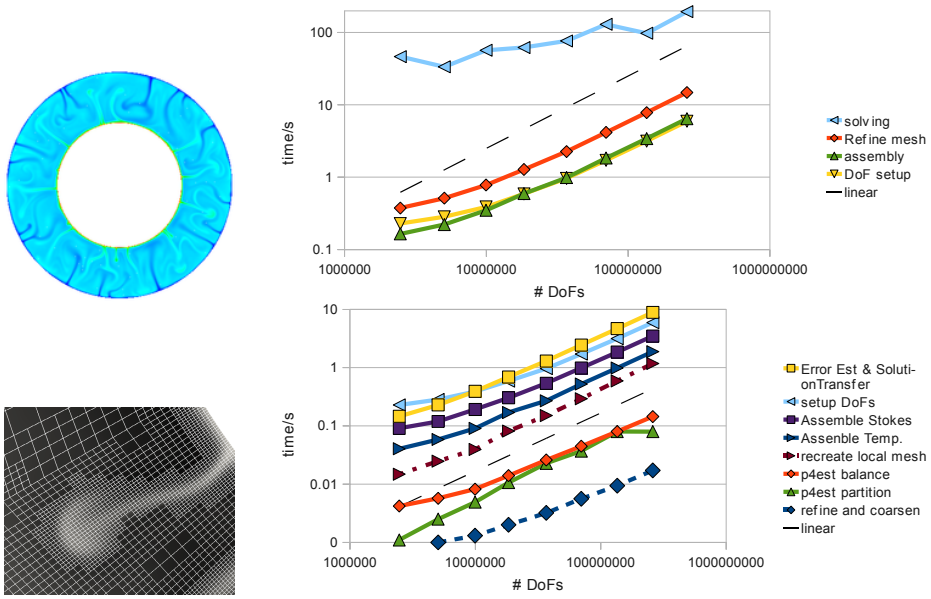

Fig. 3. 2D mantle convection. Left: snapshot of the temperature for a fixed time step and a zoom. Right: Solution times on 512 processors with overview (top) and detailed functions (bottom).

inertia is negligible at realistic velocities of a few centimeters per year. Thus the buoyancy-driven flow can be described by the Boussinesq approximation:

$$\begin{aligned} -\nabla \cdot (2\eta\varepsilon(\mathbf{u})) + \nabla p &= -\rho\,\beta\,T\,\mathbf{g}, \\ \nabla \cdot \mathbf{u} &= 0, \\ \frac{\partial T}{\partial t} + \mathbf{u}\cdot\nabla T - \nabla\cdot\kappa\nabla T &= \gamma. \end{aligned} \tag{1}$$

Here, $\mathbf{u}, p, T$ denote the three unknowns in the Earth's mantle: velocity, pressure, and temperature. The first two equations form a Stokes system for velocity and pressure with a forcing term stemming from the buoyancy through the

temperature T. The third is an advection-diffusion equation. Let η be the viscosity of the fluid and κ the diffusivity coefficient for the temperature (both assumed to be constant here for simplicity), $\varepsilon(\mathbf{u}) = \frac{1}{2}(\nabla\mathbf{u} + (\nabla\mathbf{u})^T)$ denotes the symmetrized gradient, ρ is the density, β is the thermal expansion coefficient, g is the gravity vector, and γ describes the external heat sources. Fig. 3, left, shows a snapshot from the evolution of the turbulent mixing within the Earth's mantle.

In figure 3, right, we present timing of seven adaptive refinement steps for a single fixed time step. We observe good scalability; the solver itself scales better than linearly due to the relatively fine mesh for the two dimensional solution and because the solution on the coarser meshes is reused as a starting guess.

5 Conclusions

We present a general framework for massively parallel finite element simulation. The results are convincing, showing that even complex problems with more than a billion unknowns can be solved on a large cluster of machines. The developments in `deal.II` outlined here make the maximum solvable problem size two orders of magnitude larger than previously possible.

There are several reasons for the good scalability results. Most importantly the workload is distributed evenly, because every processor has roughly the same number of locally active cells. In addition the algorithms described in section 3 introduce no significant overhead in parallel. This includes the total memory usage. As described in section 3.6 most of the communication is restricted to the neighbors.

Acknowledgments. Timo Heister is partly supported by the German Research Foundation (DFG) through GK 1023. Martin Kronbichler is supported by the Graduate School in Mathematics and Computation (FMB). Wolfgang 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 through Award No. EAR-0426271, and by an Alfred P. Sloan Research Fellowship.

The computations were done on the Hurr[3] cluster of the Institute for Applied Mathematics and Computational Science (IAMCS) at Texas A&M University. Hurr is supported by Award No. KUS-C1-016-04 made by King Abdullah University of Science and Technology (KAUST).

References

1. Balay, S., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., Curfman McInnes, L., Smith, B.F., Zhang, H.: PETSc users manual. Technical Report ANL-95/11 - Revision 3.0.0, Argonne National Laboratory (2008)

[3] 128 nodes with 2 quad-core AMD Shanghai CPUs at 2.5 GHz, 32GB RAM, DDR Infiniband, running Linux, OpenMPI, gcc 4.

2. Balay, S., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M.G., Curfman McInnes, L., Smith, B.F., Zhang, H.: PETSc Web page (2009), http://www.mcs.anl.gov/petsc
3. Bangerth, W., Burstedde, C., Heister, T., Kronbichler, M.: Algorithms and Data Structures for Massively Parallel Generic Finite Element Codes (in preparation)
4. Bangerth, W., Hartmann, R., Kanschat, G.: deal.II Differential Equations Analysis Library, Technical Reference, http://www.dealii.org
5. Bangerth, W., Hartmann, R., Kanschat, G.: deal.II — a General Purpose Object Oriented Finite Element Library. ACM Transactions on Mathematical Software 33(4), 27 (2007)
6. Bruaset, A.M., Langtangen, H.P.: A comprehensive set of tools for solving partial differential equations; DiffPack. In: Dæhlen, M., Tveito, A. (eds.) Numerical Methods and Software Tools in Industrial Mathematics, pp. 61–90. Birkhäuser, Boston (1997)
7. Burstedde, C., Burtscher, M., Ghattas, O., Stadler, G., Tu, T., Wilcox, L.C.: Alps: A framework for parallel adaptive pde solution. Journal of Physics: Conference Series 180(1), 012009(2009)
8. Burstedde, C., Wilcox, L.C., Ghattas, O.: p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. Submitted to SIAM Journal on Scientific Computing (2010)
9. Heister, T., Kronbichler, M., Bangerth, W.: Generic finite element programming for massively parallel flow simulations. In: Eccomas 2010 Proceedings (submitted, 2010)
10. 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., Stanley, K.S.: An overview of the Trilinos project. ACM Trans. Math. Softw. 31, 397–423 (2005)
11. Heroux, M.A., et al: Trilinos Web page (2009), http://trilinos.sandia.gov
12. Kirk, B., Peterson, J.W., Stogner, R.H., Carey, G.F.: libMesh: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. Engineering with Computers 22(3-4), 237–254 (2006)
13. Kronbichler, M., Bangerth, W.: Advanced numerical techniques for simulating mantle convection (in preparation)
14. Langtangen, H.P.: Computational Partial Differential Equations: Numerical Methods and Diffpack Programming. Texts in Computational Science and Engineering. Springer, Heidelberg (2003)
15. Logg, A.: Automating the finite element method. Arch. Comput. Methods Eng. 14(2), 93–138 (2007)
16. Patzák, B., Bittnar, Z.: Design of object oriented finite element code. Advances in Engineering Software 32(10–11), 759–767 (2001)
17. Renard, Y., Pommier, J.: Getfem++. Technical report, INSA Toulouse (2006), http://www-gmm.insa-toulouse.fr/getfem/