A Modular Design for Parallel Adaptive Finite Element Computational Kernels

Krzysztof Banaś

Section of Applied Mathematics ICM, Cracow University of Technology, Warszawska 24, 31-155 Kraków, Poland, Krzysztof.Banas@pk.edu.pl

Abstract. The paper presents modular design principles and an implementation for computational kernels of parallel adaptive finite element codes. The main idea is to consider separately sequential modules and to add several specific modules for parallel execution. The paper describes main features of the proposed architecture and some technical details of implementation. Advanced capabilities of finite element codes, like higher order and discontinuous discretizations, multi-level solvers and dynamic parallel adaptivity, are taken into account. A prototype code implementing described ideas is also presented.

1 Introduction

The often used model for parallelization of finite element codes is to consider a library of communication routines that handle transfer of finite element data structures, taking into account complex inter-relations between them [1]. After the transfer of e.g. an element data structure, all required connectivities (such as, for example, constituting faces and vertices, neighboring elements, children and father elements) must be restored, either directly from transfered data or by suitable computations. In such a model, main modules of a finite element code, most importantly mesh manager, must handle parallelism explicitly, by calling respective transfer procedures. As a result, despite the splitting between a communication library and a finite element code, both have to be aware of finite element technical details and parallel execution details.

In the second popular model [2] standard communication routines are employed. Then, parallelization concerns the whole code (or its main parts). This effectively means that sequential parts are replaced by new parallel components.

In the present paper an alternative to both approaches is proposed. The main modules of sequential finite element codes (except linear solver) remain unaware of parallel execution. Additional modules are added that fully take care of parallelism. These modules are tailored to the needs of parallelization of sequential parts, in order to achieve numerical optimality and execution efficiency.

The paper is organized as follows. In Sect. 2 some assumptions on finite element approximation algorithms and codes, that are utilized in parallelization process, are described. The next section concerns assumptions on a target environment for which parallel codes are designed. Algorithms fitting the proposed

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3037, pp. 155–162, 2004.

model of parallel execution are described in Sect. 4. Section 5 presents an architecture of parallel codes, with main parallel modules specified, while Sect. 6 considers in more detail the main tasks performed by parallel modules. Section 7 concerns implementation of parallel modules. Section 8 describes some numerical experiments. Conclusions are presented in Sect. 9.

2 Sequential Algorithms and Codes

The model of parallelization presented in the paper is applicable to a broad class of finite element codes, including complex adaptive codes for coupled multiphysics problems. It is assumed that several meshes and several approximation fields may be present in a simulation. Meshes may be adaptive and nonconforming. Approximation fields may be vector fields and may provide higher order approximation. All types of adaptivity, including anisotropic and *hp*, can be handled. The interface between the finite element code and a linear solver allows for the use of multi-level (multigrid) solvers.

In a prototype implementation, described in later sections, it is assumed that the finite element code is split into four fundamental modules, based on four separate data structures [3]: mesh manipulation module with mesh data structure, approximation module with finite element approximation data structure, linear equations solver (or interface to an external solver) with multi-level matrix data structure and problem dependent module with all problem specific data. Although this splitting is not necessary in order to apply parallelization process described in the paper, it facilitates the presentation of the process as well as its practical implementation.

3 Target Parallel Execution Environment

The architecture is developed for the most general to-date execution environment, a system with message passing. Any hardware system that supports message passing may be used as a platform for computations.

Naturally for PDEs, the problem and program decomposition is based on spatial domain decomposition. The computational domain is partitioned into subdomains and main program data structures are split into parts related to separate subdomains. These data structures are distributed among processes executed on processors with their local memories. Processes are obtained by the Single Program Multiple Data (SPMD) strategy and realize main solution tasks in parallel. The most natural and efficient is the situation where there is one-to-one correspondence between processes and processors in a parallel machine, but other mappings are not excluded. In the description it is assumed that there is a unique assignment: subdomain–process–processor–local memory.

4 Parallel Algorithms

From the three main phases of adaptive finite element calculations, creating a system of linear equations, solving the system and adapting the mesh, only solving the system is not "embarrassingly" parallel. Numerical integration, system matrix aggregation, error estimation (or creation of refinement indicators), mesh refinement/derefinement are all local processes, on the level of a single mesh entity or a small group of entities (e.g. a patch of elements for error estimation).

Thanks to this, with a proper choice of domain decomposition, it is possible to perform all these local (or almost local) tasks by procedures taken directly from sequential codes. There must exist however, a group of modules that coordinate local computations spread over processors.

The only part of computational kernels that involve non-local operations is the solution of systems of linear equations. However, also here, the choice of Krylov methods with domain decomposition preconditioning guarantees optimal complexity with minimal number of global steps.

5 An Architecture for Parallel Codes

Fig. 1 presents an architecture for parallel adaptive finite element computational kernels. Four fundamental sequential modules are separated from additional, parallel execution modules. The structure of interfaces between all modules is carefully designed to combine maintainability, that require minimal interfaces, with flexibility and efficiency, for which more intensive module interactions are often necessary.

The main module to handle tasks related to parallel execution is called domain decomposition manager, according to the adopted strategy for parallelization. It has a complex structure that reflects the complex character of performed operations.

6 Main Parallel Solution Tasks

Main tasks related to parallel execution of finite element programs include:

- mesh partitioning
- data distribution
- overlap management
- maintaining mesh and approximation data coherence for parallel adaptivity
- load balancing and associated data transfer
- supporting domain decomposition algorithms

Mesh partitioning, algorithms and strategy, is not considered in the current paper. It is assumed that there exist an external module that provides non-overlapping mesh partitioning according to specified criteria. The criteria must include the standard requirements for keeping load balance and minimizing the

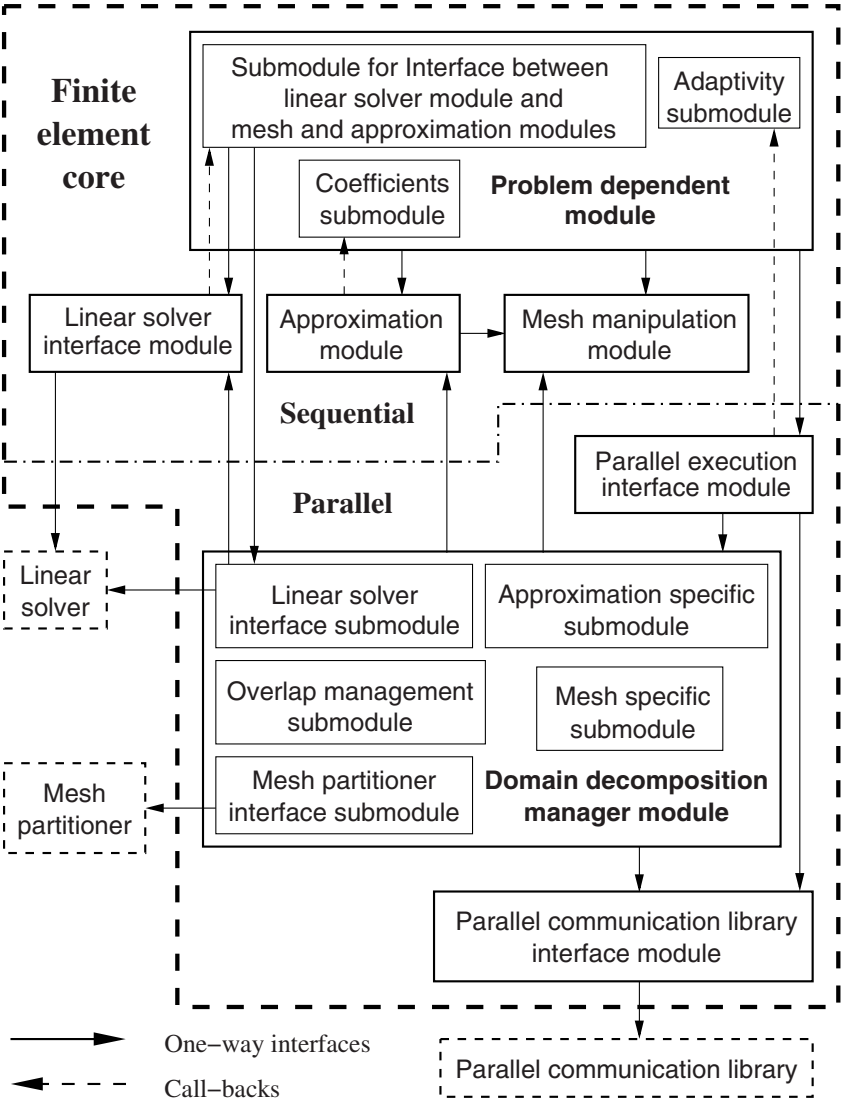

Fig. 1. Diagram of the proposed modular architecture for computational kernels of parallel adaptive finite element codes

extent of inter-subdomain boundary. Keeping load balance for all stages of computations, especially taking into account multi-level linear equations solvers [4], may be a difficult, if not impossible, task. Usually some compromise is postulated among requirements posed by different phases of computations.

Each mesh entity (and in consequence the related approximation data structure) is assigned to a single submesh (subdomain). Subdomains are distributed among processes (processors, local memories), creating an ownership relation between mesh entities and processes (processors, local memories). Each local memory stores all data related to owned entities and each processor performs main solution tasks operating on owned entities.

The existence of overlap (i.e. storing in local memory not owned, "ghost", mesh entities) is advantageous for several tasks in the solution procedure. These tasks include obviously multi-level overlapping domain decomposition preconditioning. Also error estimation, mesh refinement and derefinement benefit from storing data on neighbors of owned entities. The existence of overlap allows for utilizing more local operations and reduces the inter-processor communication. In exchange, more storage is required locally and some operations are repeated on different processors.

The amount of overlap depends on the profits achieved from local storage, which further depends not only on utilized algorithms, but also on computer architectures and interconnection networks employed. For implementation it is assumed that the amount of created overlap is indicated by the maximal extent of data, not available in the initial non-overlapping decomposition, necessary for *any* task operating on local data. Such a choice was made to adapt codes to slower parallel architectures based on networks.

It is a task of domain decomposition manager to create an overlap and to ensure that the overlap data is in a coherent state during computations. Proper values have to be provided, despite the fact that different modules and routines use and modify different parts of overlap data at different times. This task is important for parallel mesh modifications, especially when irregular (non-confirming) meshes are allowed.

Mesh modifications create load imbalance in the form of improper distribution of mesh and approximation entities between subdomains. It is assumed that in the code there is a special, possibly external, module that computes "proper" data distribution. The original mesh partitioner or a separate repartitioner can be used. Additionally to standard partitioning requirements, the module should also aim at minimizing data transfer between processors when regaining balance.

Taking the new partition supplied by the repartitioning module as an input, the domain decomposition module performs mesh transfer. To minimize data traffic, mesh entities must not be transfered separately, but grouped together, to form a patch of elements. Necessary parts of data structure, related to whole patches, are then exchanged between indicated pairs of processors.

Supporting domain decomposition algorithms consist in performing standard vector operations in parallel (such as scalar product or norm) and exchanging data on degrees of freedom close to inter-subdomain boundary between processors assigned to neighboring subdomains. Once again the operations can be cast into the general framework of keeping overlap data (approximation data in this case) stored in local memories in a coherent state. A proper coordination of data exchange with multi-level solution procedure has to be ensured.

7 Implementation

The basis for parallel implementation is formed by an assumption that every mesh entity (together with associated approximation data structure containing degrees of freedom) is equipped with a global (inter-processor) identifier (IPID). This identifier can be understood as a substitute for a global address space used in sequential codes. The IPID is composed of a processor (subdomain) number and a local (to a given processor) identifier. IPIDs are not known to sequential modules of the code. The domain decomposition manager creates an overlap and assigns IPIDs to all mesh entities. Whenever data not stored locally is necessary for computations, domain decomposition manager can find its owning processor and requests the data using suitable calls. With this implementation, keeping the local data structures in a coherent state means keeping a unique assignment of IPIDs to all mesh and approximation entities and data structures.

According to the design assumptions, the changes in the sequential routines are kept minimal. During refinements, children entities remain local to the same processor as their parents. During derefinements, all children entities are either already present locally or are transferred to one chosen processor (e.g. if multi-level computations are performed, the chosen processor may be the one assigned to a parent entity). To assign IPIDs to newly created entities, their lists are passed from mesh manipulation module to domain decomposition manager.

For the linear solver, additional routines are created for performing global vector operations and for exchanging data on overlap DOFs. In the proposed implementation these routines are simple wrappers for domain decomposition manager routines that perform actual operations.

7.1 Interfaces with Communication Libraries

It is assumed that codes use a set of generic send/receive and group communication operations. Additionally, initialization and finalization procedures are specified. All these have to be implemented for various *standard* communication libraries. In the example implementation a model of buffered send/receive operations is employed. The data to be sent are first packed into a buffer and then the whole buffer is sent. Procedures in that model can easily be implemented for MPI standard, as well as packages like PVM.

8 Numerical Examples

Two, simple from numerical point of view but demanding from the point of view of technical difficulties, computational examples are presented as a proof of concept. Both use a prototype implementation of the presented architecture in a discontinuous Galerkin *hp*-adaptive parallel code for 3D simulations.

The first example is a pure convection problem, with a rectangular pattern traveling through a 3D medium. Dynamic adaptivity is employed in this case with two levels of refinement, 1-irregular meshes and adaptations performed after

each time step. To minimize interprocessor communication for small fluctuations of subdomain sizes, load imbalance (measured by the ratio of the maximal or minimal number of degrees of freedom to the average number of degrees of freedom in a subdomain) up to 10% is allowed. When this limit is exceeded, repartitioning takes place and the balance is regained through the transfer of mesh entities.

In the example run, four processors and four subdomains were used that resulted in the average number of degrees of freedom around 5000 per subdomain. Mesh transfers were performed on average after each three steps. As a hardware platform a 100Mbit Ethernet network of PCs was used. PCs were equipped with 1.6 GHz Pentium 4 processors and 1 GByte memory. An average mesh transfer involved several thousand mesh entities. The overall speedup for four processors was equal to 2.67, taking into account times for repartitioning and mesh transfer.

Table 1. Parallel performance for 10 iterations of the preconditioned GMRES method and discontinuous Galerkin approximation used for solving Laplace's equation in a box domain (description in the text).

Single level preconditioner						
N_{DOF}	N_{proc}	Error*10^9	Rate	Time	Speed up	Efficiency
3 129 344	2	48.041	0.738	70.76	1.00	100%
	4	47.950	0.738	35.63	1.98	99%
	8	48.748	0.739	17.71	3.99	100%
Three level preconditioner						
N_{DOF}	N_{proc}	Error*10^9	Rate	Time	Speed up	Efficiency
3 129 344	2	0.027	0.350	111.16	1.00	100%
	4	0.027	0.350	57.76	1.92	96%
	8	0.027	0.348	33.15	3.35	84%

The second example is Laplace's equation in the box $[0,1]^3$ with assumed known exact solution. Results of two experiments are presented. In the first experiment the same network of PCs as for convection problem was used. The experiment consisted in solving the problem for a mesh with 3 129 344 degrees of freedom, obtained by consecutive uniform refinements of an initial mesh. Single level and three level multigrid preconditioning for the GMRES solver with Schwarz methods as smoothers was employed for solving linear equations. Table 1 presents results for *10 iterations* of the preconditioned GMRES method, to focus on the efficiency of parallel implementation of the code. N_{proc} is the number of workstations solving the problem. *Error* is the norm of residual after 10 GMRES iterations and *Rate* is the total GMRES convergence rate during solution. Execution time *Time* is a wall clock time, that includes generation of linear systems (numerical integration) as well. Speed-up and efficiency are computed in the standard way. The run with 2 PCs is taken as a reference since the problem was too large to fit into the memory of a single PC.

The second experiment for the second example was undertaken to test the scalability of the system. The experiment was performed on a cluster of 32 Pentium 3 PCs with 512 MByte memory each and 100 Mbit Ethernet interconnection. The mesh was obtained by another uniform refinement of the mesh from the previous experiment yielding 25 034 752 degrees of freedom. The data structure occupied 4.5 GBytes of memory and parallel adaptations were *necessary* to reach this problem size. Because of memory constraints (16 GBytes) a single level Schwarz preconditioning for GMRES was used, resulting in convergence rate equal to 0.9. The error reduction of 10^{-9} was obtained in 200 iterations that took 20 minutes to perform. Despite the increase in the number of iterations, the scalability of parallel implementation (related to the time of a single iteration) was maintained.

9 Conclusions

The presented model allows for relatively easy parallelization of existing finite element codes, with much of sequential parts of codes retained. The results of numerical experiments with the prototype implementation show good efficiency, making a model feasible solution for migrating finite element codes to high performance parallel environments.

Acknowledgments. The author would like to thank Prof. Peter Bastian from IWR at the University of Heidelberg for invitation to IWR and granting access to IWR's computational resources, used in the last described numerical experiment. The support of this work by the Polish State Committee for Scientific Research under grant 7 T11F 014 20 is also gratefully acknowledged.

References

1. Bastian, P., Birken, K., Johannsen, K., Lang, S., Neuss, N., Rentz-Reichert, H., Wieners, C.: UG - a flexible software toolbox for solving partial differential equations. Computing and Visualization in Science **1** (1997) 27–40
2. J.-F.Remacle, O.Klaas, J.E.Flaherty, M.S.Shephard: A Parallel Algorithm Oriented Mesh Database. Report 6, SCOREC (2001)
3. K.Banaś: On a modular architecture for finite element systems. I. Sequential codes. Computing and Visualization in Science (2004) accepted for publication.
4. Bastian, P.: Load balancing for adaptive multigrid methods. SIAM Journal on Scientific Computing **19** (1998) 1303–1321