FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Technical Report

Proceedings of the Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'03)

*Jörg Striegnitz, Kei Davis** *(Eds.)*

FZJ-ZAM-IB-2003-09

July 2003

(last change: 06.07.2003)

(*) Modelling, Algorithms, and Informatics Group, CCS-3 MS B256
Los Alamos National Laboratory
Los Alamos, NM 87545, USA

Preface

This report comprises the Proceedings of the Workshop on Parallel / High Performance Object-Oriented Scientific Computing (POOSC'03) that was held at the European Conference on Object-Oriented Computing (ECOOP) in Darmstadt, Germany, on 22 July 2003. The workshop was a joint organization of Research Centre Jülich and Los Alamos National Laboratory.

Scientific programming has reached an unprecedented degree of complexity. Sophisticated algorithms, a wide range of large-scale hardware environments, and an increasing demand for system integration and portability have shown that language-level abstraction must be increased without loss of performance.

Work presented at previous POOSC workshops has shown that the OO approach provides an effective means for the design of complex scientific systems, and that it is possible to design abstractions and applications that fulfill strict performance constraints.

However, OO still isn't fully embraced in high performance computing and there is still a need for research, development, and discussion. Previous POOSC workshops have proven that a workshop is an ideal venue for this.

For the last few years there have been no official proceedings for ECOOP workshops, these having been replaced by free-form summaries published as the ECOOP Workshop Reader. As in the past the POOSC organizers have reserved the option of independently printing a workshop proceedings if warranted by the quantity and quality of submissions. This year's contributions have demonstrated the depth, variety, and multidisciplinary character of this research field; this volume comprises a selection of those papers.

We thank all the contributors, referees, attendees, and the ECOOP workshop organizers for helping to make this workshop a highly successful event.

July 2003

Jörg Striegnitz
Kei Davis

Workshop Organizers

Kei Davis	Modeling, Algorithms, and Informatics Group, Computer and Computational Sciences Division, Los Alamos National Laboratory Los Alamos, NM 87545, USA `Kei.Davis@lanl.gov`
Jörg Striegnitz	Research Centre Jülich John von Neumann Institute for Computing (NIC) Central Institute for Applied Mathematics (ZAM) 42425 Jülich, Germany `J.Striegnitz@fz-juelich.de`

Programme Committee

Kei Davis	Los Alamos National Laboratory, Los Alamos, NM, USA
Vladimir Getov	University of Westminster, London, UK
Andrew Lumsdaine	Indiana University, Bloomington, IN, USA
Bernd Mohr	Research Centre Jülich, Germany
Jörg Nolte	Technical University Cottbus, Germany
Jörg Striegnitz	Research Centre Jülich

Table of Contents

Object-Oriented Framework for Multi-method Parallel PDE Software

Pieter De Ceuninck[1], Tiago Quintino[1,2], Stefan Vandewalle[1], and Herman Deconinck[2]

[1] Dept of ComputerScience, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium
[2] von Karman Institute for Fluid Dynamics, Waterloosesteenweg 72, B-1640 Sint-Genesius-Rode, Belgium

Abstract. We study the application of object oriented (OO) techniques in the design of parallel software for the numerical solution of partial differential equations (PDEs).
The differences between the emerging multitude of methods to solve PDEs numerically frequently force developers to limit themselves to one, or at most, to a family of related methods. The solution for this seems obvious: tackle the complexity with flexibility and code reuse, by using standard OO practices.
But at first glance, different methods appear to require incompatible interfaces. However, when looking at them from a software perspective, it is possible to devise compatible interfaces. This is accomplished by surpassing the bounds of software engineering and working with the mathematical abstractions that describe these methods, allowing us to see beyond the differences. When searching for shared functionality between different families of methods, it is straightforward to reuse linear algebra or parallel interface packages. It is hard though, to find this at the algorithmic level and it's even harder at the data storage level.
In this paper, we address how to provide clear interfaces and come up with common high-level functionality that allows to maximize code reuse and share common data structures between completely different numerical methods (e.g. Finite Element and Finite Volume Methods).
We also address the ever conflicting issue of dependency between the numerics and the physical models. Although we do not claim to solve it, we encourage the minimisation of such links by providing some design guidelines to avoid common pitfalls.

1 Introduction

In the last decades, the reuse of scientific software has gained a lot of attention and several scientific manuscripts by Ahlander, Haveraaen and Munthe-Kaas ([1]), and Cai ([3]) have addressed this issue. Researchers want to add components to an existing software package in order to be able to rapidly test new features. A multitude of methods to solve partial differential equations (PDEs) numerically, combined with a multitude of physical models call for a software environment where both the numerical methods and the physical models can easily be exchanged.

The object oriented (OO) paradigm, as described by Meyer ([10]), suits these needs perfectly, offering a set of key features to tackle the complexity with flexibility and code reuse: encapsulation, inheritance, polymorphism and dynamic binding. Gamma et al. present a catalog of solutions to software design problems in the form of design patterns ([6]).

As stated by Pree ([11]) creating a framework demands clear interfaces, defined by abstract classes. However, the different numerical methods appear to require incompatible interfaces. Mathematical abstractions describing these methods allow us to see beyond the differences and distill a common interface, at the software level. We discuss these abstractions in section 3.2. They support the development of one single multimethod solver, capable of performing simulations based on numerical methods that have not yet been put together into one single software package, as far as could be investigated.

The scientific work discussed in this paper has been done in the COOLFluiD project. COOLFluiD stands for **C**omputational **O**bject **O**riented **L**ibrary for **Flui**d **D**ynamics and is intended as a framework for numerical simulations of physical processes, targeted at conservation laws:

$$\frac{\partial \boldsymbol{U}}{\partial t} + \nabla \cdot \boldsymbol{F}(\boldsymbol{U}) = 0. \tag{1}$$

Here, $\boldsymbol{U}$ is the vector of conserved quantities and $\boldsymbol{F}(U)$ is the flux vector. For the 2D Euler model we have

$$\boldsymbol{U} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{pmatrix}, \qquad \boldsymbol{F}_x = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho H u \end{pmatrix}, \qquad \boldsymbol{F}_y = \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho H v \end{pmatrix}, \tag{2}$$

with p and H the pressure and enthalpy of the fluid.

This requires the coming together of a number of different disciplines: physics, mathematics, applied mathematics, numerical linear algebra, and last but not least software development.

This paper is organised as follows. First we sketch the background of the COOLFluiD project. In section 3 we discuss the most important elements of the multimethod design: the mathematical abstractions, the datastructure and the dependency between numerics and physics. Thereafter, we give a short overview of the preliminary results and future work. We end with some concluding remarks in the last section.

2 COOLFluiD Project

2.1 Motivation

The COOLFluiD project is a collaboration between three institutes: the von Karman Institute for Fluid Dynamics (VKI), the Centre for Plasma Astrophysics of the K.U.Leuven (CPA) and the Scientific Computing group of the department of Computer Science of the K.U.Leuven (SciCo). Their main common interests lie in computational fluid dynamics (CFD) and magnetohydrodynamics (MHD, see Bittencourt, ([2])) simulations.

In the past, the partners of VKI and CPA have developed a software package called THOR, capable of performing explicit and implicit CFD and MHD simulations on unstructured meshes in 2D and 3D. The architecture of this package however, does not allow adding and combining more complex functionality like multigrid accelleration, error estimation, grid adaptivity. Just like many other research development packages, THOR has grown from a small piece of software, designed to do just one type of simulations, into a large monolithic code, capable of performing a whole assortment of numerical simulations, but rather difficult to maintain. Numerous versions of the code have been developed, each of them adding new functionality, but none of them offering all of the functionality together.

Mastering this level of complexity, using a procedural programming language (i.c. C) is nearly impossible. Conditional expressions are abundant all over the code and adding a minor feature may require knowledge of a major part of the software, since many aspects have been hardcoded and global variables are accessed everywhere.

2.2 Object Orientation vs Procedural Programming

A possible approach to extend the procedural THOR-code, would be to create an object oriented wrapper around it and call blocks of THOR-code from within that new structure. Although this is technically achievable, it requires the original procedural code to be very well modularized. However, in the development history of THOR, high coupling and low cohesion have slipped in. While they do give a performance gain, they certainly do not support the necessary modularity. Refactoring the THOR-code is impossible, because the changes cannot be done incrementally (one of the most important characteristics of refactoring, according to Fowler et al. ([5])). The choice for a tabula rasa approach, a complete rebuild using the object oriented paradigm is clear-cut.

2.3 The Need for a Framework

As a first step, the goals of the project and the requirements of the new software package are stated. COOLFluiD is intended as a framework for numerical simulations of physical processes, defined by conservation laws. This demands the combined insight in several scientific fields: physics, mathematics, applied mathematics, numerical linear algebra, and software development.

One of the main goals of COOLFluiD is to provide a high level of exchangeability for the different components of a simulation: space discretisation, time discretisation, physical model, error estimation, multigrid, adaptivity, etc. Any combination of these components can be regarded as a single application.

Similar vs Shared Features. Based on work on numerical simulations we have come to think that a lot of the actual code of the different single applications of the family could actually be shared. The *similarities* in the resulting code have fostered this expectation. A framework however, is based on *shared features* between a number of working applications. Such shared, common parts of code need to be generalized, while the working applications remain in working condition by refactoring them to use the generalized code.

To create the generalisation on the basis of an analysis of domain semantics, without first creating a number of single applications of the family, has most often shown to be impossible. Although often semantic commonalities can be seen in the different applications of the family, these commonalities are no ideal basis for generalisation into a framework. A common theory that exists in a domain outside software development often seems to generalize different applications. However, when trying to put such a scheme into code, it turns out that the generalisation actually appeals to human subjectivity, and is not of an algorithmic nature. The basis for generalisation of code in a framework can only be the shared software structure.

In order to be able to use the shared code in a single application, fundamental redesign of that application might be necessary, and certain trade-offs in the area of performance will need to be made. It is even probable that certain detailed decisions in the domain of physics, mathematics and numerics need to be standardized to gain access to the common code base. This is also where the mathematical abstractions (see section 3.2) come into play. Yet, given the enormous cost of the development of these applications, and the current progress in computer performance, the feeling is that these trade-offs are worthwhile.

2.4 History and Experiences

At the start of the project, two people are working directly on COOLFluiD: a member of the K.U.Leuven and a member of the von Karman Institute. In the first stage of the project the typical Unified Modeling Language (UML) artifacts are created, such as a conceptual model and collaboration diagrams, following the ADI-method. ADI stands for Analysis, Design, Implementation, the three basic steps in the iterative process of OO software development. The ADI-approach, described by e.g. Larman ([9]), was chosen since it is a standard way of tackling software development problems.

The main focus was put on implementing the mesh structure in the second stage, reinforced by a third team member (from VKI). After that, we switched to another approach, because we experienced the lack of power of the ADI-approach. ADI is suitable for small projects, where you actually do not need ADI. However, it does not work well for bigger projects, and it certainly does not scale up for frameworks. The fourth team member (from K.U.Leuven) recognized this and proposed a more agile process.

eXtreme Programming. The best known type of Agile Processes is eXtreme Programming (XP). XP, as discussed by Wells ([12]) and Jeffries ([8]), is based on pair programming (two persons sitting behind one screen) and the one-room way of working. So, all persons involved in the project are in the same room and can consult eachother continuously. The big block of analysis that has to cover the whole problem in one shot is skipped and small problems are pitched into, one at a time. Small parts of software are built constantly, adding new bits and scrapping others. Short development and test cycles are inherent to XP. In this way, the framework can grow from experience. Most of the problems only arise when turning thoughts into real, compilable, running code and not during a high level, abstract analysis phase. This way we implemented the first physics and numerics modules and the communication between them.

3 Multimethod Design

In this section, we explain where the power of the multimethod design comes from. We first give an overview of the most important design patterns, then elaborate on the mathematical abstractions, necessary for this new design, but warn for a few common pitfalls. Thereafter, we discuss the design of the datastructure essential for the multimethod design. We also treat the concepts of Topological Region Sets and the TensorField, crucial for the flexibility of the design.We conclude with several design guidelines for decoupling physics and numerics.

The power of our multimethod design of the presented solver does not lie in some clever low-level optimisation tricks or implementation details. It is achieved by a fresh look at the numerics of a simulation and results in new concepts at a considerably higher level of abstraction in the software design.

3.1 Design Patterns

The implementation of the multimethod design is based on a few well known design patterns, quintessential in numerical software. Here, we only list them concisely, but we refer to Gamma et al. ([6]) for more details.

Strategy: Strategies provide a way to *configure a class with one of many behaviours.* The many variants of spatial and temporal discretisation schemes are encapsulated and interchangeable via a common interface, the blueprint of every simulation.

Command: The Command pattern is typically used *when you want to structure a system around high-level operations built on primitive operations.* The different components of the numerical schemes (Timestepper, SpaceMethod) are built from NumericalCommands, functions encapsulated in an object. Every command has the necessary intelligence to get the information it needs to perform its task on the entities in the mesh. Since all NumericalCommands respond to the same interface, they can be exchanged without trouble. This strongly promotes the modularity of the multimethod design.

Abstract Factory: All separate substituents of a Simulation are built by an Abstract Factory. This allows the *construction of a family of related or dependent objects, without specifying the concrete subclasses.*

3.2 Mathematical Abstraction

A *multimethod* PDE solver has to be able to deal with Finite Element Methods (FEM), Finite Volume Methods (FVM) and possibly a mixture of both, like Fluctuation Splitting Methods (FSM or Residual Distribution Schemes, RDS). The algorithms, representing the different methods, must be compatible with one single interface, in order to be fully transparantly callable by the PDE solver. This has not yet been done in other projects, for as far as we could check it. Our multimethod solver offers this interface.

An object oriented software interface contains a set of signatures for operations that classes, wishing to provide this interface need to implement. The interface only contains the signatures and does not administer a concrete implementation. That is left to the classes, inheriting from that abstract class (C++) or implementing that interface (Java).

This *abstract software* interface however, requires *abstraction* at the *mathematical* level. In our design of the multimethod PDE solver, we solve this with two crucial design decisions.

States vs Nodes. First, we make a distinction between states and nodes. A *state* is a vector of unknowns, stored at some location in a cell of the mesh. The state typically represents the values of the variables, needed to describe the physical model of the current simulation, e.g.: (ρ, u, v, p) for the 2D Euler model and $(\rho, u, v, w, B_x, B_y, B_z, p)$ for the 3D MHD model. A *node* is a geometrical point in space, part of the supplied or generated mesh and defining the geometry of cells in the mesh. A node does not contain any information concerning the physical model.

This decoupling is no novelty when dealing with FVM, since these methods typically store the values of the unknowns of a cell in the cell, while the cell is built from geometrical nodes. The grid of the nodes and the grid of the states are a staggered. In traditional FVM, states are not really considered as a separate entity. In FEM nevertheless, states and nodes coincide often but not always. This happens not only in the mathematics, but also in software.

No clear distinction is being made between the different views on the entities, putting them into one single class, disallowing a clear software design. There are 3 disparate cases:

– a node has to be looked at as a geometrically acutely defined point in space, with a set of unknowns (P1 elements in FEM),
– a node has only a geometrical meaning (FVM),
– a node has a set of unknowns, but the geometrical information can only be inferred and is not used explicitly (a node that is not a corner node of a P2 element in FEM).

This indisputably calls for a new software design acknowledging the distinction.

Shape Functions. The second crucial design decision affects the role of shape functions. Shape functions are well known within the FEM framework, but they are seldomly brought into relation with FVM. One can however consider FVM as having P0 shape functions, i.e. constant functions per grid cell. Higher order reconstruction for FVM can still be dealt with, by taking a stencil of cells and treating them as nodes for a higher order FEM.

From this point of view, all actions performed on a cell, whether it's in a FEM or FVM framework, pass through the shape function interface. Again, it is this *mathematical abstraction* that leads to a *software interface*, capable of dealing with two different numerical methods, never brought together into one single software package as far as we could investigate.

Pitfalls. We warn for some pitfalls with this new approach. Consistent nomenclature is essential. Mixing up between "cell" and "volume" might look like a slip of the tongue, but is a lot worse. The word "volume" bypasses the notion of the shape function, while a "cell" has a more profound background than just being a volume in space. Therefore, in FVM, we do not store the unknown in a volume, but in a cell. A cell has nodes and states, characterised by shape functions. Bypassing the notion of the shape function is equal to denying a key feature of the multimethod design.

In FVM one should work on the unknowns through the shape function, which means an extra layer of indirection but allows using different types of elements more easily and transparently.

Concerning higher order methods next to FEM, FVM uses reconstruction based on outside cells (the stencil), and Discontinuous Galerkin (DG) has extra states *in* the cell. This difference is handled transparantly by our approach, combining shape functions and the distinction between states and nodes.

3.3 Datastructure

The design capable of dealing with the multitude of methods (the algorithmic part) has to be complemented with a datastructure (the storage part), designed in an evenly transparent way, to satisfy the different numerical methods.

Every MeshGenerator configures the datastructure in such a way that all data, needed for the computation, can be retrieved easily. The connectivity tables (e.g. element-element, element-state connections) are set up when appropriate. The datastructure can be queried for any specific part, like states, edges, cells, connectivities, etc. All these aspects are stored in a DataStorage, a container that can be queried for its content with a key.

Every NumericalCommand is provided with a CommandData and has the built in intelligence to gather all necessary information. Looping over all Cells to calculate volume integrals, looping over all edges to compute the line integral of a quantity, visiting all faces in order to determine a scaled normal,... all of this is possible thanks to the encapsulation of the computation in a command and the design of the datastructure.

A timestepping method can easily build extra states at startup in the datastructure and query for them later on. A typical example is the temporary state for the second order Runge-Kutta method.

3.4 Topological Region Set and TensorField

Next to the multimethod design and its accompanying datastructure, a third design element, crucial to the flexibility of the solver are the concepts of the TopologicalRegionSet (TRS) and the TensorField.

A TRS is a set of geometric entities that require a special treatment, like the boundaries. This way, we can formulate apply different NumericalCommands to different TRSs, belonging to the same Domain.

A TensorField represents the discretised view on the Domain and serves for the transition from continuous functions to discretised numerics. A TensorField has several TRSs attached to it and can filter these, when applying NumericalCommands. This ensures that the NumericalCommands are applied to the correct TRSs. Using this approach, boundary conditions are dealt with properly.

3.5 Dependency between Numerics and Physics

Different physics often demand different numerics, so decoupling physics from numerics is no trivial mission. For our class of physics (conservation laws), we manage to decouple them, to the point that we still need symmetrisable flux jacobians.

The PhysicalModel (representing the physics) and the SpaceMethod (representing the spatial discretisation method) do not have datamembers inside pointing to eachother. Instead, the coupling is done at a higher level, in the Domain. There is one PhysicalModel and a SpaceMethod per Domain. This approach also permits **multiphysics** simulations, by defining several Domains and looping over them.

4 Preliminary Results

4.1 Conceptual Model of COOLFluiD

We briefly discuss the design of COOLFluiD, based on the conceptual model of Fig. 1. The Simulator controls the Simulation, that is defined by a NumericalModel, a DomainModel and a PhysicalModel. The NumericalModel is determined by a ConvergenceMethod (e.g. an explicit Timestepper or Multigrid) and a SpaceMethod (e.g. FSM). The main component of a PhysicalModel is the StateVector with the values of the unknowns describing the physical model and the computation of the local flux jacobian. The DomainModel takes care of the Mesh and stores the StateVectors. The TopologicalRegionSet is an important part of the DomainModel, as discussed in section 3.4. It offers a lot of the so desired flexibility.

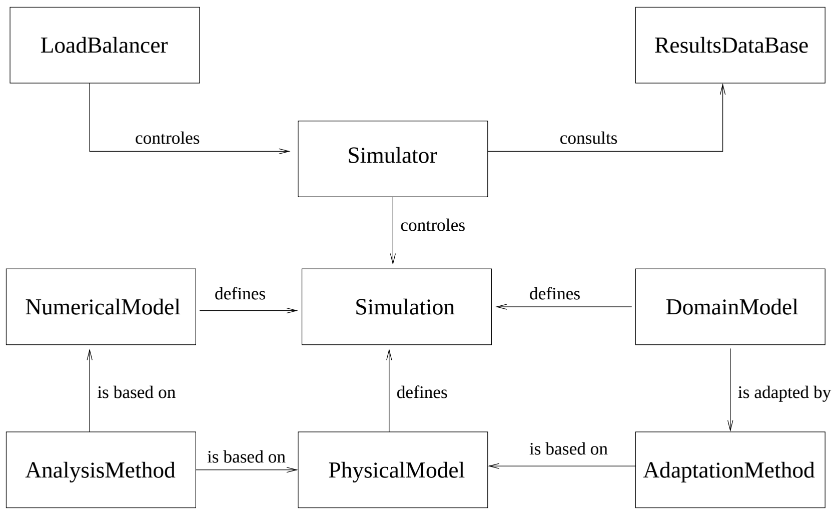


Fig. 1. Simplified conceptual model of the multimethod PDE solver COOLFluiD.

The AnalysisMethod analyses the temporary solution, based on the NumericalModel and the PhysicalModel to check e.g. if another iterationstep is necessary. The AdaptationMethod queries the DomainModel and the PhysicalModel to adapt the Mesh (h- or p-adaptivity).

The ResultsDataBase and the LoadBalancer can be considered as external code. The former is consulted by the Simulator to restart a simulation or to save results to, while the latter strives for a balanced workload on the CPUs in cooperation with the Simulator.

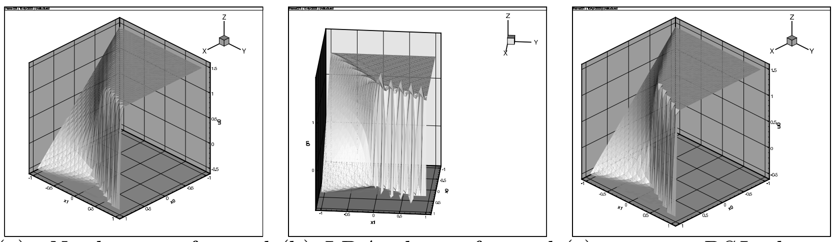
4.2 Current Implementation

As a first numerical method to implement, the Fluctuation Splitting Method (or Residual Distribution Scheme) is chosen. FSM incorporates ideas from both FEM and FVM: the representation of the unknowns originates from FEM and the discrete conservation of the flux contour integral along closed surfaces is typical for FVM. A successful implementation of FSM is a clear indication that our design will also satisfy both FEM and FVM. This justifies the decision to choose FSM first. In our multimethod solver we support the N, B, PSI and LDA schemes in a transparant way. A comprehensive overview of FSM is discussed by Deconinck et al.([4]).

A few explicit Timesteppers are implemented to illustrate the independency between the Timesteppers and the SpaceMethods: forward Euler and Runge-Kutta second order. Extensions to higher order Runge-Kutta methods are foreseen and easily realizable.

The currently implemented PhysicalModels are a scalar linear advection problem ($\partial_t u + \nabla \cdot (au, bu) = 0$), a non-linear advection equation of the Burgers type ($\partial_t u + \nabla \cdot (u^2/2, u) = 0$) and a system containing both equations. Figure 2 shows the simulation of the Burgers equation. One can clearly see the formation of the shock wave. The non-monotone character of the *LDA*-scheme is illustrated in Fig. 2(b). The boundary conditions are:

$$\begin{aligned} x0 &= -1 : u = 1.5, \quad x0 = +1 : u = -0.5, \\ x1 &= -1 : u = 0.5 - x0, \quad x1 = +1 : \text{outflow}. \end{aligned}$$



(a) *N*-scheme, forward Euler (b) *LDA*-scheme, forward Euler (c) *PSI*-scheme, RungeKutta2

Fig. 2. Simulations of the Burgers equation on a triangulated square mesh.

5 Conclusion

In this paper, we illustrated how mathematical abstractions can lead to clear software interfaces and thus form the basis of a multimethod solver for partial differential equations, PDEs. We studied the application of object oriented techniques in the design of a framework for the multimethod PDE solver.

A fresh look at the numerics of a computational simulation results in a considerably higher level of abstraction in the software design. The key design decisions in our multimethod solver are the distinction between *states* and *nodes* and the use of *shape functions* for Finite Element, Finite Volume and Fluctuation Splitting Methods. The algorithmic part in the design of our multimethod solver is complemented with a datastructure (the storage part), designed in an evenly transparent way, to satisfy the different numerical methods.

The flexibility of the design benefits also from the concepts of the Topological Region Set and the TensorField, offering a means to treat some regions in a special way (e.g. the boundaries). The coupling between numerics and physics is brought to a higher level in the design (the Domain), avoiding a tight link in the algorithms. This also permits multiphysics simulations, by defining several Domains.

The eXtreme Programming approach is the basis for the implementation of our multimethod design. The inherent constant communication and short development/test cycles provide a better modus operandi than the classic iterative Analysis-Design-Implementation scheme for a software project of this scale.

Acknowledgement

We would like to thank Andrea Lani and Jan Dockx for their contribution to the COOLFluiD project and Tom Schrijvers for his help, writing this paper.

References

[1] Ahlander, K., Haveraaen, M., Munthe-Kaas, H.: On the role of mathematical abstractions for scientific computing. Accepted for publication in the proceedings of the IFIP WG 2.5 Working Conference on Software Architectures for Scientific Computing Applications (October 2000).

[2] Bittencourt, J.A.: Fundamentals of Plasma Physics. Pergamom Press (1986).

[3] Cai, X.: An object-oriented model for developing parallel PDE software. Preprint 1998-4, Department of Informatics, University of Oslo (1998).

[4] Deconinck, H., Sermeus, K., Abgrall, R.: Status of Multidimensional Upwind Residual Distribution Schemes and Applications in Aeronautics. AIAA Paper 2000-2328 (AIAA Accession number 33804) (2000).

[5] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, Massachusetts (1999).

[6] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, Addison-Wesley Professional Computing Series (1999).

[7] Gropp, W., Lusk, E., and Skjellum, A.: Using MPI - 2nd Edition Portable Parallel Programming with the Message Passing Interface. MIT Press, Scientific and Engineering Computation Series (1999).

[8] Jeffries, R.: What is eXtreme Programming?
http://www.xprogramming.com/xpmag/whatisxp.htm (2002).

[9] Larman, C.: Applying UML and Patterns. Prentice Hall PTR, Upper Sadle River (1998).

[10] Meyer, B.: Object Oriented Software Construction. Prentice-Hall, 2nd edition (1997).

[11] Pree, W.: Design Patterns for Object-Oriented Software Development. Addison-Wesley, Reading, Massachusetts (1995).

[12] Wells, D.: eXtreme Programming: A gentle introduction.
http://www.extremeprogramming.org/index.html (2001).

Object-Oriented and Parallel Library for Natural Convection in Enclosures

Luis M. de la Cruz Salas[1] and Eduardo Ramos Mora[2]

[1] Applied Computing, DCI, DGSCA-UNAM, Mexico.
[2] Energy Research Centre, UNAM, Mexico

Abstract. In this work we present advances in the construction of a class library for solving the governing equations of natural convection in enclosures. We concentrate on newtonian and incompressible fluids. The governing equations are partial, non-linear and coupled, and are solved using the finite volume method. The pressure coupling is solved with SIMPLEC method, where a pressure correction equation is used to account for mass conservation.
The finite volume method transforms the three governing equations in linear systems that contain coefficients for diffusive and convective terms. The algebraic systems are then solved using an iterative methods.
All conservation equations are written in a general form, in such a way that it is possible to make an abstraction of the concept on a base class for all equations. In this way, it is possible to define energy, Navier-Stokes and pressure-correction equations as subclasses of the general equation class. These clases are polymorfic and their behavior is handled with "adaptors". The adaptors implement different numerical schemes and are derived from each equation using the Barton and Nackman trick.
The main issue of OOP is the performance. Many C++ constructions produce hidden temporaries that affect the peformance. In the past few years the use of the technique known as "Expresion Templates" has mitigated and in some cases solved the problem. The Blitz++ library, which includes expression templates is used in our library for array manipulation and has shown very good performance.
The library also contains a subset of classes for running in parallel architectures via MPI. The class Domain decompose the global domain in several subdomains and each one is managed for a different processor. All subdomains must get ghost boundary conditions from its neighbors. Decomposition is made in cartesian fashion in such a way that every subdomain has information from all its neighbors. This feature permits us make a persistent communication from the beginning in order to avoid the overhead of initializing sockets between processors in each iteration.

Generic Programming Support for Mesh Partitioning Based Parallelization

Guntram Berti

C&C Research Laboratories, NEC Europe Ltd.
Rathausallee 10, 53757 St. Augustin, Germany
berti@ccrl-nece.de

Abstract. We present a domain-specific, library based approach for parallelizing mesh-based data-parallel applications (like numerical PDE solution) using the domain partitioning paradigm. Concepts are presented to formalize the notions of domain and mesh partitioning. Generic programming is used to implement reusable software components that encapsulate the common core of domain partitioning and make it available to application programmers. A novel aspect of our approach is its independence of the underlying sequential structured or unstructured mesh data structures, which makes reuse of most pre-existing sequential application code possible.

1 Introduction

Despite continuously increasing power of computer hardware, parallel computing is — and will remain — the only viable option for many scientific simulation challenges. The proper use of parallel architectures poses a number of problems to the programmer, who in general has to be more explicit about the underlying hardware then he wishes. This constitutes a significant barrier to the construction of efficient parallel scientific applications.

Although research for general-purpose automatic parallelization support has been pursued for decades now, success has been limited. In our opinion, this stems from the (unavoidable) diversity of different parallelization patterns (see e.g. the work in [1]) which withstands a uniform treatment.

On the other hand, when we focus on a *specific* domain, like parallel PDE solution, or more generally, structured or unstructured mesh-based "data-parallel" applications, the situation changes. Analyzing typical parallel applications in this area, we can see that they all exhibit strong structural similarities, as they all use the same parallelization paradigm called domain partitioning.

Hence, there is hope of providing some general support for parallelizing such applications, and even to encapsulate the repetitive task entirely into reusable components, which is what we propose here.

The essential feature of our approach is that it is purely library based and works in a bottom-up way: The parallel application programmer gets supplied high-level constructs to express data distribution, but retains full control over parallel execution. No extra tools like special pre-processors or compilers are needed. Most of the existing code and data structures can be reused, the "parallel" enhancements are simply "wrapped around" it, with a very low memory overhead which is close to what a manual parallelization would need. This "wrapping" is enabled by leveraging generic programming techniques in C++, which allow to abstract from concrete data representation issues.

The reuse of original sequential data structures is a key difference to the top-down approach followed by parallel frameworks for PDE solution, like Overture [2] or AMROC [3], which concentrate on (hierarchies of) structured grids. Here, an application programmer has to reuse the (data) structures provided by the framework. This makes programming more comfortable for new developments, but also more restrictive.

The dynamic distributed data (DDD) library by Birken [4] in contrast has distributed graphs as underlying abstraction and is therefore more general than our approach. Lacking the more specialized notion of distributed meshes, it cannot directly offer specific support for them.

An approach related to our one is the GRIDS library [5], which uses scripts and a special pre-processor to generate parallel Fortran code. Other approaches centered on unstructured meshes like DIME [6] or PMO [7] are more like frameworks in that they require reuse of their native data structures.

The organization of this paper is as follows: First, we describe the domain partitioning paradigm (Sec. 2.1) and then general concepts for distributed meshes in a general way (Secs. 2.3 – 2.6). Next, their implementation using generic programming is discussed in Sec. 3. Finally, we discuss further research issues.

2 Concepts for Distributed Meshes

2.1 The Domain Partitioning Paradigm

Domain partitioning (also known as geometric partitioning) is the parallelization paradigm of choice for mesh-based applications for PDE solution, for instance using finite elements, finite volume or finite differences. It exploits the fact that the data dependencies of these spatial discretizations (*stencils*) are merely local. The computational domain (i.e. the mesh representing it) is divided into pieces of roughly the same size, and the discretization algorithms are applied to each piece in parallel. In order to guarantee efficient access to the needed information (*data locality*), some data has to be replicated on different processes by adding an overlap region (or halo) to each mesh part. Data associated to the overlap mesh region must be updated regularly in order to provide a consistent view of the global state. In general, it is most efficient to perform these updates altogether to minimize communication costs.

The domain partitioning approach implicitly assumes *data-parallel* algorithms, in which case it can produce results identical with the sequential case (except effects due to different orders of floating point operations). While the assumption of data-parallelity is often satisfied, for instance finite element stiffness matrix assembly or finite volume flux balance computation, it is violated in some important cases, most notably the solution of linear equation systems which introduce a global data dependency. In this case, an application of the paradigm produces a modified algorithm, which may or may not be appropriate (Jacobian iteration is about the only broadly used data-parallel algorithm for solving linear systems, which explains its low efficiency). More often, one uses algorithms specialized to domain partitioning, like interface iteration using Schur complements [8]. Nonetheless, the concepts like overlap, ownership and generalized stencils we are going to develop are useful for this more general situation, too.

Now, while the general ideas are common to virtually all implementations of parallel PDE solution, individual applications differ in the details:

- How large is the overlap?
- Which data has to be exchanged when?
- How is the data transferred?
- What kind of data structures are used for grids and grid functions?

In the following sections, we will see how to formalize these intuitive concepts. Further on, we will show how generic components can parameterize the associated choices, in order to provide the application programmer with exactly those high-level abstractions needed for implementing data-parallel algorithms on arbitrary grids. As the parallelization is driven by mesh distribution, distributed data structures for grids and grid functions are the essential components which can encapsulate most of the technical details.

In this paper, we will not discuss issues related to the partitioning itself, see e.g. [9] for a survey.

2.2 Some Grid-related Terminology

In order to understand the following sections, we some introduce some terminology. By a (combinatorial) grid (or equivalently, mesh) $\mathcal{G}$ of dimension d we understand a tuple $(\mathcal{G}^0, \ldots, \mathcal{G}^d)$ of element ranges of dimensions $k = 0$ (vertices) to $k = d$ (cells), together with an incidence partial ordering. Two elements e_1, e_2 are *incident*, if e_1 one is contained in the topological closure of e_2, that is, e_1 is on the boundary of e_2.

A grid function F on $\mathcal{G}$ is a mapping $F : \mathcal{G}^k \mapsto \mathcal{T}$ for some set $\mathcal{T}$. Grid functions (often called fields) are a means of storing data on grid elements, and play a crucial role in parallelization as this data has to be kept in sync on different parts.

2.3 Distributed Overlapping Grids

We can take two complementary views on distributed grids (Fig. 1). The *global point of view* sees a global grid, partitioned into local *parts* (with overlap).

We use the term parts here to emphasize the independence of the concepts of the physical distribution aspects. The *local point of view* starts from local parts, equipped with correspondence maps between overlaps of neighboring parts. This correspondence gives rise to a (virtual) global grid. Thus, the local view in general corresponds more closely to the actual physical situation, where the global grid does not actually exist. It represents also a good compromise between a more idealized global view and the actual physical distribution, which leads to a simple yet powerful and efficient programming model.

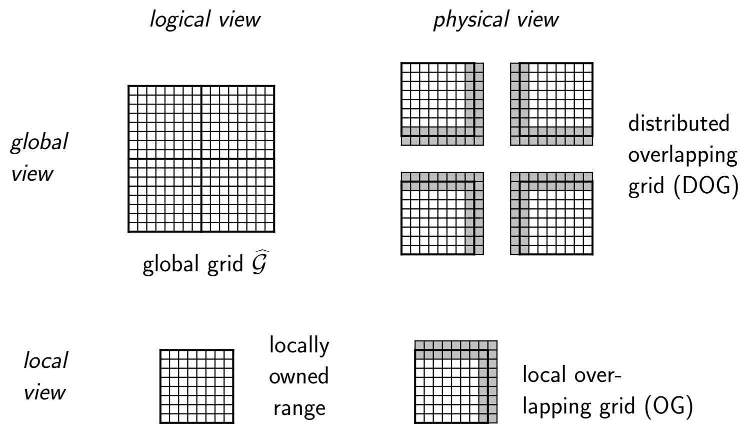

logical view
physical view
global view
global grid $\widehat{\mathcal{G}}$
distributed overlapping grid (DOG)
local view
locally owned range
local overlapping grid (OG)

Fig. 1. Components for representing a distributed grid. The global grid $\widehat{\mathcal{G}}$ is in general not physically represented. The local grid $\mathcal{G}_i$ underlies the overlapping grid OG.

The basic data structure of each local part is a *local grid* $\mathcal{G}_i$, represented by an arbitrary sequential grid data structure typically provided by the sequential application, which contains an appropriate amount of overlap $\mathcal{O}_i$. A *local overlapping grid* divides this overlap into a system of subranges, making them available to the application programmer, thus allowing him to control the scope of local computation (see Figs. 16 and 18). The top layer, *distributed overlapping grids*, complements local overlapping grids with system of correspondence mappings Φ between overlap portions of neighboring parts.

More formally, let $\mathcal{G}_i, 1 \leq i \leq N$ be a family of local grids. An *overlap structure* on $(\mathcal{G}_i)_{1 \leq i \leq N}$ is a system of grid isomorphisms on *bilateral overlaps* $\mathcal{O}_{ij}$

$$\Phi_{ij} : \mathcal{O}_{ij} \subset \mathcal{G}_i \mapsto \mathcal{O}_{ji} \subset \mathcal{G}_j$$

satisfying

$$\Phi_{ij}(\mathcal{O}_{ij}) = \mathcal{O}_{ji}, \qquad \Phi_{ij}^{-1} = \Phi_{ji}, \qquad \text{(symmetry)} \qquad (1)$$

and

$$\Phi_{ij} \circ \Phi_{jk} = \Phi_{ik} \qquad \text{on} \quad \mathcal{O}_{ij} \cap \Phi_{ji}(\mathcal{O}_{jk}) \qquad \text{(transitivity)} \qquad (2)$$

for $1 \leq i, j, k \leq N$. The local overlap of part i is given by the union $\mathcal{O}_i = \bigcup_{j=1}^{N} \mathcal{O}_{ij}$.

The functions Φ_{ij} give rise to a *equivalence relation* between elements of the local grids, with $e_i \sim e_j$ iff $e_j = \Phi_{ij}(e_i)$. The equivalence classes $\hat{e}$ form the elements of the global grid $\widehat{\mathcal{G}}$, which can thus be reconstructed from the local grids and the correspondence Φ.

2.4 Ownership on Overlaps

The overlap $\mathcal{O}_i$ of part $\mathcal{G}_i$ is now further divided into subranges reflecting ownership: We distinguish between *exposed* ranges $\mathcal{E}_{ij}$, which are owned by part i, *shared* ranges $\mathcal{S}_{ij}$, which belong to both part i and part j, and *copied* ranges $\mathcal{C}_{ij}$, that belong to part j. The following symmetry relations must hold between these ranges:

$$\begin{aligned}
\mathcal{E}_{ij} &= \Phi_{ji}(\mathcal{C}_{ji}) \\
\mathcal{S}_{ij} &= \Phi_{ji}(\mathcal{S}_{ji}) \\
\mathcal{C}_{ij} &= \Phi_{ji}(\mathcal{E}_{ji}) \\
\mathcal{O}_{ij} &= \mathcal{E}_{ij} \cup \mathcal{S}_{ij} \cup \mathcal{C}_{ij}
\end{aligned}$$

and $\mathcal{E}_{ij}, \mathcal{S}_{ij}, \mathcal{C}_{ij}$ are pairwise disjoint. See figure 4 for the general picture.

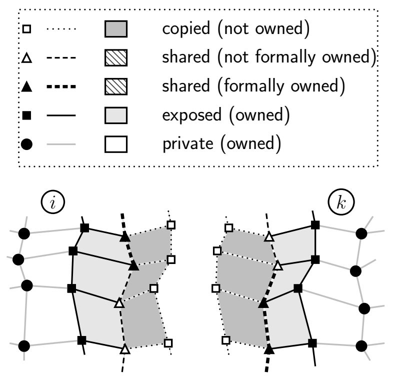


Fig. 2. Typical 1-cell overlap example, without shared cells.

For the classical Schwarz domain decomposition technique, there are no shared ranges, and the overlaps are disconnected (Fig. 6). For the so-called non-overlapping domain decomposition, we have only shared ranges (Fig. 5).

These bilateral ranges are useful primary for data exchange between different parts. For deciding where calculation has to take place, and where not, so-called *total ranges* are needed, which are roughly the unions of the bilateral ranges:

$$\begin{aligned}
\mathcal{P}_i &= \mathcal{G}_i \setminus \bigcup_{j=1}^{n} \mathcal{O}_{ij} && \textbf{p}\text{rivate} \\
\mathcal{C}_i &= \bigcup_{j=1}^{n} \mathcal{C}_{ij} && \textbf{c}\text{opied} \\
\mathcal{S}_i &= \bigcup_{j=1}^{n} \mathcal{S}_{ij} \setminus \mathcal{C}_i && \textbf{s}\text{hared} \\
\mathcal{E}_i &= \bigcup_{j=1}^{n} \mathcal{E}_{ij} \setminus (\mathcal{S}_i \cup \mathcal{C}_i) && \textbf{e}\text{xported}
\end{aligned}$$

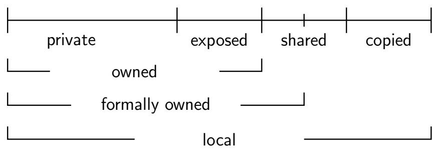


Fig. 3. Logically layered structure of overlap ranges

On exposed elements in $\mathcal{E}_i$, the local part is exclusively responsible for performing any calculation. On the shared range, several parts will in general contribute to the calculation, or calculations are done redundantly on each part. On copied elements, typically no local calculations are performed. Besides these elementary ranges, we often need a *formally owned* range $\mathcal{F}_i \subset \mathcal{P}_i \cup \mathcal{E}_i \cup \mathcal{S}_i$ (Fig. 3) which further split the shared elements in an arbitrary way, such that $\mathcal{F}_j \cap \Phi_{ij}(F_i) = \emptyset$ for $i \neq j$. This is useful e.g. for implementing global reduction operations.

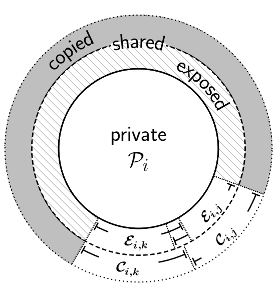


Fig. 4. Generic overlap configuration

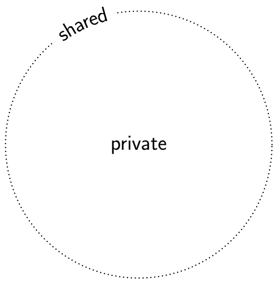


Fig. 5. Configuration for "non-overlapping" domain decomposition

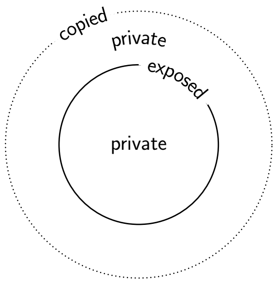


Fig. 6. Configuration for Schwarz domain decomposition

2.5 Distributed Overlapping Grid Functions

Data *on* the grid is given by *grid functions* in the sequential case, and by *distributed grid functions* (DGFs) in the distributed case. Because a DGF is multiply defined for overlap elements, we must define its *consistency state.*

With an overlapping grid function F (the local representative of a DGF), we associate a write range W a read range R, and a function f defining the algorithm used to define the values of F. Typical values for W is the owned range, and for R the local range (cf. Fig. 3).

We now assume f to be a function

$$f : \mathcal{G} \times S_f \mapsto \mathcal{T}$$

which depends on a state S_f of variables, for example other grid functions. If $F \notin S_f$, we call the loop

```
for all $e \in W$ do
    $F(e) \leftarrow f(e, S_f)$
```

a *data-parallel loop*, and f a data-parallel algorithm for F and S_f. For example, if F is a vector on grid vertices, and f is the local Jacobi relaxation on a vertex, then the loop is data parallel. In contrast, for the Gauss-Seidel relaxation, the loop is *not* data-parallel, the corrected values are immediately used.

It is now straightforward to define consistency on the tuple (F, f, W, R), given a distributed grid $(\mathcal{G}_i)_{1 \le i \le N}$ with correspondence relation Φ: F is *locally consistent* on a range $L \subset W$, iff $F(e) = f(e) \quad \forall e \in L$. A distributed grid function $(F_i, f, W_i, R_i)_{1 \le i \le N}$ is *globally consistent*, iff each (F_i, f, W_i, R_i) is locally consistent on W_i and for each pair $e_i \in R_i, e_k \in R_k$ with $\hat{e}_i = \hat{e}_k$ we have $F_i(e_i) = F_k(e_k)$. In other words,

$$F_i = F_k \circ \Phi_{ik} \quad \text{on} \quad R_i \cap \mathcal{O}_{ik}$$

The action of making a DGF globally consistent is called *synchronization*.

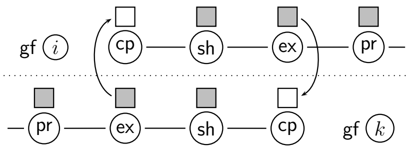


Fig. 7. Redundant calculation on shared ranges

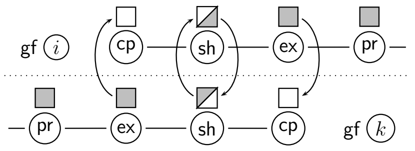


Fig. 8. Partial calculation on shared ranges

On shared elements, two different strategies of computation are possible: Either redundant (Fig. 7), in which case each part calculates its results on the shared range locally, or partial (Fig. 8), in which case the results from all sharing parts have to be combined afterwards. For example, when computing a finite element stiffness matrix in a non-overlapping configuration like in Fig. 5, the rows belonging to shared vertices have to be added if a consistent representation of the matrix is needed.

The concepts presented in this section are surprisingly powerful for implementing parallel grid-based algorithms. However, in order to provide a complete solution of the parallelization problem, there must be a means of generating the data structures like overlap ranges defining the distributed grid. This task is tackled in the following section.

2.6 Stencils and Overlap Generation

In finite difference terminology, the term *stencil* is used to denote the local data dependency of an algorithm (discretization), and typically described by (the non-zero pattern of) a matrix. If we want to generalize stencils to unstructured meshes, we must abandon the matrix notion.

```
**for all** Cells $c \in \mathcal{G}^d$ **do**
  flux($c$) = 0;
  **for all** Neighbor cells $c'$ of $c$ **do**
    flux($c$) += numflux($c$,$c'$,U)
```

Fig. 9. A simple finite volume flux balance

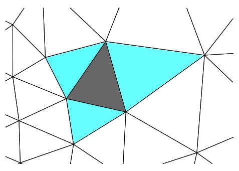


Fig. 10. its associated stencil $(2, 1, 2)$, or $(d, d-1, d)$ dimension-independently

Looking at the algorithm in figure 9, we see that this algorithm contains a global loop over all cells, and for each cell c, every neighbor cell is accessed, i.e. every cell sharing a facet with c. The stencil of this algorithm is shown graphically in figure 10. It can be described algebraically by the sequence $(d, d-1, d)$ (where d is the dimension of the mesh), meaning "start from grid elements of dimension d, go over incident elements of dimension $d-1$ (the facets) to incident elements of dimension d (neighbor cells)".

If we analyze each algorithm in a program this way, we arrive at a number of different stencils. These can be used to find the necessary overlap for a partitioned mesh.

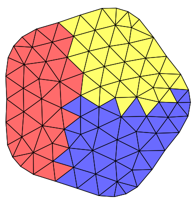


Fig. 11. A tiny mesh, partitionend into 3 parts.

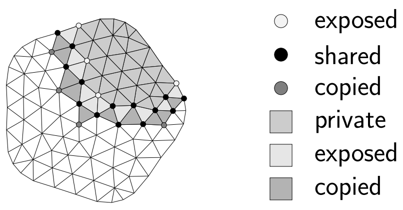


Fig. 12. Overlap for one part generated from the stencil of Fig. 10

3 Generic Components for Distributed Meshes

Our aim is to develop software components which implement the concepts presented before in a way which is as general as possible, while being easily adaptable to the specific application at hand, and inducing as few changes as possible in existing application code.

To achieve this aim, we extend the functionality of the sequential program layer by layer by the additional functionality needed for a parallel application.

A given sequential grid data structure forms the core layer. All local, application-dependent, grid-related algorithms continue to be based solely on this data structure and thus do not need to be changed. Any global, distribution-dependent functionality is provided by an additional layer – distributed grids. This layer itself is configurable in order to be adaptable to different physical distribution contexts and communication libraries.

This layering leads to a clean separation of algorithmic from distribution aspects, and is crucial for the a-posteriori parallelization of existing applications, where algorithmic code is based on a specific sequential grid data structure.

The technical basis for this data structure independent approach is given by a *generic programming* technique, which we discuss very briefly in the next section. The solution we are going present is based on C++ and is applicable directly only to applications also written in C++, however, providing interfaces to C and Fortran would be possible.

3.1 Generic programming

Generic programming aims at separating data representation details from higher level components like algorithm implementations. Thus, a new level of generality and reusability is achieved. One of the most prominent examples for generic programming is the C++ Standard Template Library (STL) [10] part of the C++ standard library, which deals with sequences (containers) and algorithms on sequences.

```
IN: $U : \mathcal{G}^d \mapsto \mathbb{R}^p$
OUT: avg $: \mathcal{G}^0 \mapsto \mathbb{R}^p$
  for all Vertices $v \in \mathcal{G}^0$ do
    avg($v$) = 0, vol = 0
    for all cells $c$ incident to $v$ do
      avg(v) += volume(c)*U(c)
      vol += volume(c)
    avg(v) = avg(v)/vol
```

Fig. 13. Algorithm for vertex averaging of cell values

```
grid_function<Cell,  state> U;          // IN
grid_function<Vertex,state> avg(grid,0);// OUT

for(VertexIterator v(grid); v; ++v) {
  double vol = 0;
  for(CellOnVertexIterator cv(*v); cv; ++cv) {
    avg[*v] += geom.volume(*cv) * U(*cv);
    vol_sum += geom.volume(*cv);
  }
  avg[*v] /= vol_sum;
}
```

Fig. 14. Generic implementation of the vertex averaging

An approach similar in spirit suitable for meshes has been investigated by the author within the open source Grid Algorithms Library GrAL [11]. With GrAL, it is possible to develop algorithms and derived data structures which are independent of the underlying mesh data structures, a feature which is crucial for implementing the concepts presented before in a truly reusable and general way.

The concepts of GrAL cannot be discussed here in detail, the interested reader is referred to [12] or [13]. Here, we restrict ourselves to an example. Looking at figure 14, we see a generic implementation using the GrAL interface of the algorithm of Fig. 13, which volume-averages cell values on vertices.

The generic class template `grid_function` allows to store any type of data on any grid element type (e.g. cell or vertex). With a `VertexIterator`, we can iterate through all vertices of a grid (or a subrange!); a `CellOnVertexIterator` allows to access all cells incident to a vertex.

We notice that this code is independent of the mesh dimension, type (structured/unstructured) and representation. It only requires the two iterator types mentioned before to be defined for the actual grid type. In a similar way, the data structures and algorithms for distributed grids discussed below can be implemented, and "wrapped around" a user supplied sequential grid data structure. To enable this wrapping, a user has to provide a description of his grid data structure in terms of a GrAL-compliant interface.

3.2 Components for Distributed Grids and Grid Functions

In the following, we discuss several generic software components which can be used to enhance a given sequential grid data structure with support for distributed programming. In order to keep the exposition short, we do not give syntactic details – the reader is referred to the reference implementation available in GrAL. A high-level view on the components relationship in an UML-like notation is given in Fig. 15.

Overlap ranges A data structure for representing the overlap of a distributed grid must provide efficient access to any segment (copied, shared etc., see Fig. 3) of both total and bilateral ranges, for any grid element type needed (cells, edges, vertices etc.). As the segments can be combined by simple unions of adjacent elementary segments (i.e. private, exposed, shared, and copied, Fig. 3), it is straightforward to derive an efficient representation for both total and bilateral overlaps for each element dimension by using arrays of *element handles* (a sort of minimal element representation, typically an integer, defined in GrAL).

Using the overlap data structure (`Overlap<Grid>` in Fig. 15) combining these layered ranges, a programmer of a parallel application can then access total ranges, for instance all copied cells, all formally owned vertices etc., for constraining iteration in a parallel loop. Also, the distributed grid functions can access bilateral ranges for communication. Bilateral ranges are used to implicitely represent the correspondence mappings Φ_{ij} by a consistent ordering or their elements.

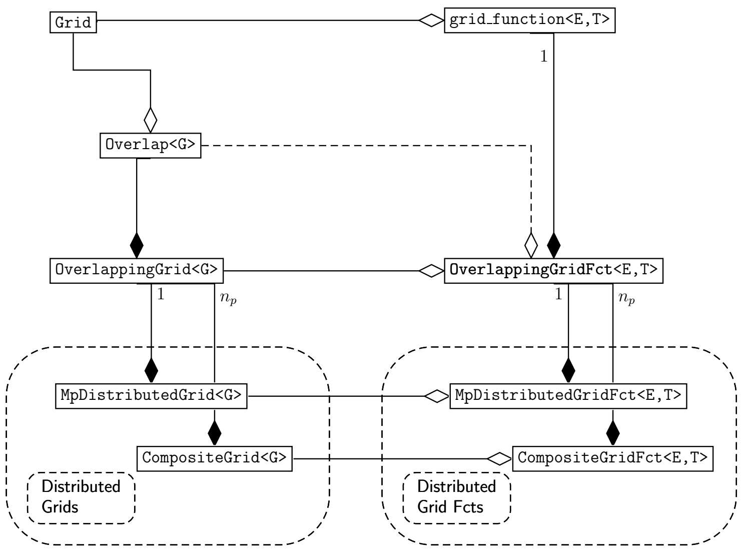


Fig. 15. Components for distributed overlapping grids and their relationships. Overlapping grid functions may contain their own overlaps (dashed line).

For the total overlap, it is in principle possible to manage with $O(1)$ memory overhead if we can sort the mesh elements in a way consistent with the elementary segments.

Overlap ranges are contained in overlapping grids, and optimized (i.e. smaller) overlap ranges may also be part of overlapping grid functions.

Distributed overlapping grids We distinguish between basic *overlapping grids* (OGs) with local semantics, and *distributed overlapping grids* (DOGs) with global semantics, representing the complete grid. Depending on the physical nature of distribution, there are different versions of distributed overlapping grids, for example message passing DOGs for distributed memory architectures, which typically contain a single OG, or composite grids, which are a set of OGs living in a single global memory. Such composite grids can be used for shared-memory parallelization using threads, to avoid cache conflicts and false sharing, and to test a parallel application sequentially.

Overlapping grids are used to factor out common functionality that is independent of the physical distribution. They consist of a user-supplied mesh data structure, enhanced with a GrAL-compliant interface, and an overlap range data structure like that described before.

Distributed overlapping grid functions For grid functions, the same distinction between local and global points of view as in the case of grids leads to *overlapping grid functions* (OGFs) on the one hand, and *distributed grid functions* (DGFs) on the other hand, where the latter group has further ramifications depending on the underlying DOG type, for instance message passing distributed grid function or composite grid function.

An overlapping grid function contains a user-defined, GrAL-compliant sequential grid function, has a reference to an overlapping grid, and optionally might contain an own overlap which is a subset of the overlap provided by the OG. Thus, an overlapping grid function which is used only on formally owned cells can avoid superfluous data communication.

A distributed grid function adds communication handlers to an OGF, which encapsulate the protocol for copying data, for instance using MPI or simple memory copy. It provides methods `begin_synchronize()` and `end_synchronize()` for bracketing the updating process, and thus allows for asynchronous communication and overlapping of calculation and communication.

A DGF is responsible for performing synchronizations in the right order for correctness, for instance, shared ranges must be updated before copied/exposed ranges if partial calculation is active.

Global reduction operations are defined on DGFs, parameterized by a binary reduction operator (for instance sum or maximum). These operation rely on formally owned ranges. In some cases, the binary reduction operator may require additional information, in which case the user may want to perform an explict local reduction by hand (cf. CFL-number computation in Fig. 17). A global reduction on the scalar values is then performed by a generic routine `global_reduce()`.

3.3 Overlap Generation

A cornerstone of the whole approach is the ability to generate the overlap with correct extent automatically, based on a stencil, and starting from a partitioned grid. For doing so, the concept of the *hull* generated by a stencil and a set of germ elements is crucial. Intuitively, the hull is defined by starting at the germ elements and successively adding incident elements according to the stencil. This is described in more detail in [14].

Overlaps can now be computed either from a partitioned global grid, or from an already distributed grid containing the correspondence information for shared elements. In general, a user only has to provide a partitioning and the stencil, all subsequent steps can be handled automatically.

3.4 Application Examples

Below, we present some simple but typical examples for parallel algorithms using the components presented before. In Fig. 16, we see the parallelized version of the sequential vertex avering of Fig. 13. We observe that the basic parallelized algorithm is almost identical to the sequential version, only the loop bounds and the final synchronization operation have to be added. A similar observation applies to the explicit reduction loop in Fig. 17.

```
IN: U: $\mathcal{G}^d \mapsto \mathbb{R}^p$
OUT: flux: $\mathcal{G}^d \mapsto \mathbb{R}^p$
  for all Cells $c \in$ owned$(\mathcal{G})^d$ do
    flux($c$) = 0;
    for all Neighbor cells $c'$ of $c$ do
      flux($c$) += numflux($c$,$c'$,U)
  synchronize(flux)
```

Fig. 16. Simple parallel version of Algorithm 9

```
IN: U: $\mathcal{G}^d \mapsto \mathbb{R}^p$
OUT: maxcfl: maximal CFL value
  for all Cells $c \in$ formally owned$(\mathcal{G})^d$ do
    maxcfl ←max(maxcfl, cfl(c,U));
  maxcfl ←global_reduce(maxcfl, max);
```

Fig. 17. Parallel reduction code: Explicit loop

If we want to optimize it by overlapping computation and communication, the straightforward solution is to duplicate the loop (Fig. 18), which is ugly. A better possibility is to use iteration over different ownership ranges, as indicated by figure 19 (not yet implemented in GrAL). Yet another possibility would be to implement a sort of `for_each_cell()` template function, which takes the loop body as an argument. This has the advantage of making parallelity more explicit in the interface, while allowing an arbitrary ordering of computation behind the scenes. While it is in principle possible to implement such a loop template in C++ by writing a specialized class executing the loop body (local algorithm), it turns out to be clumsy, and to require substantial changes to program organization. Unlike functional languages, C++ does not support closures or unnamed "lambda functions". Approaches like the boost lambda library [15] are probably only of limited help here.

```
IN: U: $\mathcal{G}^d \mapsto \mathbb{R}^p$
OUT: flux: $\mathcal{G}^d \mapsto \mathbb{R}^p$
  for all Cells $c \in$ exposed$(\mathcal{G})^d$ do
    flux($c$) = 0;
    for all Neighbor cells $c'$ of $c$ do
      flux($c$) += numflux($c$,$c'$,U)
  begin_synchronize(flux)
  for all Cells $c \in$ private$(\mathcal{G})^d$ do
    flux($c$) = 0;
    for all Neighbor cells $c'$ of $c$ do
      flux($c$) += numflux($c$,$c'$,U)
  end_synchronize(flux)
```

Fig. 18. Parallel version of Algorithm 9, overlapping communication and computation. Note the unwanted duplication of the loop body.

```
IN: U: $\mathcal{G}^d \mapsto \mathbb{R}^p$
OUT: flux: $\mathcal{G}^d \mapsto \mathbb{R}^p$

  for all Rge. $R \in \{\text{exp}(\mathcal{G})^d, \text{prv}(\mathcal{G})^d\}$ do
    for all Cells $c \in R$ do
      flux($c$) = 0;
      for all Neighbor cells $c'$ of $c$ do
        flux($c$) += numflux($c$,$c'$,U)
    begin_synchronize(flux, R)


  end_synchronize(flux)
```

Fig. 19. Parallel version of Algorithm 9, using range iteration to overlap computation and communication. Here, only one loop body is needed.

The components described here have been used to develop a generic two-dimension solver for one- and two-components Euler solver which is described in [16]. This application uses the same generic code for sequential and parallel executables, and achieves typical parallel efficiency for explicit discretizations. Also, an application for solving the two-dimensional, incompressible, stationary Navier-Stokes equations using a SIMPLE pressure correction method has been parallelized *a posteriori*, i.e. using the original data structures. Only a few dozens of lines of the original code had to be changed, plus the creation of a GrAL adaptation layer for the original mesh data structure, which could be developed and tested completely separatly. The straight forward parallelization of the nested iteration of the SIMPLE algorithm worked correctly, but is not very efficient. Here, algorithms better suited to the distributed case have to be applied.

4 Conclusion

We have presented a set of concepts formalizing the notions of domain and mesh partitioning based parallelization. Building on that, we have shown how generic, reusable software components can encapsulate many of the related technical issues, thus making high-level parallel programming possible in the context of mesh-based, data-parallel applications. A key feature of these components is their non-intrusiveness: They allow to continue reusing existing sequential application code and data structures to a great extent, thus radically cutting down the parallelization work.

A necessary step for applying the generic components is the definition of a GrAL-complying interface for the mesh data structures of the sequential application. Albeit in principle not difficult, it can be a substantial barrier. GrAL already contains support for such adaptation, but it may be worthwhile to go a step further and create almost ready-made solutions for typical cases.

A useful enhancement to distributed grid functions would be to allow separate overlap ranges per grid functions, which could be organized in an overlap data base to save memory. Also, iteration over ownership ranges (Fig. 19) will be useful.

The concepts and components described here are for static mesh distribution. While for the concepts, nothing would change for dynamic mesh distribution, components for dynamic distributed grids would be highly useful. As for the automatic overlap generation, the complicated machinery could probably completely hidden behind a simple interface.

We plan to base the parallelization of an octree based mesh generator on top of the parallel components described here. The somewhat more irregular algorithmic patterns of a mesh generator may result in additions to the components, such as distributed partial grid functions.

References

[1] Massingill, B.L., Mattson, T.G., Sander, B.A.: Patterns for parallel application programs. In: Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP 1999). (1999)

[2] Brown, D.L., Chesshire, G.S., Henshaw, W.D., Quinlan, D.J.: OVERTURE: An object-oriented software system for solving partial differential equations in serial and parallel environments. In: 8th SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis. (1997)

[3] Deiterding, R.: AMROC – blockstructured Adaptive Mesh Refinement in object-oriented C++. http://www.math.tu-cottbus.de/ deiter/amroc (2002)

[4] Birken, K.: Semi-automatic parallelisation of dynamic, graph-based applications. In D'Hollander, E., Joubert, G., Peters, F., Trottenberg, U., eds.: Parallel Computing: Fundamentals, Applications and New Directions, Elsevier Science (1998)

[5] Geuder, U., Härdtner, M., Reuter, A., Wörner, B., Zink, R.: GRIDS — a parallel programming system for grid-based algorithms. The Computer Journal **36** (1993) 702–711

[6] Fox, G.C., Williams, R.D., Messina, P.C.: Parallel Computing Works! Morgan Kaufmann Publishers (1994)

[7] Terascale LLC: Parallel Mesh Object (PMO) home page (2002)

[8] Kuznetsov, S., Lo, G.C., Saad, Y.: Parallel solution of general sparse linear systems. Technical Report UMSI 97/98, Minnesota Supercomputer Institute, University of Minnesota (1997)

[9] Elsner, U.: Graph partitioning – a survey. Technical Report SFB393/97-27, Technische Universität Chemnitz (1997)

[10] Lee, M., Stepanov, A.A.: The standard template library. Technical report, Hewlett-Packard Laboratories (1995)

[11] Berti, G.: GrAL – the Grid Algorithms Library. http://www.math.tu-cottbus.de/ berti/gral (2001)

[12] Berti, G.: GrAL – the Grid Algorithms Library. In Sloot, P.M.A., Tan, C.J.K., Dongarra, J.J., Hoekstra, A.G., eds.: Proceedings of ICCS 2002, part 3. Volume 2331 of LNCS., Springer (2002) 745–754

[13] Berti, G.: Generic software components for Scientific Computing. PhD thesis, Faculty of mathematics, computer science, and natural science, BTU Cottbus, Germany (2000)

[14] Berti, G.: A calculus for stencils on arbitrary grids with applications to parallel pde solution. In Sonar, T., Thomas, I., eds.: Proceedings of the GAMM Workshop on Discrete Modelling and discrete Algorithms in Continuum Mechanics, Logos Verlag Berlin (2001) 37–46

[15] Järvi, J., Powell, G.: The lambda library: Lambda abstraction in C++. In: Proceedings Second Workshop on C++ Template Programming, Tampa Bay, OOPSLA2001 (2001)

[16] Schenk, K., Bader, G., Berti, G.: Analysis and approximation of multicomponent gas mixtures. In Feistauer, M., Kozel, K., Rannacher, R., eds.: Proceedings of the 3rd Summer Conference Numerical Modelling in Continuum Mechanics, Prague (1997)

PEDPI as a Message Passing Interface with OO support

Zoltán Hernyák

Eszterházy Károly College
Department of Information Technology
1. sqr. Eszterházy , Eger 3300 Hungary
aroan@ektf.hu
http://sztech.ektf.hu/~aroan

Abstract. [1]
The Parallel Event-driven Programming Interface is a middleware - developed in the Microsoft .NET environment - that supports Object-oriented parallel program development. By using it applications can be created that are based on the type-checked and multi-parted message communication system. Addressing the messages can be independent from the physical positions and the platform of the nodes. Resolving the address is handled in the background according to standard TCP/IP. Taking advantage of this and the .NET architecture, the binary distribution of the application is capable of running on grids and clusters, having various platforms (currently Windows and Linux platforms are supported). Developed in the .NET environment, these distributed applications are backed up with plenty of supply by the services of .NET's Base Class Library and even with the advanced features of OO programming (garbage collector, exception handling etc). With the help of the already defined interfaces in the system, the objects of the user's program can be easily linked to the process of sending and receiving messages. Beside the usual way of sending and receiving blocking messages, callback based message handling is also available. The incoming messages can be handled seperately by these callback functions. The system supports the writing of multi-threaded programs, and the calling of these callback functions can be within a thread or in a thread group. An easily customizable possibility of logging the messages for debugging purposes is included in the system. . . .

1 The Distributed Programming System

P.E.D.P.I. is a message passing interface, by using it we can develop a distributed application in Microsoft .NET environment.

First the environment, where these applications can be run, will be described.

The "**Developer's computer**" is the computer, on which the developer works.

[1] Partially supported by OTKA T037742 and 0089/2002 IKTA

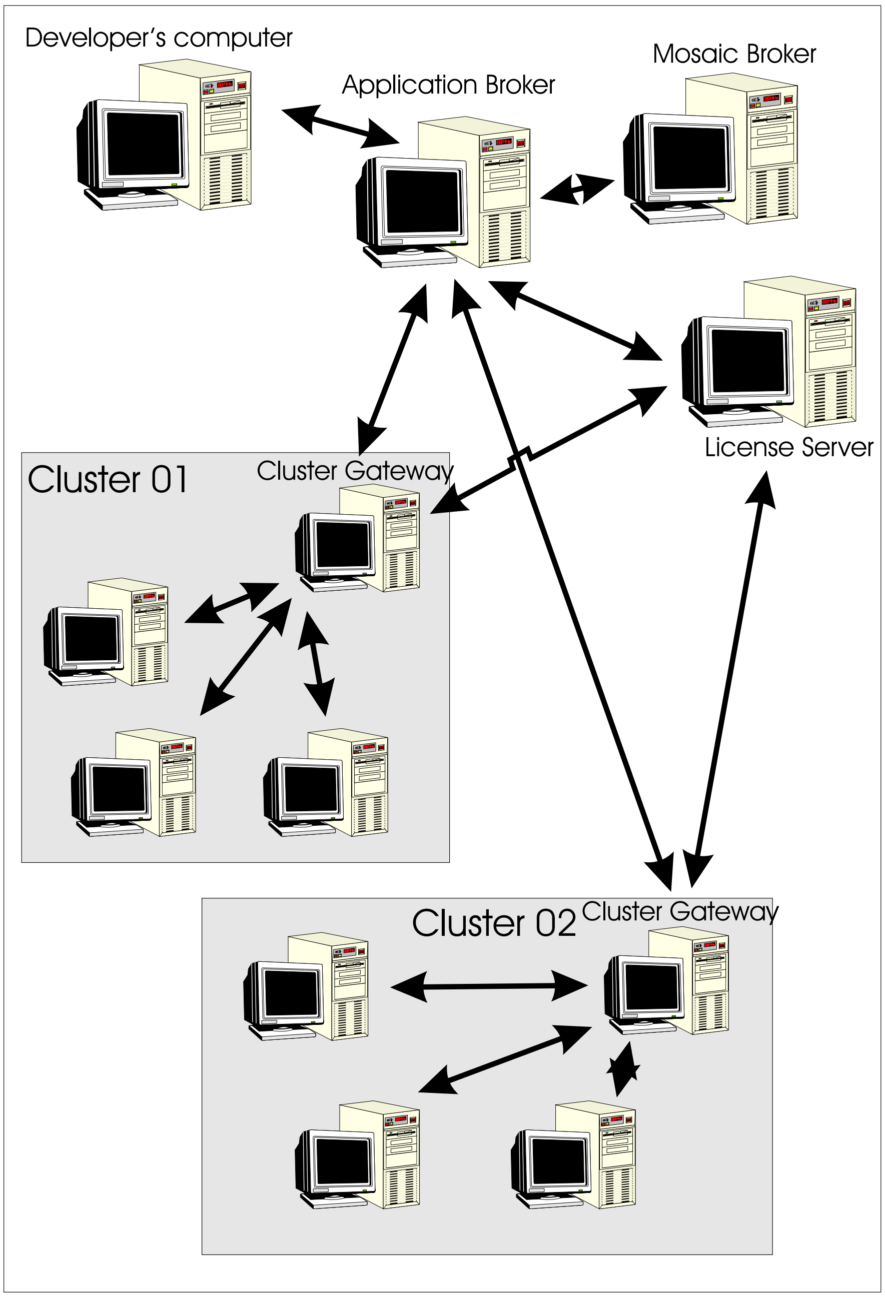


Fig. 1. The Distributed Programming System

The "**Application Broker**" is a server, to whom the developer can send the application with small descriptive configuration remarks. In these remarks the developer writes down the resources needed by the application, and the licenses he has to run. The broker checks the licenses using a "**License Server**", and determines the clusters where the license grants the rights for the resources, and a scheduler looks after the application.

The "**Cluster Gateway**" is the main computer which can start applications on the cluster controlled by it.

The "**Mosaic Broker**" is another server with an Open Programming Interface support. In this server OO classes can be stored with a well-defined assignment. The distributed application can specify which class is needed for its run. The Application Broker asks the Mosaic Broker for a class, and completes the application with the best class matching the actual conditions.

This system is under development: several functions are working now, but a number of points and protocols are under construction.

2 The P.E.D.P.I. itself

Physically PEDPI is a DLL (Dynamic Link Library) developed in C#. It is a middleware with several useful functions - and independent from the environment introduced previously. The connection between the two products is that the environment is the one which schedules the PEDPI applications. The next version of PEDPI, which is under development, will be closer to the environment: dynamically allocates nodes for computing, using the Servers and the scheduler in the process.

2.1 Messages in PEDPI

Since PEDPI is a message passing interface, understanding the message is essential. In contrast to MPI, PEDPI sends typed messages. A message can consist of one or more parts. Every part has a unique type ID. The built-in types in C# can be completed with user defined types (classes).

2.2 User-defined objects as the part of a message

A user defined class can be part of a message when it fulfils the following requirements:

– has to implement an "`ISelfDescriptive`" interface with two simple methods:
  - `public bool WriteMyselfToStream(BinaryWriter write)`
  - `public void ReadMyselfFromStream(BinaryReader read)`

– has to have a default (parameterless) public constructor

– has to be registered by `RegisterUserObject( Type myType)` method

The "WriteMyselfToStream" is analog to the serialization technic, (read and write itself to a stream). The reason for not using the serialization methods is that this is not an object serialization - it is the sending of a message. An object may not want to send all the data about itself as a message.

When a user-defined object wants to be a part of a message, the method "WriteMyselfToStream" will be called during the sending procedure. On the other side, during the receiving, an object will be created using the default constructor, then the "ReadMyselfFromStream" will be called. The object has to control its writing and reading from the stream.

An object can not cause an error during reading itself from the stream, because the stream behaves as if it contained only that part of the message. It can't read more data than that was written for this part.

When the receiver has no registration to that user defined class, this part of the message will be a raw data (byte array).

Example of a simple PEDPI Program

```
class myClass:PPE.PEDPI.ISelfDescriptive
{
   private int myData=0;
   //...............................................
   public myClass()
   {
   }
   //...............................................
   public bool WriteMyselfToStream(BinaryWriter write )
   {
      write.Write(myData);
      return true;
   }
   //...............................................
   public void ReadMyselfFromStream( BinaryReader read )
   {
      myData = read.ReadInt32();
   }
   //...............................................
   //... myClass's computation methods are here ...
   //...............................................
}

static void Main()
{
  TContext myContext = PEDPI.Start();
  Message msg = new Message();
  double d = 1.1;              msg.Add( d );
  myClass c = new myClass();   msg.Add( c );
```

```
    myContext.SendMsg( msg, myTag, targetNode );
    PEDPI.Stop();
  }
```

When a message is constructed, it can be send to a node. A node must be a member of a context. A context is a group of nodes. Inside a context every node has a unique ID, which is an integer value (like in MPI). In the PEDPI there is a context node table, which stores the additional information on the target node - eg. its IP address and port. This table is synchronized between the nodes, by using system level messages, and can be modified when a node wants to define a new context, or a node is attaching dynamically to the running distributed application.

In the beginning, every node is in the default context. Any node can initiate to built up a new context specifying the nodes in the new set.

On the receiver side the arriving stream will be automatically separated into the original data types and data values. Therefore the receiver application can use the data immediately:

```
Message msg = mpi.BReceive( myTag );
Console.WriteLine("Number of parts={0}",msg.MsgLength);
```

The parts of the message can be processed incrementally:

```
if (msg.isDouble()) // is the next part a double?
   Console.WriteLine("1.st part={0}",msg.asDouble());
```

or by its index:

```
if (msg.isBool(2)) // is the 2nd part a bool ?
   Console.WriteLine("2nd part={0}",msg.asBool(2));
```

A message can be sent only in asynchronous mode. During sending, a communication channel will be built up between the sender and the receiver node. If it is impossible to create, an exception will be raised. Thus the sender node will know if the message has arrived correctly to the target node or not, but will not know if this message is processed or not. The message on the receiver node will arrive into a ”Message Store” object, which will store the message until the application wants to process the message.

On the other side, the node can receive the message in a blocking or in a non-blocking mode, or through a callback function. The blocking and non-blocking mode is the same as in other systems: the application can use a ”peek” function to check if a special message has arrived or not, or can wait until a specified message arrives.

3 Callback functions

The callback functions can become a new technique to process incoming messages. The application can contain several callback functions. These functions can be placed into groups, which has attributes.

A callback function can be:

- **Single Grouped** callback function: single means that only one method from the group set can be active at the same time. The developer can create many groups, and place the callback methods into separate groups. The methods in the same group act like a one-threaded application. The methods in different groups act like a multi-threaded application.
- **Single Standalone** callback function: this method can only be activated when it has finished processing the previous message. This is like a Single Grouped function, which stands in a group alone.
- **Multi-threaded** callback function: this method can be activated immediately when the message is arriving - even if it is running on processing the previous message. This type of callback functions can be used to monitor or debug messages and the application itself.
- **Blocking receive**: in fact, it is not a callback function, however, when a thread uses a blocking receive the thread will be suspended, and a callback function will be automatically installed. When a message, which satisfies the requirements of the blocking receive, is arriving, this callback function will be activated, and will resume the thread, so the thread can continue its run.

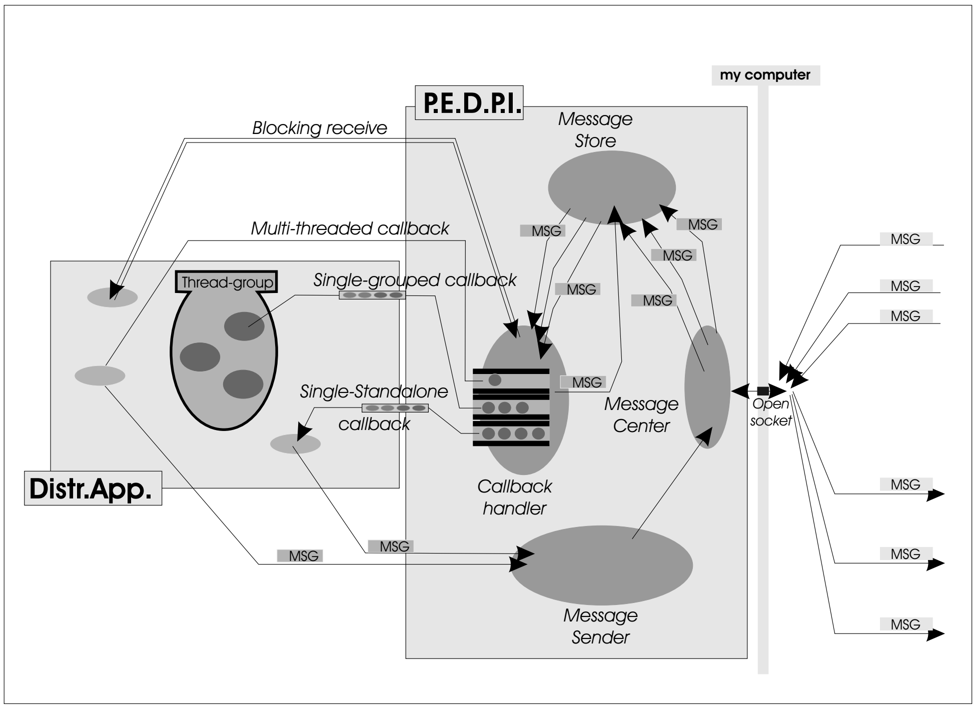


Fig. 2. Inside P.E.D.P.I.

The callback functions must be registered into a context, specifying the messages it can process. This is a message filter, where one can specify the tag-s. A "tag" is a user- defined integer value attached to the message. Using it the application can scatter the messages very quickly in a "`switch`".

PEDPI uses a TCP/IP socket, through which it can receive the incoming messages. The socket is listened by the "**Message Center**" object, which can receive system level messages and application messages.

The "**Message Store**" stores the incoming messages until its process is finished. The incoming message is stored immediately, then the "**Callback Handler**" is informed about the message. It selects a registered callback function from its own list, and activates one. When none can be activated - it will not do anything. It means that the message needs to remain stored.

When a callback function is registered, the handler checks the message store to see if any stored message satisfies the requirements of the new callback function. In this case the callback function will be activated to process the "old" messages first.

When more than one callback function can process an incoming message (because their message filter is matching the incoming message) a decision will be made:

1. Is there any blocking receive points? If yes - it will get the incoming message
2. Is there any multi-threaded callback functions? If yes - it will get the incoming message
3. Is there any single standalone or single grouped callback functions? If yes - it will get the incoming message

Observing what has been covered; the distributed application can easily be a multi- threaded application - even if the programmer does not want it to be. By using the callback functions the programmer can implement an event-driven application - where the events are incoming messages. But he has to be aware of the possibility that it might become multi-threaded.

4 Log

In order to help the development, PEDPI contains a built-in message logging feature. It can be easily activated and used. It produces ASCII text files (log files) about the incoming, outgoing and processed messages. The developer can register his own log objects, and implement any kinds of logging and debugging.

5 The speed of the .NET and C#

The next graph proofs, that the handling of network under the .NET environment on Windows XP is not slower than Linux/MPI at all. The following diagram shows how many times we need to send and receive 100.000 messages

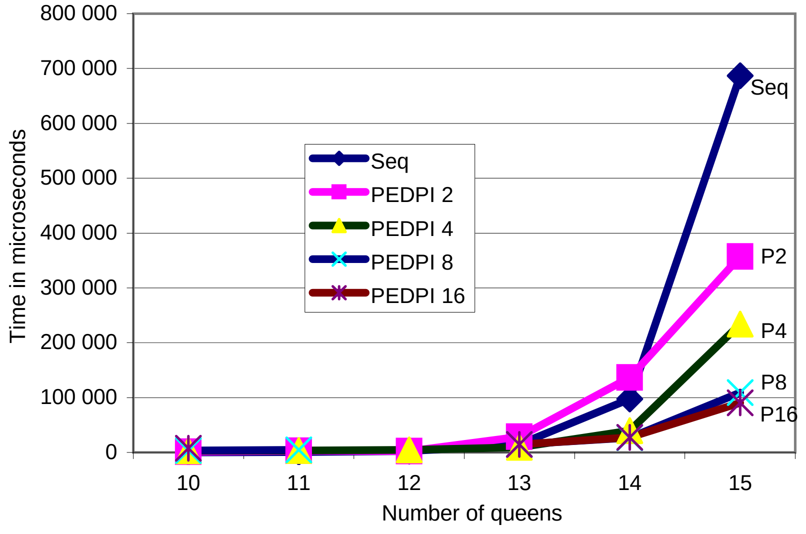


Fig. 3. 100.000 messages send and receive

between two nodes varying the length of the message body from 1,2,4,8,16 up to 16384 bytes. The axis y signs the time of sending and receiving in milliseconds. The axis x signs the size of one message in bytes.

- **NSock** means: low level network Socket was used in the communication (C#, .NET, WinXP)
- **NStream** means: Network Stream over a socket was used in the communication (C#, .NET, WinXP)
- **MPI** means: MPI was used in the communication (C, MPI, Linux)

Unfortunately, this test can't be run on PEDPI, because PEDPI builds the message channel up every time a message is sent, and after transferring a number of messages Windows generates an exception that too many network sockets are open - in spite of closing every socket immediately after it was used. It might be the fault of GC, but it will be corrected in the next version of PEDPI.

6 An example - N queens on the chessboard

The classical "N queen on the chessboard" problem solved in C# using PEDPI. The following diagram shows the speed-up varying the number of the queens (and the number of columns and rows of the chessboard) and the number of nodes assisting the calculation of all the solutions on the problem.

- **Seq** means: the sequential algorithm
- **PEDPI 2** means: parallel algorithm with PEDPI on 2 nodes
- **PEDPI 4** means: parallel algorithm with PEDPI on 4 nodes

– **PEDPI 8** means: parallel algorithm with PEDPI on 8 nodes
– **PEDPI 16** means: parallel algorithm with PEDPI on 16 nodes

During solving the problem the nodes share the backtrack interval among themselves. Increasing the number of nodes the intervals will be shorter, and the communication becomes more frequent, and it decreases the performance.

The axis y signs the time in microseconds, the axis x signs the number of queens on the chessboards.

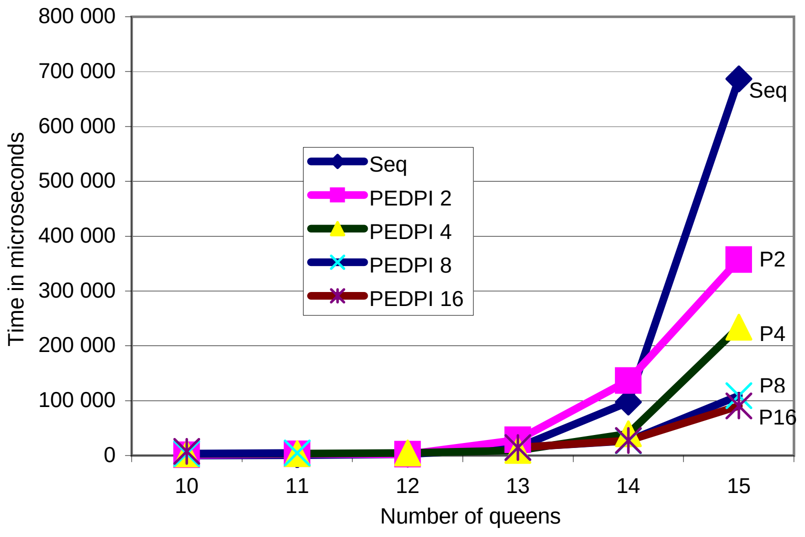


Fig. 4. N queens on the chessboard

In the future we would like to cooperate with the Hungarian JINI project to join our forces together to develop a much more reliable, robust and flexible message passing interface.

References

[1] Microsoft .NET MSDN

[2] JiniGRID project: `http://aszt.inf.elte.hu/~grid`

[3] Horváth Zoltán - Kozsik Tamás, *Safe Mobil Code - CPCC: Certified Proved-Propery-Carrying Code*, paper presented at 16th European Conference on Object-Oriented Programming (ECOOP 2002), University of Mlaga, Spain, June 10-14, 2002. `http://aszt.inf.elte.hu/~fun_ver`

[4] Peter S. Pacheco, *Parallel programming with MPI*, Morgan Kaufmann Publishers Inc., San Francisco, California, 1997

[5] Yukiya Aoyama, Jun Nakano, *RS/6000 SP: Practical MPI Programming*, International Technical Support Organization, `http://www.redbooks.ibm.com`

[6] William Gropp, Ewing Lusk, *User's Guide for mpich, a Portable Implementation of MPI, Version 1.2.1*, Argonne National Laboratory, University of Chichago, Mathematics and Computer Science Division,

[7] *MPI-2: Extensions to the Message-Passing Interface*, Message Passing Interface Forum, July 18, 1997, University of Tenesse, Knoxville, Tenessee

An Object-Oriented Design for an Atmospheric Flow Model

Frank Schimmel

Meteorologisches Insitut, Universität Hamburg, schimmel@dkrz.de

Abstract. The application of object-oriented (OO) design to atmospheric models is presented exemplarily for the newly developed model . The model employs a finite volume discretisation, which is inherently very modular. This modularity and flexibility is preserved in the design and implementation. Some well established OO design patterns are applied, partially in modified form to accommodate the special needs of a scientific computing application, e.g. to improve performance. The model is implemented in the C++ programming language, which provides the special means of templates. These are used in some places to improve performance.
The design of a3m is developed roughly following software development processes and models of established 'best practices' of OO design. Use-case-driven software development processes are found to be not directly applicable to constructing a low-interaction application such as an atmospheric model. Instead, an extension of the software development process to include the mathematical modelling and the discretisation of the resulting equations as development activities is proposed. This provides the means to decompose the complexity of the final application, an atmospheric flow model, into manageable units. A programming technique based on partial specialisation of C++ templates to further facilitate the decomposition of complexity is demonstrated.

1 Introduction

From a certain point of view, an atmospheric model is just a computer program, a piece of software. However, new approaches for software development are rarely used in the construction of atmospheric models. The newly developed atmospheric flow model a^{3}m applies some unusual approaches and methodologies for atmospheric modelling: it is designed following the object-oriented (OO) paradigm and that design is implemented in the C++ programming language. This distinguishes a^{3}m from the vast majority of atmospheric models which are built following the structured programming / procedural paradigm implemented in Fortran.

The OO paradigm provides more and more powerful means of abstraction, namely classes, inheritance and polymorphism. In general, OO designs permit better encapsulation and separation of concerns, thus providing a high degree of modularity and flexibility and greatly increase the potential for re-use of single system components. Additionally, the C++ programming language provides the very special feature of templates: classes and functions can be parameterised by types and/or compile time constants. This feature has received a lot of attention in the scientific computing community [1, 2, 3].

The recursive acronym a^{3}m stands for "**a**^{3}m is an **a**daptive **a**tmospheric **mo**del." The model is based on the flow equations for compressible, inviscid media, the Euler equations. It employs a set of relatively recent numerical methods to obtain approximate solutions of these equations, a high-resolution finite volume (FV) discretisation using essentially non-oscillatory (ENO) recoveries to gain higher order accuracy. A detailed description of a^{3}m is given in [4].

The design presented here is essentially a design for a FV framework. Special emphasis is put on retaining the modularity inherent to FV schemes and, most importantly for a^{3}m, on flexibility with respect to the type of computational grid employed.

Some well established OO design patterns are applied in a^{3}m's design. Sometimes they have to be adapted to suit the special needs of a scientific computing application, e.g. to improve performance. The behavioural patterns Strategy, Iterator and Template Method of the "Gang of Four" (GoF), Gamma, Helm, Johnson and Vlissides, described in their book [5] are of special interest. Some modifications have to be made to the design for the sake of performance: real polymorphic behaviour can be costly in terms of execution time and is thus eliminated by the use of C++ templates in some critical places. That way, a specific choice from a set of possible classes is determined at compile time, an approach sometimes referred to compile time polymorphism.

A modified and extended version of an OO software development process similar to the "recommended processes and models" described by Larman [6] is devised for the development of a^{3}m. It should be also applicable for developing similar scientific computing applications. It was motivated by the special nature of this class of applications: driving the development by use cases (i.e. ways the system is interacted with) is not an option for a low-interaction program. Instead, an extension of the development process to explicitly include the mathematical modelling and discretisation as development activities is proposed. From these other means of driving the development, decomposing the systems' complexity into manageable units, can be derived. With this respect, a programming technique based on partial specialisation of C++ templates to aid the decomposition of complexity is demonstrated.

Section 2 gives a short description of the mathematical discretisation methods and discusses the goals for the design of a^{3}m and approaches to achieve them. In Sect. 3 the applicability of a typical OO software development process to the creation of an atmospheric model (or a similar scientific computing application) and the need for adaption of the process to this kind of application are examined. Section 4 discusses some implications of the OO approach on performance, which is one of the major constraints for atmospheric models and scientific computing application in general. Section 5 summarises the conclusions from this work and discusses some limitations of the design presented here as well as possible future directions.

2 Discretisation and Design

The atmospheric flow model a^{3}m uses a finite volume (FV) spatial discretisation. A general description of FV schemes can be found in [7], [8] and [9]. Here only some some terms and concepts to be found again in the remainder of this paper are introduced. The basic idea of FV methods is to partition the computational domain, the region of space in which to simulate, into a set of small volumes or *cells*. The governing equations, the compressible flow equations in the case of a^{3}m, are averaged over the cells yielding one value for each cell. These *cell averages* evolve over time due to *sources and sinks* within each cell and due to *fluxes* exchanged between *neighbouring* cells through their common boundaries, the *faces*. Cells and faces are essential components of the *computational grid* used for the FV discretisation. The accuracy of the method is enhanced by *recovering* approximations of the equations' solution on each cell which are better than the piecewise constant approximation by cell averages.

FV methods are inherently very modular. Several distinct functional units and concepts can be identified in their construction: numerical flux functions, recoveries and quadratures. In principle, instances of these functional units can be combined at will and with various time integration schemes.[1] Also, they are applicable with any kind of computational grid. One central design goal is to preserve this modularity and flexibility in the implementation of the atmospheric model.

2.1 Time Integration

Time integration methods differ in the way an integration step is performed, i.e. how, given a particular spatial discretisation, the solution at time $t + \delta t$ is computed from the solution at time t. The algorithm for the Euler forward scheme is given in Fig. 1 on an abstract level. Other time integration schemes require different sequences of similar operations and iterations. A simple way of providing these variations while maximising code re-use is to employ a Template Method [5] for performing the time step: an abstract base class 'TimeIntegration' provides code common to all schemes, while concrete integration schemes, as presented in Fig. 1, only provide an implementation of the 'step()' method and inherit the rest. Figure 2 shows this relationship as an unified modelling language (UML) class diagram [10].

2.2 Modularity and Separation of Concerns

The choice of recovery procedure largely determines important properties of the FV method as a whole, most importantly the amount of artificial oscillations

[1] Not all possible combinations actually make sense or are numerically stable.

1. For each cell:
compute the recovery function.

2. For each pair of neighbouring cells:
compute the flux through their common face and record the induced tendencies of the cell averages.

3. For each face-cell pair on the domain boundary:
compute the flux to or from the exterior of the domain and record the induced tendencies.

4. For each cell:
compute sources and sinks and record the induced tendencies.

5. For each cell:
update the cell average using the accumulated tendencies.

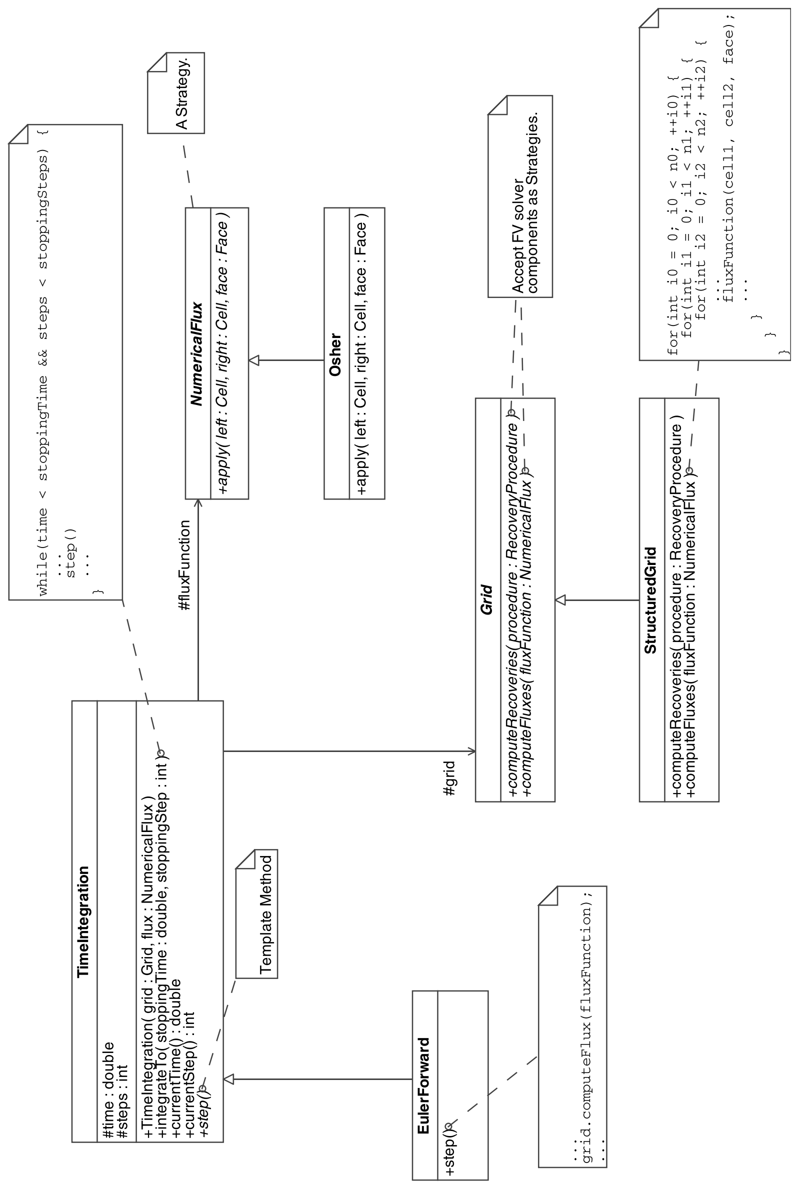
Fig. 1. Sequence of operations for one step of the Euler forward time integration.

and numerical diffusion introduced into the solution. Thus, being able to easily exchange this unit without modifying unrelated code is highly desirable. Similarly, other functional units should be changeable without impact on other parts of the code.

One of the key features of a^{3}m is a high degree of control over spatial resolution, i.e. where to have small and large cells. There is a large number of approaches to achieve this goal, e.g. by using a distorted structured grid [11, 12], a locally refined block-structured grid [13, 14, 15] or even an unstructured grid [16]. Introducing an abstraction of the computational grid enables their interchangeability, so that the type of grid, (block-)structured or unstructured, refineable or not, can be varied easily without changes to other components of the model, e.g. of the FV solver.

A general rule of thumb for program design is to assign the responsibility for a certain part of the systems' functionality to the component which has the necessary knowledge to perform the task. This is also referred to as the Expert pattern in [6]. Following this rule, it is the grid's responsibility to iterate over the combinations of it's cells and faces needed for the different operations of the FV scheme. But the concrete computations must not be coded directly into the grid classes themselves. Otherwise the interchangeability of either, the type of grid or the FV solver components, is violated and the classes are not focused in their concerns anymore: a grid class represents a particular kind of computational grid, not a complete FV scheme.

A solution to this problem is provided by the GoF Iterator pattern [5]. However, separate iterator objects were found to perform poorly [4], even with heavy use of C++ template to replace real runtime-resolved polymorphism (Sect. 4). Therefore, all of a^{3}m's grid classes provide iterator *methods* taking a FV solver component as an argument and passing the necessary cell/face combinations to them. This is also shown in the class diagram in Fig. 2, where the FV solver


TimeIntegration
#time : double
#steps : int
+TimeIntegration(grid : Grid, flux : NumericalFlux)
+integrateTo(stoppingTime : double, stoppingStep : int)
+currentTime() : double
+currentStep() : int
+step()
#fluxFunction
#grid
NumericalFlux
+apply(left : Cell, right : Cell, face : Face)
Osher
+apply(left : Cell, right : Cell, face : Face)
A Strategy.
Grid
+computeRecoveries(procedure : RecoveryProcedure)
+computeFluxes(fluxFunction : NumericalFlux)
Accept FV solver components as Strategies.
StructuredGrid
+computeRecoveries(procedure : RecoveryProcedure)
+computeFluxes(fluxFunction : NumericalFlux)
Template Method
EulerForward
+step()

```
while(time < stoppingTime && steps < stoppingSteps) {
  ...
  step()
  ...
}
```

```
for(int i0 = 0; i0 < n0; ++i0) {
  for(int i1 = 0; i1 < n1; ++i1) {
    for(int i2 = 0; i2 < n2; ++i2) {
      ...
      fluxFunction(cell1, cell2, face);
      ...
    }
  }
}
```

```
...
grid.computeFlux(fluxFunction);
...
```

Fig. 2. An UML class diagram of a^{3}m's basic design.
The boxes in this diagram represent classes, the different compartments display the class name, attributes and methods. Solid arrows with a hollow triangle head indicate *Generalisation*: they connect sub-classes (more specialised) to super-classes (more general), e.g. an instance of class 'EulerForward' *is* a 'TimeIntegration'. Other solid lines represent *Associations*, where arrow heads indicate navigability: instances of different classes referencing one another, e.g. a 'TimeIntegration *has* a 'FluxFunction'. For a detailed description of the notation see [10].
Note that the FV solver components other than numerical flux function have been left out of this diagram for the sake of simplicity.

components other than numerical flux functions have been left out of the diagram for the sake of simplicity. The FV solver components act as a variation of a GoF Strategy [5]. Thus, the grid classes provide specialised looping constructs depending on the type of iteration (for computing recoveries, fluxes or sources) and spatial dimension (see also Sect. 3.2), while the kernel of these loops, the concrete FV solver component passed to the iteration method, is injected into them.

3 A Software Development Process for an Atmospheric Model

Citing Larman [6], a software development process "is a method to organise the activities related to the creation, delivery and maintenance of software systems." The focus of this paper is the creation of an atmospheric model. Considering the creation of such a complex software system, one of the main issues addressed in a software development process is to provide methods for an *incremental* and *iterative* development, tackling one unit of manageable complexity at a time.

3.1 Driving the Development

The development process ("recommended processes and models", RPM) described by Larman, which may also be viewed as an instance of the Unified Process [17], knows several activities or disciplines: planning, analysis, design, construction and testing. Unlike the older 'waterfall approach', disciplines are not performed sequential in time: the development is organised in relatively short cycles over all of the disciplines. This allows e.g. for the requirements to change over time and to immediately account for those changes in the analysis, design and implementation in the same development cycle.

The Larman's RPM [6] is use-case-driven: the complexity of the complete software system to build is partitioned by cases of how that system is used, i.e. interacted with. The use-cases are implemented one by one, incrementally and refining the functionality and increasing the complexity towards meeting the defined requirements, each in a cycle of analysis, design, construction and testing.

For an atmospheric model like a^{3}m (and similar scientific computing applications) there is essentially only one use-case comprising the total complexity and functionality: a scientist conducts a simulation. Hence, the decomposition of complexity and driving the software development has to be achieved by different means to derive analysis, design, implementation and testing tasks of manageable complexity.

To this end, the development process is extended to explicitly include the mathematical modelling and discretisation as development activities. The modified development cycle is sketched in Fig. 3. Note that the activities shown are, again, not sequentially in time, only the iteration cycles of the process become bigger.

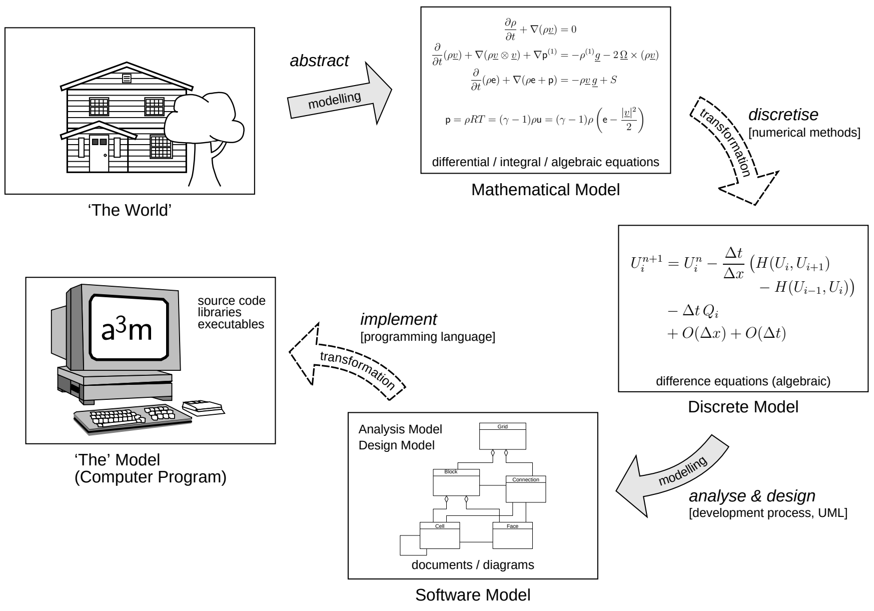

'The World'
abstract
modelling
$\frac{\partial\rho}{\partial t} + \nabla(\rho\underline{v}) = 0$
$\frac{\partial}{\partial t}(\rho\underline{v}) + \nabla(\rho\underline{v}\otimes\underline{v}) + \nabla\mathsf{p}^{(1)} = -\rho^{(1)}\underline{g} - 2\,\underline{\Omega}\times(\rho\underline{v})$
$\frac{\partial}{\partial t}(\rho\mathsf{e}) + \nabla(\rho\mathsf{e}+\mathsf{p}) = -\rho\underline{v}\,\underline{g} + S$
$\mathsf{p} = \rho RT = (\gamma-1)\rho\mathsf{u} = (\gamma-1)\rho\left(\mathsf{e} - \frac{|\underline{v}|^2}{2}\right)$
differential / integral / algebraic equations
Mathematical Model
transformation
discretise
[numerical methods]
$U_i^{n+1} = U_i^n - \frac{\Delta t}{\Delta x}\big(H(U_i,U_{i+1}) - H(U_{i-1},U_i)\big) - \Delta t\,Q_i + O(\Delta x) + O(\Delta t)$
difference equations (algebraic)
Discrete Model
modelling
analyse & design
[development process, UML]
Analysis Model
Design Model
Grid
Block
Connection
Cell
Face
documents / diagrams
Software Model
implement
[programming language]
transformation
a³m
source code
libraries
executables
'The' Model
(Computer Program)

Fig. 3. The activities of a³m's extended development process.

Now that the particular nature of the problem domain to implement, a mathematical discretisation method applied to a particular set of equations, is accounted for in the software process, new ways of driving that process are accessible. The possible approaches include:

1. start with fewer spatial dimensions,
2. start with a simpler mathematical discretisation method (leave out functional units of the FV solver, e.g. using only a first order scheme, or use trivial instances of functional units)
3. start with a different, simpler set of equations than the atmospheric flow equations and
4. start, due to the special goals for a³m (Sect. 2.2), with a simpler type of grid.

As an example, the implementation of a³m began with a first-order method (requiring no recoveries at all) solving the one-dimensional advection equation and later the Burgers equation on a simple structured grid. The later versions solve the fully compressible Euler equations in multiple space dimensions on block-structured grids.

This way of driving the development blends well with the requirements towards the design postulated in Sect. 2. It is the modularity achieved by the strict separation of concerns between the different FV solver components as well as the time integration and grids that enables their interchangeability from absent or trivial via simple to complex versions of these components needed for the approaches 2 and 4 above. Figure 4 shows the evolution of the conceptual model

[6] during a³m's development. The two diagrams show some of the concepts of the problem domain to implement, a FV discretisation method, and their interrelations. The upper diagram shows a very early stage of the development, where no recovery (approach 2) and only a simple structured grid (approach 4) were used. The lower diagram uses piecewise linear recoveries on a block-structured grid.

The full design of a³m also includes a parameterisation by the physical problem to simulate, ranging from linear advection to the full equations of atmospheric motion. This is only hinted at in form of the 'SolutionValue' concept in Fig. 4. The physical problem classes include different representations of the solution for the actual model state, the cell averages, and computing recoveries as well as those computations immediately involving the analysis of the equations (e.g. eigenvalues of the fluxes' Jacobian) which are used by some FV solver components. This parameterisation allows a variation of the equations solved, i.e. to pursue approach 3 above, but for simplicity this shall not be discussed in more detail at this point.

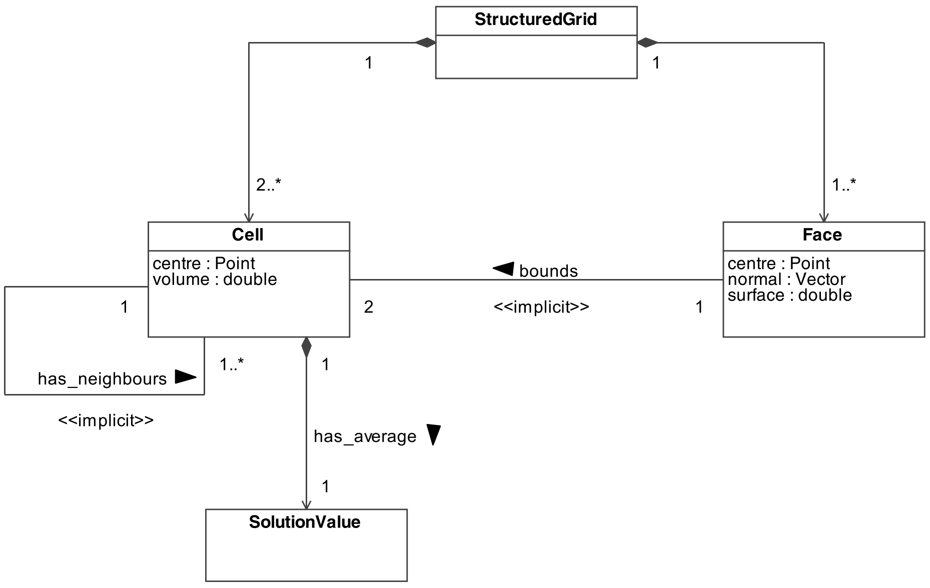
3.2 Decoupling Spatial-Dimension-Dependent Code Using Partial Specialisation of C++ Templates

Following approach 1 above, the development of a³m was begun with only one spatial dimension. It should be noted that only some parts are dependent on the spatial dimensions. All other parts can be re-used directly if decoupled properly from the dimension-dependent code. The time integration and sources do not depend on the spatial dimension. Since the design currently limits recovery functions to at most piecewise linear functions in space, the responsible methods of the FV solver components take two, three or four cells to recover in one, two and three dimensions, respectively. This can be easily achieved by overloading these methods. The flux computation is essentially the (approximate) solution of a one-dimensional Riemann problem at the face between two cells and is thus independent of the spatial dimension. In more than one dimension, a rotation to a local coordinate system is performed. This rotation is dependent on the number of dimensions. Another part depending on the spatial dimension are the grids' iteration methods which apply the FV solver components to the necessary combinations of cells and faces.

A possible method to follow approach 1 above would be to implement separate grid classes for each dimension and type: one for 1d structured, 1d block-structured, 2d structured, etc. But this approach results in very low code re-use among grids of the same type.

Taking into account that specifying the number of dimensions at compile time is not a significant restriction, a special feature of C++ for generic programming can be employed: templates. So the grids are parameterised by the number of spatial dimensions with a template parameter of type 'int'. Unlike polymorphism, which has to be resolved at the time of execution, this has no impact on performance. A grid class would look something like shown in Fig. 5.

(a) early development



(b) late development

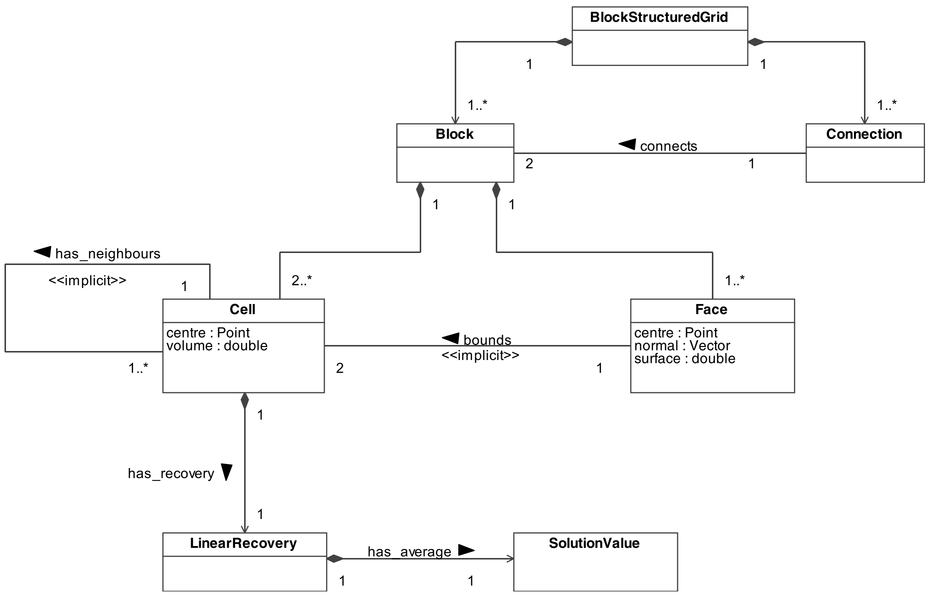


Fig. 4. Evolution of the conceptual model during the development.
The notation is the same as in Fig. 2. Additionally, filled diamonds on association ends indicate *Composition*, i.e. the class on the other end of the association is an integral part of the class on the one end. See [10] for more details.

```
template <class Problem, unsigned int dimension>
class Grid {
public:
  ...
  inline void computeFluxes(FluxFunction& fun) {
    if(dimension==1) {
      ...
    } else if(dimension==2) {
      ...
    } else if(dimension==3) {
      ...
    } else {
      throw
        "computeFluxes() only implemented for dimensions 1..3";
    }
  }
  ...
};
```

Fig. 5. A naive grid class implementation.

A decent optimiser should be able to eliminate the unused branches. Nevertheless they have to be correctly parsed and code has to be generated for them, just to be removed afterwards. Also, throwing an exception at run time when failure is already certain at compile time makes little sense.

A different approach is followed in a^{3}m using another interesting feature of C++: partial specialisation of templates. So specialised implementations of the iteration methods could be given for the number of spatial dimensions while all other template parameters are still unspecified. Usually, a compiler instantiates only those specialised functions actually used, so that no code is generated for the unused versions. If there is no specialised (and no generic) version, an error is reported latest while linking.

Unfortunately, partial specialisation is only legal for classes and not for functions, not even member functions (methods). Adhering to the proverbial "another layer of indirection will solve it", this can be circumvented by defining a specialised implementation helper class as shown in Fig. 6. The specialised implementation of **class** GridImpl in 2d is shown in Fig. 7.

This approach even enforces proper encapsulation: class GridImpl really is just an extension of class Grid and should only be accessed by that. This is achieved by declaring the computeFluxes() member function of the specialised class GridImpl<Problem, 2> private and making class Grid<Problem,2> a friend of its helper class.

Specialised versions of the rotation can be gained similarly.

```
// Just declaration: there is no generic iteration.
template <class Problem, unsigned int dimension>
class GridImpl;

template <class Problem, unsigned int dimension>
class Grid {
public:
  ...
  inline void computeFluxes(FluxFunction& fun) {
    // delegate iteration to specialised helper
    GridImpl<Problem, dimension>::computeFluxes(fun, *this);
  }
  ...
private:
  friend class GridImpl<Problem, dimension>;
  Array<Cell, dimension> cells;
  Array<Face, dimension> faces[dimension];
};
```

Fig. 6. Implementation of a grid class using a helper class.

4 Resolving Performance Issues of the Object-Oriented Design

Polymorphism is one of the major features of the OO approach and is used in many design patterns, including the Strategy and Template Method patterns used in a^{3}m's design [5]. But polymorphism, realised in the C++ language by virtual functions, may have a significant impact on performance: the actual type of an object, and thereby the exact method to call, has to be determined at run time. This is not only costly compared to a normal function call, it also prevents inlining of the function and all further optimisations inlining usually enables. Although other issues have received a higher priority in a^{3}m's development, end-to-end execution time is an important issue. Therefore, real polymorphic behaviour – involving virtual member functions – is eliminated by the use of C++ templates in some critical places, an approach is sometimes referred to as compile time polymorphism.

In general, a virtual function should be replaced if it is called often and if the function performs only few computations. Performing a virtual function call to the 'step()' method of the 'TimeIntegration' class has little impact on the total execution time, since the method performs, although mostly indirectly, a lot of complicated computations (a whole step of integration). Virtual function calls for methods performing I/O also have little impact: I/O is generally slow and streams of the C++ standard library use virtual functions anyway.

```
template <class Problem, 2>
class GridImpl {
private:
  friend class Grid<Problem, 2>;

  static void computeFluxes(FluxFunction& fun,
                            Grid<Problem, 2>& grid) {
    for(int i0 = 0; i0 < grid. cells .extent(0) − 1; ++i0) {
      for(int i1 = 0; i1 < grid. cells .extent(1) − 1; ++i1) {
        // call FluxFunction fun here...
      }
    }
  }
   ...
};
```

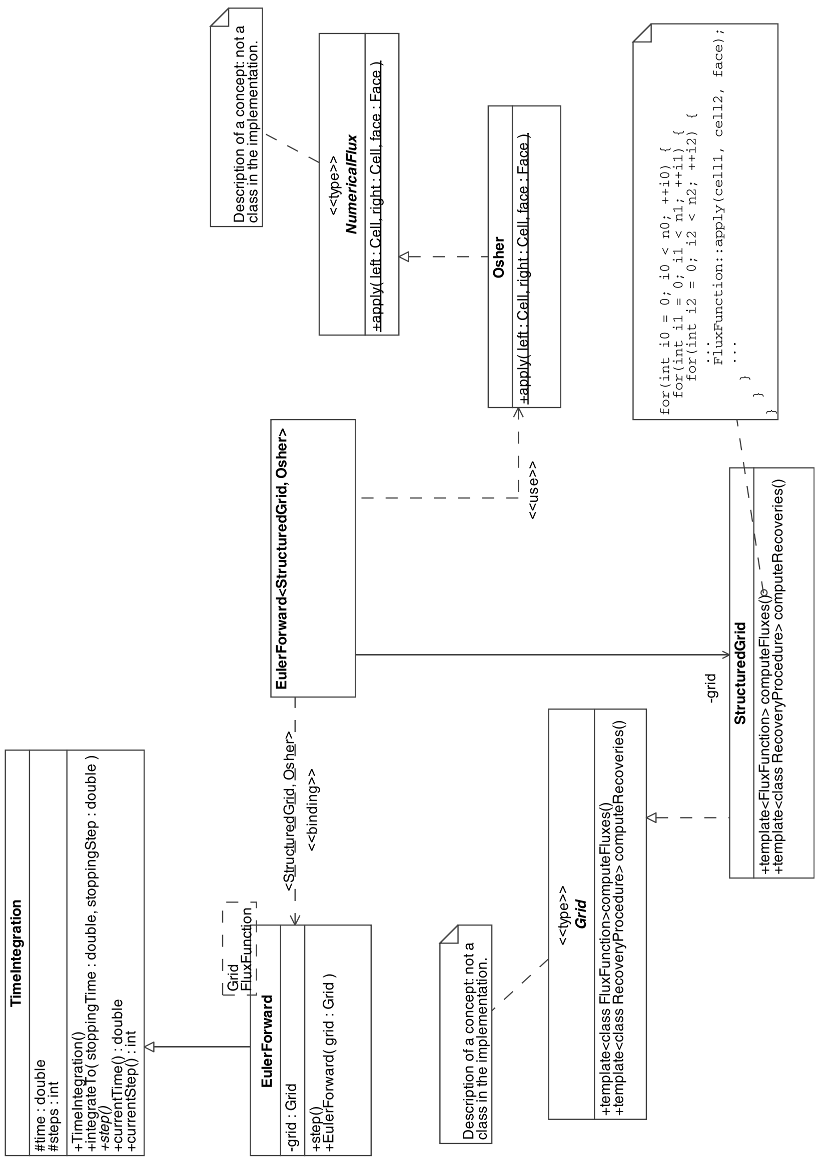
Fig. 7. Implementation of a specialised helper class used in Fig. 6.

The FV solver components are called for all cells or all certain combinations of cells and faces every time step. Hence, the polymorphism in these classes is dropped and they are passed as template parameters to the concrete time integration class and from there on to the grids' iteration methods. Applying this principle to GoF Strategies like the numerical flux classes also has some resemblance to policy-based class design [18]. In Alexandrescu's approach a host class inherits publicly from one or more policy classes and thereby also inherits the behaviour of the policies. The important difference to a^{3}m's design is that no inheritance is involved: the Strategies are template parameters to *inner* template functions of the grid classes. The modified design is shown in Fig. 8.

Summarising some important differences to the polymorphic design shown in Fig. 2, 'NumericalFlux' and 'Grid' are no classes in the implementation anymore. They just serve to illustrate what numerical flux functions and grids are, respectively. Also, the 'apply()' method of the numerical flux functions (class 'Osher') is now a static member function: numerical fluxes have no state anyway, they just provide behaviour (a piece of code to be injected in some loop). The abstract class 'TimeIntegration' has no 'fluxFunction' and 'grid' attributes anymore. The grid and numerical flux function are template parameters of the concrete time integration 'EulerForward'. Accordingly, 'EulerForward' is now constructed with a grid and the numerical flux function is accessed only by its type passed as the second template parameter.

5 Summary

The software design of the new atmospheric model a^{3}m was presented and discussed. Unlike the vast majority of atmospheric models, which adhere to a pro-


TimeIntegration
#time : double
#steps : int
+TimeIntegration()
+integrateTo(stoppingTime : double, stoppingStep : double)
+step()
+currentTime() : double
+currentStep() : int
EulerForward
Grid
FluxFunction
-grid : Grid
+step()
+EulerForward(grid : Grid)
<StructuredGrid, Osher>
<<binding>>
EulerForward<StructuredGrid, Osher>
<<use>>
-grid
Description of a concept: not a class in the implementation.
<<type>>
NumericalFlux
+apply(left : Cell, right : Cell, face : Face)
Osher
+apply(left : Cell, right : Cell, face : Face)
StructuredGrid
+template<FluxFunction> computeFluxes()
+template<class RecoveryProcedure> computeRecoveries()
<<type>>
Grid
+template<class FluxFunction>computeFluxes()
+template<class RecoveryProcedure> computeRecoveries()
Description of a concept: not a class in the implementation.

```
for(int i0 = 0; i0 < n0; ++i0) {
  for(int i1 = 0; i1 < n1; ++i1) {
    for(int i2 = 0; i2 < n2; ++i2) {
      ...
      FluxFunction::apply(cell1, cell2, face);
      ...
    }
  }
}
```

Fig. 8. Design with compile time polymorphism using C++ templates. The notation is the same as in Fig. 2. Additionally dashed arrows with a hollow triangle head indicate *Realisations* of a *Type*, which are present for conceptual purposes only and do not represent classes in the implementation, other sashed arrows indicate different kinds of *Dependencies* and dashed boxes on classes represent parameters to that class, i.e. they are template classes. Again, see [10] for more details.

cedural approach, this design follows the OO paradigm. It makes use of some well-established OO design patterns. The design was implemented in the C++ programming language. C++ templates were used to avoid runtime penalties from polymorphism while retaining encapsulation, proper separation of concerns and, thus, a very high degree of modularity.

A software development process based on that described by Larman [6] was employed. It was extended to explicitly include the mathematical modelling and discretisation as software development activities. Instead of driving the process by use cases, a strategy tailored for more interactive applications, these extensions provide the necessary means of decomposing the systems' complexity.

A programming technique employing partial specialisation of template classes, a special feature of the C++ language, to decouple code immediately depending on the number of spatial dimensions of the physical problem to solve (iteration loops for structured grids, rotations) from other program parts was demonstrated. This technique enables a further decomposition of complexity and, thus, aids driving the proposed development process.

Although, for the sake of a cleaner presentation, this paper covers only a simplified version of a^{3}m's design, some of its' limitations should not go unmentioned: currently quadratures are not explicitly modelled in the design and recoveries are limited to piecewise linear. Both limits the accuracy of the FV method to second order. A more general approach would require significant changes in the design and was postponed to potential future development cycles.

Other directions of development are well supported by the design in the current state. These include a transition from locally refined to fully adaptive block-structured grids, i.e. grids which change in their spatial distribution of resolution during a simulation and parallelisation. The former lies exactly within the stated requirement of an interchangeable computational grid. Due to the achieved separation of concerns in a^{3}m's design, this requires only the implementation of a new grid class (an extension of the existing block-structured grid class) while the rest of the code remains unchanged.

Parallelisation can be achieved similarly: the distribution of data to different computational nodes can be achieved by partitioning the grid into suitable blocks, each of which is processed in parallel. Adhering to the proverbial "another layer of indirection will solve it", a more advanced strategy would be to use distributed containers (arrays) for storage of the face and cell data within grid blocks and delegate iterations (for recovery, flux computation, etc.) to the these arrays. In either way, the rest the code remains unchanged, especially the FV solver components are not cluttered with the details of the parallelisation.

Acknowledgements

The author would like to thank K. Heinke Schlünzen, Jan Könlein and Axel Wienberg for their comments on early versions of this paper.

The development of a^{3}m was conducted as a project in the Graduiertenkolleg *Erhaltungsprinzipien in der Modellierung und Simulation mariner, atmosphäri-*

scher und technischer Systeme funded by the Deutsche Forschungsgemeinschaft (DFG).

References

[1] Veldhuizen, T.: Expression templates. C++ Report **7** (1995) 26–31 Reprinted in the book *C++ Gems*.

[2] Veldhuizen, T.L.: C++ templates as partial evaluation. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99) (1999)

[3] Haney, S., Crotinger, J., Karmesin, S., Smith, S.: PETE: The Portable Expression Template Engine. Dr. Dobb's Journal of Software Tools **24** (1999) 88, 90–92, 94–95

[4] Schimmel, F.: Development and Test of an Atmospheric Flow Model Employing Adaptive Numerical Methods. Dissertation, Fachbereich Geowissenschaften, Universität Hamburg (2002)

[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Professional, Boston, Massachusetts (1995)

[6] Larman, C.: Applying UML and Patterns: an Introduction to Object-Oriented Analysis and Design. Prentice-Hall, Upper Saddle River, New Jersey (1998)

[7] Hirsch, C.: Numerical Computation of Internal and External Flows, Volume 1, Fundamentals of Numerical Methods. Wiley, Chichester (1988)

[8] Abgrall, R., Lantéri, S., Sonar, T.: ENO approximations for compressible fluid dynamics. Zeitschrift für Angewandte Mathematik und Mechanik **79** (1999) 3–28

[9] LeVeque, R.J.: Finite Volume Methods for Hyperbolic Problems. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, UK (2002)

[10] : OMG Unified Modeling Language Specification. (2001) Version 1.4.

[11] Schlünzen, K.H.: Numerical studies on the inland penetration of sea breeze fronts at a coastline with tidally flooded mudflats. Contributions to Atmospheric Physics **63** (1990) 243–256

[12] Sündermann, A.: Die Anwendung von adaptiven Gittern in zwei einfachen Modellen für eine atmosphärische Rollenzirkulation. Berichte aus dem Zentrum für Meeres- und Klimaforschung 5, Zentrum für Meeres- und Klimaforschung, Universität Hamburg (1990) Dissertation.

[13] Lilek, Z., Muzaferija, S., Perić, M., Seidl, V.: An implicit finite-volume method using nonmatching blocks of structured grid. Numerical Heat Transfer, Part B **32** (1997) 385–402

[14] Skamarock, W.C., Klemp, J.B.: Adaptive grid refinement for two-dimensional and tree-dimensional nonhydrostatic atmospheric flow. Monthly Weather Review **121** (1993) 788–804

[15] Berger, M., Oliger, J.: Adaptive mesh refinement for hyperbolic partial differential equations. Journal of Computational Physics **53** (1984) 484–512

[16] Meister, A., Sonar, T.: Finite volume schemes for compressible fluid flow. Hamburger Beiträge zur Angewandten Mathematik, Reihe F 4, Institut für Angewandte Mathematik, Universität Hamburg (1998)

[17] Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Developement Process. Addison Wesley Professional (1999)

[18] Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. C++ In-Depth Series. Addison Wesley Professional, Boston, Massachusetts (2001)

Fast Expression Templates for the Hitachi SR8000 Supercomputer

Alexander Linke and Christoph Pflaum

Universität Würzburg, Institut für Angewandte Mathematik und Statistik,
Am Hubland, D-97074 Würzburg, Germany

Abstract. Expression templates(ET) can significantly reduce the implementation effort for mathematical software. On the Hitachi SR8000 supercomputer it can be observed, however, that classical ET implementations do not lead to an optimal performance in the case of certain expressions such as $c = aabbab$. This is because they do not assist the compiler in recognizing that variables are used several times within an expression. Therefore, we introduce the concept of enumerated variables, which are provided with an additional integer template parameter. Since different variables have different types, a modified implementation of ET leads to an optimal C++ code on the Hitachi SR8000. These so-called *Fast Expression Templates* perform better than classical ET, even when variables are not used several times. Performance results are presented on the Hitachi SR8000 supercomputer with automatic vectorization and parallelization.

1 Introduction

While object-oriented programming is being embraced in the industry, its acceptance by the High Performing Computing community is still very hesitant, mainly because of supposed performance losses. For example, supercomputer manufacturer Hitachi tells us that there is not a single user of C++ on Hitachi supercomputers in Japan. Indeed, introducing abstract data types and operator overloading empowers the software engineer to forge entire user-defined languages based on C++, which can be understood by C++ compilers. However, there are not any language features in C++ which are designed to inform the compiler about allowed transformations of mathematical expressions involving user-defined abstract data types. Therefore, though such user-defined mathematical expressions can be compiled by C++-compilers, they perform very poorly. The first solution to overcome these performance problems were expression templates in C++, whose development we will summarize in the following.

By adding templates to C++, the language gained much more than originally intended. Todd Veldhuizen recognized the potential of this facility and in 1995 and 1996 published the first articles about template meta programming [11] and expression templates [12]. Expression templates were proposed to overcome performance problems which arise from simple operator overloading in mathematical expressions. Unnecessary temporaries are avoided by performing some kind of expression dependent inlining within a single loop. For many systems the performance of expression templates competes with Fortran [13].

Soon, there was a rapid development of powerful mathematical packages based on expression templates, e.g. Blitz++ by Todd Veldhuizen [14] and the Generative Matrix Computation Library(GMCL) described in [4]. Further important steps were PETE [7] and POOMA [10], which are two projects started at the Los Alamos National Laboratories. PETE is a tool for implementing expression templates for various applications. POOMA supports the implementation of mathematical algorithms for solving partial differential equations. Users of POOMA write sequential source code similar to FORTRAN 90 and get automatic parallelization on various platforms just by using compiler switches.

Some aims of the POOMA project are shared by the EXPDE project [8]. EXPDE eases the implementation of parallel 3D finite elements codes by providing finite element and multigrid operators on arbitrarily shaped domains. Mathematical algorithms can be formulated with EXPDE in a language very close to the mathematical language. Expression templates enable automatic parallelization.

Performance problems with expression templates were discovered for the first time by Federico Bassetti, Kei Davis and Dan Quinlan in 1997, see [2] and [3].

In this article we want to present another problem with expression templates. The solution of this new problem solves prior problems, too. Let us look at a bit of C++ code with $a * b$ understood as component-wise multiplication:

```
Vector a, b, c;
c = a*a*b*b*a*b;
```

On a single node of the Hitachi supercomputer SR8000 a classical expression templates implementation of this code - as suggested in [15] - achieves a performance significantly inferior to handcrafted C code, although the expression templates code is fully vectorized and parallelized, see figure 4.1.

Generally, expression templates derive a so-called parse tree or expression object representing an expression and perform some expression dependent inlining. Classical implementations split the information about an expression into two parts. The data type of the expression object represents the structure of the expression and member variables represent the variables involved within the expression. The data type of the expression object is known at compile-time. However, use of information represented by member variables is compiler-dependent. For example, information accessible at compile-time about the expression `a*a*b*b*a*b` is equivalent to:

```
Vector*Vector*Vector*Vector*Vector*Vector
```

Therefore, classical expression templates do not assist the compiler in recognizing that the Vectors a and b are used several times within the expression and performance losses can arise.

This problem can be overcome by introducing the concept of enumerated variables. Enumerated variables are provided with an additional integer template parameter. Since different variables have different types, the compiler can perform more intelligent expression dependent inlining. Moreover, by using enumerated variables, the intermediate C++ code can be forced to be identical to handcrafted C code and performance measurements on the Hitachi SR8000 show optimal C++ performance. Even all the performance issues recognized by Bassetti, Davis and Quinlan do not occur with our new implementation of expression templates.

1.1 Structure of the Paper

In the next section we will explain in detail how expression templates are classically implemented, how they work internally and which problems they suffer from. Thereafter, we present in detail a suggestion for a modified implementation. In the last two sections we show performance results and draw conclusions about the use of expression templates in High Performance Computing.

2 Classical Expression Templates

Let us discuss a minimal classical expression templates implementation for vectors with component-wise multiplication, as suggested in [15]. Our example was implemented on the Hitachi SR8000 supercomputer. The program runs on a single node of the Hitachi, which is equipped with 8 processors. Each processor has so-called pseudo-vectorization facilities. For our presentation it is enough to imagine that each processor is a vector-processor. Parallelization and vectorization is enabled by the C99 keyword `restrict` and the compiler directive `/*voption indep*/`, see [5] and [6]. The mathematical operator `times` is encapsulated by:

```
struct times {
public:
  static inline double apply(double a, double b) {
    return a*b;
  }
};
```

The expression class is implemented as:

```
template<typename Left, typename Op, typename Right>
struct Expr {
  const Left &leftNode_;
  const Right &rightNode_;

  Expr(const Left &t1, const Right &t2)
    : leftNode_(t1), rightNode_(t2) {}

  inline double Give(int i) const {
    return Op::apply(leftNode_.Give(i), rightNode_.Give(i));
  }
};
```

Here is a simple vector class:

```
struct Vector {
  Vector(double *restrict &data, int N) : data_(data), N_(N) {}

  template<typename Left, typename Op, typename Right>
  inline void operator=(const Expr<Left, Op, Right> &expression) {
    int N = N_;

    /*voption indep*/
    for(int i=0; i < N; ++i)
      data_[i] = expression.Give(i);
  }

  inline double Give(int i) const {
    return data_[i];
  }

  double * &data_;

  int N_;
};
```

and here the `operator*`:

```
template<typename Left>
Expr<Left, times, Vector> operator*(const Left &a,
  const Vector &b) {
  return Expr<Left, times, Vector >(a, b);
}
```

Now wee see it in action:

```
  ...
  Vector
    a(a_data, N), b(b_data, N), r(r_data, N);
  r = a*b*a;
  ...
```

We will now explain how expression templates work. Let us look at the line `r = a*b*a;`. The simulated run of the compiler looks like:

```
  r = a*b*a;
    = Expr<Vector,times,Vector>(a,b)*a;
    = Expr<Expr<Vector,times,Vector>,times,Vector>(
        Expr<Vector,times,Vector>(a,b),a);
    =: expression;
```

It then matches `r.operator=`:

```
r.operator=<Expr<Expr<Vector,times,Vector>,times,Vector> >
  (expression) {
  int N = N_;

  /*voption indep*/
  for(int i=0; i < N; ++i)
    data_[i] = expression.Give(i);
}
```

Now `expression.Give(i)` is expanded by inlining `Give(i)` from each node of the expression object:

```
data_[i] = expression.Give(i);
         = times::apply(expression.leftNode_.Give(i),
                        expression.rightNode_.Give(i));
         = times:apply(times::apply(
             expression.leftNode_.leftNode_.Give(i),
               expression.leftNode_.rightNode_.Give(i)),
             expression.rightNode_.Give(i));
         = times::apply(times::apply(
             expression.leftNode_.leftNode_.data_[i],
               expression.leftNode_.rightNode_.data_[i]),
             expression.rightNode_.data_[i]);
         = expression.leftNode_.leftNode_.data_[i] *
             expression.leftNode_.rightNode_.data_[i] *
           expression.rightNode_.data_[i];
```

Although component-wise multiplication of three vectors is a very trivial application of expression templates, the intermediate C++ code inlined by expression templates is quite complex. More difficult applications like expression templates for differential operators in 3D yield even more complex intermediate code. If `operator=` is inlined, a C++ compiler can, in principle, optimize the above expression in the sense that it evolves to:

```
for(int i=0; i < N; ++i)
  r.data_[i] = a.data_[i] * b.data_[i] * a.data_[i];
```

Indeed, this requires a compiler to have good optimization facilities. Since the Hitachi C++ compiler does not cope with C++-specific optimization facilities, on the Hitachi SR8000 classical expression templates suffer from several performance problems.

3 Fast Expression Templates

After this short discussion of classical expression templates, we present an alternative approach. The main idea is to derive an expression object from an expression whose information is completely accessible at compile-time. This allows more intelligent inlining and guarantees optimal C++ performance. We achieve this by introducing enumerated variables. Enumerated variables have an additional integer template parameter, which has to be chosen uniquely for each variable. Then different variables have different types. As before, we present a minimal implementation of this new approach. This causes the examples to be somewhat cumbersome, but we feel that it makes understanding the approach easier.

The presented implementation restricts us to use only three different enumerated variables, but is easily extended to an arbitrarily high number of enumerated variables. An alternative, more elegant, approach would use typelists, as described in [1]. In this case there are no restrictions on the number of enumerated variables. Global variables are easily avoided by additionally introducing expression wrapper classes, which wrap a single enumerated variable within an expression.

First we introduce some macros, which simplify the handling of enumerated variables:

```
#define params_in double *restrict data0_, \
                  double *restrict data1_, \
                  double *restrict data2_
#define params_out data0_, data1_, data2_
#define DeclareEnumVariables double *restrict data0_, \
                                    *restrict data1_, \
                                    *restrict data2_

double *global0_, *global1_, *global2_;
```

The classes, which derive the expression object from a mathematical expression, have the same structure as in the classical case. However, they do not have any member variables and the methods can be defined as `static`. The complete information about an expression is now represented by its data type.

```
struct times {
public:
  static inline double apply(double a, double b) {
    return a*b;
  }
};

template<typename Left, typename Op, typename Right>
struct Expr {
  static inline double Give(params_in, int i) {
    return Op::apply(Left::Give(params_out, i),
     Right::Give(params_out, i));
  }
};
```

The main implementation idea consists of introducing the enumerated variable class `vector`. The expression dependent inlining is again performed by methods defined as `static`. Furthermore, it is remarkable that the new approach allows to abandon the compiler directive `/*voption indep*/`. We achieve parallelization and vectorization just by using compiler switches and the C99 keyword `restrict`:

```
template<int varnum>
struct Vector {
  template<typename Left, typename Op, typename Right>
  void operator=(const Expr<Left, Op, Right> &expression) const {
    DeclareEnumVariables;
    // double *restrict data0_, *restrict data1_, ...;

    int N = global_N;
    // Preparations, can be implemented in a more elegant manner
    data0_ = global0_; data1_ = global1_; data2_ = global2_;

    for(int i=0; i < N; ++i)
      GiveData(params_out)[i] = expression.Give(params_out, i);
  }

  static inline double Give(params_in, int i) {
    return 0.0;
  }

  static inline double*restrict GiveData(params_in) {
    return NULL;
  }
};

template<> inline double Vector<0>::Give(params_in, int i) {
  return data0_[i]; }
template<> inline double Vector<1>::Give(params_in, int i) {
  return data1_[i]; }
template<> inline double Vector<2>::Give(params_in, int i) {
  return data2_[i]; }

template<>
inline double*restrict Vector<0>::GiveData(params_in) {
  return data0_; }
template<>
inline double*restrict Vector<1>::GiveData(params_in) {
  return data1_; }
template<>
inline double*restrict Vector<2>::GiveData(params_in) {
  return data2_; }
```

The following piece of code is analogous to the classical code:

```
template<typename Left, typename Right>
inline Expr<Left, times, Right> operator*(const Left &a,
  const Right &b) {
  return Expr<Left, times, Right>();
}

template<typename Left, int varnum>
Expr<Left, times, Vector<varnum> > operator*(const Left &a,
  const Vector<varnum> &b) {
  return Expr<Left, times, Vector<varnum> >(a, b);
}
```

Last but not least, an excerpt from the main program:

```
  ...
  Vector<0> a; Vector<1> b; Vector<2> r;
  r = a*b*a;
  ...
```

We will now explain how this new approach differs from classical expression templates. To this end, let us look at the expression `c=a*b*a`. The expression object for this expression has the following type:

```
Expr<Expr<Vector<0>,times,Vector<1> >,times,Vector<0> >
```

Since this expression type contains the complete information about the expression `a*b*a`, a more intelligent inlining is possible. The expression is evaluated as:

```
  r.operator=Expr<Expr<Vector<0>,times,Vector<1> >,times,
               Vector<0> >(expression) {
    DeclareEnumVariables;
    int N = global_N;
    // Preparations, can be implemented in a more elegant manner
    data0_ = global0_; data1_ = global1_; data2_ = global2_;

    for(int i=0; i < N; ++i)
      GiveData(params_out)[i] = expression.Give(params_out, i);
  }
```

Now we investigate how `expression.Give(params_out, i)` is inlined:

```
data2_[i] = times:apply(
         times::apply(
           expression::Left::Left::Give(params_out, i),
        expression::Left::Right::Give(params_out, i)),
      expression::Right::Give(params_out, i));
  = times:apply(
      times::apply(data0_[i], data1_[i]), data0_[i]);
  = data0_[i]*data1_[i]*data0_[i];
```

This inlined source code is just the code a programmer would naively write. Since `expression::Left::Left` and `expression::Right` are of type `Vector<0>`, and since `expression::Left::Right` is of type `Vector<1>`, enumerated variables allow the most precise static inlining. This static inlining is very important, because it assures that the expression object is created at compile-time, not at run-time. This has an important impact on performance.

The new approach for expression templates presented here is much more powerful than classical implementations. This is not only true for our trivial minimal implementation with component-wise multiplication, but also for complex expression templates applications on 3D-grids with 3D-stencil-evaluations.

4 Performance Results

In this section we will present performance results for three different vector expressions on a single node of the Hitachi SR8000 supercomputer in Munich. A single node is equipped with eight RISC processors and each processor can perform vector operations with floating point numbers. The peak performance of a Hitachi node is 12 GFLOPS per second, see [5].

On the Hitachi SR8000 we use the Optimizing C++ compiler sCC by Hitachi, because it is the only C++ compiler which can vectorize C++ programs on this platform. Unfortunately, this compiler is available only as a beta version 5. It has some bugs and is not fully ANSI C++ compliant. However, it optimizes numerical computations, actually written in C code, fairly well. Examples were compiled with the compiler options `-restrict -Os`. The keyword `restrict` is part of ANSI C99, see [5].

For each example we compare three different implementations. The first consists of handcrafted C code compiled by a C++ compiler, called *No Expression Templates*(NET). The second is a *Classical Expression Templates*(CET) implementation and the third is our new approach, called *Fast Expression Templates*(FET). For performance measurement of our FET implementation we used a minimal implementation with five variables and component-wise vector addition and multiplication. Performance measurements are presented for the three following examples:

$$c = a * a * b * b * a * b \tag{1}$$

$$\begin{aligned} a = a + a * a + a * a * a + a * a * a * a + a * a * a * a * a +\\ a * a * a * a * a * a + a * a * a * a * a * a * a \end{aligned} \tag{2}$$

$$a = b * c + d * e \tag{3}$$

Each expression was evaluated 1000 times and the elapsed time was measured. In the case of NET and FET code the sCC compiler can optimize away superfluous floating point operations, but not in case of CET.

4.1 First Example

The first example $c = a*a*b*b*a*b$ makes clear that FET is better in advising the sCC compiler that variables are used several times within an expression. NET and FET code achieve the same performance. The logfiles created by sCC concerning loop unrolling, parallelization and vectorization, show that the intermediate C++ code inlined by FET is compiled very similar to NET code. Performance of CET is measurably lower than of NET or FET.

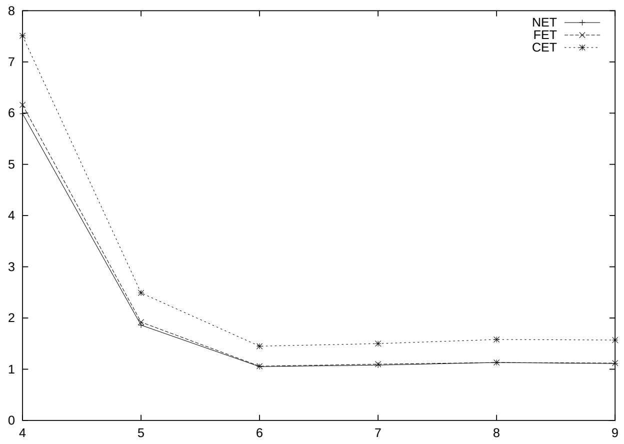


Fig. 1. Elapsed time per vector component for three implementations of the expression $c = a*a*b*b*a*b$ on a single node of the Hitachi SR8000. With N the vector length, we plot elapsed time per vector component against $\log_5 N$. The abbreviations NET, CET and FET stand for *No Expression Templates*, *Classical Expression Templates* and *Fast Expression Templates*

The following two tables show the performance improvement achieved by FET in comparision to CET for (1) and the absolute performance of FET measured in units of MFLOPS per second.

$\log_5(\textit{Vector length})$	4	5	6	7	8	9
Performance ratio FET/CET	1.22	1.30	1.37	1.36	1.40	1.40

$\log_5(\textit{Vector length})$	4	5	6	7	8	9
Performance of FET	812	2600	4720	4550	4430	4460

4.2 Second Example

The second example

$$a = a + a*a + a*a*a + a*a*a*a + a*a*a*a*a + a*a*a*a*a*a + a*a*a*a*a*a*a$$

again makes clear that FET is better in advising the sCC compiler that variables are used several times within an expression. However, in this example the sCC compiler can exploit this much more. Hardware counters show that, with N the vector length, NET and FET perform $12N$ floating point operations to evaluate the expression once. According to the Horner scheme this is the minimum number of operations needed. Unlike NET and FET, the CET implemention performs $27N$ floating point operations, since in this case the sCC does not recognize that the variable a is used several times in this expression.

Again, NET and FET achieve the same performance and their logfiles are identical.

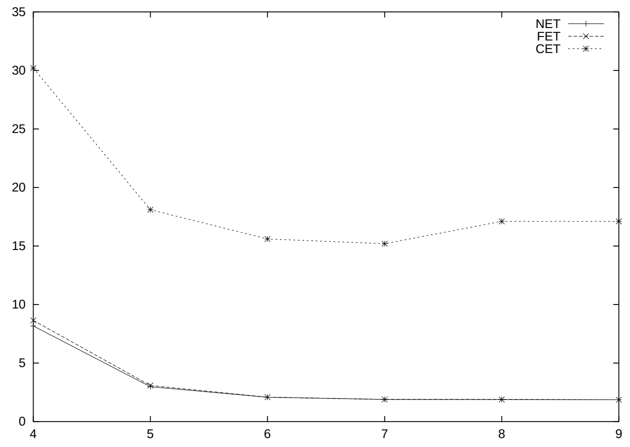


Fig. 2. Elapsed time per vector component for three implementations of the expression $a = a + a*a + a*a*a + \cdots + a*a*a*a*a*a*a$ on a single node of the Hitachi SR8000. For explanations, see figure 4.1

.

The following two tables show the performance improvement achieved by FET in comparision to CET and the absolute performance of FET measured in units of MFLOPS per second.

$\log_5(Vector\ length)$	4	5	6	7	8	9
Performance ratio FET/CET	3.50	5.86	7.46	8.04	9.05	9.14

$\log_5(Vector\ length)$	4	5	6	7	8	9
Performance of FET	1390	3880	5730	6340	6350	6420

This example shows that for the sCC compiler there are expressions, for which FET will perform arbitrarily better than CET.

4.3 Third Example

The third example shows that FET perform better then CET, even when variables are not used several times within an expression. Therefore, FET seem to solve the problems recognized by Bassetti, Davis and Quinlan in [3], too.

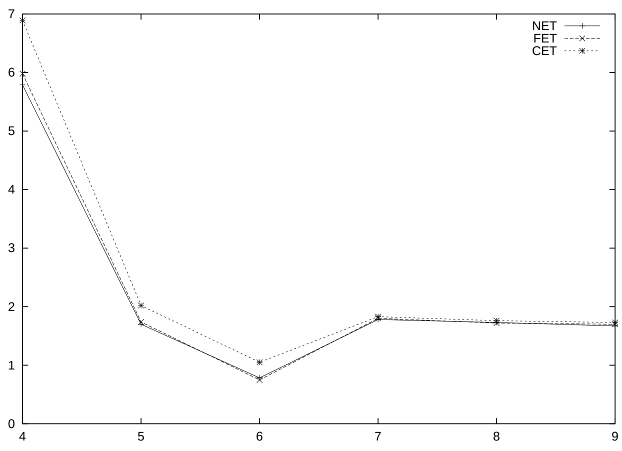


Fig. 3. Elapsed time per vector component for three implementations of the expression $a = b*c+d*e$ on a single node of the Hitachi SR8000. For explanations, see figure 4.1

The following two tables show the performance improvement achieved by FET in comparision to CET and the absolute performance of FET measured in units of MFLOPS per second.

$\log_5(Vector\ length)$	4	5	6	7	8	9
Performance ratio FET/CET	1.15	1.16	1.40	1.02	1.02	1.02

$\log_5(Vector\ length)$	4	5	6	7	8	9
Performance of FET	501	1720	4010	1660	1740	1770

5 Conclusions and Perspectives

In this article we wanted to show that classical implementations of expression templates exploit the potential of the approach only partially, its real potential being far from recognized. Expression templates should be investigated further in many directions. Optimal C++ performance is possible if expression templates are implemented in an optimal way. Since the complete information represented by an expression object is accessible at compile-time, optimal C++ performance should be achievable also for more complex expression templates applications like 3D finite elements expressions. Therefore, the prospects of expression templates seem to be very interesting. Two possible new applications shall be outlined in short.

First, let us look at our 3D finite element codes on the Hitachi SR8000. At the upper level of abstraction it is possible to combine expression templates with template meta programming. We can then define complex transformation rules for expressions by template meta programming in order to achieve high-level optimizations. An expression like `DxDx_FE(u) + DyDy_FE(u) + DzDz_FE(u)` could be transformed by template meta programming into `Laplace_FE(u)`.

At the machine-oriented level it should, with expression templates, be possible to calculate the number of vectorization streams necessary for any particular mathematical expression. If the number of necessary streams exceeds the number of available streams, inlining can be controlled such that the entire expression is divided into several subexpressions. Then, although temporaries are generated, such expressions perform better on the Hitachi SR8000 due to full vectorization.

Such automatic optimization facilities introduce modularity in the implementation of mathematical software. The formulation of algorithms can be close to the usual mathematical language. Optimization, parallelization and vectorization of mathematical expressions can be masked completely by expression templates implementation. This seems to us a great gain.

References

[1] Alexandrescu, A: Modern C++ Design : Generic Programming and Design Patterns applied. Addison-Wesley, Boston, 2001.

[2] Bassetti, F, Davis, K, and Quinlan, D: Toward Fortran 77 Performance from Object-Oriented C++ Scientific Framework: HPC '98 April 5-9, 1998.

[3] Bassetti, F, Davis, K, and Quinlan, D: C++ Expression Templates Performance Issues in Scientific Computing. CRPC-TR97705-S, October 1997.

[4] Czarnecki, K, and Eisenecker, U: Generative Programming : Methods, Tools, and Applications. Addison-Wesley, Boston, 2000.

[5] Leibniz-Rechenzentrum München: The Hitachi SR8000-F1, System Description. http://www.lrz-muenchen.de/services/compute/hlrb/system-en

[6] Numerical C Extensions Group: Restricted Pointers in C. Final Report, Draft 2, X3J11/94-019, WG14/N334, http://www.lysator.liu.se/c/restrict.html

[7] Los Alamos National Laboratories: PETE - Portable Expression Templates Engine. http://www.acl.lanl.gov/pete/html/introduction.html

[8] Pflaum, C: Expression Templates for Partial Differential Equations. Comput Visual Sci **4**, 1–8, (2001).

[9] Pflaum, C, and Falgout, R: Automatic Parallelization with Expression Templates. Lawrence Livermore National Laboratory technical report UCRL-JC-146179, November 2001.

[10] Los Alamos National Laboratories: POOMA: www.acl.lanl.gov/pooma

[11] Veldhuizen, T: Using C++ Template Metaprograms. *C++ Report* Vol. 7 No 4. (May 1995), pp. 36–43.

[12] Veldhuizen, T: Expression Templates. C++ Report **7** (5), 26–31 (1995).
[13] Veldhuizen, T: Will C++ be faster than Fortran? Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97).
[14] Veldhuizen, T: Blitz++. http://oonumerics.org/blitz/index.html
[15] Veldhuizen, T: Techniques for Scientific C++. Indiana University Computer Science Technical Report No 542, Version 0.4, August 2000.

On orthogonal specialization in C++

Dealing with efficiency and algebraic abstraction in Vaucanson

Yann Régis-Gianas, yann.regis-gianas@lrde.epita.fr,
Raphaël Poss, raphael.poss@lrde.epita.fr

LRDE: EPITA Research and Development Laboratory
14-16, rue Voltaire - F-94276 Le Kremlin-Bicêtre Cedex - France
http://www.lrde.epita.fr/

Abstract. The Vaucanson library works on weighted finite state machines in an algebraic framework. As computing tools, FSMs must provide efficient services. Yet, abstraction is needed to obtain genericity but also to define properly what objects we are working on.
Even if parameterized classes are a known solution to this problem, the different kinds of algorithm specializations are limited when using usual template techniques.
This paper describes a new design pattern called ELEMENT which enables the orthogonal specialization of generic algorithms w.r.t. the algebraic concept and w.r.t. the implementation. The idea is to make concept and implementation explicitly usable as object instances.
First, we show how it solves the specialization problem. Then, we detail its implementation and how we deal with some technical pitfalls.

Vaucanson is a C++ generic library for weighted finite state machine manipulation. For the sake of generality, in Vaucanson FSMs are defined using algebraic structures such as alphabet (for the letters), free monoid (for the words), semiring (for the weights) and series (mapping from words to weights) [5]. As usual, the challenge is to maintain efficiency while providing a high-level layer for the writing of generic algorithms. One of the particularities of FSM manipulation is the need for a fine grained specialization power on an object which is both an algebraic concept and an intensive computing machine.

Vaucanson is the core of a project initiated in 2001 by Jacques Sakarovitch of the École Nationale Supérieure des Télécommunications (ENST, Paris). The project is now a collaborative work between the ENST and the École Pour l'Informatique et les Techniques Avancées (EPITA, Paris).

1 Algorithms for weighted finite state machines

1.1 Two points of view

On the one hand, the mathematical aspect of automata requires the definition of a precise context. Indeed, an algorithm must specify on what kind of semiring or alphabet it works. A hierarchy of algebraic concepts is necessary to make their context explicit. Such a hierarchy can be found in any book about algebraic structures.

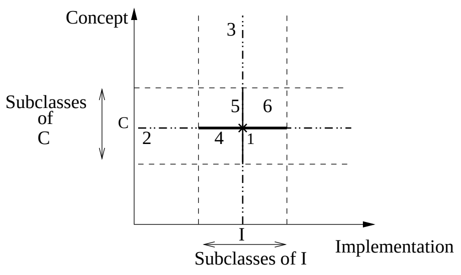
On the other hand, weighted finite state machines are used to process large amount of data. In addition, algorithms on automata can have exponential complexity, so primitive operations should be as fast as possible. Efficiency cannot be sacrificed to gain the convenience of abstraction. Choosing the most relevant data structure is essential. However, many data structures exist to represent letters, alphabets, words, weights, series and automata. Furthermore, they are highly correlated since an automaton is built with series, a series is defined by words and weights, and a word by letters. Then each implementation is parameterized by some other implementations leading to something like a nest of dolls. Such implementations cannot be easily mixed in a monolithic hierarchy. Also, we want to reuse data structures from external libraries.

Thus, the design problem is to unify these two points of view into the same object to enable both implementation-driven and algebraic-driven writing of algorithms.

1.2 Generic algorithms and specialization power

Abstraction has lead to many algorithms with a general formulation. Generic programming is relevant, because general algorithms should be written once. More precisely, an algorithm can be generic w.r.t the mathematical concept and w.r.t the underlying data structure used as implementation of that concept.

However, some theoretical results are restricted to a precise algebraic context. Thus, we must be able to bound the algorithm input to a particular family of concepts. Likewise, algorithms can be written using the properties of a particular implementation, so restriction facilities over implementation parameters must be available. The figure 1 sums up some desirable specializations.



1. I, C are fixed ;
2. all I and C is fixed ;
3. all C and I is fixed ;
4. all sub-classes of I and C is fixed ;
5. all sub-classes of C and I is fixed ;
6. all sub-classes of C and I.

Fig. 1. Different type boundings of algorithm input

To be transparent to the final user and to improve genericity, these different specializations must be compatible with overloading.

1.3 Plan

The sequel of this paper is organized as follows. The section 2 shows the lack of answer in the well known C++ template techniques. Then, the section 3.1 presents our design pattern. In section 4, we apply it and observe that it fills our requirements. Finally, in section 5, the implementation techniques are presented to explain precisely how we deal with some pitfalls.

2 Confronting our desires with C++ template techniques

In practice, polymorphism implemented with late binding is too expensive for intensive computing. The generative power of C++ template mechanisms is known to enable abstraction with limited efficiency loss. The Standard Template Library (STL) has shown the workability of such polymorphism [4].

Yet, parameterization *à la* STL is unbounded. We cannot define two generic functions with the same name and the same arity because type variables are free. The Barton and Nackman trick [6] and other works [2] tend to reproduce the object oriented programming. The idea is to compel the open recursion to be static, *ie* the static type system knows exactly the subclasses that are used as instantiations of a particular abstract class. The following code attempts to illustrate this idea:

```
// This version is valid for any sub-class of A.
template <class C>
void algorithm(const A<C>& i);

// This version is valid for any sub-class of B.
template <class C>
void algorithm(const B<C>& i);
```

However, the one-dimensional discrimination of a single object hierarchy is not enough to design both the mathematical concept and the implementation. At first sight, the Bridge design pattern [3], or more precisely the Generic Bridge [1] design pattern could be suitable. Yet, the Generic Bridge is asymmetric, it is centered on the concept. Consequently, if the object is a concept parameterized by its implementation, specialization of type 4 is forbidden because of the invariance of the template argument. The following code illustrates this:

```
struct Matrix {};
struct CompressedMatrix : public Matrix {};

// This cannot be called with arguments of type A¡CompressedMatrix¿.
void algorithm(const A<Matrix>& i);
```

3 The Element design pattern

3.1 Presentation of the design pattern

Traditionally, concepts and implementations are separated by using abstract and concrete classes in hierarchies. Doing so, the chosen implementation for member function calls only depends on the actual class type. Then, a concept can be denoted by an abstract class whose subclasses are its implementations.

Our idea is to make explicit the separation between the concept and the implementation at the object level. We compose our entities with an instance of a concept class and an instance of an implementation.

By separating concepts and implementations in different hierarchies, we allow separate refinements of concepts and implementation algorithms. Moreover, we allow to use the same data type to implement distinct concepts, without the hassle of defining whole new concrete classes.

For example, elements of a tropical semiring are distinguished by their association with an instance of the concept $(\mathbb{Z}, max, +)$, and can be implemented by several basic C++ integer types. Conversely, basic C++ integer types can either represent elements of a tropical semiring or elements of a “classical” semiring $(\mathbb{Z}, +, \times)$, depending on the concept instance they are linked to. In either case, a single class *Element* is responsible for the composition.

As demonstrated later, this design entails more freedom and specialization facilities.

For the sake of simplicity, we denote the abstract concept associated to an entity, instance of class *Element*, its *structure* and the corresponding instance the *structural element*.

3.2 A two-component generic object

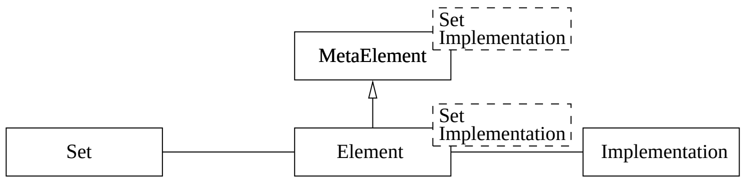


Fig. 2. Class diagram of the ELEMENT design pattern

The main item in this pattern is the *Element* class, a generic class which acts as a glue between a concept and an implementation. Indeed, the pattern can be read as *Element*<S, T> is *the type of an element of set S implemented by T*, or in other words *Element*<S, T> *structures the value type T by S*.

All services except construction and assignment are provided by decoration through the parent class *MetaElement*. Class *MetaElement<S, T>* serves three roles:

– *to specify the interface* for the class *Element<S, T>* viewed as an implementation of concept S denoting *how any instance of T must fulfill the requirement of the concept S*,
– to offer additional abstract services implemented using only services defined in the specified interfaces,
– to *link* services to their external implementations.

Of course, this class must be specialized over S and T. For additional genericity, the hierarchy between concepts should be mapped to a hierarchy between their specializations of *MetaElement*, so that the final *Element<S, T>* is decorated in correlation to the inheritance graph of concepts.

Figure 2 describes the design pattern in UML, while figure 3 details the decoration mechanism.

3.3 External functions as an adaptation layer

To reduce the number of *MetaElement* specializations, default code for concept requirement implementation is needed. We could define it directly in the specialization of *MetaElement<S, T>* for S fixed and T free, assuming the presence of methods and inner types (such as `begin() / end()`, `iterator`, *etc*). Yet, this solution inhibits implementation with partial default behavior. Moreover, this forces implementations to be C++ classes, whereas C++ builtin types or externally-defined structures can also be wanted as implementation types.

We decided to use external functions to define how the implementation fulfills concept requirements. Therefore, any *MetaElement* specialization is just a way to choose what external functions are to be used. Doing so, we introduce a fine grained specification of implementation services. At the same time, we also solve some binary method problems.

4 Applying the pattern: decomposition for specialization control

Given class *Element*, we can decompose any entity, or *Element* instance, for typing purpose. The following listing illustrates generic way to implement algorithms over *Element*:

```
// Generic wrapper
template <typename S, typename T>
void algorithm(const Element<S, T>& e)
{
  // Call the implementation, decomposing
  // the Element instance along the way.
  return algorithm_impl(e.set(), e.value(), e);
}
```

Once this framework is set up, implementations of `algorithms` can be specialized in any of the directions illustrated in figure 1. This is done by the following constructions:

```
// Type 1: the concept and value type are fixed.
void algorithm_impl(const S1& s, const T1& v, const Element<S1, T1>& e);

// Type 2: concept fixed, generic implementation for any value type.
template <class T>
void algorithm_impl(const S1& s, const T& v, const Element<S1, T>& e);

// Type 3: value type fixed, generic implementation for any concept.
template <class S>
void algorithm_impl(const S& s, const T1& v, const Element<S, T1>& e);

// Type 4: generic implementation for any sub-concept of S1.
template <class S, class T>
void algorithm_impl(const S1& s, const T& v, const Element<S, T>& e);

// Type 5: generic implementation for any value sub-class of T1.
template <class S, class T>
void algorithm_impl(const S& s, const T1& v, const Element<S, T>& e);

// Type 6: generic implementation for any sub-class of (S1,T1).
template <class S, class T>
void algorithm_impl(const S1& s, const T1& v, const Element<S, T>& e);
```

5 Implementation of class *Element*

5.1 Design considerations

The implementation of class template *Element*, and therefore the whole structure of the design pattern, was subject to the following guidelines:

- object instances of class *Element* should behave as "naturally" as possible w.r.t. the user. Especially, a user who has no experience with the library should be able to infer most of the use cases of *Element* from simple examples.
- the behavior and the set of available services in class *Element* can change depending on its static parameters. For example, instances of *Element* intended to represent values in an algebraic semiring have a `star()` method. Similarly, instances intended to represent automata have an `add_state()` method.
- at any time, a reference to the structural element of an *Element* instance can be retrieved with no computation cost. For example, it is possible from an instance of *Element* intended to represent a word to retrieve the whole alphabet over which it is defined.

– singleton structures should induce no memory footprint in *Element* instances. For example, there is no run-time data associated with the canonical semiring structural element surrounding the basic C++ types (`int`, `short`, ...). Therefore corresponding *Element* instances should be as small (from the C++ compiler's point of view) as the basic C++ types used as value types, for optimization purposes.

There are three facets in the current implementation of class *Element*, closely related to the requirements presented above.

5.2 *Element<S, T>* as a wrapper around *T*

Because *Element<S, T>* is actually a wrapper around type T, its main role is to aggregate a value of type T. Therefore, a number of basic services to handle the value data are provided by class *Element*, presented in table 1. Their use is valid iff the corresponding requirements over type T are met.

Description	Example use	Requirements
Referencing	`Element<S,T>& e;` `const Element<S,T>& ce;`	(none)
Access to the aggregated value	`T& v = e.value();` `const T& cv = ce.value();`	
Default construction	`Element<S,T> ev;`	T default-constructible
Copy construction	`Element<S,T> ev(ce);`	T copy-constructible
Construction from a value	`Element<S,T> ev(cv);`	
Assignment	`ev = e;`	T assignable
Destruction		T destructible

Table 1. Services of class *Element<S, T>*

These basic services are trivially implemented using only the properties of type T. They are therefore distinct from all additional services presented below, which also depend on type S and on the availability of related operators.

5.3 *Element<S, T>* as an element of a set

The power of our design pattern is that the same data type T can be structured by several distinct structural elements.

However, parameterization of class *Element* by the static type of its structure S is usually not sufficient. Indeed, a structure type S may denote several different structural elements with distinct behavioral influences on *Element<S, T>*. For instance, this can be observed in Vaucanson when using tropical semirings where the special infinity value is dynamically defined: the static type information (*TropicalSemiring*) is not sufficient to express the correct computation of addition and multiplication in the semiring, because the actual, dynamic value for infinity must be tested.

Because of this, we chose to hold a reference to the structural element in each object instances of *Element*<S, T>. For this purpose, *Element*<S, T> aggregates a reference to an instance of S *via* a mechanism presented in section 5.5. This reference can be retrieved with the `set()` method. For consistency purposes, the following properties must hold:

- once defined, the structural element of an *Element* instance cannot be changed nor modified; this is virtually ensured by `set()` returning a "const" reference.
- structural elements must be classifiable by means of `operator==`; this helps keeping[1] global instances of structures, using unique references to designate unique structural elements, for efficient by-reference comparisons.

Linking *Element* instances to structural elements is done at instantiation time, using the following construct:

```
Element<S,T> e(/* structural element */ s, /* value */ v);
```

Take note of the additional argument `s` given to the constructor of class *Element*. This construction does not invalidate the construction style presented in table 1; in fact, *Element* instances that have been constructed without giving a reference to the structural element are in a state called "transitional", during which only the basic operations are valid. Passing to the normal state is done by post-construction binding to a structural element with the `attach()` method:

```
Element<S,T> e(v);
// Here e is in the transitional, incomplete state.
e.attach(s);
// Now e is fully defined.
```

5.4 Subjecting the behavior of values to structures

The design of class *Element* targeted maximum extensibility via template specialization and method overloading, as presented in section 1.2. It was achieved by delegating computation for *all* services offered by class *Element* to global functions with special names (of the form `op_X`, for each operation `X`). These can then be refined *via* template specialization and function overloading (as in section 4).

By default, this delegation is set up for all standard C++ operations; table 2 shows how delegations are expressed and table 3 shows the mapping between standard C++ operations and special function names.

Distinction between sets of delegations is made by parameterized inheritance of class *Element*. Indeed, *Element*<S, T> inherits from *MetaElement*<S, T>,

[1] The uniqueness is ensured by a global type table discussed in section 5.6.

Description	Operation	Function call
binary operations	`e1 Op e2`	`op_##OpName(e1.set(), e2.set(), e1.value(), e2.value())`
	`e1 Op v1`	`op_##OpName(e1.set(), e1.value(), v1)`
	`v1 Op e1`	`op_##OpName(e1.set(), v1, e1.value())`
	`e1 Op v2`	`op_##OpName(e1.set(), e1.value(), op_convert(e1.set(), SELECT(T1), v2))`
	`v1 Op e2`	`op_##OpName(e2.set(), op_convert(e2.set(), SELECT(T2), v1), e2.value())`
difference	`e1 != e2`	`!(e1 == e2)`
comparison	`e1 > e2`	`e2 < e1`
	`e1 >= e2`	`!(e1 < e2)`
	`e1 <= e2`	`!(e2 < e1)`
negation	`- e1`	`op_neg(e1.set(), e1.value())`
prefix incr. and decr.	`Op e1`	`op_in_##OpName(e1.set(), e1.value())`
postfix incr. and decr.	`e1 Op`	`Element<S1,T1> copy(e1); Op copy`

`e1`: *Element*$<S_1, T_1>$, `e2`: *Element*$<S_2, T_2>$, `v1`: T_1, `v2`: T_2

Table 2. Delegation of standard C++ operations to function calls

which is by default empty but can be specialized to provide additional methods. For example, in Vaucanson delegations such as `star()` (`op_star`) for semiring elements or `add_state` (`op_add_state`) for automata, have been added.

As a matter of fact, all the standard delegations are set up in *Element*'s root parent class, *SyntacticDecorator*, from which each specialization of *MetaElement* must inherit directly or indirectly.

Figure 3 shows a UML description of the model.

5.5 Eliding references to structural elements

As presented in section 5.3, *Element*$<S, T>$ holds a reference to its structural element, an instance of type S. However in many cases a structural element is entirely defined by its static type S, i.e., there is no useful dynamic data associated to instances of S.

In these cases, a simple aggregation of a C++ reference (pointer) in *Element*$<S, T>$ would be a waste of memory space and time (for allocation and copy of the unneeded reference).

We avoided this waste by the encapsulation of the aggregation through a dedicated class, *SetSlot*. *SetSlot* derives from class *SetSlotAttribute*, parameterized by S and a Boolean value: the specialization of *SetSlotAttribute* for the Boolean true actually has a pointer attribute, whereas its default specialization has no such attribute but an accessor that returns a null reference.

`operator+()`	`op_add`	`operator+=()`	`op_in_add`
binary `operator-()`	`op_sub`	`operator-=()`	`op_in_sub`
`operator*()`	`op_mul`	`operator*=()`	`op_in_mul`
`operator/()`	`op_div`	`operator/=()`	`op_in_div`
`operator%()`	`op_mod`	`operator%=()`	`op_in_mod`

`operator=()`	`op_assign`
`operator==()`	`op_eq`
`operator<()`	`op_lt`
prefix `operator++()`	`op_in_inc`
prefix `operator--()`	`op_in_dec`
unary `operator-()`	`op_neg`
`swap()`	`op_swap`

Table 3. Mapping between C++ operator names and function names

When instantiating *SetSlot*, the Boolean attribute passed to the parent instance of *SetSlotAttribute* is taken from the value of `dynamic_traits<S>::ret`, *dynamic_traits* being a helper which defaults its attribute `ret` to `false` but can be specialized for any structure type S.

This mechanism is illustrated on figure 4.

5.6 Ensuring unique instances of structural elements

To allow efficient by-reference comparison of structural elements, a mechanism was set up to ensure that all *Element* instances sharing the same *dynamic type* (structurally equal structural elements) share the same reference to a unique structural element.

Practically, it ensures that if any two distinct *Element*<S, T> instances `e1` and `e2` were instantiated from distinct structural elements `s1` and `s2` verifying `s1 == s2`, then the property `&e1.set() == &e2.set()` always holds even if `&s1 != &s2`.

This was done by implementing an operator that keeps, for each static type S, a list of all distinct instances. It reports for any instance of S the address of the equivalent instance in the list, adding it to the list if necessary.

This mechanism therefore implies two requirements over S for *Element*<S, T>:

– S must possess an equivalence `operator==`,
– S must be copy-constructible, and values created by copy construction must be equal by means of `operator==`.

It is important to notice that this implementation is only efficient when there are few distinct instances of any structure. When a structure has many instances, containers such as `std::set` (requiring `operator<` over structural elements) or hash maps (requiring a hash function) could replace the list.

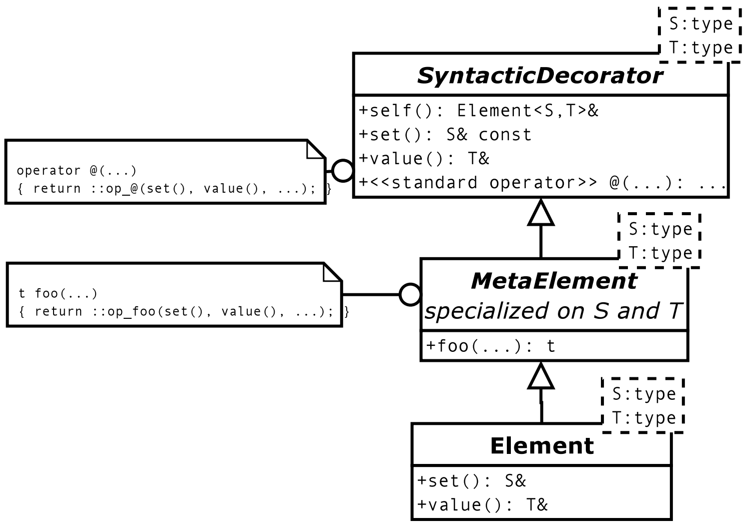


Fig. 3. Model for *Element*

5.7 Return types for operators

Most operations over *Element* instances return values whose types are independent from their arguments. That is, the return type can either be the same *Element* type or another basic or compound C++ type. However, some operators, especially arithmetical operators, should return a type computed from the types of its arguments. In Vaucanson this is shown, for instance, in the multiplication of a polynom by a weight or the lazy transposition of an automaton, which returns its argument encapsulated in a dedicated *TransposeView* adapter.

For this purpose, most operators are associated with a dedicated trait structure which computes the return type from both the structure type and value type; the generalized form for operators is thus:

template<**typename** S1, **typename** S2, **typename** T1, **typename** T2>
typename op_##OpName##_traits<S1, S2, T1, T2>::ret_t
operator Op(**const** Element<S1,T1>&, **const** Element<S2,T2>&)

with `op_##OpName##_traits` being specialized as needed.

Of course, since it represents the most widely used case, the default return type for `op_##OpName##_traits<S, S, T, T>` is *Element*<*S*, *T*>.

6 Conclusion

The Element design pattern is relevant for the orthogonal algorithm specialization problem, and we are thus using it successfully in Vaucanson. The idea

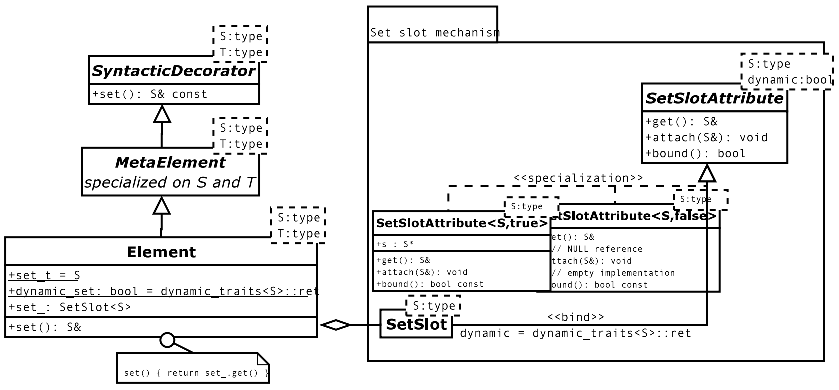


Fig. 4. Intelligent aggregation of references to structural elements

can be generalized to problems where objects are built with more than two orthogonal components. Therefore, we hope that this design pattern will be used in other fields.

Finally, we want to thanks Astrid Wang-Reboud, David Lesage, Nicolas Burrus, Niels Van-Vliet and Akim Demaille for their advises about both technical and writing issues.

References

[1] Alexandre Duret-Lutz, Thierry Géraud, and Akim Demaille. Design patterns for generic programming in C++. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, 2001.

[2] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL, the computational geometry algorithms library. Technical Report 3407, INRIA, April 1998.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.

[4] David R. Musser and Alexander A. Stepanov. Algorithm-oriented generic libraries. *Software - Practice and Experience*, 24(7):623–642, 1994.

[5] Jacques Sakarovitch. *Elment de thorie des automates*. 2003.

[6] Todd Veldhuizen. Techniques for scientific C++. Technical report, Indiana University Computer Science, 2000.

To OO or not to OO

Manuel Kessler

Institute for Aerodynamics and Gasdynamics (IAG), University of Stuttgart,
Pfaffenwaldring 21, D-70550 Stuttgart, `kessler@iag.uni-stuttgart.de`

Abstract. The High Performance Computing (HPC) community lags between 10 and 20 years behind other Information Technology (IT) areas. We figure out some of the reasons for this sluggish acceptance and incorporation of new technologies and try to find some arguments and prerequisites for a more lively and finally successful embracement of new insights, in order to be able to tackle the challenges of future HPC applications. However, OO—as any other technology—is no silver bullet, and some care is indicated.

Object-Oriented Design—and even more so OO Implementation—is still a new and for some people even suspicious approach to software development in the HPC field. The reasons for this unfortunate situation are many, among them

– real performance loss
– perceived performance loss (rumor, outdated experience)
– lack of training
– lack of time
– complexity of OO languages, like C++ or Java
– plain laziness, or, less offensive, inertia.

Let me elaborate on them a little more, and augment the argumentation with the experience of some past project using C++ or FORTRAN and some hints out of a starting project using C++. I will concentrate therefore basically on these two languages, although others may be similar to these.

Real performance loss This is a pity. If we look at the language definitions themselves, there is little reason why a piece of code expressed in FORTRAN should not be as fast as the same code in C++ syntax. Furthermore, the FORTRAN language provides little functionality not available in C++. The only real difference is the promise mandated by the FORTRAN standard, that function arguments may not be aliased, which enables some optimisations not available for a C++ (or C, for that matter) compiler. By the way, this peculiarity of FORTRAN has bitten countless programmers. The keyword `restrict`, which allows a selective definition of unaliased pointer or arguments, is standardised currently only for a new revision of C. Compilers, that really make use of that information are even more rare than others, that accept, but silently ignore this hint. However, while benchmark cases can be created in favour of FORTRAN for even an order of magnitude (on sufficient weird hardware), in real world aliasing or not is more of an academic question. Aliasing basically conflicts with loop unrolling and software pipelining. Speculative and out-of-order execution render these techniques mostly unnecessary with the possible exception of very tight, small loops. Even then, a good optimising compiler could easily figure out such cases, generate another instantiation of this function for unaliased arguments, and generate code to select between the two at runtime according to actual aliasing information.

Things are a little different when actual compiler implementations are taken into account. In my limited experience with free and commercial compilers, this is not yet settled. On the PC, the C++ compiler seems to be ahead of its FORTRAN counterpart, while on a workstation the ratio tends to be reverse. Understandably, because in the PC market the overwhelming majority of programmers use C/C++ over FORTRAN, while the situation on a supercomputer is quite contrary. This creates an interesting chicken-and-egg problem. The FORTRAN compiler has undergone many revisions and is now quite mature, including its optimiser, which has received decent regard. The C++ compiler is a new technology, and the compiler writers still struggle with the implementation of the language itself (often enough unsuccessful) instead of improving the performance. Consequently, the result is disappointing, an equivalent piece of code is some small or large factor slower than the FORTRAN program. Accordingly the programmer attributes this compiler inadequacy to the language, "C++ is dead slow" and goes back to its trusted FORTRAN thing. So nobody uses C++ on the supercomputer, and therefore no manufacturer is interested to invest money in improving the compiler. Unless some brave—or stupid—people coming from the PC world stick with their language of choice, kick the vendor and therefore create some market pressure, there is no way out of this vicious circle. Fortunately, this seems to happen right now and there is some progress, at least from some manufacturers.

Perceived performance loss There is no argument against that besides convincing hard data, showing the same (or better) performance for exactly the same application. "An uncle of mind had a colleague who knew someone who tried it and he was not satisfied." Of course, the information content of such rumour is very close to zero. However, more often than not this is the most prominent obstacle to the acceptance of C++ (or OO in general), since it creates a feeling of inappropriateness. And only a fool would use a tool considered inappropriate.

On the other hand, a seemingly real performance loss is in many cases only a perceived performance loss, as a different compiler or only a newer release would have been much more capable than the tried one. As stated before, compiler technology for C++ proceeds quickly, first to support the standard language, but recently more on optimisations. Therefore a benchmark run three or five years before is basically useless these days. Furthermore, even if the system compiler is still inadequate in its newest incarnation, some third

parties make a living from providing decent compiler technology for a number of platforms, and in the HPC market as well.
A still different reason for perceived performance loss is the inappropriate use of language features, which is a pretty straightforward consequence of the complexity of the language combined with the lack of training of people trying to use C++. Again, this is some kind of a chicken-and-egg problem, as there is little training material of high quality, so nobody gets some proper training, so nobody is able to produce some material and provide training. There are several books about using C++ for engineering and computational intensive applications, but none so far (to the best of my knowledge) with concepts and techniques useful in HPC for performance on par with FORTRAN. Implementation elegance, that leads to three pointer indirections and two virtual function calls for each floating point operation is not the way to go if even hand optimised code on a supercomputer runs for days. So only a very skilled person using the appropriate features of the language and a recent sophisticated compiler on the destination platform can do a trustworthy benchmark and thus enable a solid differentiation between real performance loss, which is to be attributed to the language, or language concept, and perceived performance loss, which is the user's fault.

Let me substantiate these claims regarding performance loss (real or perceived) with some real figures. As a test case I used the well known Stepanov benchmark, which just adds a number of double values. This is done with increasing levels of abstraction, by hiding the value in a struct, using an iterator instead of a pointer, and masquerading the iterator again with a reverse_iterator, in every possible combination. The time required for each variant is compared to a plain C function, using array syntax in the most basic way. The ratio for each variant is listed, and as a final number (every benchmark necessarily has to boil down to a final number) the geometric mean of all these ratios is output. Of course this benchmark is as artificial as any one, being able to prove nearly anything. You certainly know the old saying: there are lies, damned lies, and benchmarks. The actual code was taken from [1] and slightly modified to check the performance in native C and FORTRAN as well in one go.

First I collected some older results from the Internet [2], generated with an older generation of C++ compilers, approximately 2-5 years old. Note that the Power Mac results are taken directly from Apple [3] and thus to be read with a grain of salt, especially regarding the superior performance of their own compiler compared to Metroworks CodeWarrior.

Machine	Compiler	MFLOPs	Abstraction penalty
IBM	xlC	4.3	4.96
	KCC	9.5	2.34
Power Mac	CW 11	5	12.44
	MrC 3.0	59	1.00
SGI #1	KCC	22.4	1.00
SGI #2	CC	110	1.69
T3E	CC	10.2	3.85
	KCC	53.4	0.74

This data could be interpreted as somewhat encouraging, since one can easily see that there do exist compilers reaching an abstraction penalty of 1.00, thus optimising away all the data abstractions. This is supported by other authors, for example using the Blitz++ library [4]. However, the absolute level of performance in our case is partly disappointing. For example, even as I do not know the exact model of the T3E, even the smallest model has a theoretical peak performance of 600 MFLOPs per CPU. This special machine is notoriously difficult to push higher than 10% of the peak performance, even in FORTRAN 77, as some colleagues of mine had to find out after struggling with it for much longer than they would have liked to. They gave up at approximately 15% peak (using FORTRAN).

For comparison of more recent compilers I had several workstation type computers at my disposal, as well as some high performance machines. Although I played a little bit with optimisation options, some obscure combination may improve the numbers dramatically on one machine or the other.

Machine	Compiler	MFLOPs	Abstraction penalty
Athlon XP 2000+	gcc 3.2	415	1.00
	icc 7.1	426	0.98
SGI, R10K	gcc 2.95.2	64	1.04
	CC 7.3.1	190	1.01
SGI, R12K	gcc 3.0.2	159	0.85
	CC 7.4	108	3.44
SUN, Ultra SPARC III	gcc 2.95.3	211	0.53
	CC 5.3	114	8.22
CRAY T3E-900	CC 3.5.0.1	23	6.85
Hitachi SR8000	gcc 2.95.2	22	1.00
	CC	14	4.1
NEC SX-5	CC 1.0 rev 0.55	15	60.25

The absolute values again are quite disappointing. The Athlon is pretty much on its maximum level, besides assembly language, but all the other machines show certainly room for improvement. Our sole comfort is that FORTRAN and C versions of this simple loop are often relatively slow as well. The Power Mac could probably come close to 360 MFLOPs, the two SGI's 400 respectively 800 MFLOPs (179 in FORTRAN, 194 in C respectively 370 and 400), SUN should be able to reach 900 MFLOPs (625 in FORTRAN, 740 in C), this T3E model 900 MFLOPs (218 in FORTRAN, 182 in C), the Hitachi 1000 MFLOPs on a single CPU (59 in FORTRAN, 61 in C), and the NEC even 4 GFLOPs (930 MFLOPs in FORTRAN, 950 in C). The NEC result is insofar understandable, as the C++ compiler is obviously not able to vectorise the loops, being thus forced to use the much slower scalar unit, but even then the 3-5 MFLOPs of the higher abstract test cases are a pity. The SGI's are a little bit peculiar. In principal the native compiler does a really good job, as the R10K case proves. But there has been a regression going from version 7.3.1 to 7.4, where the compiler obviously lacks scalar replacement for all cases where the double wrapped inside a struct. Leaving this cases out, or running the 7.3.1 code on the R12K machine gives near optimal results of 395 MFLOPs and an abstraction level of 1.00.

As said before, I certainly do not claim these numbers to be exact, and as the machines are not dedicated, there is always some error margin of 5-10%. However, the general trend is quite clear. Workstation type machines have come to a level where C++ is on par with C and even FORTRAN regarding performance, so any performance loss is probably either based on outdated data or misusing the language. Moving on to the high performance machines, speed severely breaks down, proving my statements about the real performance loss on those type of computers. Let me now proceed to less measurable factors, which are often even more important in real life than hard numbers.

Lack of training The training problem has already been brought up in the last section. Most of the development for HPC happens on universities and research institutes, which are quite similar actually (at least here in Germany). They want to get a problem solved, in our case some fluid dynamics or aeroacoustics problem, and use HPC as a tool to do this job. They are engineers, not programmers, and only have some limited experience in coding, let alone in OO languages. Consequently, they use what they have at their disposal, and what they are told. The instructors were in the same situation only a few years before, and consequently can provide help only with the tools they were told to use, and so on. More often than not, HPC development is a little of copy-and-paste of some previous code, messing around with it and tailoring it to the current needs. I have seen this everywhere I have been working so far again and again. New programming techniques are thus adopted only if a new colleague brings in her own experience, but can hardly grow internally. And the old way has been successful, for so many years now, so there seems to be no need to invest in training.

Lack of time Closely connected to this issue is the lack of time to make the transition to OO. Training needs time, re-implementation of old and grown applications with a new design and a new language takes even more time, and time is a scarce resource. Most of the people in the HPC only work for some limited time there, for example to complete a Ph.D., and they desperately need all the time to familiarise themselves with the field of study, to understand the problem, to solve it in the framework provided, and to write their thesis. Often there is only little temporal overlap between two people, which results in less communication and thus more time needed to become acquainted with everything necessary. Few people ensure continuity to such a project for, say, a decade or more, and thus would be candidates for extensive training besides their other duties.

This lack of time for training is complemented by the lack of time to rewrite the applications. Many of them have grown over many years and many people working on it, and have reached a level of maturity and plethora of features, that a complete rewrite of these applications in a more modern design and language would require a huge effort, without any gain in functionality. Of course, later on new functionality could be added much more quickly (at least we hope so), but the start-up costs are prohibitively expensive. Combined with the situation of high fluctuation in staff, and their destination to solve a new problem in limited time, this adds to a nearly insurmountable amount of inertia.

Complexity C++ is undoubtedly one of the most complex languages available, probably even *the* most complex one. There are countless ways to solve a problem in C++, as several programming styles or paradigms are supported: procedural (inheritageheritage of C), modular (as with Modula), object oriented (taken from Simula, and partly Smalltalk), generic (with templates inspired by Ada), and of course any combination of them. The basic philosophy in the design of the language was to provide the programmer every detail she might need, and to give control over pretty much everything. That is the reason why there is no garbage collection in C++, for example. Consequently, you can do nearly everything with C++, even on a very low level, but there are also countless ways to shoot yourself into the foot. The initial learning curve is quite steep, and you need very much experience to choose sensibly between all implementation variants. Things are getting worse in the HPC area, because it does not lead to success to just hack up an application, and expect high performance from such an ad-hoc attempt. Some language features strictly necessary for adequate performance are unfortunately challenging enough to be above the level of common programmers. Together with the lack of training this is probably the most serious obstacle to widespread acceptance.

Laziness Well, little can be done about that. But we should remember, that software, in C++ or in FORTRAN, is written by human beings, and by their very nature they tend to be less fervent than their managers would like them to be (the same is true for the managers themselves, of course). As a matter of fact, learning is laborious, and there is little interest to invest much effort as long as there is no or little benefit obvious. New concepts mean to say good bye to old and trusted habits, and human nature is that much inert. Therefore motivation is everything, and while we do not want to follow the general trend to short-lived hypes in IT (like Java, B2B, J2EE, E-commerce, M-commerce, ...), it could help to establish some feeling of “coolness” of C++ and OO in the HPC community, but only if we simultaneously appreciate the hard work required to reach the level of sophistication necessary for successful applications.

After having seen all this more or less reasonable arguments, one could easily conclude that OO is just not worth the effort. And in our opinion, this is probably true for some, perhaps even for the majority of applications. Remember, always use the right tool for the job, and sometimes this definitely is FORTRAN77.

Our group is now in the process of designing and writing a new parallel fluid dynamic solver, drawing from experience gained previously. I have implemented myself three such solvers (for unstructured meshes) in C++ before [5, 6, 7], with a more limited application area, but basically in the same field. Moreover, I have seen and worked with two more solvers implemented in C++ by other people, and four in different FORTRAN dialects. If I take my last solver as an example, the advantages of using C++ are quite obvious:

– At first I used a direct solver for sparse, irregular matrices. With the problems growing in size, a new handling of these matrices got necessary, to improve the performance to acceptable levels. Therefore the solver was replaced by a sophisticated renumbering scheme and Cholesky decomposition with little fill-in (using the Math Template Library [8]). Besides the new implementation of this algorithm, the rest of the program was nearly unaffected. Later on, for even larger problems, this direct solver was too memory intensive, so I replaced it with an iterative conjugate gradient method. Again, the global impact was negligible.
– Another part of the problem to solve was to compute generalised eigenvalues and eigenvectors of large matrices. A direct method (with LAPACK++[9]) was used at first and worked well until the problem size exceeded RAM space. After that I switched (at runtime, depending on problem size) to an iterative Arnoldi method [10], provided by ARPACK++ [11], which is a wrapper for ARPACK [12]. In this latter case, I did not have to adapt my data structures to the ones required by ARPACK++, but only provided some fundamental operations to the library, like multiplication of a vector with the matrix. Again, this was only a very localised change.
– Very late in the project I decided to do (limited) three dimensional problems as well. More specifically, only domains of a two dimensional mesh extruded along a perpendicular axis were allowed. This quite significant increase in functionality was accomplished in less than two weeks, retaining the efficiency of the code for planar problems. Even in this fundamental case the global impact was minimal, as the entire equation solution and eigenvalue computation was unaffected.

There is no comparison with a procedural design and a FORTRAN implementation possible. However, based on my experience collected along the project I would roughly estimate a doubling in effort, and possibly much more. Others may have found different results, but as I do not know any real world benchmark between two highly skilled programmers in their respective language, it is difficult to arbitrate objectively.

For our new solver we try to separate concerns as much as possible. Extensive use of templates at the computational heart facilitates the exchange of equivalent components (for example viscous/inviscid data, different flux computations etc.) without sacrificing performance. However, in less performance critical paths inheritance and virtual functions are used as well in order to decouple different modules from each other. The design of HPC applications is in some aspects radically different from more interactive programs, where one of the most important rules is to optimise later. If we would do so in our context, the overall structure would probably completely dissimilar and could hardly be polished to satisfactory speed without a major redesign. As high performance is a key feature of HPC, it has to be taken into consideration from the beginnings.

However, not only technical reasons and design decisions contribute to the success or failure of a project. In our opinion, some basic requirements should be fulfilled, such that a sensible application of C++ can help us reach new levels of functionality—and in rare cases even of performance, as more complex and possibly superior algorithms are getting into reach, or new hardware features can be exploited. If these premises do not hold now, or are at least expected in the near future, the potential benefits should be very carefully weighed up against the investment of time and money into a transition to OO.

Availability Application development never happens in empty space, but is tailored to one or several platforms. In the HPC market the platforms are diverse, and many different technologies compete with each other, for example highly parallel machines with vector computers, distributed with shared memory, common-off-the-shelf components and special purpose equipment. If no C++ compiler is available, there is little to discuss. And if there is a compiler at our hands, but full of bugs and with a lousy optimiser, and the vendor does not show any initiative or competence to improve it in the very near future, the situation is only slightly better. So a serious evaluation of the destination platform(s) is the very first step before going any further. On workstations, or clusters of workstations for that matter, the tendency today is that there is little loss in performance, if the language is used properly. For the remaining 5-20% that might be left other factors are much more important, as cache issues, loop ordering, I/O, parallelisation opportunities. On supercomputers, prospects are still dismal.

Necessity It is difficult to advocate the adoption of new technology just for the sake of technical superiority or elegance. A gradual transition to an OO implementation is usually very difficult and results often in a complete rewrite, but with the old structure still retained, and thus little advantage. Therefore a new startup from scratch is the better alternative, but only if we can justify the time to lift the rewrite to already available functionality by clear advantages for further, already scheduled improvements. The best date for such a rewrite is when such an improvement would be result in a major redesign of the existing code anyway. We have this for example when we wanted to move a monolithic code, written for a shared memory vector supercomputer, to a new platform with many nodes and distributed memory. The monolithic data structures would have to be completely redesigned and broken up for a message passing paradigm with explicit communication instead of data exchange in shared memory, and at this point experience tells us that this results more or less in a complete rewrite. In such a case, a rewrite with a new technology and new design is much simpler to account for than for an evolutionary process.

Support The powers that be have to provide the necessary setting for the redesign and a new implementation. This includes training for the people working on the project, availability of time to reach the goals and some tolerance against one or the other complicacy throughout. Especially if such a task is undertaken for the first time, many issues will eventually come up and hinder progress. A certain long sight is then needed not to give up half-way through before the profits can be gathered.

Continuity Along the same lines is the requirement of some continuity in the project, especially for the people. A high fluctuation in early phases of design and implementation (where the design often has to be adjusted) is clearly undesirable, as new personnel needs adjustment to the job and thus distract others from their work. At least the project leader making all the major design decisions should stay until a level of functionality comparable with the old code is reached. Later on, when the design and most of the code base have settled and it comes to a continuing evolution of the software this continuity is less imperative, but even then somebody with longer experience should guide the others in their strategies on how to improve the code.

Competence This is probably the most important one prerequisite, and the most difficult one to guarantee. I consider a project doomed without at least one highly skilled person with experience in a similar subject, since nobody else is able to devise a sound design and head the rest of the team through early stages. Such people are very rare, as is the application of OO and C++ in HPC in general. Therefore only very few have the chance to gain experience in a similar subject, since an excellent programmer for banking software, for example, is not necessarily equally qualified for HPC.

Taking these requirements into account, only part of all HPC applications seem to qualify for a transition to OO. Some of these requirements will certainly improve over time, as the lack of decent compilers, as well as the scarcity of competent programmers. Others are more difficult to estimate, as the support in management and the motivation of people to learn new skills. So if we try to answer the question initially stated in the title, all we can reply responsibly is:

It depends.

References

[1] Stepanov, A.: KAI's version of Stepanov benchmark code – version 1.2. `http://www.physics.ohio-state.edu/~wilkins/computing/benchmark/stepano% v.html` (1997) Checked June 2003.

[2] Wilkins, J.: Effect of standard template classes on performance. `http://www.physics.ohio-state.edu/~wilkins/computing/benchmark/STC.htm% l` (1998) Checked June 2003.

[3] Apple, Inc.: stepanov. `http://developer.apple.com/tools/mpw-tools/compilers/benchmarks/mrc-3.% 0f1/stepanov-3.0f1.html` (1997) Checked June 2003.

[4] Veldhuizen, T.: Blitz++ home page. `http:www.oonumerics.org/blitz` (1996) Checked June 2003.

[5] Keßler, M.: Numerisch berechnung zäher strömungen. Studienarbeit, Universität Würzburg (1995)

[6] Keßler, M.: Uzawa-Methoden zur numerischen Behandlung von Strömungsproblemenn. Diplomarbeit, Universität Würzburg (1997)

[7] Keßler, M.: Die Ladyzhenskaja-Konstante in der numerischen Behandlung von Strömungsproblemen. Dissertation, Universität Würzburg (2000)

[8] Lumsdaine, A., Siek, J.G., Lee, L.Q.: Mtl home page. `www.osl.iu.edu/research/mtl` (1999) Checked June 2003.

[9] Dongarra, J., Pozo, R., Walker, D.: Lapack++: A design overview of object-oriented extensions for high performance linear algebra. In: Proceedings of Supercomputing '93, IEEE Press (1993) 162–171

[10] Arnoldi, W.: The principle of minimized iterations in the solution of the matrix eigenvalue problem. Quart. Appl. Math. **9** (1951) 165–190

[11] Gomes, F.M., Sorensen, D.C.: ARPACK++: A C++ Implementation of ARPACK Eigenvalue Package. (1997)

[12] Lehoucq, R.B., Sorensen, D.C., Yang, C.: ARPACK User's Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods. (1997)

Functional Programs on Clusters[⋆]

Viktória Zsók, Zoltán Horváth, Zoltán Varga

Department of General Computer Science
University of Eötvös Loránd, Budapest
e-mail: hz@inf.elte.hu, Zoltan.2.Varga@nokia.com, zsv@inf.elte.hu

Abstract. The implemented Clean-CORBA and Haskell-CORBA interfaces open a way for developing parallel and distributed applications on clusters consisting of components written in functional programming languages, like Clean and Haskell.
We focus on a specific application of this tool in this paper. We design and implement an abstract communication layer based on CORBA server objects. Using this layer we can build up computations in form of distributed process-networks consisting of components written in several programming languages, some components written in functional style in Clean, while other components written in an object-oriented language like Java or C++.
The speed-up of computations is investigated using a simple example.

1 Introduction

One of the easiest way to provide powerful infrastructure for parallel and distributed computing is to build a cluster and interconnect clusters via the internet into a Grid.

Functional programming is very suitable for expressing parallelism. Composition of functions is an associative operation, so evaluation of functional programs can be done in parallel or distributed way. So functional programs are inherently parallel but the evaluation in parallel of an expression is not always worthwhile.

There are several elements in the functional programming language Clean which support to control parallel and distributed evaluation and communication [11, 1, 13]. Also the Haskell language has several dialects with parallel features: GpH [9], pH [10], Eden [4], Distributed Haskell with Ports [8]. These solutions are different in efficiency and in power of expressiveness and require different hardware and software infrastructure.

A higher degree of abstraction level expressing parallelism can be achieved by parameterizing computational skeletons with evaluation strategies. Evaluation strategies [12, 6] may be applied in parallel computations separating dynamic evaluation issues from static requirements. Evaluation strategies are appropriate tools in order to control the evaluation order and degree, the dynamic behaviour

[⋆] Position paper. Technical paper to appear in Proceedings of SPLST'03 [7]. Supported by the Hungarian National Science Research Grant (OTKA), Grant No. T037742 and by IKTA 89/2002 (JiniGrid).

and the parallelism [12]. A skeleton is a parameterized algorithmic scheme. Skeletons in functional languages are higher order functions parameterized by functions, types and evaluation strategies. There were several studies regarding skeletons [3, 12] from the apparently very simple but very useful skeleton `parmap`, to the more complex skeletons like the parallel elementwise processing [6].

Functional programs can also be developed and tested on cluster systems. The first study was the comparison of the GpH and the Eden languages regarding their performances [9]. The GpH and Eden comparison was done on a Beowulf cluster. A Haskell version of parallel elementwise processing implemented on a cluster was presented in [5].

Our intention is to test and to verify how the Clean functional programming language fits into the parallel programming framework offered by clusters. We use an architecture, which allows to build up applications consisting components written in several programming languages, some components written in pure functional style for example in Clean, while other components written in an object-oriented language.

A Clean-CORBA interface [13] is used as an infrastructure for parallel communication. The interface implements a language mapping from Clean to IDL. Our Clean-CORBA interface uses the MICO CORBA implementation and allows to write CORBA clients and servers in the lazy functional programming language Clean.

We designed and implemented an abstract communication layer based on this software architecture. The distributed computation is built up from components implemented in form of CORBA clients. These components communicate via channels which are CORBA server objects. The channel object is written in two variants, in Clean and in C++. The clients may be written in any language with CORBA interface.

We have chosen an implementation of a pipeline computation as an example in this paper to present the main features of our approach. We implemented the clients of this example in functional style, in Clean. We measured the performance of the application on a cluster consisting of 16 processors.

Section 2 describes the Clean-CORBA interface. The mapping from the CORBA IDL to the Clean functional language is described according to the language elements.

The third section presents an implementation of asynchronous communication channel, which can be used for connecting Clean programs and other programs in a cluster environment.

The pipeline skeleton is very suitable for the computation of functions which can be built by the composition of small components, for the detailed description of the problem see the fourth section.

The last section (section 5) concludes.

2 Clean-CORBA interface

To access CORBA from a programming language a language mapping for the particular language is needed. This mapping should contain the following elements: an IDL module mapping to the specific language, the simple and composed types of IDL association with the types of the language, the projections of the definitions and operations of the IDL interface, the implementation of services offered by the CORBA server and of the pseudo-objects of the CORBA into the language.

In Clean-CORBA interface the operations are associated with functions, CORBA objects with Clean records. For communication through TCP ports and for IP identification the services of MICO Binder are used. Interfaces are generated differently for clients and for servers.

The identifiers of the IDL are the same in Clean, the names of the modules are included in the identifiers. IDL constants are mapped to Clean constants.

The different integer types are associated with the `Int` type of Clean, in the same way the real types are projected into the `Real` type of the Clean language.

Enumeration types are mapped to Clean algebraic types.

IDL Structures are mapped to Clean records. The field names remain the same. If the structure contains an 'anonymous' field (like `sequence <long> m3`), then the IDL compiler will create a new Clean type (in this case `Foo__m3`), and this will be the type of the corresponding field in the Clean record. Recursive structures and unions are supported too. IDL unions map to Clean algebraic data types, with one data constructor for each legal discriminator value. IDL sequences map to Clean lists.

The most interesting is the mapping of `TypeCode`, which gives us information about the IDL types during runtime.

IDL Interfaces map to abstract Clean types, which contain the object reference in their hidden parts. Each interface type has a corresponding `<T>__nil` function which returns a NIL object reference of the given type. Conversions between interface types are supported through `<T>__narrow` and `<T>__widen` functions generated by the IDL compiler.

Each IDL operation maps to a Clean function which performs the CORBA call (for examples see [13, 7]). The first argument of each function is the receiver CORBA object. Since these functions have side effects, they both take and return a unique `World` argument which represents the environment of a Clean program. The operation my fail, so the result belongs to the algebraic type `ResultOrException`, which is an union type. If the IDL operation has `out` or `inout` arguments, the functions return them, too. For example:

```
Account_balance2 :: Account *World
->(((ResultOrException (CORBA_Void,CORBA_Long) CORBAException),
     *World))
```

For each IDL attribute, the IDL compiler will generate both a getter and a setter function.

The Dynamic Invocation Interface (DII) is supported through the following function:

```
CORBA_invoke :: CORBA_Object String [CORBAArg] TypeCode [TypeCode]
*World -> (Any, CORBAException, [CORBAArg], *World)
```

The meaning of the arguments: target CORBA object, the name of the operation, a list of the arguments, return type of the operation, typecodes of IDL exceptions, the unique environment: the world.

The result is a tuple with the following parts: the return value of the operation, the exception raised by the operation, if any, the value of the `out` and `inout` arguments, the new World.

The server side mapping uses a simplified version of the Object IO framework [1]. The IDL compiler generates servant types for each IDL interface. A servant is a record type with one field for each IDL operation in the interface. The programmer must create an instance of this servant type, and register it with the system before it can answer CORBA requests.

The implementation consists of a CORBA-CLEAN interface library, and an IDL-TO-CLEAN compiler. The interface library consists of three layers:

1. The lowest layer is a collection of C functions giving access to CORBA functionality.
2. The middle layer simply consists of Clean wrapper functions around the C functions in the previous layer.
3. The third layer contains the high level interface described above.

The implementation uses CORBA DII and DSI for communication, similarly to the MICO-TCL interface software TclMico.

The IDL compiler works by first uploading the contents of the IDL file into a CORBA Interface Repository daemon, then reading this data using normal CORBA calls into an intermediate representation, and finally generating Clean code.

For detailed description end examples of the mapping see [13, 7].

3 The implementation of a channel object

Many problems can be viewed as networks of message-communicating processes, therefore it is very useful to implement an abstract channel object for asynchronous message passing.

To interconnect processes or distributed programs we can implement communication primitives for asynchronous message passing using CORBA server objects. We store the messages in the local state of the server.

The program has to import the `channel` interface, which defines the channel operations. The program also has to import the Clean standard environment and the `Corba` package. These are the basic modules for our Clean-CORBA interface.

The initialization of the CORBA system uses the `CORBA_ORB_init` function, which returns a `CORBA_ORB` object. `CORBA_Server_run` initializes the CORBA server.

In our model the channel initializes the ORB and starts a CORBA event handler. By the `ServerInit` we create a servant, which will be registered by the ORB. `ServerInit` transforms the general object reference into the desired type. The event handler system will assure that the requests of the clients are passed to the servant objects.

```
Start w
   # (orb,_,w) = CORBA_ORB_init args w
   = CORBA_Server_run orb Void ServerInit w
where
 ServerInit ps w
  # (obj, ps, w) = Channel__servant_open ps servant w
  # w
    = WriteIORToFile (CORBA_Server_get_orb ps) obj "channel.ior" w
  = (ps, w)
 servant = { Channel__servant |
                      ls             = messages,
                      impl_send      = my_send,
                      impl_receive   = my_receive,
                      }
        my_send (ls, ps) what w
                = ((ls ++ [what], ps), Result Void , w)
        my_receive ([x:xs], ps) w
                = ((xs, ps), Result x, w)
```

`Channel__servant_open` registers the servant at the IO system. The servant defines the operations of the channel. These operations are state transition functions, which modifies the local state of the channel (`ls`). The `messages` is the sequence containing the elements of the channel. The function `my_send` is the implementation of the channel operation `send` and adds to the sequence an element sent by the client. The `my_receive` function implements the channel function `receive` and sends to the client one data from the sequence.

4 The pipeline skeleton

The pipeline skeleton is a special type of process network usually applied for calculating a composite function. The processes are organized linearly. A processes running on a pipeline element calculates a component function and sends intermediate results to its immediate successor. The data input is processed at the beginning of the pipeline.

We consider a simple description of the pipeline problem [2].

Let $D = \ll d_0, d_1, \ldots, d_M \gg$ be a sequence of data, where $M \gg N$, and let $F = \ll f_0, f_1, \ldots, f_N \gg$ be a sequence of functions.

Let $f^i(x)$ denote $f_i(f_{i-1}(\dots f_0(x)\dots))$; we assume that $f^i(x)$ is defined for all i, $0 \leq i \leq N$ and all x in D.

We compute the sequence $f^N(D)$, where $f^N(D) = \ll f^N(d_0), \dots, f^N(d_M) \gg$.

The pipeline problem is implemented in the following form: the Clean programs are Corba-clients and calculate the components of F.

The computation can be parameterized by the component function f_i and by the type of its argument (skeleton). The send and receive functions are implemented by the abstract channel CORBA server, the object presented in the previous section.

For sending data on the channel we have the following function:

```
sendf x obj w
    # (Result l, w) = Channel_full obj w
    | l             = sendf x obj w
    = Channel_send obj (f x) w
```

The function checks if the sequence of data is full. In case is full will try again, in case it is not full will send the data to the server object. For receiving data we have the following function:

```
receivef obj w
    # (Result l, w) = Channel_empty obj w
    | l             = receivef obj w
    = Channel_receive obj w
```

The function verifies if the sequence is empty. If it is empty then will try again, otherwise receives a data from the server.

As an example we compute $sin(x) \approx \sum_{i=0}^{n} (-1)^i * \frac{x^{2i+1}}{(2i+1)!}$. For this we use the following data structure: $d = (xx : Real, s : Real, e : \{1.0, -1.0\}, h : Real)$. The function $sin(x) \approx sin_n \circ \dots \circ sin_0(x)$, where

$sin_0(x) = (x^2, x, -1.0, x)$

$sin_i(d) = (d.xx, d.s + d.e * d.h * \frac{d.xx}{(2i)*(2i+1)}, d.e * (-1), d.h * \frac{d.xx}{(2i)*(2i+1)})$

$sin_n(d) = d.s + d.e * d.h * \frac{d.xx}{(2n)*(2n+1)}$

The following lemma can be proved:

$f^i(x) = f_i \circ \dots \circ f_0(x) = (x^2, \sum_{j=0}^{i} (-1)^j * \frac{x^{2j+1}}{(2j+1)!}, (-1)^{i+1}, \frac{x^{2i+1}}{(2i+1)!})$

for all $i = 0, \dots, n-1$.

According to the lemma the pipeline skeleton will produce a correct result.

The evaluation order of Clean programs is lazy, so the evaluation of some expressions may be postponed by the run-time system. In case of distributed applications the order of evaluation may be important in several cases, so for some expressions a strict evaluation should be enforced. Functions returning the value of the system clock has to be evaluated strictly for example.

5 Performance measurement

The cost of the communication via the CORBA server objects is relatively high compared to the cost of this simple computation. If we slow down the computation sin_i functions simulating a most complex computation (we apply a weighted function), then we can observe even a small speedup (see figure 1).

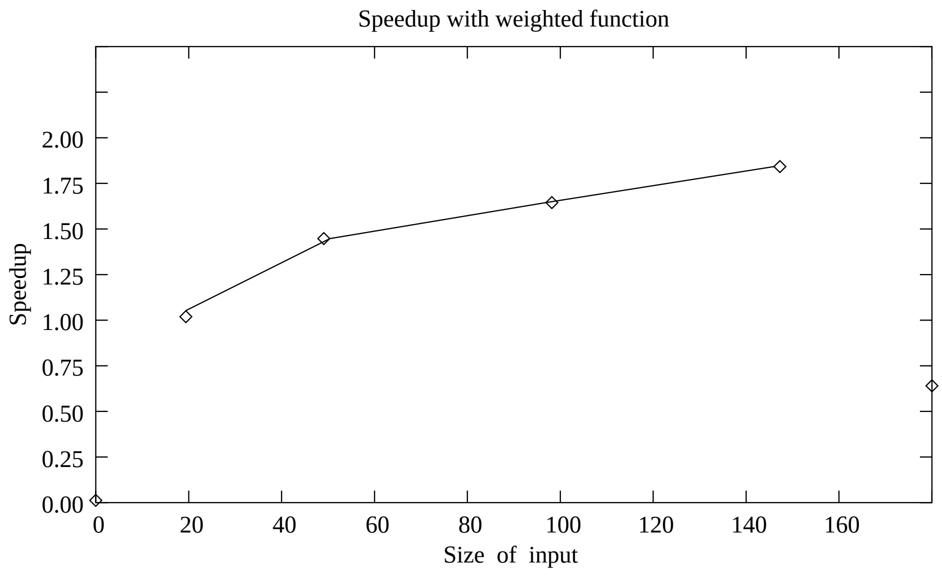


Fig. 1. Speedup with different number of input data

6 Conclusions

The presented Clean-CORBA interface and the abstract communication layer on top of it is applicable for implementing computations in form of distributed process-networks. The application may consist of components written in several programming languages. We presented a simple pipeline computation written in the pure functional language Clean. We observed a small speed-up of this computation on a 16 processor cluster.

References

[1] Achten, P., Wierich, M.:*A Tutorial to the Clean Object I/O Library*, University of Nijmegen, 2000. http://www.cs.kun.nl/˜ clean

[2] Chandy, K. M., Misra, J.: *Parallel Program Design*, Addison-Wesley, 1989.

[3] Cole, M.: Algorithmic Skeletons, In: Hammond, K., Michaelson, G., eds., *Research Directions in Parallel Functional Programming*, pp. 289-303, Springer-Verlag, 1999.

[4] Galán, L.A., Pareja, C., Peña, R.: Functional Skeletons Generate Process Topologies in Eden, In: *Int. Symp. on Programming Languages, Implementations Logics and Programs PLILP'96*, Aachen, Germany, LNCS, Vol. 1140, pp. 289-303, Springer-Verlag, 1996.

[5] Horváth Z., Hernyák Z., Kozsik T., Tejfel M., Ulbert A.: A Data Intensive Application on a Cluster - Parallel Elementwise Processing, In: Kacsuk P., Kranzlmüller D., Neméth Zs., Volkert J. (eds.): *Distributed and Parallel System - Cluster and Grid Computing, Proc. of 4th Austrian-Hungarian Workshop on Distributed and Parallel Systems*, Kluwer Academic Publishers, The Kluwer International Series in Engineering and Computer Science, Vol. 706, pp. 46-53, Linz, Austria, September 29-October 2, 2002.

[6] Horváth Z., Zsók V., Serrarens, P., Plasmeijer, R.: Parallel Elementwise Processable Functions in Concurrent Clean, to appear in *Computers & Mathematics with Applications*, Elsevier.

[7] Horváth Z., Varga Z., Zsók V.: Clean-CORBA Interface for Parallel Functional Programming on Clusters. To appear in: Proceedings of SPLST'03.

[8] Huch, F., Norbisrath, U.: Distributed Programming in Haskell with Ports, *Implementation of Functional Programming Languages, 12th International Workshop, IFL2000*, Aachen, Germany, September 4-7, 2000, LNCS, Vol. 2011, pp. 107-121, Springer 2001, http://www-i2.informatik.rwth-aachen.de/hutch/distributedHaskell.

[9] Loidl, H.W., Klusik, U., Hammond, K., Loogen, R., Trinder, P.W.: GpH and Eden: Comparing Two Parallel Functional Languages on a Beowulf Cluster in Gilmore, S. (ed.): *Trends in Functional Programming*, Vol. 2, pp. 39-52, Intellect, 2001.

[10] Rishiyur S. Nikhil, Arvind: *Implicit Parallel Programming in pH*, Morgan Kaufmann, 2001.

[11] Serrarens, P.R.: *Communication Issues in Distributed Functional Computing*, PhD Thesis, Catholic University of Nijmegen, 2001.

[12] Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.J.: Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, Vol. 8, No. 1, pp. 23-60, 1998.

[13] Varga Z.: Clean-CORBA Interface, Master thesis, University of Eötvös Loránd, Budapest, 2000. (Supervisor: Horváth Z.)