OBJECT-ORIENTED MODELING OF PARALLEL PDE SOLVERS*

Michael Thuné, Krister Åhlander†, Malin Ljungberg, Markus Nordén, Kurt Otto, Jarmo Rantakokko
Uppsala University
Uppsala, Sweden

Abstract This is a status report of a long-term research effort focusing on object-oriented modeling of parallel PDE solvers, based on finite difference methods on composite, structured grids. Two previous results of this effort are reviewed, the class libraries *Cogito* and *Compose*. Cogito is implemented in Fortran 90, with MPI for the message passing, and provides abstract data types for parallel composite-grid methods. Compose is in C++ and allows for fully object-oriented construction of PDE solvers by composition of objects. The object model behind Compose is described, and some research issues related to the refinement of the model are outlined. Finally, some recent results are presented, which are initial steps in addressing these issues.

Keywords: object-oriented, parallel, PDEs, Fortran, C++, framework

1. INTRODUCTION

The traditional programming style in scientific computing is procedural, plain Fortran. This leads to very efficient programs, but the process of *constructing* the programs is time-consuming and error-prone. The latter drawback has become increasingly accentuated as both the computer architectures and the scientific applications have become more complex. In fact, the lack of adequate software tools has been regarded as a serious impediment to the realization of the strategic potential of high performance scientific computing [10]. Consequently, during the last decade there has been a large number of research projects focusing on software issues in this field.

*The research was supported by the Swedish Research Council for Engineering Sciences, and by Uppsala University via a faculty grant.
†Currently at the Dept. of Informatics, University of Bergen, Norway

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35407-1_22
R. F. Boisvert et al. (eds.), *The Architecture of Scientific Software*

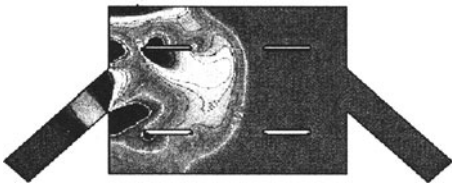

Figure 1 Simulation of airflow through an expansion pipe.

Many of the most successful projects have used an *object-oriented* approach. By now, the advantages of object-oriented scientific programming are widely recognized. The *feasability* of the approach was demonstrated in early projects, which consisted in enriching the programming language with abstract data types (ADTs) suitable for scientific computing, see, e.g., [16], [26]. The overall programming style was still procedural. In a second phase came projects aiming for a *fully* object-oriented approach, where the main program essentially vanishes, see, e.g., [4, and the references therein], and [6]. The role of the main program reduces to creating objects and eventually activating one object, which subsequently activates other objects. The actual algorithm consists in interactions between objects.

Our research group has been part of this development. We focus on the numerical solution of partial differential equations (PDEs). The major goals are:

1 to construct complete PDE solvers via composition of objects;

2 that this should not be restricted to predefined numerical operators, i.e., that new numerical methods could be implemented via composition of objects;

3 that this way of constructing PDE solvers should be applicable to scientific and industrial problems on parallel computer platforms.

Primarily we consider finite difference methods. In PDE solvers based on finite difference approximations, complicated geometries are handled via the insertion of composite, structured grids, for example multiblock grids. Such a grid is a collection of structured grids, the union of which covers the geometry at hand. To represent a grid function on the entire composite grid, the grid functions on the different element grids are tied together through interpolation at the grid boundaries. In a parallel computing context, this interpolation can lead to communication between processors.

Fig. 1 shows a simplified model problem to illustrate this. It is a simulation of airflow through an expansion pipe, and the geometry re-

quires a five-block grid. The simulation is based on the compressible Navier–Stokes equations. In the following, this application will be used for illustration.

The present paper gives a status report of our work. We begin by reviewing the results of our previous software projects *Cogito* [25] and *Compose* [2]. Both of them are class libraries, to be used for constructing PDE solvers. Cogito represents the ADT approach, whereas Compose is fully object-oriented (in the sense discussed above). Our current research aims at elaborating the object model of Compose. We discuss some research issues in that context, and present some recent results.

2. REVIEW OF PREVIOUS RESULTS

2.1. COGITO: ABSTRACT DATA TYPES FOR COMPOSITE GRIDS

For the problems we are considering, with complicated data structures, and the additional aspect of parallelization, the traditional way of constructing a PDE solver (from scratch, in Fortran) is inadequate. There is an apparent need for software tools that raise the level of abstraction considerably.

To this end, we developed Cogito [25], a Fortran 90 library supporting implementation of parallel PDE solvers. Cogito has an object-oriented design. The core classes are *Grid*, *Composite Grid*, and *Grid Function*. Each individual instance of these classes is automatically distributed over several processors. The parallelism is of SPMD type, and uses message passing via MPI. The message passing takes place within the object and is invisible to the user.

We note in passing that although Fortran 90 is not an object-oriented language, it allows for an object-oriented *style* of implementation, via its mechanisms for modularization, data abstraction, and dynamic memory allocation. This has been noted by several authors, and [8] gives an exposition of "object-oriented" Fortran 90 programming.

The *data partitioning* in Cogito is handled by a data partitioning module with an object-oriented design [22]. It is based on our own framework for partitioning of composite grids [23]. Within this framework, it is possible to implement a wide range of specific partitioning algorithms. Fig. 2 shows two examples of partitionings for the multiblock grid of the model problem discussed above. To the left is a straightforward approach, where each block is partitioned into rectangles, one per processor. Thus, each processor will get one part of each block. The alternative shown to the right in Fig. 2 is the result of a more sophisticated approach, which combines structured and unstructured parti-

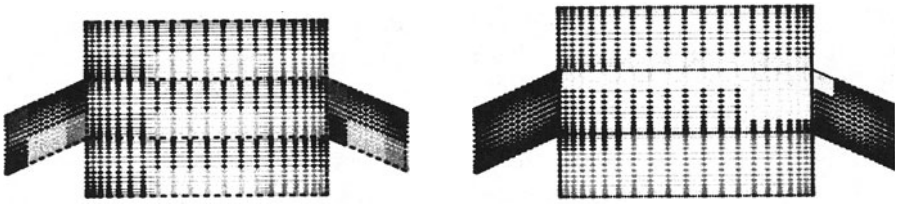

Figure 2 Example of two ways of partitioning and mapping the multiblock grid for the model problem in Fig. 1. Both of these alternatives, as well as many others, can be expressed in the partitioning framework contained in Cogito.

```
! Initiation
   call Create_CG(cg,'pipe.dat','rectangle')
   call Create_GF(u,'u',4,cg)
   ...
! Runge-Kutta time marching
   do s = 1, nstages
     call spdisc(v, R)
     call Saxpy_GF(v, a(s), R, u)
     call Interpolate_GF(v, 'pre')
     call bound(v)
     call Interpolate_GF(v, 'post')
   end do
```

Figure 3 Code example with calls to Cogito. For explanations, see the text. The overall style of programming is procedural. The program is parallel. The user specifies what strategy to use for partitioning and distributing the grid. Apart from this, all the parallelism is hidden to the user.

tioning techniques, giving *one* connected subdomain to each processor, thus reducing the communication. Both of these partitionings, as well as many others, can be expressed in the partitioning framework which is contained in Cogito.

The present classes in Cogito are essentially data stores with associated operations. They do not initiate interaction with other objects. Thus, the overall style of programming is procedural. In a PDE solver based on Cogito, the numerical method is expressed as operations on grid function objects. Fig. 3 shows, as an example, the initiation of some objects, and a subsequent code sequence expressing the Runge–Kutta time marching scheme used in our 2D compressible Navier–Stokes solver. We use a naming convention, such that the Fortran routine implementing the operation B on class A gets the name B_A. (In order to avoid extremely long names, we use standardized abbreviations of the class names.)

It is assumed that the grid has been generated by a separate grid generator, and is stored in a file. In Fig. 3, *Create_CG* initiates a composite grid object, reading the grid data from the file `pipe.dat`. The grid is to be partitioned into rectangles (see Fig. 2, left-hand side) and these are distributed over the processors. Next, *Create_GF* creates a grid function u on the composite grid. This grid function will represent the numerical solution of the 2D compressible Navier–Stokes equations, and consequently it has four components per grid point. The grid function is automatically distributed in the same way as the corresponding grid. Subsequent operations on the grid function are carried out on the entire grid, and the message passing is hidden from the user.

The second code sequence in Fig. 3 implements a Runge–Kutta time marching scheme, which computes a number of Runge–Kutta stages $v^{(s)} = u^n + a(s)R(v^{(s-1)})$. Here, R is the discretized right-hand side of the compressible Navier–Stokes equations. The discretized space derivatives are computed inside the subroutine `spdisc`, which is supplied by the user. This subroutine, as well, is implemented via calls to operations on Grid Function objects. The *Saxpy_GF* operation computes `u + a(s)*R`, where `u` and `R` are grid functions, and the result is stored in the grid function `v`.

2.2. COMPOSE: OBJECT-ORIENTED COMPOSITION OF PDE SOLVERS

Writing a PDE solver based on Cogito relieves the programmer of a considerable amount of low level details, concerning the data structures and the parallelization. However, the numerical method still has to be hand coded, as a sequence of operations on grid function objects. Moreover, the coupling between the numerical method and the PDE problem remains, which leads to limited reusability of the code. In the code example above (Fig. 3), the Runge–Kutta code segment is reusable, but the user-supplied subroutine `spdisc` is specific for a certain discrete approximation of a certain PDE problem. Thus, if we wish to address the same equations with a different approximation, or apply the same approximation to another set of equations, the subroutine `spdisc` has to be rewritten from scratch.

In order to increase the reusability of the code, we developed Compose [2]. The goal was to allow for "component-based" construction of PDE solvers. The approach remained object-oriented, so the "components" were to be objects.

The object-oriented framework Compose is implemented in C++ and contains classes representing the mathematical equations, boundary con-

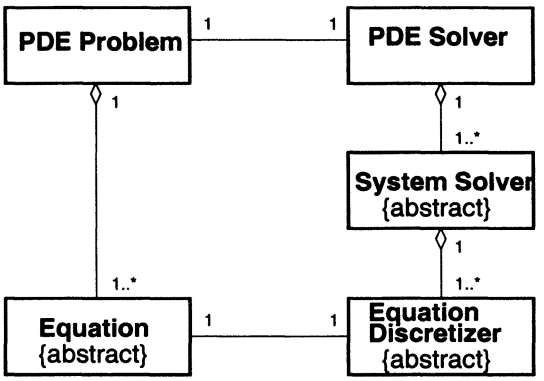

Figure 4 The key classes of Compose.

ditions, etc., as well as various aspects of the numerical methods. Such objects can be composed into a complete PDE solver, which thus becomes fully object-oriented in the sense discussed above.

The Compose project emphasized the object-oriented analysis, and the resulting object model was a main result of the project. This model can be regarded as a general framework for the construction of PDE solvers. The key classes on the uppermost level of abstraction are shown in Fig. 4. On this level of abstraction, the model applies to a variety of PDE solvers, with different underlying numerical approaches. Our *implementation* of the model, however, focusses on finite difference methods on composite, structured grids.

The Compose object model distinguishes between the *PDE Problem* and the *PDE Solver*. There is an association between the two, reflecting that a PDE solver is associated to the problem it is to solve. The PDE Problem is an aggregate of *Equations*, and the PDE Solver is an aggregate of *System Solvers*, each of which is an aggregate of *Equation Discretizers*. The *Equation Discretizer* is associated to an *Equation*.

Fig. 4 does not show the lower level classes Grid, Grid Function, etc. However, such classes are present in Compose as well, and provides a supporting lower layer. In our implementation of the Compose object model, Overture [5] serves this purpose. Overture is a C++ library similar to Cogito. In Overture, the discrete grid function has differential operators. For example, `u.x()` represents the differentiation of u with respect to x. This notation is used to express the PDE problem. However, the actual *computation* of the differential operators is done via discrete approximations. Overture has a class that represents a *package of discrete space operators*. When the grid function is initiated, it

gets associated to such a package, which then specifies *which* discrete approximations to use for the various derivatives.

We now explain the dynamic behavior of the Compose object model for the case of an explicit time marching method, as in the example of our 2D compressible Navier–Stokes solver above. In Compose, the compressible Navier–Stokes equations would be an inheritor of the Equation class. The PDE Solver would have an Explicit System Solver (an inheritor of System Solver). The space operator package object is an argument to the System Solver constructor. Thus, the Explicit System Solver will be able to establish the connection between the solution (grid function) and the discrete space operators. When the Explicit System Solver is to advance the solution to the next time level, it tells the Runge–Kutta Discretizer to update the solution. This particular equation discretizer, an inheritor of the base class Equation Discretizer, knows what time marching algorithm to use, but it tells the equation object to compute the right-hand side (cf. `R` in Fig. 3). The equation object then applies the various differential operators (and other operators) occurring in its right-hand side. The solution (an instance of Grid Function), via its associated space operator package, knows what *discrete approximations* to use for these derivatives.

In the case of an implicit time marching method, the dynamic behavior is similar, but more complicated. Then, the Equation Discretizer contributes to the construction of an algebraic system. The various contributions from different equation discretizers are assembled by the Implicit System Solver, which subsequently solves the system.

It should be noted that each of the classes Equation, System Solver, and Equation Discretizer is the abstract base class of an *inheritance hierarchy.* As an example, the Equation hierarchy has a first level of inheritors representing various *kinds* of equations. The actual equations are on the second level of the inheritance tree. Not only the partial differential equations are represented as equation objects, but also the boundary and initial conditions.

The Compose model has been demonstrated to work in practice, for example in the case of the 2D *incompressible* Navier–Stokes equations [1]. In the Compose-based implementation of a solver for these equations, the PDEs (a system of two convection-diffusion equations for the velocities, and an elliptic equation for the pressure) are represented as independent objects, as are the initial conditions and various boundary conditions needed. Moreover, the elliptic equation for the pressure could reuse an existing class for the Poisson equation, and the solver for that equation. However, since the solver objects are separate from the equation objects, we could as well reuse the equation only, and connect it to a new solver.

This distinguishes Compose from Diffpack [4, Chapter 11], which also has a fully object-oriented structure, but does not handle the equations as independent objects. Another software library with similar scope, ELEMD [4, Chapter 4], has separated the equations, and also includes a class Equation Discretizer. There are differences in details, mainly due to the fact that ELEMD emphasizes finite element methods, whereas Compose focusses on finite difference methods.

Another related effort is POOMA [6]. There is no apparent counterpart to Compose in the five-layer model of POOMA. The top layer, the application layer, differs between applications, and there is no general model for composing applications, whereas this is precisely where Compose has its focus.

Finally, Compose has a built-in support for code validation and monitoring. The equation classes include operations that allow for testing the equations with known solutions (via the technique of forcing) [2]. This, in turn, is based on support in Overture for this kind of testing. Moreover, Compose includes the concept of *Monitor* classes, which can be used for computational steering [2]. None of these features seem to be available in the related work discussed above.

3. FINE-GRAINED MODELING OF NUMERICAL OPERATORS

3.1. OVERVIEW

With respect to the goals stated in § 1, the prototype implementations of Compose and Cogito address the first and third goals. In our ongoing work we are exploring different directions for extending and improving the Compose model in view of the second goal. That is, the aim is to make it possible to construct numerical operators incrementally, starting out with a set of basic objects. In this way, the user will be able to design new numerical algorithms *within* the object-oriented framwork, without having to fall back on "low level" programming in C++ or Fortran 90.

Primarily, we are focusing on the following types of operators.

- *Stencils.* We have recently equipped Cogito with a Stencil class, which allows for incremental construction of stencil operators.

- *Coordinate invariant operators.* In the same spirit as a group at Bergen University [3], with which we are interacting, we aim at distinguishing between the mathematical formulation of the differential equations and the actual evaluation in a specific coordinate system. Since the equations occurring in applications can often be expressed in terms of coordinate invariant operators (gradient,

divergence, curl, etc.), software support for such operators would increase the reusability of the software. As an example, a problem with a cylindrical geometry may then be simplified to a 2D problem without changing the equations.

The support for coordinate invariant operators is also appealing in the context of *curvilinear* structured grids, where the actual computations take place on a rectangular grid. Then, there is by necessity a coordinate transformation involved, between the computational grid and the physical grid. This calls for a reformulation of the PDE, involving the metric tensor [7, p. 68] of the mapping. However, if the equations are expressed in terms of coordinate invariant operators, *and* if such operators are available in the object-oriented software library, then the reformulation of the equations can be avoided.

The expression for a coordinate invariant operator in a specific coordinate system includes derivatives of various quantities. For a user who wishes to fine-tune the algorithm, it is desirable to be able to decide precisely *how* these derivatives are going to be approximated in the discretized operator. Consequently, we want to allow for *composition* of coordinate invariant operators, using basic building blocks such as difference stencils.

It can be noted that Overture [5], as well, addresses the issue of automatizing the mapping between the computational and physical coordinate systems in the case of curvilinear grids. However, they do not support coordinate invariant operators in the way discussed above. Neither do they allow for "component-based" design of operators, as envisaged here.

- *General difference operators.* A natural extension of these ideas would be to allow for incremental construction of general difference operators. For example, the complete right-hand side of a difference method for a nonlinear PDE could be expressed as a single difference operator.

- *Preconditioners.* The idea of fine-grained modeling also carries over to other types of operators. Preconditioners is an example. So far preconditioners have been regarded as atomic units in object modeling. In a pilot study [18], we went beyond that limit, in that we presented a way of constructing a certain family of preconditioners from "smaller" objects. We are currently generalizing these ideas.

Remark: The fine-grained decoupling of operators that we are aiming for is motivated by the needs of algorithm developers. Many users do not require this flexibility, but are satisfied with using a standard variant of an operator. For them, we envisage that a library of predefined operator objects will be available.

In the following, we discuss in more detail two cases of fine-grained modeling of numerical operators: stencils and preconditioners.

3.2. GENERALIZED STENCIL OPERATORS

We have recently extended Cogito with stencil operators [19]. Objects of the class *Stencil* can be created as combinations of "smaller" objects of the same class. There is a library of basic objects, representing the identity operator, the shift operators in different space directions, and the standard difference approximations of the first derivative. By operations such as composition of stencils, multiplication of stencils with coefficients (scalar or matrix), etc., complex stencils can be built. This makes it possible for the user to design new stencil operations within the framework, and to store them in the library for subsequent reuse.

As an additional benefit, the ability to collect a large number of arithmetic operations into a single stencil operation leads to improved cache utilization. In our experiments, on a Sun Wildfire parallel platform, we observed reductions of 50% in execution time when the new stencil class was introduced [19]. This is explained partly by the cache effects, partly by additional code restructuring that helped the Fortran 90 compiler do a better job.

The stencils are "generalized" in the sense that a single stencil object can act on several components of a grid function, and can have different actions on different components. Moreover, the number of components in the *result* can be different from the number of components of the operand. Thus, in general, we allow for stencils where the coefficients are rectangular matrices.

When a stencil is going to be used, it is *connected* to a grid function. Internally, in the grid function object, the stencil is then stored in a table, and persistent MPI objects for the communication are set up. The actual application of the stencil takes place via subsequent calls to *Apply Stencil*, which is an operation of the class *Grid Function*. By locating the application *inside* the grid function object, the internal data structures of that object can be accessed directly, which is important for efficient execution of the stencil operation.

The idea of a stencil class is not new. The point of our particular design is the support for incremental construction of stencils. POOMA

[13] has a stencil class that allows for efficient application of stencils, but the design of new stencils requires C++ coding. Karpovich et al. [14] have no separate stencil representation, but provide classes for applying a sequence of stencils to a matrix.

3.3. PRECONDITIONERS BASED ON FAST TRANSFORMS

As a second example of fine-grained modeling of numerical operators, we discuss preconditioners based on orthogonal transforms, also known as *normal block preconditioners* [11],[20]. This is linked to other research activities at our department, where the aim is to design new normal block preconditioners for discretizations of systems of PDEs. By modeling the preconditioners as described in the following, we will provide these colleagues with a laboratory in which they can conveniently compose new preconditioners and experiment with them. The savings, in terms of human efforts, will be huge, and with careful implementation the additional overhead in execution time will be negligible.

The construction of a normal block preconditioner for the discrete system $Bu = g$ goes as follows:

- Select a set of discrete transforms, and decide which transform to apply in what space direction.

- Form a normal block operator M with the same block structure as B, but where the blocks at one or more levels of M are diagonalizable by the transform matrices, and where M is the best approximation to B measured in the Frobenius norm [20].

The subsequent application of a normal block preconditioner can be efficiently implemented in parallel, in terms of fast transforms and solution of narrow-banded systems [15].

Traditionally, the system $Bu = g$ is interpreted as a linear algebraic system with coefficient matrix B, and with u and g as vectors. However, since u and g are actually grid functions with four indices each—one for the components and one for each space direction—B is a *tensor* [7] with four upper indices ("row" indices) and four lower indices ("column" indices). It is largely the underlying grid that determines the block structure of the tensor B. It would be obscured by a conversion to matrix form. Consequently, the construction and application of the normal block preconditioner is much more conveniently expressed if B is maintained in its original tensor form [20]. This is a motivation for introducing the tensor as a new basic data type in Cogito.

In our tentative object model for the construction of normal block preconditioners, the *Normal Block Solver* constructs the preconditioner, *and* knows how to carry out subsequent preconditioner solve operations. The construction of the normal block operator M is based on the *Band Tensor* B, and on a *Polytransform*, which is an ordered set of *Transform* objects. The resulting preconditioner is also a band tensor, and is tightly connected to the normal block solver. Each transform is a discrete trigonometric transform, which is associated with the discrete Fourier transform, which is further associated to the radix-2 fast Fourier transform.

We have recently made a serial pilot implementation of this model, in Fortran 90. It contains seven orthonormal and three nonorthonormal transforms. Moreover, it has operations such as applying a polytransform to a grid function, computing the inner product between a band tensor and a grid function, and applying the normal block solver. The serial code will be used for validation (and possibly modification) of the model, before we go on to a parallel implementation. Note, that these implementations do not begin from scratch, but can build on our previous experiences [12, 21, 15].

4. MIXED C++/F90 IMPLEMENTATION OF FLOW SOLVERS

The revised object model we aim for is to be implemented on the basis of our previous software Cogito (Fortran 90) and Compose (C++). The intention is to develop a flexible framework for construction of parallel PDE solvers using the object-oriented capabilities of C++, which will execute with high parallel efficiency via the Fortran 90 components of Cogito.

As a first step in this direction, we have made a C++ embedding of the "object-based" Fortran 90/MPI version of Cogito [17]. We have demonstrated for a scalar advection problem in 2D that the C++/Fortran 90 version gives almost exactly the same execution time as the pure Fortran 90 code.

For further validation of the mixed-language approach, we reimplemented the 2D compressible Navier–Stokes code by calling the new, mixed language version of Cogito. In this case as well, there was a negligible difference in execution time between the pure Fortran 90 and the C++/Fortran 90 version of the Navier–Stokes solver. In addition, we measured scalability, in terms of sizeup [24] for the two versions of the Navier–Stokes code. The two codes show (almost) identical behavior.

The maximum discrepancy is 4.5% and there is no trend of an increasing discrepancy as the number of processors goes up [17].

Using calls to Fortran from within C++ classes is relatively straightforward. Our situation is more complicated, since we wrap C++ around an *object-based* Fortran 90 code. This problem was discussed in [9]. Our approach [17] is similar, but avoids one of the steps of wrapping. Moreover, we have demonstrated that our approach works in practice for a relatively large object-based Fortran 90 library, i.e., Cogito.

5. CONCLUSIONS

In previous work, we have explored the potential of object-oriented programming in the context of numerical solvers for partial differential equations. First, we demonstrated that an object-oriented style of implementation of parallel PDE solvers in Fortran 90 is feasible, and relieves the programmer of many low-level details. Next, we proposed the object-oriented framework Compose, which allows for fully object-oriented construction of PDE solvers. In Compose, the PDEs are represented as separate objects, independent of the numerical approach to be used. A pilot implementation of Compose, in C++, shows that the model is applicable to realistic application problems, such as the incompressible Navier–Stokes equations on composite, structured grids.

We conclude that it is relevant to continue elaborating the Compose model. In particular, we have discussed the introduction of a more fine-grained modeling of numerical operators, so that complicated operators can be constructed via composition of simpler ones. As a preliminary result in this direction, we described a generalized stencil class that has recently been implemented. Moreover, we discussed the object-oriented modeling of normal block preconditioners. Finally, we presented the ambition to base the next implementation of the C++ library Compose on our "pseudo object-oriented" Fortran 90 library Cogito, which executes efficiently on parallel platforms. The new mixed C++/Fortran 90 version of Cogito promises to be a suitable basis for this development.

References

[1] K. Ahlander, *An extendable PDE solver with reusable components*, in Computational Technologies for fluid/thermal/structural/chemical systems with industrial applications, V. V. Kudriavtsev and C. R. Kleijn, eds., vol. 397-1, New York, 1999, ASME, pp. 39–46.

[2] K. Ahlander, *An object-oriented framework for PDE solvers*, PhD thesis, Uppsala University, Information Technology, Dept. of Scien-

tific Computing, Uppsala, Sweden, 1999.

[3] K. ÅHLANDER, M. HAVERAAEN, AND H. MUNTHE-KAAS, *On the role of mathematical abstractions for scientific computing.* Proceedings of the IFIP WG 2.5 Working Conference on Software Architectures for Scientific Computing Applications, Kluwer, 2000.

[4] E. ARGE, A. M. BRUASET, AND H. P. LANGTANGEN, eds., *Modern Software Tools for Scientific Computing*, Birkhäuser, 1997.

[5] D. L. BROWN ET AL., *Overture: An object-oriented framework for solving partial differential equations on overlapping grids*, in Object Oriented Methods for Interoperable Scientific and Engineering Computing, M. E. Henderson, C. R. Anderson, and S. L. Lyons, eds., SIAM, Philadelphia, 1999, ch. 26.

[6] J. C. CUMMINGS ET AL., *Rapid application development and enhanced code interoperability using the POOMA framework*, in Object Oriented Methods for Interoperable Scientific and Engineering Computing, M. E. Henderson, C. R. Anderson, and S. L. Lyons, eds., SIAM, Philadelphia, 1999, ch. 29.

[7] D. A. DANIELSON, *Vectors and Tensors in Engineering and Physics*, Addison–Wesley, Reading, MA, 2nd ed., 1997.

[8] V. K. DECYK ET AL., *How to express C++ concepts in Fortran 90*, Scientific Programming, 6 (1997), pp. 363–390.

[9] M. G. GRAY ET AL., *Shadow-object interface between Fortran 95 and C++*, Computers in Science and Engineering, 1 (1999), pp. 63–70.

[10] A. H. HAYES, *The changing face of high-performance computing in the United States*, Wuhan Journal of Natural Sciences, 1 (1996), pp. 420–429. Keynote Lecture at the International Conference on Parallel Algorithms, Wuhan University, Wuhan, P. R. China, October 1995.

[11] S. HOLMGREN AND K. OTTO, *A framework for polynomial preconditioners based on fast transforms I: Theory*, BIT, 38 (1998), pp. 544–559.

[12] ——, *A framework for polynomial preconditioners based on fast transforms II: PDE applications*, BIT, 38 (1998), pp. 721–736.

[13] S. KARMESIN ET AL., *Array design and expression evaluation in POOMA II*, in Proceedings of ISCOPE'98, Lecture Notes in Computer Science, Vol. 1505, D. Caromel, R. Oldehoeft, and M. Tholburn, eds., Berlin, 1998, Springer-Verlag.

[14] J. F. KARPOVICH ET AL., *A parallel object-oriented framework for stencil algorithms*, in Proceedings of the Second International Sym-

posium on High-Performance Distributed Computing, 1993, pp. 34–41.

[15] E. LARSSON AND S. HOLMGREN, *A parallel domain decomposition method for the Helmholtz equation*, Tech. Rep. 2000-006, Dept. of Information Technology, Uppsala Univ., Uppsala, Sweden, 2000.

[16] M. LEMKE AND D. QUINLAN, *P++, a parallel C++ array class library for architecture-independent development of structured grid applications*, ACM SIGPLAN Notes, 28 (1993), pp. 21–23.

[17] M. LJUNGBERG AND M. THUNÉ, *Mixed C++/Fortran 90 implementation of parallel flow solvers.* Accepted for publication in the proceedings of Parallel CFD 2000.

[18] E. MOSSBERG, K. OTTO, AND M. THUNÉ, *Object-oriented software tools for the construction of preconditioners*, Sci. Programming, 6 (1997), pp. 285–295.

[19] M. NORDÉN, *Stencils for parallel object-oriented PDE solvers*, Master's thesis, Uppsala University School of Engineering, 2000. Report No. UPTEC F 00 067.

[20] K. OTTO, *A tensor framework for preconditioners based on fast transforms.* Manuscript.

[21] K. OTTO AND E. LARSSON, *Iterative solution of the Helmholtz equation by a second-order method*, SIAM J. Matrix Anal. Appl., 21 (1999), pp. 209–229.

[22] J. RANTAKOKKO, *Software tools for partitioning of block-structured applications*, in Proceedings of ISCOPE'98, Lecture Notes in Computer Science, Vol. 1505, D. Caromel, R. Oldehoeft, and M. Tholburn, eds., Berlin, 1998, Springer-Verlag.

[23] ——, *Partitioning strategies for structured multiblock grids*, Parallel Computing, 26 (2000), pp. 1161–1680.

[24] X. SUN AND J. L. GUSTAFSON, *Towards a better parallel performance metric*, Parallel Computing, 17 (1991), pp. 1093–1109.

[25] M. THUNÉ ET AL., *Object-oriented construction of parallel PDE solvers*, in Modern Software Tools for Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhäuser, Boston, MA, 1997, pp. 203–226.

[26] R. D. WILLIAMS, *DIME++: A language for parallel PDE solvers*, Tech. Rep. CCSF-29-92, Caltech, Pasadena, CA, 1993.

DISCUSSION

Speaker: M. Thuné

Fred Gustavson : Did you use parallel narrow band solvers in your implementation?

M. Thuné : The pilot implementation of the Normal Block Solver and related classes is serial, but of course we intend to carry on this work with a parallel implementation. As I mentioned, Cogito is parallel. Moreover my co-author Kurt Otto and his colleagues have had for many years a non-object-oriented implementation of parallel preconditioners of this type.