C++ Code Design for Multi-Purpose Explicit Finite Volume Methods: Requirements and Solutions

Nadejda Kirchner*, Oliver Herzog, Sascha Knell, Volkmar Holzwarth, Udo Ziegenhagel, Arno Klomfass

EMI, Fraunhofer Institute, Eckerstr. 4, 79104 Freiburg, Germany

Nadejda.Kirchner@emi.fraunhofer.de

Abstract: The Ernst-Mach-Institute (EMI) of the Fraunhofer-Society is dealing with a wide spectrum of subjects in the fields of applied physics, mechanical and civil engineering. The EMI department for numerical simulation supports the institute and the external customers with the high-performance software applications in the fields of compressible flows, structural dynamics, electro dynamics and multi-disciplinary couplings of these applications. A majority of the in-house codes is written in FORTRAN 95 for efficiency reasons. Nevertheless, potential benefits of object-oriented programming in C++ are recognized. Performance studies for numerical simulations in terms of explicit finite element methods have shown that FORTRAN provides much better efficiency than C++. Here we analyze the factors contributing to the code performance for the explicit finite volume scheme and present pro and contras of possible C++ solutions.

Introduction

The numerical solution of partial differential equations (PDEs) requires spatial and temporal discretization with subsequent solution of a large algebraic system of equations. Solution algorithms based on the cell-centered finite volume method (FVM) has recently been the subject of considerable research in field of computational fluid dynamics, because the FV-numerical model can be directly derived from integral form of conservative equation and designed for quite various grids: structured and unstructured, Cartesian and body-fitted (curvilinear), stationary and moving/deforming. Finite volume methods are traditionally divided in two groups: explicit and implicit ones, according to the way of discretization used for the time derivative. Implicit schemes offer numerical stability at the extra cost of having to deal with the resolution of an algebraic system with as many unknowns as grid cells at any time step requiring inversion of a large scale matrix spread over the whole grid. The principal idea of explicit FVM (eFVM) is evaluation of characteristic variables at a future time in every single gird cell in terms of these variables known at the former time step. Here no matrix inversion is needed, but a smaller time step is required. The allowed time step size of explicit FVM is restricted by stability reasons to fulfill the Courant-Friederichs-Levy (CFL) condition. Contrarily to implicit schemas, the explicit schemes tend to require fewer computations per time step, but the time saved on a per-time-step basis can be lost on per-simulation-period basis because the time step is constrained by CFL condition. For further details we refer the reader to [Ref. 1-3].

Multi-purpose solver: requirements to implementation

The advantages of object-oriented programming for implementation of the explicit FVM have been especially revealed by the need to develop a multi-purpose solver. The multi-purpose solution assumes an ensemble playing of distinct physical models specified on the system of complex geometry within one solution phase. Thus, the multi-purpose solver is a

software application with implemented heterogeneous physical model spread over the macrogrid based on non-uniform fine meshes. Thus, the architecture and the choice of the underlying data structures are constrained by complete physics and topology of the system. Thereby the mathematical formulation of each single physical model is based on the same flow equation written for a volume domain V with boundary surface A in an integral form as following:

$$\frac{D}{Dt}\int_V U\,dt = \oint_A F\vec{n}\,dA + \int_V S dV \qquad (1)$$

Here $\frac{D}{Dt}$ is material time derivative, U is solution variable in conservative form, $\vec{n}$ is outgoing unit normal vector, F is a total flux through the domain surface A, and S is the source term. As follows from Eqn. (1), flexible and easy definition of each physical model is possible through specification of U, F and S. Design of the grid consisting of zones with quite different U, F and S definitions is required for development of the multi-purpose solver. In addition, spacing of the macrogrid determines the accuracy and the cost of calculations. Diffizil grid areas (caused by shocks, discontinuities etc.) require fine spacing that cannot be determined a priori and trails the system evolution in space and time (i.e., the adaptive mesh refinement). Thus, complex, non-uniform, dynamic data structures are needed for development of the multi-purpose solver based on explicit finite volume scheme, which are difficult to realize in FORTAN with its default rigid data formats.

Example of explicit FV: numerical model of the Fourier heat conduction low

We consider the Fourier heat conduction law as a simple illustrative example to find the most efficient C++ solution for simulation of spatially 3D transient heat flow in terms of the explicit FVM. The integral form of the flow equation Eqn. (1) contains no source term and its cell centered numerical model can be written as Eqn.(2) for unstructured grid of – for example – hexahedron type in terms of the explicit FV scheme:

$$U_i^{n+1} = U_i^n + \frac{dt}{V_i}\sum_{l=1}^{6} F_{l,i} * a_{l,i} * s_{l,i} \quad (2)$$

Here i is the cell index, n indicates the time step number, U is a volume specific total energy calculated as $U = c * \varrho * T$ with c, ϱ, T being the cell centered specific heat, density and temperature, respectively. In Eqn. (2) dt is the CFL restricted global time step, $F_{l,i}$ represents l-component of the fluxes normal to the cell face of area $a_{l,i}$ surrounding i –cell given by:

$$F_{l,i} = F\vec{n} = -k\,grad\,T\,\vec{n} = -\frac{k_R + k_L}{2}\,\frac{T_R - T_L}{h_{RL}} \qquad (3)$$

$$s_{l,i} = \begin{cases} -1, & incoming\ \overrightarrow{n_{l,i}} \\ 1, & outgoing\ \overrightarrow{n_{l,i}} \end{cases} \quad (4)$$

Parameters of Eqn. (3) are the properties of two adjacent grid cells (R and L staying for right and left, respectively) that share l-face of i –cell. Here k denotes the heat conduction coefficient and h_{RL} is a distance between two adjacent cells. The boundary conditions are defined by fixed value either of flux or temperature.

Multi-purpose solver: C++ code organization

Despite of the obvious simplicity of Eqn (2), it puts a challenging task for C++ in the performance competition with FORTRAN even in terms of generic programming paradigm [Ref. 4-6]. The origin of the problem is the dominant role of *access* to the object properties (even by using of STL iterators) over the efficiency of the *computation algorithm* itself reaching the same or better performance as FORTRAN by using of generic support [Ref. 7]. To show it, we present below our C++ implementation in details.

A 3D unstructured grid of hexahedron type is produced by in–house mesh generator that records the connectivity relations between the indices of the grid entities (in form of cell–adjacent faces–adjacent nodes and face–adjacent cells– adjacent nodes) as well as the node positions into external file. Doing so, we avoid the need to parallelize the grid connectivity matrix by cluster computing and simply prepare a portion of the mesh for calculations on each single CPU. The grid part for one computational node includes usually 10^5 –10^6 nodes, 10^6 – 10^7 faces, 10^5 –10^6 cells, which we denote in general as data objects. Each data object possesses a list of properties shown in Table 1 and uses a separate numbering system.

Table 1: Properties of the data objects constituting the numerical grid

CELL	FACE	NODE
- Index - Geometry type - Volume - Center mass - Content - Conserv. Variables - Adjacent faces - Adjacent nodes	- Index - Geometry type - Area - Flux - Centroid - Normal - Adjacent cells - Adjacent nodes	- Index - Position - Velocity

We apply the concept-bounded (static) polymorphism to achieve a better performance and introduce three concepts Cell, Face and Node within the namespace Object in terms of abstract types as Base, Topology, and Physics. The last ones model the concept, i.e. they implement the requirements on data types describing:

– some basic characteristics (e.g. value–, counter–, Cartesian coordinate types etc.);

– distinct topologies (e.g. the underlying data structure for the type cell–adjacent nodes has the length of 8 for the cells of hexahedron type and 4 for the cells of tetrahedron type);

– the sense of the physical problem to be solved (e.g. the internal representation of the solution variable – scalar in the case of heat flow or vector for more complicated problems – as well as constitutive data structures that mimic the cell content like multifluid or embedded in the cell object).

For example the Object::Cell possesses a list of associative types, each determined by a certain modeling type, the interplay of which defines the complete behavior of the concept:

```
typedef typename Base::counter counter_t;
```

```
typedef typename Base::value value_t;

typedef typename Base::coords coords_t;

typedef typename Base::identifier identifier_t;

typedef typename Base::boundary boundary_t;

typedef typename Topology::cell_adjacent_nodes_t AdjacentNodes;

typedef typename Topology::cell_adjacent_faces_t AdjacentFaces;

typedef typename Physics::multifluid_t MultiFluid;

typedef typename Physics::parameter_t Parameters;

typedef typename Physics::conservative_variables_t ConsVar;
```

We achieve the maximal software reuse by implementing of concepts parameterized over these 3 groups of types to represent each grid entity. Thereby, the types encapsulated in the models must fulfill the performance requirement, i.e. they have to provide the *fast access* to the numeric data stored. The specific of explicit FV scheme is caused by the need for multiple accesses (say, about 10^8 times on single CPU) to each single property of the data objects listed above. Here the access process dominates over computations during the solution phase. The situation is opposite to the implicit schemes, where the intensive calculations on large scale numeric structures prevail over the access and where e.g. PETs [Ref. 8], Janus [Ref. 9] or DUNE [Ref. 16] provide an extensive support.

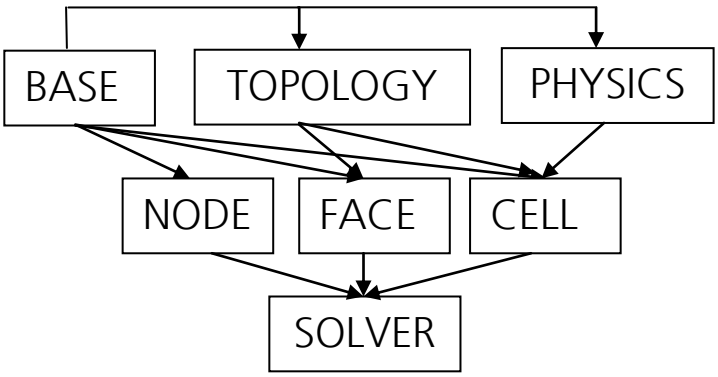
In details, the data object properties for explicit FVM are usually stored in the form of short numeric vectors of a fixed size. For example each grid face of quadrilateral type has 4 adjacent nodes and the corresponding data structure must represent a mathematical vector of 4 elements reflecting the sequence of node indices adjacent to the face. Another example is the Cartesian coordinate type, i.e. a vector of length 3 with the x–, y– and z– components of the grid node position. Thus, the problem is related to selection of the C++ data type for implementation of a short numerical sequence of fixed size. Surprised? Indeed, the standard– and boost C++ provide a wide spectrum of data structures representing a mathematical vector. We summarize in the Table 2 the results of a simple experiment: the clock time is measured, spent by access to – for example – the second element of a numerical vector of size 5 implemented by using of various C++, C and Fortran data structures. This test shows that never mind how intelligent we implement the interaction of the grid data structures during the solution, the FORTRAN code will provide a better efficiency at least of factor 2.76 by implementation of the numeric vectors in terms of C arrays, of factor 3.45 by using of boost::array and of factor 5.45 by application of the boost::fusion::vector of the fixed size. The only vector C++ data structure providing a better performance than FORTAN is the Dune::FieldVector that will be implemented in our future tests.

Another concept SOLVER was introduced for implementation of solution phase of Eqn (2). It is modeled by abstract types CELL, FACE, NODE that can be specified by the types Object::Cell, Object::Face and Object::Node described above. The complete inheritance scheme applied is shown in Figure 1.

Table 2: Comparative performance study of a targeting vector element access. A vector of 5 elements is filled with random numbers. Then we compare the clock time spent by 10^8 for–cycles, each calculating the value equal to the sum of cycle counter (i) and the second vector element. Results are listed in the 4th column (built under bjam with <toolset>, gcc, <optimization>speed: <define>USE_INLINE_ASSEMBLER) and 5th column (build under GNU make). The tests were conducted on 1 x AMD Phenom(tm) 9850 Quad-Core Processor ; 2,5 GHz; 4 GB RAM; gcc version 4.1.2 20071124 (Red Hat 4.1.2-42).

Library	**Implementation**	**Access**	**bjam Time [sec.]**	**g++ –O Time [sec.]**
STL C++	std::vector<double> vec (5);	i+vec[1]	3.23	3.28
	std::vector<double> vec (5);	i+vec.at(1)	7.42	7.62
	std::vector<double> vec (5);	*(vec.begin()+1) + i	2.99	2.84
	std::valarray<double> vec (5);	i+vec[1]	0.97	0.99
	std::valarray<double> vec (5);	i + *(&vec[0]+1)	1.1	1.08
[Ref. 10]	kvector<double, 5> vec;	i + vec[1]	0.9	0.89
[Ref. 16]	**Dune::FieldVector<double, 5> vec;**	i + vec[1]	-	**0.15**
[Ref. 11]	boost::array<double, 5> vec;	i + vec[1]	0.84	0.85
[Ref. 12]	boost::array<double,5>	i+at_c<1>(vec)	1.36	1.37
[Ref. 13]	boost::ublas<double> vec(5);	i + vec[1]	1.74	2
[Ref. 14]	mtl::dense_vector<double> vec(5);	i + vec[1]	2.28	2.35
[Ref. 12]	boost::fusion::vector5<double, double, double, double, double > vec;	i + at_c<1>(vec)	1.33	1.85
C array	double vec[5];	vec[1]	0.68	0.67
[Ref.15]	gsl_vector * v= gsl_vector_alloc(5)	i + gsl_vector_get(v,1)	1.06	-
[Ref. 8]	Vec x; PetscScalar *avec; … VecSetSizes(x,PETSC_DECIDE,5); VecSetType(x,VECSEQ); … VecGetArray(x,&avec);	(PetscScalar)i + avec[1];	-	0.64
			gfortran –o3 Time[sec.]	
Fortran	double dimension(5) :: vec	i +vec(2)	**0.243**	

Figure 1: Concept-bounded polymorphism implemented for solution of the heat flow equation in terms of explicit finite volume scheme.



Generally, the solution phase includes the following steps:

1. The flux update Eqn (3) requires traverse over all faces, access to indices of the cells adjacent to each single face (stored by using of data structure deviated from Topology, e.g. Topology::face_adjacent_cells_t), search for the right/left cell by index and access to the properties of each found cell listed in the Table 1 above.
2. The global time step dt is estimated as a minimum over all local time steps, calculated for each single cells of the numerical grid. This stage requires traverse over all cells with subsequent access to sequence of type Topology::cell_adjacent_faces_t, search of - and access to the properties of each face adjacent to the single cell.
3. Update of the solution variable U according to Eqn. (2) is based on traverse over all cells, traverse over adjacent faces for each cell, access to the cell and face properties.

Thus, another problem of explicit FV scheme is related to the choice of a suitable container to store the whole set of data objects featuring the grid. The container type must support not only the fast search of the data object by index. The algorithms for insert of new-, remove and dispose of the old data objects within the container must be high efficient for adaptive mesh refinement. Finally, the choice of container is restricted by performance of access to the object properties. Table 3 presents comparison of std::hash_map container [Ref. 10] with the boost::intrusive::unordered_set [Ref. 17]. We refer the reader to [Ref. 17] for details about distinctions in intrusive and non-intrusive containers. Our study shows that the boost::intrusive::unordered_set provides a factor 2.45 better performance than the std::hash_map by access to the object properties. It is about 10 times faster by deleting of the objects from container. We implemented boost::intrusive::unordered_set to store the numeric grid (cells, faces, nodes) needed for solution of Eqn(2) with the properties listed in Table 1.

Table 3: Benchmarking of std::hash_map and boost::intrusive::unordered_set containers. For illustration we store the objects that represent a simple cuboid as a class template constructed by cuboid index (that does not coincide with the map key, keeping in mind adaptive mesh refinement) with member functions dx(), dy(),dz() and volume() referring to the cuboid unit length in each dimension and its volume, respectively. The test includes the clock time measurements of the following processes. We initialize 10^6 objects and fill with them the container (1). Then we traverse the container by using of iterator and measure access time to the x-unit length and to the volume of each cuboid stored (2). Finally we delete all objects from container (3). The tests were conducted on 1 x AMD Phenom(tm) 9850 Quad-Core Processor ; 2,5 GHz; 4 GB RAM; gcc version 4.1.2 20071124 (Red Hat 4.1.2-42).

	boost::intrusive::unordered_set, time [sec]	std::hash_map, time [sec]
1	1.07	1.05
2	0.53	1.3
3	0.03	0.25

In conclusion, we developed a static library build under Boost.Build [Ref. 18] for the task managing. Manipulation of the processes for the grid initialization and solution is driven by intrusive pointers of the Boost Smart pointer library [Ref. 19] that provide automatic life time management of the objects.

Results and conclusion

We measured the clock time for complete solution phase at single time step for the following 2 cases: all vectors of fixed size are implemented as a boost::array (1) and as a numbered vector boost::fusion::vector (2), since the Boost Fusion library provides a broad spectrum of high-efficient run time algorithms for the work on sequences. As expected from the analysis above (s. Table 2) our C++ solution is slower than the FORTAN one. The results are presented in Table 4 in comparison with FORTRAN solution.

Table 4: Performance comparison of C++ and Fortran implementations for single simulation period by numerical solution of the heat flow equation in terms of explicit FVM. The tests were conducted on 1 x AMD Phenom(tm) 9850 Quad-Core Processor ; 2,5 GHz; 4 GB RAM; gcc version 4.1.2 20071124 (Red Hat 4.1.2-42).

	C++ Time [sec]	FORTRAN Time [sec]
boost::array	0.25	0.06
boost::fusion::vectorN	0.63	

Generic programming paradigm is an established technique for development of high-performance computing algorithms. Nevertheless, not only the algorithms, but also access must be optimized within the numeric libraries to achieve the at least the same efficiency as in FORTRAN for simulations in terms of explicit finite volume scheme. Our study shows that only the recently discovered DUNE project [Ref. 16] provides the promising for our purpose data structures, which we plan to implement in our further tests.

Acknowledgements

N. Kirchner thanks Dr. P. Gottschling, TU Dresden, for fruitful discussions.

References

1. Versteeg, H.K., Malalasekera, W., „An introduction to computational fluid dynamics: the finite volume method“, Pearson Education Limited, 2007
2. Löhner, R., “Applied computational fluid dynamics techniques: an introduction based on finite element methods”, John Willey & sohns, LTD, 2001
3. Löhner, R., Cebral, J.R., Camelli, F.E.,Appanaboyina, S., Baum, J.D., Mestreau, E., Soto, O., “Adaptive embedded and immersed unstructured grid techniques”, Comp. Methods. Appl. Mech. Engrg.197 (2008) 2173-2197
4. Abrahams, D., Gurtovoy, A., “C++ template metaprogramming: concepts, tools, and techniques from boost and beyond”, Addison-Wesley, 2007
5. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Reis, G, D., Lumsdain, A., “Concepts: Linguistic support for generic programming in C++”, ACM SIGPLAN , 41 (10) 2006, 291–310
6. Gregor, D., Järvi, J., Kulkarni, M., Lumsdain, A., Musser, D., Schupp, S., Generic programming and high-performance libraries, Int. J. Parallel.Prog., 33,2-3, 2005
7. Gottschling, P., Wise, D.,S., Joshi, A., “Generic support of algorithmic and structural recursion for scientific computing” Int.J. Parallel,Emergent, Distributed Systems, 2008
8. Portable, Extensible Toolkit for Scientific Computations, www.mcs.anl.gov/petsc/petsc-as/
9. Gerlach, J., Sato, M.,”Generic Programming for Parallel Mesh Problems”, Lecture Notes in Computer Science, 1732, 108-119, 1999
10. Stephens, R., Diggins, Ch., Turkanis, J, Cogswell, J, „C++ Cookbook Solutions and Examples for C++ Programmers”, O’Reilly, 2005
11. Josuttis, N., Boost Array, http://www.boost.org/doc/libs/1_38_0/doc/html/array.html
12. De Guzman, J., Marsden, D., Schwinger,T., Boost Fusion, http://spirit.sourceforge.net/dl_more/fusion_v2/libs/fusion/doc/html/index.html
13. Walter, J., Koch, M., „Basic Linear Algebra“, http://www.boost.org/doc/libs/1_38_0/libs/numeric/ublas/doc/index.htm
14. Gottschling, P., Lumsdaine, A., „The matrix template library 4”, http://www.osl.iu.edu/research/mtl/mtl4/
15. GNU Scientific library, http://www.gnu.org/software/gsl/
16. Distributed and Unified Numerics Environment, http://www.dune-project.org/index.html
17. Krzikalla, O., Gaztañaga, I., Boost.Intrusive, http://www.boost.org/doc/libs/1_38_0/doc/html/intrusive.html
18. Abrahams, D., Prus, V., Boost.Build, http://www.boost.org/doc/tools/build/index.html
19. Colvin, G., Dawes.B., Adler, D., Boost. Smart Pointers, http://www.boost.org/doc/libs/1_38_0/libs/smart_ptr/smart_ptr.htm