In print, Engineering with Computers (2007)

Generic Programming Techniques for Parallelizing and Extending Procedural Finite Element Programs

Fehmi Cirak[*]
Department of Engineering
University of Cambridge
Cambridge, CB2 1PZ, U.K.

Julian C. Cummings[†]
Center for Advanced Computing Research
California Institute of Technology
Pasadena, CA 91125, U.S.A.

Abstract

We outline an approach for extending procedural finite-element software components using generic programming. A layer of generic software components consisting of C++ containers and algorithms is used for parallelization of the finite-element solver and for solver coupling in multi-physics applications. The advantages of generic programming in connection with finite-element codes are discussed and compared with those of object-oriented programming. The use of the proposed generic programming techniques is demonstrated in a tutorial fashion through basic illustrative examples as well as code excerpts from a large-scale, finite-element program for serial and parallel computing platforms.

Key words: finite elements, parallelization, generic programming, object-oriented programming, multi-physics coupling

[*]fc286@cam.ac.uk, Tel: ++44-1123-332716, Fax: ++44-1123-339713
[†]cummings@cacr.caltech.edu

1 Introduction

Computer simulation of multi-physics problems often requires the simultaneous use of different mathematical models on parallel computing platforms. There are powerful discretization methods and extensive computational kernels available for computing each physical field, such as finite-element software for solid mechanics and finite-volume or finite-difference software for computational fluid dynamics. The availability of coupling techniques for the dissimilar mathematical models and discretization methods, which are concurrently employed on the same physical domain or interface, is clearly crucial. In addition, as the number of utilized software components increases, efficient and robust interfaces between the various solver-specific, high-level data structures gain in importance. Further adding to the complexity, pre-existing solver codes are often written in a procedural style using C or Fortran, with poor encapsulation of data structures. Parallelizing such computational kernels or coupling different solvers for multi-physics simulations based upon the existing data structures can be extremely difficult and error prone. In academic research, moreover, the physical as well as mathematical models are mutable and change relatively frequently (*e.g.*, based on verification and validation or availability of computational resources), which may entail modification of the software.

To address these issues, the use of modern programming languages and software concepts is considered essential. In particular the object-oriented programming (OOP) paradigm has proven to be crucial for software that requires several layers of abstraction, such as graphical user interfaces, visualization tools, and (to some extent) finite-element codes [1, 2]. In scientific research codes, meanwhile, it is widely accepted that the copious use of OOP can lead to severe performance penalties and codes that are difficult to maintain. In response to some of the shortcomings of OOP, generic programming has gained widespread acceptance during the last decade as a complementary programming paradigm, especially in the C++ community. The popularity of generic programming is mainly motivated by the success of the C++ *Standard Template Library* [3, 4], or STL, which provides the language with a standardized set of generic programming components. The collection of *Boost* libraries [5, 6] is also worth mentioning in this context. The Boost libraries started out as an extension to the STL and have been partially integrated into the next version of the C++ standard. There are also an increasing number of high-level software packages in computational science and engineering that successfully exploit the generic programming paradigm, such as CGAL for computational geometry [7], Blitz++ for array operations and finite differences [8], and Getfem++ for generic finite elements [9].

In the work being presented here, our basic approach is to retain existing (non-generic and non-object-oriented) data structures and computational kernels wherever possible and to add high-level container and wrapper classes written in C++. These classes provide improved data encapsulation and a simple means of prototyping new applications. Moreover, this approach allows one to make use of generic container types and algorithms provided by the Standard Template Library. Our code makes extensive use of the STL concepts of *iterator* and *functor* to drastically reduce the complexity of application driver codes. We have constructed a library for finite-element modeling of solid shells that illustrates our approach, along with several serial and parallel sample applications. We have also developed parallel multi-physics applications that couple the shell solver with an explicit parallel fluid solver, in which fluid/solid boundary information is exchanged at each time step.

It is evident that any software design with several abstraction layers (*e.g.*, in the form of C++

class hierarchies) adds complexity to the numerical software and mostly lengthens the learning period for new developers. Thus more advanced designs are only profitable in situations where they are truly warranted. Examples of such scenarios include complex multi-physics applications, non-standard discretization schemes such as subdivision shell elements [10], or the use of adaptive geometry and mesh representations [11].

The remainder of the paper is organized as follows. Section 2 begins with the discussion of the basic object-oriented and generic programming concepts that are important for scientific computing. In a tutorial fashion, we illustrate how such concepts are put into practice with some simple code fragments. The concept of a generic unstructured mesh container and accompanying generic algorithms are introduced in Section 3. Specific attention is given to how existing data structures and algorithms can be interfaced to the previously introduced generic components by using wrapper classes (see Sec. 3.2). In Section 4 the generic abstraction layer is used to implement a basic parallelization layer on top of the serial finite-element solver. Section 5 introduces the *adaptor* technique that we use for simple prototyping and construction of coupled multi-physics applications. Code performance and parallel scaling numbers are given in Section 6 for several high-performance computing platforms, indicating a minimal performance penalty for the use of such generic programming techniques.

2 Review of Software Design Concepts for Scientific Computing

2.1 Object-Oriented Programming

The main focus of object-oriented programming is on data abstraction, data encapsulation, relationships among data, and the design of large-scale systems. OOP is a well-established paradigm and is supported by most modern programming languages. In OOP, software is organized around objects that consist of data structures and methods needed to access and manipulate this data. By encapsulating data within objects and explicitly controlling the means by which it is accessed, the programmer reduces the risk of unforeseen side effects that are a typical cause of errors in complex applications. The OOP approach also provides a natural means of decomposing a complex application into semi-autonomous code components that interact in a well-defined manner.

Let us now briefly review a few essential aspects of object-oriented programming as realized in C++ that are relevant to the present discussion. One key concept of OOP is inheritance, [1] which allows the programmer to create new classes by extending an existing base class. The base class defines the interface, and the derived class provides the implementation specific to that derived class. Importantly, the interface of the base class is always a subset of the interface of the derived class. Extensibility in OOP is achieved by syntactically allowing the substitution of an object of the derived class wherever an object of the base class is required. The actual type of the object, and hence the particular function executed, is determined at run time; therefore, the name *run-time polymorphism* or *dynamic binding* is used to describe this process. The run-time decision making imposes a small performance penalty and requires additional memory for

[1] Public inheritance with virtual member functions in C++.

the object, which can be a problem with fine-grain objects. Furthermore, compiler optimizations are hindered, since the compiler does not know which version of the function might be called at run time.

To make this discussion concrete, let us consider a basic finite-element software framework that can be implemented by using a base class called `FiniteElement`, which describes only the interface for a typical finite element. Later on, the framework user can add new elements to the framework (*e.g.*, an element for one-dimensional problems `FiniteElement1D`) by simply deriving from `FiniteElement`. Since the user does not need to perform any modifications on the framework, [2] the inheritance functionality of OOP ensures the easy extensibility of the framework. The base class, and hence the framework, needs to be modified only if a fundamentally different kind of finite element with a different interface requirement must be implemented. Nevertheless, modifying the interface of the base class entails the consideration of all the derived classes and can be a cumbersome task in a multi-level inheritance hierarchy.

Generalization or extension of class specifications typically involves adding new levels to the inheritance hierarchy, which inadvertently increases the software complexity and impairs code maintainability as well as performance [12]. Therefore, it is widely accepted that using purely object-oriented software design for scientific research codes, such as finite-elements, can have severe limitations. In the literature, there are many concepts available in order to deal with the complexity generated by OOP. There is also a rather extensive set of well-documented design and analysis patterns, which give OO solutions for recurring software design problems [13].

2.2 Generic Programming

Generic programming is mostly centered around algorithms and data structures as basic building blocks. In C++, generic programming is supported through class template and function template mechanisms. Templates are a powerful language feature and allow the development of classes and functions without specifying the data type used during implementation. For example, templated functions enable the development of generic algorithms, such as searching or sorting routines, without specifying the type of data that is being manipulated. The template parameters can be understood as formal placeholders, which are automatically replaced with actual types during the compilation. Importantly, the actual type that is used does not need to be aware of the generic algorithm. In contrast, in an OO design the generic algorithm would be implemented using a common base type, and the actual type that is used must be derived from this base type. The resulting strong dependency between the different classes impedes the extensibility and reuse of software packages.

The process of replacing the template parameters with actual types during compilation is called *template instantiation*, or simply *instantiation*. If the same templated function or class is instantiated with different types, the compiler generates a separate function or class for each type. Note that in order to use a function or class from a templated library, the definition of that function or class has to be visible at compile time. In effect, it is not possible to combine templated functions and classes into pre-compiled libraries.[3] This slight disadvantage is greatly

[2]If shared libraries are used, the framework user does not even need to create a new executable.

[3]One may, however, pre-instantiate a templated function or class with specific template arguments and compile

compensated by generation of a more efficient binary code than is possible with OOP. Since the definitions of all templated functions are visible to the compiler, it can perform techniques like inlining (to eliminate function call overheads) or hardware-specific optimizations more exhaustively. In the context of numerical analysis and fine-grain objects, such as libraries for small matrices, the performance difference between OOP and generic programming can translate into significant overall run-time differences. Furthermore, in generic programming only static types are used, and the compiler can perform strong type checking with the attendant increase in robustness.[4]

By far the most successful generic library in use is the Standard Template Library in C++ [3, 4]. The STL consists of various containers for a collection of objects and a number of independent algorithms. The algorithms, such as `sort` or `count`, can be applied to any STL container. To ensure interoperability, the STL utilizes the concept of an `iterator`, which can be roughly interpreted as the generalization of pointers in C. For a more comprehensive introduction to generic programming with C++, we refer the reader to standard textbooks such as [4, 14, 15, 16].

2.2.1 Illustrative Example

To fix terminology and to introduce the basic concepts of generic programming, we discuss an implementation of a range query algorithm in C++ using the STL (Listing 1). The range query algorithm could be used as part of a collision or intersection search algorithm, (e.g., see [17, 18, 19]). The STL container `vector` is used, as well as the algorithms `sort`, `lower_bound`, and `upper_bound`.

Node coordinates are stored in a `struct` named `Point`. In order to make `Point` independent of dimensionality, it has been templated or parameterized in terms of the spatial dimension (*i.e.*, `DIM`). The template argument `DIM` needs to be specified wherever `Point` is instantiated (as on line 21 of Listing 1). Importantly, it is not possible to specify template parameter `DIM` at run time (*e.g.*, depending on user input parameters). The nodes are stored in the `pts` object, which is an instance of the STL `vector` container class. To shorten the long type names, which inevitably are produced when templates are used, frequent use of the `typedef` (type definition) construct is made. After adding all the points to `pts`, the container is sorted with the STL algorithm `sort`. The `sort` function is generic and can be applied to user-defined data types stored in various STL containers. In the present example, the search criterion is specified with a parameterized function object, or *functor*, named `ComparePoints`. A functor is simply a `struct` or `class` with an overloaded function call operator `()`, like on line 9 in Listing 1. Functors may be understood as the generalization of conventional C function pointers. Importantly, they can be full-fledged classes and store additional internal variables, like `dir_` in `ComparePoints`.

The `ComparePoints` functor compares for two points of type `Point<DIM>` the `dir_`-component of the coordinates. On line 25, points are sorted with respect to their first coordinate component. The sort algorithm has better than $\mathcal{O}(N \log N)$ average complexity [3]. On lines 28–30, for determining the points in the range [`low`, `upp`] the `lower_bound` and

such instantiations into a library. This can be helpful in cases where particular instantiations are commonly used.

[4]Note that in OOP, a pointer to a base class may contain the address of an object of a derived class. Consequently, the actual type of the object referred to by the pointer can be known only at run time.

`upper_bound` algorithms are used, both of which have $\mathcal{O}(\log N)$ complexity. Finally, the number of points within the range is counted on line 33.

```
 1  template <int DIM>
    struct Point { double coor_[DIM]; };
 3
    // comparison functor for point coordinates
 5  template <int DIM>
    struct ComparePoints {
 7      ComparePoints(int dir) : dir_(dir) {}
        const int dir_;
 9      bool operator() (const Point<DIM>& p0, const Point<DIM>& p1) {
            return (p0.coor_[dir_] < p1.coor_[dir_]);
11      }
    };
13
    int main() {
15     typedef Point<3> Point3D;
       std::vector<Point3D> pts;
17
       // add points to pts container
19
       // instantiate two points for storing the range of interest [low, upp]
21     Point3D low, upp;

23     // sort pts with respect to their 0-th coordinate
       ComparePoints<3> compareXdir(0);
25     std::sort(pts.begin(), pts.end(), compareXdir);

27     // find the points within [low, upp]
       std::vector<Point3D>::iterator itu, itl;
29     itu = std::upper_bound(pts.begin(), pts.end(), upp, compareXdir);
       itl = std::lower_bound(pts.begin(), pts.end(), low, compareXdir);
31
       // count the number of points in the range (itl, itu]
33     int pointsInRange = std::distance(itl, itu);

35     return 0;
    }
```

Listing 1: Sketch of a basic range query implementation.

Remark 1 *Instead of the* `ComparePoints` *functor, it is also possible to use a conventional C function pointer. Using the following comparison function*

```
bool compareXdir(const Point<3>& p0, const Point<3>& p1) {
    return (p0.coor_[0] < p1.coor_[0]);
}
```

the sort, lower_bound, and upper_bound function calls in Listing 1 do not need any modifications. Note that the functor is able to store additional information (in this case, the coordinate component `dir_` *to be compared), in contrast to the function. Furthermore, the functor implementation is usually more efficient, since the compiler can perform optimization techniques (e.g., inlining) that are not possible with the function pointer [3].*

The presented orthogonal range query implementation is tailored with respect to the structure `Point`. Although this does not pose a problem for this small example, if we decide to factorize the orthogonal range query into an independent function as shown below, this could restrict the extensibility.

```
template <typename IT, typename PT>
void oRQ(IT begin, IT end, const PT& low, const PT& upp)
{
    ComparePoints<3> compareXdir(0);
    std::sort(begin, end, compareXdir);

    IT itu, itl;
    itu = std::upper_bound(begin, end, upp, compareXdir);
    itl = std::lower_bound(begin, end, low, compareXdir);
    // ...
}
```

Listing 2: Range query algorithm factorized into a function.

As a rule, in generic programming the dependence of the algorithm on the data types should be minimal and well defined. In contrast, the `oRQ` function is tightly coupled to the `Point` structure, mainly through the use of the `ComparePoints<3>` functor (Listing 2). We can correct this problem by endowing the `Point` type with the specification of how to perform comparisons, applying a technique known as *static* or *compile-time polymorphism*. There is both an intrusive and a non-intrusive way to make the `oRQ` function generic. In the intrusive approach, the `Point` type is augmented with a `typedef` of its comparison functor.

```
template <int DIM>
struct Point {
    typedef ComparePoints<DIM> Compare;
    double coor_[DIM];
};
```

Then in the `oRQ` function in Listing 2, `ComparePoints<3>` can be replaced with `typename PT::Compare`.[5] In effect, the intrusive approach requires that any type to be used with the `oRQ` function must have the type `Compare` defined.

Alternatively, it is possible to use the non-intrusive *traits* technique, which crucially relies on the template specialization functionality [20]. In the traits approach, an extra layer of indirection is introduced by defining a `PointTraits` type.

```
template <typename PT>
struct PointTraits {};

template <int DIM>
struct PointTraits<Point<DIM> > {typedef ComparePoints<DIM> Compare;};
```

In the preceding listing, the general definition of `PointTraits` is empty because there is no meaningful comparison functor for an arbitrary `PT` type. The second definition is a specialization of the `PointTraits` template for `Point<DIM>` types. During compilation the specialized template classes or functions have the first priority because they are a more exact match to

[5]The keyword `typename` is used to clarify that the entity `Compare` qualified by the type parameter `PT` is itself a type.

the requested instantiation. For the non-intrusive approach to be functional, in `oRQ` the functor `ComparePoints<3>` has to be replaced with `typename PointTraits<PT>::Compare`. If the user decides to use `oRQ` with a new class or structure for storing the coordinates, only a new specialization of `PointTraits` needs to be provided.

2.2.2 Advanced Generic Programming: Metaprogramming and Expression Templates

As previously described, the compiler template instantiation mechanism effectively replaces placeholders (template parameters) with specific types as requested by the user. In this context, the overall compilation process can be envisaged as a two-step procedure: first the template instantiation mechanism creates a non-templated program by resolving all the template parameters, and subsequently the resulting program is compiled into the binary format. With so-called *template metaprogramming*, the compiler template instantiation mechanism can be used for creating programs that are evaluated at compile time. In numerical computing, related techniques can be used for unrolling loops and creating highly efficient algorithms [21].

A widely used C++ feature in numerical libraries is operator overloading, which enables the evaluation of expressions (*e.g.*, `a=b+c+d`) for user-defined vector and matrix types. A naive implementation of operator overloading in combination with standard expression evaluation forces the compiler to create temporary objects to contain the intermediate subexpression results. For fine-grain operations, the associated memory allocation and deallocation and the necessary copies can impose severe performance penalties. The *expression templates* technique eliminates the temporaries by automatically constructing expression parse trees and evaluating the whole expression at once rather than one operation at a time [22].

2.3 Multiparadigm Programming

Despite the appeal of generic programming, its exclusive use for large-scale software is not viable, since the excessive use of templates usually leads to cryptic software that is hard to understand and has very long compilation times. Therefore, a new concept that has crystallized during the last several years is multi-paradigm programming, a combination of object-oriented and generic programming. In particular for engineering software, it is beneficial to use for higher-level objects (*e.g.*, a solver or mesh) an object-oriented representation and for lower-level objects (*e.g.*, elements, nodes and coordinate tuples) a generic programming approach. This multi-paradigm approach exploits the run-time flexibility provided by OOP and the efficiency and type safety provided by generic programming. In addition, using static polymorphism judiciously enables flatter class hierarchies and alleviates one of the difficulties with OOP.

3 Generic Containers and Algorithms for Unstructured Meshes

The development of large-scale numerical engineering software is evolutionary and incremental; hence, most of the already established components have been implemented over several years and often do not make use of advanced software engineering concepts. It is not reasonable to rewrite such complex computational kernels in view of recent advances in software engineering. In the following work, we introduce an approach for encapsulating such computational kernels and augmenting them with a thin interface layer in order to make them interoperable with STL-type generic mesh algorithms and containers. Our overall approach can be summarized as follows:

- Retain existing computational kernels in C or Fortran where possible.
- Encapsulate existing procedural code or non-generic data structures with STL-style containers.
- Perform typical FEM operations on unstructured mesh data using STL-style generic algorithms.

3.1 Generic Mesh Class

For the purposes of finite-element computation, it suffices to have a very lightweight mesh class that essentially stores the mesh entities and has a few templated member functions for iterating over these mesh entities. As will be elaborated in detail, the genericity is achieved through the extensive use of traits and functor techniques. In the presented framework, it is assumed that the finite-element mesh has been generated with an external solid modeling and/or mesh generation package. The output of such mesh generation tools usually consists of a simplicial complex and tagged simplices (for indicating boundary or material types). In order to make the mesh class sketched in Listing 3 independent of a particular data structure, it has been parameterized with respect to the element, face, and vertex types. The mesh container shown in Listing 3 reads the simplicial complex from an input stream, instantiates the simplices, and stores pointers to them in STL `vector` containers. In order to use a pre-existing vertex, face, or element class with the proposed `Mesh` class, it is necessary to implement the specializations of traits classes `VertexTraits`, `FaceTraits`, and `ElementTraits`.

The most common operations for finite elements in the solution phase are iteration over the elements (*e.g.*, to assemble the stiffness matrix) or iteration over the vertices (*e.g.*, to compute nodal forces). Therefore, the `Mesh` class has parameterized member functions `iterateOverVertices`, `iterateOverFaces`, and `iterateOverElements` for iterating over the mesh entities. The algorithms for performing operations specific to finite elements will be provided in the form of functors to the iteration methods.

```
template <typename V, typename F, typename T>
class Mesh {
    std::vector<V* >  vertices_;
    std::vector<F* >  faces_;
    std::vector<T* >  elements_;

public:
    typedef VertexTraits<V>  Vertex;
    Mesh(std::istream& is) {
        int numVtx;
        is >> numVtx;

        for (int i=0; i<numVtx; ++i) {
            vertices_.push_back(Vertex::createVertex(is));
        };
        // read the elements from the input stream
    }

    template <typename OP>
    void iterateOverVertices(const OP& op) {
        std::for_each(vertices_.begin(), vertices_.end(), op);
    }

    //
};
```

Listing 3: Sketch of a basic mesh container.

The non-condensed version of the lightweight mesh container sketched in Listing 3 contains additional functionalities, such as the ability to mark certain mesh entities as active, inactive, or ghost. The mesh container and its methods access the vertex data or related functions only through the `VertexTraits<V>` entity. Note that `iterateOverVertices` itself uses the STL `for_each` algorithm to apply the functor to all the vertices.

Remark 2 *Instead of using generic member functions for iterating over the mesh entities, it would be possible to use the non-member STL for_each algorithm. However, this would require direct access to the private members of the class.*

3.2 Wrapper Classes

The principal motivation for developing the present framework was to extend pre-existing finite-element kernels written in C with a generic layer. As is typical in C code, the computational kernels consist of data structures in the form of `struct` data types and functions (*e.g.*, see Listing 4). In order to make the high-level C data types and functions accessible to a generic programming layer, small traits classes, also known as *wrapper classes*, are used. A sample wrapper class and its specialization for accessing the data and functions related to the vertices is shown in Listing 5. Note that keeping the implementation of the wrapper class template empty or incomplete effectively prohibits its use (as on Line 1 of Listing 5). In the template specialization, an independent type is defined for each variable or array in the C struct. The specialized type can be understood also as a functor for accessing the variable in a generic way. Furthermore, the indirection provided by the C++ layer over the C functions can effectively be used to preprocess the function arguments. For example, in the `createVertex` function in Listing 5, the data is read from a C++ input stream and given to the C function in form of an array.

```
typedef struct {
    double xyz[3];      /* position of the vertex */
    int index;          /* node number */
    /* arrays for displacements, velocities, accelerations */
} Vertex3dC;

extern "C" Vertex3dC *createVertex3dC(int id, double xyz[3]);
```

Listing 4: Existing C data structure and function declaration.

```cpp
template <typename V> struct VertexTraits {};

template <>
struct VertexTraits<Vertex3dC> {
    typedef Vertex3dC VertexType;

    struct Coordinates {
        typedef double VarType;
        static const int NumVar = 3;
        VarType* operator()(VertexType* vtx) {return vtx->xyz;}
    };

    struct Index {
        typedef int VarType;
        static const int NumVar = 1;
        VarType* operator()(VertexType* vtx) {return &(vtx->index);}
    };

    static VertexType* createVertex(std::istream& is) {
        // read the index and coordinates from the input stream is
        return createVertex3dC(id, xyz);
    }
};
```

Listing 5: Vertex traits class and its specialization.

Note that the unabridged implementation of `VertexTraits<Vertex3dC>` contains further types for accessing the displacements, velocities and accelerations. They have been omitted from Listing 5 in order to keep the discussion compact.

In addition to the `Vertex` data type, it is necessary to wrap key high-level C functions, such as the time integration function in the case of explicit dynamics or the stiffness matrix assembly algorithm.

3.3 Sample Algorithms

The introduced wrapper classes provide a simple abstraction layer and are the basis for the utilization of powerful generic algorithms. In the following, we introduce two such generic algorithms: the `VertexCollector` and the `VertexDistributor` functors. In the finite-element program, the vertex variables, such as displacements, velocities, *etc.*, are assigned to the vertices and are only available in a distributed sense. It is frequently necessary to collect these distributed variables into containers, *e.g.*, during output of the mesh for visualization or the communication phase on parallel computers. Before introducing the `VertexCollector` algorithm, one commented example usage is given in Listing 6.

```cpp
// Define the Vertex type
typedef VertexTraits<Vertex3dC> Vertex;
// STL vector object for storing the coordinates
typedef std::vector<Vertex::Coordinates::VarType> CoordinateCont;
CoordinateCont coordinates;
// STL iterator adaptor for inserting an item after the last coordinate
typedef std::back_insert_iterator<CoordinateCont> CoordinateInserter;
```

```
CoordinateInserter coordinateInsert(coordinates);
// Functor for collecting the coordinates (second argument)
// into the coordinate container using inserter (first argument)
VertexCollector<CoordinateInserter, Vertex::Coordinates>
    collectFunctor(coordinateInsert);
// request the mesh object m to apply the functor to all vertices
m->iterateOverVertices(collectFunctor);
```

Listing 6: Collect the vertex coordinates into an STL container.

Although the preceding usage looks rather cryptic, it is generic and hence versatile in several ways. First, it is not necessary to deal with dynamic memory management because it is accomplished by the `vector` container.[6] Second, essentially the same code snippet can be used for collecting the coordinates into a different type of container, such as an STL `list`. Moreover, simply changing the iterator type to an `ostream_iterator` will print the data to a file or to the standard output as in the following:

```
std::ostream_iterator<double> output(std::cout, "\n");
VertexCollector<std::ostream_iterator<double>,
                Vertex::Coordinates> collectFunctor(output);
m->iterateOverVertices(collectFunctor);
```

Third, with only a few changes the same algorithm will collect the vertex indices or other physical variables. Finally, the algorithm is completely independent from the underlying vertex data structures and does not need to be changed if the vertex representation changes.

The implementation of the `VertexCollector` is rather simple and is shown in Listing 7. It is parameterized with respect to an STL insert iterator type and the vertex functors, such as `Coordinates` and `Index`, defined in Section 3.2. As is typical in the STL, the independence of the algorithm from the container type is achieved by using iterators instead of directly using the container.

```
template <typename Inserter, typename OP>
struct VertexCollector :
    public std::unary_function<typename OP::VertexType *, void>
{
    Inserter ins_;
public:
    VertexCollector(Inserter ins) : ins_(ins) {}
    void operator()(typename OP::VertexType * vtx) const {
        typename OP::VarType *vtxVar = OP()(vtx);
        for (int i=0; i<OP::NumVar; ++i) {*ins_++ = *vtxVar++;}
    }
};
```

Listing 7: Collector for data stored at the vertices.

The corresponding functor `VertexDistributor`, which distributes data from a container or C array to the vertices is shown in Listing 8. Similar to `VertexCollector`, it is parameterized with respect to the input iterator type and the vertex functors. The input to the constructor consists of a iterator range `[begin, end)` for the data that is to be distributed.

[6]In order to avoid the successive memory reallocations during the collection process with the STL `vector` container, one can use the `reserve` method.

```
template <typename Iterator, typename OP>
class VertexDistributor :
    public std::unary_function<typename OP::VertexType *, void>
{
    Iterator        begin_;
    const Iterator  end_;
public:
    VertexDistributor(Iterator begin, Iterator end) :
        begin_(begin), end_(end) {}
    void operator()(typename OP::VertexType * vtx) {
        typename OP::VarType *vtxVar = OP()(vtx);
        for (int i=0; i<OP::NumVar; ++i) {
            assert(begin_ != end_);
            *vtxVar++ = *begin_++;
        }
    }
};
```

Listing 8: Distribute data stored in a container to the vertices.

A sample usage of this algorithm is as follows:

```
typedef std::vector<Vertex::Coordinates::VarType> CoordinateCont;
CoordinateCont coordinates;
// ....
VertexDistributor<CoordinateCont::iterator, Vertex::Coordinates>
    distributeFunctor(coordinates.begin(), coordinates.end());
    m->iterateOverVertices(distributeFunctor);
```

Changing the iterator to an `istream_iterator` enables reading and distributing the data from an input stream, such as a file or terminal. The related changes to the preceding code snippet are left as an exercise for the reader.

3.4 Code Performance Issues

It is clear from the examples discussed in Section 3.3 that the abstraction layer introduced through the use of simple wrapper classes for a pre-existing set of finite-element-modeling data types and functions can greatly simplify the development of FEM applications. The resulting application code is much more flexible and can easily be extended to handle new geometric data representations or apply new algorithms. However, one might wonder what cost in code performance, if any, is incurred through the use of such an abstraction layer. Any loss in code performance relative to the efficiency obtained by using the original FEM data types and functions directly is commonly referred to as the *abstraction penalty*.

In order to assess the performance of generic FEM algorithms utilizing our abstraction layer, we composed functors similar to the previous `VertexCollector` example named `NewmarkPredictFunctor` and `NewmarkCorrectFunctor`. These two functors perform the basic vertex displacement (u), velocity ($\dot{u}$), and acceleration ($\ddot{u}$) updates of the explicit Newmark time integration scheme in the predictor and corrector stages.

$$\begin{array}{ll} \text{Predictor:} & u_{n+1} = u_n + \Delta t \dot{u}_n + \frac{1}{2}\Delta t^2 \ddot{u}_n \\ & \widetilde{\dot{u}}_{n+1} = \dot{u}_n + (1-\gamma)\Delta t \ddot{u}_n \\ \text{Corrector:} & \ddot{u}_{n+1} = \frac{r_{n+1}}{\underset{\sim}{m}} \quad \text{with} \quad r_{n+1} = F\{u_{n+1}\} \\ & \dot{u}_{n+1} = \widetilde{\dot{u}}_{n+1} + \gamma \Delta t \ddot{u}_{n+1} \end{array}$$

The constructors for these Newmark functors take as arguments and store internally any needed parameters of the time integration, such as the time step size Δt and the Newmark damping factor γ. The `operator()` method then uses the accessor structures provided by the `VertexTraits` class to read and write the variables associated with each given vertex object in a generic style. In a FEM simulation, the residual vector r_{n+1} is assembled from the internal and external forces on an element given the predicted displacements u_{n+1}.

A simple test code was written using the Newmark functors to update a large number of randomly initialized `Vertex` structures. The total CPU time required to perform the update using the functors was compared with that required using a C function pointer approach, with C functions acting directly upon the data structures. For the sake of simplicity, a trivial linear functional form $F = -cu_{n+1}$, with a scalar stiffness c, was assumed. Exercising this test code on a wide variety of processor architectures and using several different C/C++ compilers, we found no measurable abstraction penalty for the use of our generic functor-based implementation of the Newmark time integration scheme.

Remark 3 *In the used finite element code the data associated with vertices of a mesh, such as displacements, velocities and forces, are locally stored at vertex objects. This approach allows one to easily add or remove vertices to the set of finite-element nodes, as may be required in the case of fracture propagation, mesh adaptation, or dynamic load balancing across parallel processors. In many conventional finite-element codes, the vertex variables are stored in separate, contiguous data arrays in order to enhance code performance and minimize cache misses on cache-based architectures. Significant complexity and overhead costs are introduced by the need to maintain these contiguous data arrays as the vertex set changes. In contemporary non-linear engineering computations, the efficiency increase attainable through the optimization of the data layout is fairly limited. This is largely due to the fact that the bulk of the computing time is typically spent in assembling and solving matrices rather than performing per-vertex operations. Performance comparisons using the test code described above indicate that at most a factor of ∼2 speedup can be obtained using contiguous data arrays instead of per-vertex data structures, for the case of a trivial residual force computation.*

4 Generic Parallelization of the FEM Solver

The parallelization of serial finite-element programs for distributed-memory computing architectures relies on the partition of the discretized domain into subdomains and computing each subdomain on a different processor. The number of subdomains is usually chosen to be equal to the number of processors, and on each subdomain essentially a serial solver is applied. An efficient partitioning is characterized by a minimum interprocess communication and maximum processor utilization. Depending on the discretization scheme, the mathematical model, and the solution method, the discretized subdomains can be overlapping or non-overlapping (*e.g.*, see [23]). For example, one discretization scheme that leads to (one-element-wide) overlap-

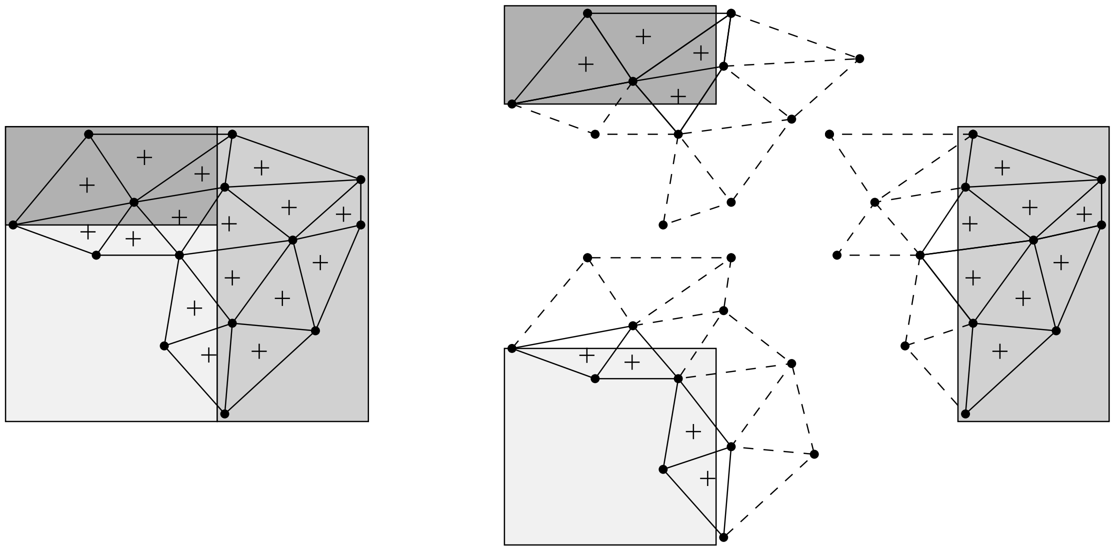


Figure 1: Partitioning of a finite-element mesh (left) into three subdomains using the RCB algorithm. The three subdomains have an overlap that is one element wide.

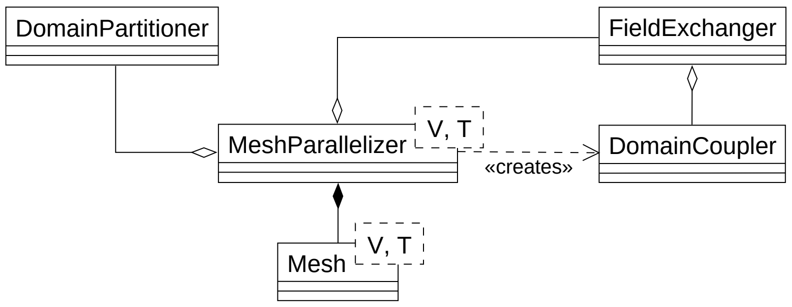


Figure 2: UML diagram for the parallelization layer.

ping subdomains involves the use of subdivision shape functions for thin-shell computations [24, 25]. Each subdomain is equipped with a one-element-wide ghost layer at the subdomain boundaries. The subdomain solvers only communicate through the exchange of data at the subdomain boundaries or intersections. Along the lines of this high-level algorithmic description, it is possible to add a simple parallelization layer on top of the serial, generic, finite-element framework introduced so far.

A good parallelization strategy ideally should not annihilate the serial functionality of the finite-element code. In the present framework, the parallelization layer is added on top of the generic mesh abstraction layer in a non-intrusive way (see Fig. 2). The details of the parallelization, consisting of domain partitioning and interprocess communication, are encapsulated in a `MeshParallelizer` class. It uses the associated `DomainPartitioner` class to partition the domain and the `FieldExchanger` class to perform the interprocess communication.

```
template <class V, class T>
class MeshParallelizer {
```

```
public:
    MeshParallelizer(MPI_Comm communicator, Mesh<V,T> *mMesh);
    enum VarType {mass = 0, residual = 1};
    void initializeCommunication();
    void exchange(const VarType& var);
private:
    Mesh<V,T>                                       *mMesh_;
    DomainPartitioner                               *psetPartitioner_;
    FieldExchanger                                  *exchanger_;
};
```

Listing 9: Mesh parallelizer.

The `DomainPartitioner` implements the straightforward and simple recursive coordinate bisection (RCB) method for partitioning the mesh [26]. The input for the algorithm is the center position of the elements. The algorithm recursively subdivides the input point set with a cutting plane into two halves. The cutting plane is always placed orthogonal to the most elongated coordinate direction. Despite the shortcomings of RCB in comparison to other state-of-the-art algorithms (*e.eg.*, see [27]), it has been chosen here because it is easily implemented.

The `DomainCoupler` conceals the point-to-point exchange of data between the different processes based on a communication map computed during the initialization phase. The communication map is computed by first exchanging the axis-aligned bounding boxes of all the domains and computing their intersections.[7] Two processes need to communicate only if the intersection of the assigned subdomains is non-empty. Note that the `DomainCoupler` is only aware of the extent of the bounding box and is uninformed about the underlying finite-element mesh. For a particular process, the task of the `DomainCoupler` is to publish a given data buffer (*i.e.*, a contiguous memory block of data) and to receive the data buffers of all relevant processes. For data transfer between the processes, the low-level Message Passing Interface (MPI) library is used. MPI provides only a primitive functionality and is awkward to use without a generalization layer, as provided by the `DomainCoupler` class.

```
class FieldExchanger {
public:
    FieldExchanger(DomainCoupler* coupler,
                   std::vector<int>::iterator lGBegin, int nodes);
    template <int NVAR, template <typename> class OP, typename IT>
    void exchange(IT begin, IT end);
private:
    // mapping from imported node IDs to local node IDs
    std::vector<int>        importedNodeIDs_;
    // domain coupler instance which handles the communication
    DomainCoupler          *coupler_;
};
```

Listing 10: Synchronization of nodal data shared by different proccesses.

The `FieldExchanger` class is the next abstraction layer above the `DomainCoupler`, and it performs the exchange of data between finite-element nodes located on different processes (see Listing 10). To that end, it is required that the finite-element nodes on all processes have

[7]The axis-aligned bounding box of a subdomain is the smallest rectangular box with sides parallel to the x, y, and z axes that encloses the subdomain.

a unique global numbering. The functionality of the `FieldExchanger` is best illustrated by means of an example: assembling the internal force contributions during the explicit time integration. Each element contributes to the internal force, which is stored at the nodes. For the shared nodes at the subdomain boundaries, the contribution of the elements located on the adjacent subdomain(s) need to be exchanged and added. The following example shows the synchronization of the internal forces stored in a container:

```
FieldExchanger *ex = new FieldExchanger(...);
std::vector<double> intern(...);
const int dofPerVertex = 3; // intern components per node
ex->exchange<dofPerVertex, std::plus>(intern.begin(), intern.end());
```

The range between the iterators `[intern.begin(), intern.end())` contains all the internal forces for one subdomain. `FieldExchanger` requires that the used container is an STL sequence (*e.g.*, `vector`, `deque` or `list`) so that the iterator associated with an element implies a unique local numbering (`current-intern.begin()`). Other useful operations besides `std::plus` include `std::max` and `std::min`. The `FieldExchanger` ensures that only relevant data is communicated to neighboring subdomains and performs the necessary mappings between the local IDs and global IDs.

Note that the `DomainCoupler` and `FieldExchanger` are independent components and can also be used for non-finite-element purposes.

5 Coupling the Finite-Element Solver to Other Solvers

Computational models that require the interaction of several disparate physical or mathematical models inherently lead to large and complex software. One common conception is to use object-oriented frameworks for integrating the individual domain solvers (*e.g.*, see [28, 29]). Although object-oriented frameworks are well suited for implementing new software or integrating stable, mature software components, they tend to be overly restrictive for research purposes. In addition, the individual components have frequently been implemented independently, and they have a different level of maturity and/or use different programming models.

To remedy the code-coupling problem, a non-intrusive generic programming approach can be developed. The UML diagram in Fig. 3 describes a sample fluid-structure interaction framework for coupling a solid solver (represented by a `Solid` class) and a fluid solver (represented by a `Fluid` class). The framework provides the `SolverBase` class, which encapsulates coarse-grain functionality typical to both solvers, such as `visualize`, `advance`, and `solve` methods. Furthermore, there is a derived class `SolverAdaptor`, which is templated with respect to the solver type. The methods of `SolverAdaptor<SolverType>` only provide an indirection layer for calling the methods of `SolverType` (*e.g.*, see the implementation of the `visualize` method). In contrast to the proposed non-intrusive technique, which is commonly referred to as *external polymorphism*, conventional OOP only permits an intrusive solution by deriving the `Fluid` and `Solid` classes from the `SolverBase` class.

A possible usage of this framework is in parallel computing, where one group of processors is assigned to the fluid domain and the remaining processors are assigned to the solid domain. The solver is instantiated on the fluid processors with `SolverBase *sp = new SolverAdaptor<Fluid>` and on the solid processors with `SolverBase *sp = new`

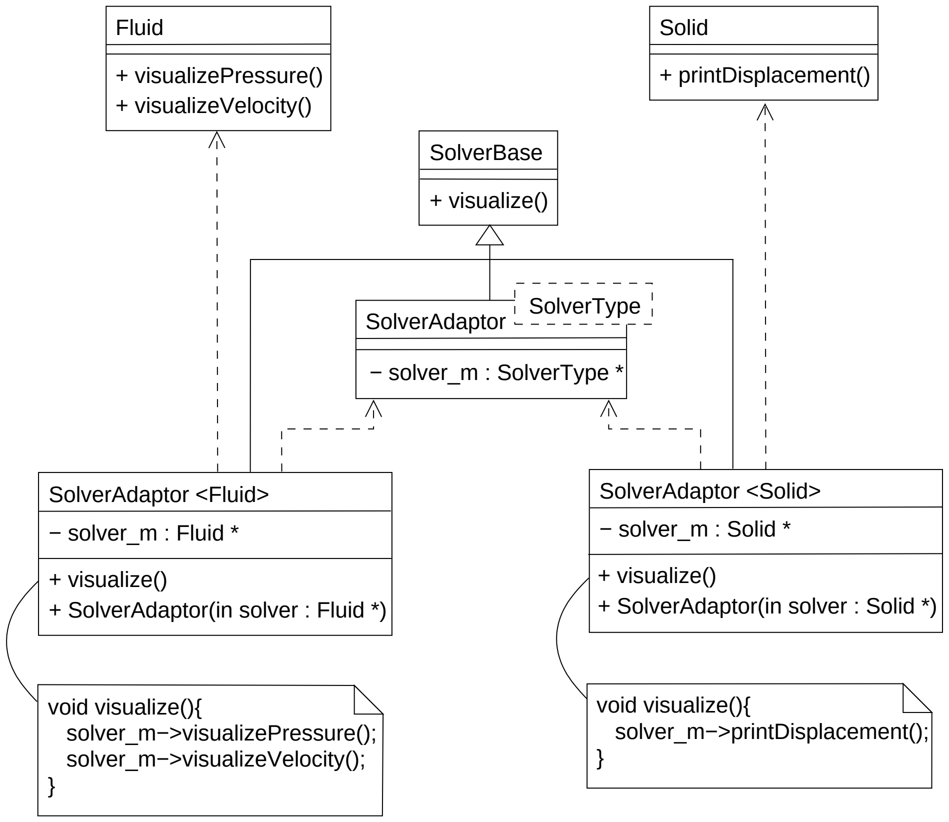

Fluid
+ visualizePressure()
+ visualizeVelocity()
Solid
+ printDisplacement()
SolverBase
+ visualize()
SolverAdaptor
SolverType
− solver_m : SolverType *
SolverAdaptor <Fluid>
− solver_m : Fluid *
+ visualize()
+ SolverAdaptor(in solver : Fluid *)
SolverAdaptor <Solid>
− solver_m : Solid *
+ visualize()
+ SolverAdaptor(in solver : Solid *)
void visualize(){ solver_m−>visualizePressure(); solver_m−>visualizeVelocity(); }
void visualize(){ solver_m−>printDisplacement(); }

Figure 3: UML diagram for non-intrusive coupling of two independent solvers.

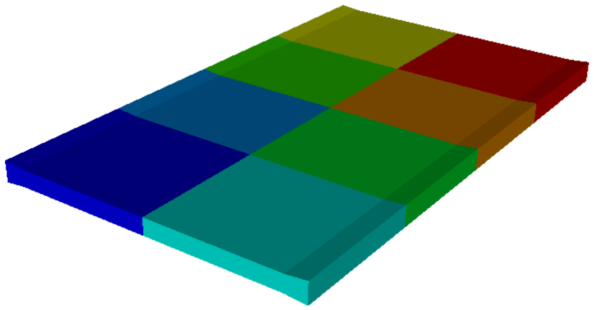
`SolverAdaptor<Solid>`. The run-time polymorphism ensures that `sp->visualize()` automatically calls the proper method for visualization on each processor. This specific framework shows the advantages of combining of run- and compile-time polymorphism for non-intrusive solver coupling.

6 Applications

In this section we introduce selected applications computed with the discussed software framework. In all of these examples, the solid components are discretized with subdivision shell elements [10, 24] and the non-stationary equilibrium equations are integrated in time with the explicit Newmark scheme. Furthermore, the material model used is a J_2-plasticity model for large deformations [30]. The description of the discretization methods and physical models used here has been consciously kept brief. For details the interested reader is referred to the cited publications.

6.1 Double-hull Plate Structure Subjected to Pressure Loading

The first example is the deformation of a double-hull structure subjected to external pressure loading (Fig. 4). It consists of two skin plates and a stiffener in the form of a girder placed


(a) Partitioning for eight processors.

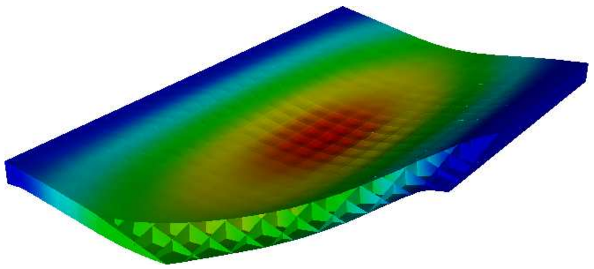

(b) Cut through the deformed structure.

Figure 4: Double-hull subjected to external pressure loading.

between the plates. The non-manifold topology of the structure requires special consideration for geometric description as well as discretization. In the present framework, the whole double-hull is considered first as a collection of non-connected, quadrilateral plate structures. The movement in unison of the non-connected plates is enforced during the explicit time integration by proper averaging of the vertex masses and forces at the boundaries. The previously described non-intrusive software architecture enabled us to accomplish this by simply modifying the time integration loop, without affecting the core components of the library.

Figure 4(a) shows the partitioning of the double-hull structure for eight processors. The double-hull is deformed by the pressure loading. The two boundaries along the longer edges are fixed, while the other two boundaries are free. The deformation of the double-hull with a discretization consisting of 325116 degrees of freedom is shown in Figure 4(b).

Code profiling of this double-hull FEM application using the *gprof* profiling tool on a Linux Pentium-based cluster indicates that nearly 98.5% of the computing time is spent computing the internal forces on each element. Other tasks such as computing the external pressure loading, performing the predictor/corrector time advance, and interprocessor communications account for the small remainder. Of the time spent computing the internal forces, 99.78% occurred within the computational kernels written in C, indicating a mere 0.22% abstraction penalty for the most crucial section of code.

6.2 Tube Subjected to Moving Pressure Loading

As a more complex application, we consider the fracture of a thin cylindrical tube due to traveling pressure loading (Fig. 5). Fracture initiation and propagation is accomplished by placing special cohesive interface elements between standard elements of the initial mesh [25]. The interface elements constrain the opening of the crack flanks to the deformation of the shell and reproduce the mechanics of the fracture-opening process.

Figure 5(a) shows the partitioning of the undeformed tube with a discretization of approximately 65300 degrees of freedom for 32 processors. The pressure loading travels from the left end to the right end of the tube with a constant velocity. As the pressure wave progresses, the tube fractures into elongated strips as depicted in Figure 5(b).

The parallel performance of this example was studied on a symmetric multi-processor (SMP) cluster of nodes containing Compaq/HP Alphaserver ES45 processors and using a Quadrics Qs-

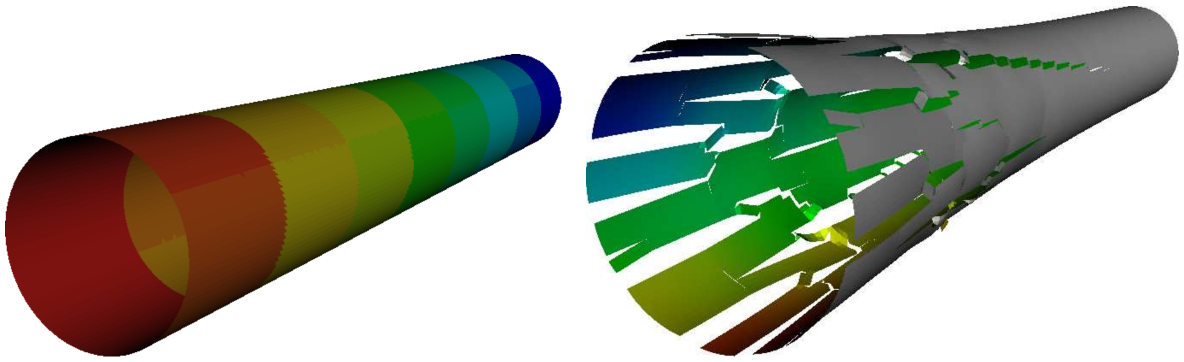

(a) Partitioning for 32 processors. (b) Partially fractured tube.

Figure 5: Tube subjected to non-stationary pressure loading.

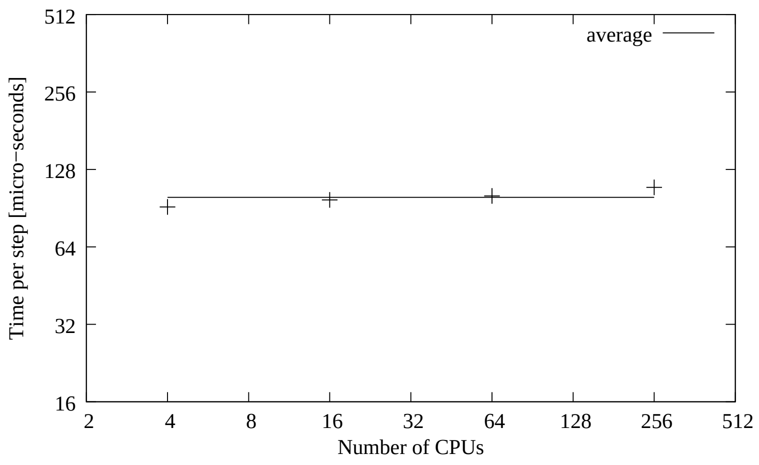


Figure 6: Weak scaling with approximately 580 elements per processor.

Net high-speed interconnect. The overall algorithm is nearly perfectly scalable from 4 to 256 processors, as can be seen from the weak-scaling results in Figure 6. During these runs the number of elements on each processor was approximately constant, with a value of 580. During the strong-scaling runs, shown in Figure 7, the total number of elements is kept constant. As expected, the performance degrades if the number of elements per processor is too few, less than about 35. With so few elements, the interprocessor communication overhead becomes large compared to the processor computational workload. The relatively low number of 35 elements means that even fairly small problems can be run very fast using many processors.

6.3 Tube Fracture

Our last example is the fluid-shell coupled simulation of detonation-driven rupture of an aluminum tube (Fig. 8). The tube is filled with an explosive ethylene-oxygen mixture, which creates a propagating detonation wave after it is ignited at one end of the tube. The detonation wave is computed with a gas dynamics solver on a Cartesian mesh using the reactive Euler equations

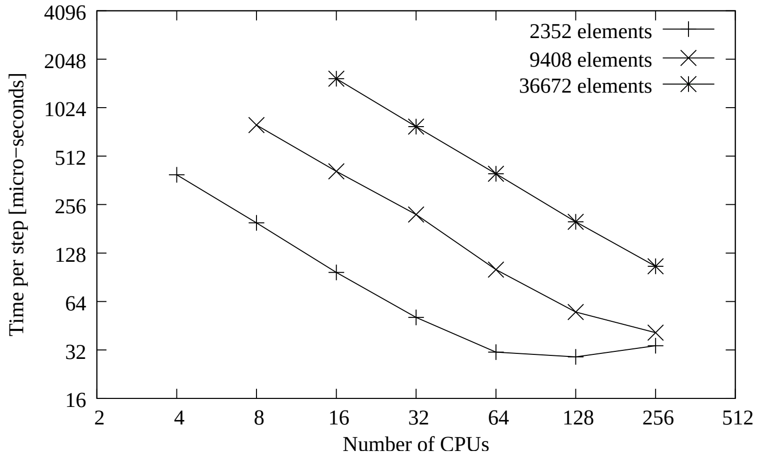


Figure 7: Strong scaling for three different mesh sizes.

[31]. The fluid and shell solvers are weakly coupled by applying appropriate interface boundary conditions during the explicit time integration. For further details on the coupling technique used and the computational specifics we refer to [32, 33, 34]. Figure 8 shows a snapshot of a typical fluid-structure interaction simulation where the venting of the high-pressure reacted gas out of the cracked tube is visible.

7 Conclusions

We have presented a generic programming approach in C++ for extending existing finite-element software components that were previously written in procedural programming languages such as Fortran and C. The initial purpose of the generic programming layer was to facilitate the parallelization of an existing thin-shell solver and its coupling to other solvers. Beyond that, the generic programming layer was used as a fresh starting point for developing and implementing new algorithms (*e.g.*, for fracture of finite-element meshes and contact search). Code performance tests show that the performance penalty incurred due to the use of a generic programming layer is minimal in comparison with the overall running time of the code. Components of the introduced software can be downloaded from the Internet at `http://www-g.eng.cam.ac.uk/csml/software.html`.

Acknowledgments

The support of DoE through Caltech's ASCI Center is gratefully acknowledged (DOE W-7405-ENG-48, B523297).

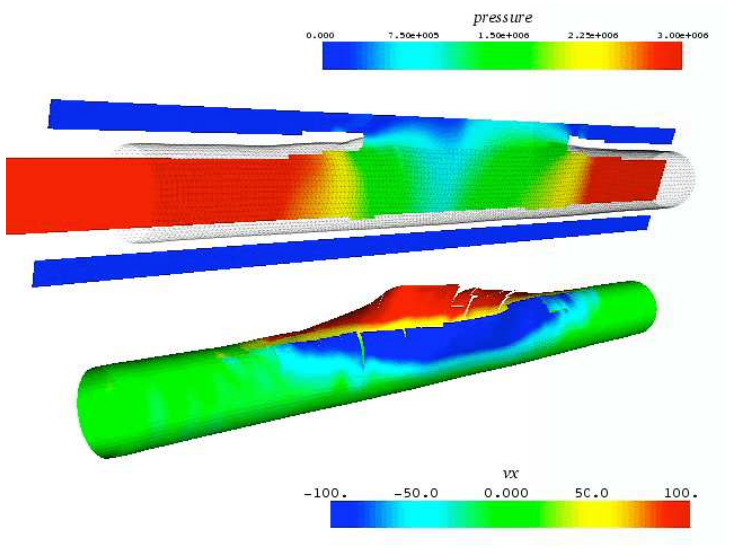


Figure 8: Fluid-shell coupled simulation of detonation-driven fracture of an aluminum tube.

References

[1] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit – An Object-Oriented Approach to 3D Graphics*. Prentice Hall, 2nd edition edition, 1998.

[2] R.I. Mackie. Object oriented programming of the finite-element method. *International Journal for Numerical Methods in Engineering*, 35(2):425–436, 1992.

[3] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd. edition, 1997.

[4] N.M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, 1999.

[5] B. Dawes, D. Abrahams, and R. Rivera. Boost. http://www.boost.org/.

[6] B. Karlsson. *Beyond the C++ Standard Library – An introduction to Boost*. Addison-Wesley, 2005.

[7] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of cgal, a computational geometry algorithms library. *Software-Practice and Experience*, 30:1167–1202, 2000.

[8] T. Veldhuizen. Blitz++. http://www.oonumerics.org/blitz/.

[9] Y. Renard and J. Pommier. Getfem++. http://www-gmm.insa-toulouse.fr/getfem/.

[10] F. Cirak, M. Ortiz, and P. Schröder. Subdivision surfaces: A new paradigm for thin-shell finite-element analysis. *International Journal for Numerical Methods in Engineering*, 47(12):2039–2072, 2000.

[11] J.-F. Remacle and M.S. Shephard. An algorithm oriented mesh database. *International Journal for Numerical Methods in Engineering*, 58(2):349–374, 2003.

[12] J. Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, 1996.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] D. Vandevoorde and N.M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2003.

[15] M.H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1998.

[16] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[17] M.W. Heinstein, F.J. Mello, S.W. Attaway, and T.A. Laursen. Contact-impact modeling in explicit transient dynamics. *Computer Methods in Applied Mechanics and Engineering*, 187:621–640, 2000.

[18] F. Cirak and M. West. Decomposition-based contact response (dcr) for explicit finite element dynamics. *International Journal for Numerical Methods in Engineering*, 64:1078–1110, 2005.

[19] S.P. Mauch. *Efficient Algorithms for Solving Static Hamilton-Jacobi Equations*. PhD thesis, California Institute of Technology, Pasadena, CA, 2003.

[20] N.C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[21] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, May 1995.

[22] T. Veldhuizen. Expression templates. *C++ Report*, June 1995.

[23] A. Quarteroni and A. Valli. *Domain Decomposition Methods for Partial Differential Equations*. Oxford University Press, 1999.

[24] F. Cirak and M. Ortiz. Fully c^1-conforming subdivision elements for finite deformation thin-shell analysis. *International Journal for Numerical Methods in Engineering*, 51(7):813–833, 2001.

[25] F. Cirak, M. Ortiz, and A. Pandolfi. A cohesive approach to thin-shell fracture and fragmentation. *Computer Methods in Applied Mechanics and Engineering*, 194:2604–2618, 2005.

[26] M.J. Berger and S.H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, 36:570–580, 1987.

[27] B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 184:485–500, 2000.

[28] J.R. Stewart and H.C. Edwards. A framework approach for developing parallel adaptive multiphysics applications. *Finite Elements in Analysis and Design*, 40:1599–1617, 2004.

[29] M.W. Beall and M.S. Shephard. An object-oriented framework for reliable numerical simulations. *Engineering with Computers*, 15:61–72, 1999.

[30] A. Cuitino and M. Ortiz. A material-independent method for extending stress update algorithms from small-strain plasticity to finite plasticity with multiplicative kinematics. *Engineering Computations*, 9:437–451, 1992.

[31] R. Deiterding. *Parallel adaptive simulation of multi-dimensional detonation structures*. PhD thesis, Brandenburgische Technische Universität Cottbus, September 2003.

[32] F. Cirak and R. Radovitzky. A lagrangian-eulerian shell-fluid coupling algorithm based on level sets. *Computers & Structures*, 83:491–498, 2005.

[33] F. Cirak, R. Deiterding, and S.P. Mauch. Large-scale fluid-structure interaction simulation of viscoplastic and fracturing thin-shells subjected to shocks and detonations. *Computers & Structures*, In press, 2006.

[34] R. Deiterding, R. Radovitzky, S.P. Mauch, L. Noels, J.C. Cummings, and D.I. Meiron. A virtual test facility for the efficient simulation of solid material response under strong shock and detonation wave loading. *Engineering with Computers*, In press, 2006.