Janus — a C++ Template Library for Parallel Dynamic Mesh Applications

Jens Gerlach, Mitsuhisa Sato, and Yutaka Ishikawa
{jens,msato,ishikawa}@trc.rwcp.or.jp

Tsukuba Research Center of the Real World Computing Partnership
Tsukuba Mitsui Building, 1-6-1 Takezono, Tsukuba-shi 305-0032, Japan

Abstract. We propose *Janus* — a C++ template library of container classes and communication primitives for parallel dynamic mesh applications. The paper focuses on *two phase containers* that are a central component of the Janus framework. These containers are quasi-constant, i.e., they have an extended initialization phase after which they provide read-only access to their elements. Two phase containers are useful for the efficient and easy-to-use representation of finite element meshes and generating sparse matrices. Using such containers makes it easy to encapsulate irregular communication patterns that occur when running finite element programs in parallel.
Keywords: data parallel programming, object-oriented programming, generic programming, finite element methods.

1 Introduction

If we think of a finite element program as a collection of related objects on which operations are performed we recognize that there are basically two types of application objects. The first type are sets that represent spatial structures and the second type are numerical functions on these sets. Here are some examples of spatial structures and functions on them.

- Given the node set N, many physically relevant data are represented by functions $f : N \mapsto \mathbb{R}$.
- The element matrices are a function from the triangulation $e : T \mapsto \mathbb{R}^{p \times p}$ where p denotes the number of degrees of freedom per element.
- The system matrix m is a function $m : \mathcal{N} \mapsto \mathbb{R}$ where $\mathcal{N}$ is a subset of $N \times N$ that represents the sparsity pattern of the matrix.

The parallelism of finite element methods is mainly data parallelism with respect to the meshes. Using parallel computers with a distributed memory architecture requires therefore a partition of the triangulation T and the node set N. Whatever communication occurs when running a finite element program in parallel, it is caused by a relation of the meshes. A problem hereby is that due to their irregularity and size the relations must be partitioned themselves.

The driving motivation behind the design of the Janus framework is to provide application-oriented, easy-to-use and efficient abstractions for the above mentioned fundamental components of finite elements methods. Janus offers building blocks to represent (possibly partitioned) spatial structures and functions on them. Moreover communication must be expressed explicitly based on the mesh relations.

Janus is implemented as a C++ template library. The library is generic in respect to the numerical types, the way the user wants to represent mesh points, and the mapping information that is used for partitioning the meshes.

A fundamental concept in Janus is that of *two phase containers* which are used to represent spatial structures with a non-trivial initialization phase. The lifetime of a two phase container is separated into a *generation phase* and an *access phase*. The transition from one to the other phase is marked by a call to its `freeze` method. Here are some reasons why such a concept is useful for the implementation of finite element methods:

1. An adaptive finite element method can be considered as a succession of *generation* and *computation* phases (cf. [6]). The same holds for the underlying patterns of sparse matrices.
2. The necessary initializations are usually too complex to be performed in one call of a C++ constructor. Extending the initialization phase helps to make the change of data structures transparent to the application programmer.
3. Communication that might occur while generating distributed meshes can be delayed until the call of the `freeze` method.

Another important concept is that of an *associated container* that is used for the representation of numerical functions on (distributed) spatial structures.

An overview of these containers and their usage in a sequential context are represented in section 2. Aspects of the parallel implementation and optimizations that are enabled by the use of two-phase containers are discussed in section 3. We explain how the use of two phase containers allows one to analyze irregular communication patterns as they occur in finite element sparse matrices. This is an important optimization for the iterative solution of parallel finite element problems.

2 Concepts and Classes in Janus

Both from a conceptional and implementation point of view Janus is based on the containers and algorithms of the Standard C++ Library [1] – also known under the name Standard Template Library (STL) [2]). The STL is not only a collection of fundamental data structures, generic classes, and algorithms. It defines *concepts*, i.e. generic sets of type requirements and its container classes are models of these concepts, i.e., they are types that satisfy these requirements. The idea is that: "Using concepts makes it possible to write programs that cleanly separate interface from implementation" [2].

2.1 Two Phase Containers

A *two phase container* is a variable-sized container that supports insertion of elements. However, all insertions must have been finished *before* any element of the container can be *accessed*. Only *non-mutating* access is allowed.

The phase in which insert operations are allowed is called the *generation phase* or *first phase*. The phase in which access is allowed is called *access phase* or *second phase*. The transition from the first to the second phase is marked by a call to the `void freeze()` method of a two phase container.

Containers that follow these type requirements represent application objects that have a non-trivial yet clearly distinguishable initialization procedure. Typical examples are finite element meshes or sparse matrices patterns whose structure is not known at compile time.

A two phase container can be "frozen" only once and it has no `thaw` method that would allow new insertions. This means that a two phase container cannot be used to implement meshes that are meant to be modified after their initialization. However, this is only an apparent restriction, since from a conceptional point of view it is often easier to represent mesh modification by creating a new mesh out of an existing one (cf. [6]).

The `FixedSet` Container Family provides two template classes that are models of the two phase container concept, namely the `OrderedFixedSet` and the `HashedFixedSet` templates. For both containers it holds that no two of their elements may be the same.

The main difference between these two classes is that the `OrderedFixedSet` uses the STL `set` template class to initially store its data, whereas the other one is be implemented by the STL `hash_set` container. Both containers provide read-only random access to their elements. This is very natural since in the second phase, i.e., when the container is frozen, it is no problem to number its element from 0 to `size()`. This property of two phase containers can be exploited for a very efficient implementation of vector classes for finite element methods which is explained in section 2.2.

Note that the actual details of the representation of the sets (red-black tree or hash table) are hidden from the user. It is very easy to switch between both implementation strategies or even to mix them. This is in contrast to implementation strategies that expose such low-level details to the application programmer [8].

Use of Two Phase Containers. The following code fragment (see figure 1) shows a typical use of a two phase container. Given the triangles of a finite element mesh, its nodes (in this particular case the vertices of the triangles) shall be generated. We use a six-tuple of integers to denote triangles and their nodes. This allows us to express the triangle-vertex relation by simple index arithmetic [4, 5]. To get the vertices of a triangle on a certain level of an adaptively refined mesh the short inline function `vertices` must be called. This is done for each

triangle and the resulting nodes are inserted in the node set `nodes`. Even if the same node is inserted several times the implementation of the container assures that it occurs only once.

```
typedef OrderedFixedSet<Index<6>,less<Index<6> > > Triangles;
typedef OrderedFixedSet<Index<6>,less<Index<6> > > Nodes;

Nodes create_nodes(int level, const Triangles& triangles)  {
  Nodes nodes;
  Triangles::const_iterator i;
  for(i = triangles.begin(); i != triangles.end();i++)  {
     Tuple<Index<6>,3> v = vertices(*i, level);
     for(size_t j = 1; j <= 3; j++) nodes.insert(v[j]);
  }
  nodes.freeze();
  return nodes;
}
```

Fig. 1. Using a two phase container for the representation of the nodes of a finite element mesh

After all vertices have been inserted the `nodes` container is frozen. In case of an `OrderedFixedSet` container its elements are copied from a STL-`set` container that was used during the initialization phase to a dynamically allocated fixed-size array represented by the STL-`vector` container.

The `FixedRelation` Container Family consists of two phase containers to represent relations between two sets. Therefore they store pairs of elements of other sets. Except for some additional methods and type information about the underlying sets the interface of these classes is the same as for `FixedSet` containers.

There is a special member of this family called `IndexedFixedRelation`. When calling `freeze()` the position of the components of its pairs with respect to the underlying sets are determined. It is shown in section 2.3 how this can be used for the efficient implementation of sequential finite element methods.

2.2 Associated Containers

Associated containers are primarily used for the efficient representation of numerical functions on sets represented by two-phase containers.

An associated container is by definition a random-access container whose size is determined by that of a another container that represents the underlying set. When an associated container is initialized it gets a reference to its underlying set object which must be a fixed-size container or a frozen two phase container. This

allows for efficient storage of the elements, for example the STL `valarray<T>` could be used.

Elements of associated containers can be accessed by random access or by access through elements of the underlying set (the `at` method).

The `SetArray` class template is Janus' standard model of an associated container. It offers no direct support for numerical operations. These services are provided by the template classes `SetVector` and `SetMatrix` which are wrappers around `SetArray`. The main difference between both containers is that `SetMatrix` requires that the underlying set is a member of the `FixedRelation` container family.

2.3 Interaction of Two Phase and Associated Containers

For each triangle the average value of a grid function `u` on the index set `nodes` shall be computed and stored in a grid function `x` on the index set `triangles`. To determine the vertices of a triangle we use the `vertices` method again. Note that we iterate over the triangles through random access to `x.set()` which returns a reference to `triangles`.

```
void average(int level, const SetArray<Nodes,double>& u,
             SetArray<Triangles,double>& x) {
  for(size_t i = 0; i < x.size(); i++) {
     Index<6> triangle = x.set()[i];
     Tuple<Index<6>,3> v = vertices(triangle, level);
     x[i] = (u.at(v[1]) + u.at(v[2]) + u.at(v[3])) / 3.0;
  }
}
```

Fig. 2. Implementation of the function `average`

The implementation shown in figure 2 looks quite appealing, but there are two problems with this usage of the `at` method.

The first problem is that the `at` method won't work in the parallel case because the data that it tries to access may reside in another computational domain and Janus does not support (for performance reasons) remote accesses to *individual* elements. The second problem is the overhead even in the sequential case since a call of `at` causes a non-trivial search in the underlying set.

A solution to both problems is to compute in advance the relation between the triangles and their vertices and to store them in a variable of type `Tuple<Triangles_Nodes,3>`, i.e., we consider the triangle triangle-vertex relation as three separate relations.

Note that in the example in figure 3 the template `IndexedFixedRelation` (mentioned in section 2.1) is used. The precalculated positions can be accessed through the methods `index1(size_t)` and `index2(size_t)`. This means that in the sequential case the `average` procedure can be implemented as follows.

```
typedef IndexedFixedRelation<Triangles,Nodes> Triangles_Nodes;

Tuple<Triangles_Nodes,3>
triangle_vertex(const Triangles& t, const Nodes& n, int level) {
  Tuple<Triangles_Nodes,3> result(Triangles_Nodes(t,n));
  for(Triangles::const_iterator i=t.begin(); i!= t.end(); i++) {
     Index<6>          triangle = *i;
     Tuple<Index<6>,3> node = vertices(triangle,level);
     for(size_t j = 1; j <= 3; j++)
        result[j].insert(make_pair(triangle,node[j]));
  }
  for(size_t j = 1; j <= 3; j++)  result[j].freeze();
  return result;
}
```

Fig. 3. Creation of the triangle vertex relation.

```
void average(const SetArray<Nodes,double>& u,
const Tuple<Triangles_Nodes,3>& r, SetArray<Triangles,double>& x) {
  for(size_t i = 0; i < x.size(); i++)
     x[i] = (u[r[1].index2(i)] + u[r[2].index2(i)] +
             u[r[3].index2(i)]) / 3.0;
}
```

Fig. 4. Revised sequential implementation of the function `average`.

Note that this use of the precalculated indices is nothing more than the traditional "index arrays" that are typically used in Fortran programs. In Janus these helper objects are set up when the container that holds relation is frozen. This computation is therefore transparent to the user.

3 Parallel Environment

With respect to a parallel implementation the programmer should have an SPMD programming model in mind. The library supports expressing data parallelism on the level of meshes. This requires first of all that programmers have a good model to represent mesh nodes and elements. We advocate representations of meshes by so-called index spaces, i.e. sets of integer tuples [4–6].

The great advantage of our indexing technique is that it provides application-oriented global names that are independent from implementation details. This allows to express communication relations independent from the mapping of the indices onto the underlying hardware architecture. The approach of using integer tuples to place and retrieve data recalls the concept of tuple spaces in Linda [3]. However, in Janus these integer tuples are stored in two phase containers whose

access semantics are formed after the usage cycle of finite element meshes. This allows locally fast random access to the data.

3.1 Mapped Containers

In a parallel and distributed environment the finite element meshes have to be distributed over a group of abstract processes which are called *domains* in Janus. As in MPI these processes are denoted by integers [10].

To represent distributed meshes in Janus the programmer uses *mapped* (two phase) containers. Mapped containers have an additional template parameter that serves as a mapping type. As mapping type any class that has a method `domain` can be used that assigns an integer to its argument. The mapped container uses the mapping type to decide to which domain an object that is inserted shall be mapped. The idea of using mapping type template parameters has been taken from the runtime library of the PROMOTER programming model [7, 9]. It gives the user greater flexibility in choosing appropriate mapping strategies.

If an object is inserted into a mapped container then the mapping type is taken to check to which domain the object belongs. If the domain is the same as the one of the mapped container then it is inserted locally. Otherwise, it is put in a temporary buffer. When calling the freeze methods of the mapped container, the temporary buffers are sent to the appropriate domains where the objects are inserted. Delaying the communication is possible since elements are accessed only after the freeze method has been called.

3.2 Communication in Janus

To express communication in Janus it is required that the user explicitly describes which points belong to the underlying mesh relation. Figure 3 showed the example of creating the triangle vertex relation.

Note that in Janus the user describes the relation on the level of mesh points, i.e., in an application-oriented way. When creating a relation the user does not need to specify where the mesh points he refers to are actually stored. This necessary information is obtained by the library from the mapping objects of the mapped two phase containers.

Since the relation itself is stored in a two-phase container it is known that it won't change during its usage. Thus it can be examined before its first use. Analyzing the sparsity patterns allows that message buffers of the right size can be created in advance thus reducing the communication overhead. This is a very important optimization for parallel sparse matrix multiplication which is a key component of iterative methods. They are the preferred method for the solution of large scale finite element problems.

4 Concluding Remarks

We have presented the major concepts of a template library for data parallel adaptive mesh applications. The concept of a two phase container provides

simple, yet sufficient and efficient support for irregular structures such as finite element meshes and sparse matrix patterns. Two phase containers are beneficial in a sequential and parallel context and serve as a useful base for other concepts such as associated containers.

Using two phase containers for the description of mesh relations allows that irregular communication patterns can be analyzed right when they are created.

Currently we use a prototype of Janus for the parallel finite element analysis on two-dimensional meshes. For the solution of the linear systems we use the conjugate gradient method with a simple diagonal preconditioner. In future we will incorporate multilevel preconditioners and adaptive refinement into the solver. The necessary abstractions are already contained in Janus.

References

1. Bjarne Stroustrup: *The C++ Programming Language, Third Edition*, Addison-Wesley, 1997
2. *Standard Template Library Programmer's Guide*, `http://www.sgi.com/Technology/STL/`
3. David Gelernter: *Generative communication in Linda*, ACM Transactions on Programming Languages and Systems, 2(1):80–112, January 1985.
4. J. Gerlach, M. Sato, Y. Ishikawa: *A Framework for Parallel Adaptive Finite Element Methods and its Template Based Implementation in C++*, Proceedings of the 1st International Conference on Scientific Computing in Object-Oriented Parallel Environments, Marina del Rey, CA (1997), Lecture Notes in Computer Science, LNCS 1343, Springer Verlag, 1997.
5. J. Gerlach, G. Heber: *Fundamentals of Natural Indexing for Simplex Finite Elements in Two and Three Dimensions*, TR 97-008, Technical Report of the Real World Computing Partnership, Japan `http://www.rwcp.or.jp/people/jens/publications/TR-97-008`
6. J. Gerlach: *Application of Natural Indexing to Adaptive Multilevel Methods for Linear Triangular Elements*, TR 97-010, Technical Report of the Real World Computing Partnership, Japan `http://www.rwcp.or.jp/people/jens/publications/TR-97-010`
7. Giloi W.K., Kessler M., Schramm, A.: *PROMOTER: A High Level, Object-Parallel Programming Language* Proceedings of the International Conference on High Performance Computing, New Dehli, India, December 1995
8. M. Griebel, G. Zumbusch: *Hash-Storage Techniques for Adaptive Multilevel Solvers and Their Domain Decomposition Parallelization*, Contemporary Mathematics, Vol. 218, pp. 279-286
9. Bi Hua: *Object-oriented Data Parallel Programming in C++* Proc. International Conference on Parallel and Distributed Processing techniques and Applications PDPTA'97, Las Vegas, U.S.A., June 30. – July 3., CSREA 1997, RWC-D-97-015
10. W. Gropp, E. Lusk, A. Skjellum: *Using MPI*, The MIT Press, 1994