Generic Programming for Parallel Mesh Problems

Jens Gerlach* and Mitsuhisa Sato**

GMD-FIRST, Rudower Chaussee 5, 12489 Berlin, Germany

Abstract. We use the paradigm of *generic programming* together with the Standard Template Library (STL) to develop concepts and template classes that can represent finite sets and their relations as they occur in (parallel) finite element and other numerical methods. A key idea of our approach is to consider these sets as static search structures with clearly separated phases for insertions and retrievals. The resulting C++ template library *Janus* has generic `domain` and `relation` classes that allows a space-optimal and quickly traversable representation of finite element meshes and sparse matrix graphs. We discuss the usage of Janus at hand of the parallel implementation of the Poisson equation for linear triangular elements. On the Hitachi SR2201 parallel system we achieve an absolute speedup of 54 on 128 processing elements for the solution of the associated sparse matrix problem with the diagonally preconditioned conjugate gradient method. We also compare our approach with related work such as the HPC++ and POOMA projects.

1 Introduction

The efficient and flexible representation of searchable finite sets and their relations is an important part of programming finite element methods (FEM), sparse matrices, and graph problems. In the following we list the most important characteristics of these structures.

1. The description of the sets is more complex than rectangular grids.
2. The sets and relations might evolve over the process of computation.
3. It is possible to identify clearly separated *modification* and *access* phases. We have coined the term *two phases* for this observation [1].
4. These sets may have a hierarchical or distributed representation and algorithms are often performed in parallel on them.

In this paper we represent the concepts and classes of the C++ template library Janus that directly supports searchable finite sets and their relations. Janus' design is based on the paradigm of *generic programming* - which we will introduce in this section - and its containers have the same *iterator* interface as the containers of the C++ Standard library.

* jens@first.gmd.de

** msato@trc.rwcp.or.jp, Real World Computing Partnership, Japan

Janus can be considered as a small layer on top of the STL and communication primitives such as MPI. To write a complete FEM program more tools as shown in figure 1 are necessary. This also shows that Janus supports the reuse of existing tools and libraries.

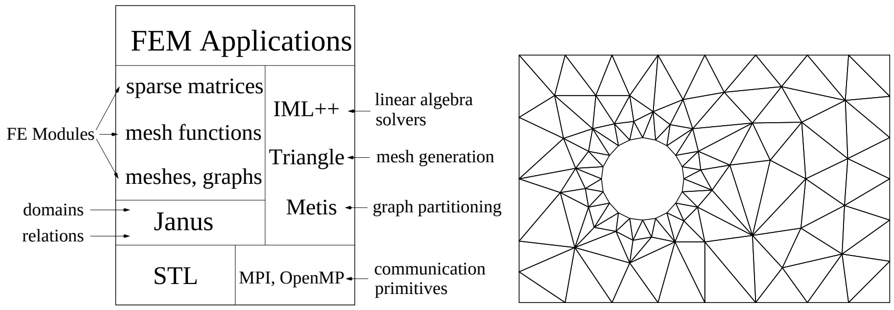

Fig. 1. Janus provides support for domains and relations which are fundamental abstractions for many numerical problems.

Readers of this papers should be familiar with the C++ programming language [9] in particular with the Standard Template Library (STL) [8]. Nevertheless we think that this work is of interest to any application programmer from the field of high performance parallel computing.

1.1 Generic Programming

For a definition of the term "generic programming" we follow D. Musser [5].

> My working definition of generic programming is "programming with concepts", where a concept is defined as a family of abstractions that are all related by a common set of requirements. A large part of the activity of generic programming, particularly in the design of generic software components, consists of concept development — identifying sets of requirements that are general enough to be met by a large family of abstractions but still restrictive enough that programs can be written that work efficiently with all members of the family.

A very well known example of applying the ideas of generic programming is the Standard Template Library (STL) [8] which is now a part of the C++ Standard Library. The STL is organized around concepts for containers, iterators, and algorithms. We follow the terminology of the STL by saying that a type that fulfills the requirements of a concept C is a "model of C".

1.2 A Short Tour of Janus

Janus introduces two major new concepts that are not directly covered by the STL, namely Domain and Relation. Janus also provides four template containers that fulfill the requirements of these new concepts.

A Domain is a *static search structure* [2] that provides random access to its elements (see section 2.1). A static search structure is a type that has clearly separated *insertion* and *retrieval* phases. This fits to the two phases of finite element meshes and graphs of sparse matrices. The transition from the insertion to the retrieval phase is marked by calling the special method `freeze`.

Janus provides two template classes to represent general domains. The first one is `complete_domain` which is tailored for finite sets that reside in a single address space. The template class `distributed_domain` represents a finite set that is partitioned over a fixed group of contexts, in this case, MPI processes.

The second major concept Relation represents the well known mathematical concept of a relation between two sets. We consider relations primarily as *communication relations* and require therefor that they provide a method `apply` that expresses a generalized matrix-vector multiplication see section 2.2.

Corresponding to the two domain classes Janus provides two template classes that are models of the concept Relation. Since we have the nearest-neighbor relations of finite element meshes and other potentially irregular relations in mind, our relation template classes are designed to store pairs of element of two domains.

In section 4 we present speedup results for the conjugate gradient kernel for our discretized FEM problem on the SR2201 massively parallel processor and on a Linux PC cluster using up to 128 processors.

Before drawing conclusions and explaining our future work we compare our approach with two other, well known C++ template frameworks for high performance computing, namely the Parallel STL (PSTL) in the HPC++ project [4] and the POOMA framework [7].

2 Concepts for Finite Sets and Their Relations

2.1 Static Search Structures and Domains

Our fundamental concept to represent finite sets and their relations is that of a *static search structure*. We us the following definition of this term [2].

> A *static search structure* is an Abstract Data Type with certain fundamental operations, e.g., initialize, insert, and retrieve. Conceptually, all insertions occur before any retrievals.

What makes static search structure interesting to us is the clear separation between insertions and retrievals.

A *Domain* is a static search structure that describes a set of n elements which are numbered from 0 to $n-1$. No two elements in a range may be the same. Therefore the numbering is a one-to-one mapping. A type that is a model of the concept domain must implement the following non-mutating member functions.

`size_type size()` returns the number of elements in the domain.
`value_type operator[](size_type k)` returns the k^{th} element of the domain.
`size_type position(const value_type&)` is the inverse of `operator[]`.
`bool contains(const value_type& v)` is true iff `v` belongs to the domain.

The ordering implied by the position of the elements of a domain should be understood as a *local* index. This is important since Janus has template classes that represent groups of domains which are not necessarily contained in the same address space.

We speak of *One Phase Domains* if the elements of the domain can be completely described at creation time. Otherwise we speak of *Two Phase Domains*. The transition from the insertion to the retrieval phase of Two Phase Domains is marked by calling the `freeze` method. A model of the Two Phase Domain concept must implement the following methods.

`void insert(const value_type& v)` inserts `v` into the container.
`void freeze()` switches from the insertion to the retrieval phase.
`bool frozen() const` returns `true` iff `freeze()` has been called.
`const_iterator find(const value_type& v) const` searches the element `v`.

Shortly said, a Two Phase Domains is a Random Access Container [8] with an extended initialization phase that allows only non-mutating access.

2.2 Relations

A primary use of relations in scientific computations is to express communication as generalized matrix-vector multiplication. As an example we consider the nearest neighbor relation on a two-dimensional grid that relates a point (i, j) with the following four points

$$(i+1, j), \quad (i-1, j), \quad (i, j+1), \text{ and } (i, j-1).$$

A typical usage of this relation is to describe stencil operations on two-dimensional arrays `a` and `b` as they occur in finite difference methods.

```
a[i][j] = b[i+1][j] + b[i-1][j] + b[i][j+1] + b[i][j-1]
```

This is our motivation to consider relations from an algebraic point of view. Let $b : B \to V$ a function and $\oplus$ a binary operation on V. Let $R \subset A \times B$ a relation then we define $R \oplus b$ as the function $a : A \to V$ for which

$$a(i) = \bigoplus_{(i,j)\in R} b(j) \tag{1}$$

hold. We assume that there is a default value 0_V in V that is taken to define $a(i)$ for points i that do not take part in the relation R. Note that equation 1 can be considered as a matrix vector multiplication where the matrix has the value 1 for pairs (i, j) that belong to the relation R and 0 otherwise. The use of relations as factors in communication products has been inspired by the PROMISE programming model [12].

The requirements of the concept Relation are based on this definition. A type `Rel` that is a model of the concept relation must have a template member function `apply` with the following signature.

```
class Rel {
public:
   template<class source_it, class target_it>
   void apply(source_it s, target_it t) const;
};
```

The types `source_it` and `target_it` must be random access iterators [8] to types that represent the functions b and a. The user can specify the binary relation $\oplus$ used for accumulating elements. The version of `apply` which is shown above uses the operator `+`. In contrast to `source_it` the iterator `target_it` must allow mutating access. The size of the containers to which the arguments of the `apply` method point must be the same as the domains between the relation object exist. They can be retrieved by the constant member functions `size_domain1` and `size_domain2`. Note that all a relation knows about its domains is their types, value types, and sizes.

3 Domain and Relation Template Classes

In this section we describe the four major template classes of the Janus library which are shown in table 1. Domains and relation classes reside in certain *contexts*. By this we mean fundamental properties of the programming environments such as existence of several address spaces or multi threading.

Concept/Context	Sequential	Parallel
Domain	`complete_domain`	`distributed_domain`
Relation	`complete_relation`	`distributed_relation`

Table 1. The domain and relation templates of Janus

The main difference between these `complete` and `distributed` classes is that the latter ones are implemented on top of MPI and can be easily used in data parallel programs.

3.1 Details of the Domain Template Classes

The class `complete_domain<T>` is implemented as a sorted array which ensures compact storage and an logarithmic average time to search elements. The implementation uses the `vector` container of the STL and its `binary_search` algorithm to implement the method `complete_relation::find`. Note that sorting and removing duplicated elements happens when calling the `freeze` method. This means that these operations are almost transparent to the user and are performed only once.

The template class `distributed_domain<T,Compare = less<T> >` describes a domain that is *distributed* over a fixed group of MPI processes. Locally the class behaves similar to the `complete_domain` template.

The context (process) to which an element of type `value_type` shall be mapped must be specified at insertion time. The communication that is necessary to put the elements in the desired contexts is performed when this container is frozen. This means that the low level details of building and maintaining this distributed data structure are hidden from the user.

Note however, that access to elements is strictly local, that is, this container provides only local and no *parallel* iterators such as the `distributed` containers of the Parallel STL (see section 5). In Janus we strive for the use of iterators in local contexts and offer the concept of Relations (see section 2.2) to access groups of objects.

3.2 Details of the Relation Template Classes

Janus' relation classes `complete_relation` and `distributed_relation` are also models of the concepts Two Phase Domain. They can explicitly store the ordered pairs an irregular relation which makes them very attractive for the implementation of FE programs. Both classes provides not only the member `apply(source_it, target_it)` as required by the concept Relation 2.2, but also has the member template function

```
template<class matrix_it, class source_it, class target_it>
void apply_matrix(matrix_it m, source_it s, target_it t) const
```

that implements a matrix-vector multiplication $a(i) = \sum_{(i,j)\in R} c(i,j)\cdot b(j)$ where the matrix c is stored in the random access container to which the iterator `matrix_it m` points. The order in which the matrix is represented must be equal to the one in which the elements of this relation container is stored. For the latter the Compressed Row Storage (CRS) format is used. As in the case of the class `complete_domain` the final storage scheme is build transparently to the user when the `freeze` method is called.

3.3 Arrays in Janus

As in the POOMA [7] framework we consider an *array* as a function that maps from a input domain to a range of outputs. Since domains are linearly ordered we can fall back on one-dimensional arrays to represent general arrays on them. A value in an one-dimensional array and a particular element of a domain are associated if they have the same position in their respective structures. Since random access is supported only local within each context this implies that an array is distributed according to the underlying domain.

This simple and and intuitive interface makes it unnecessary to provide an own array class in Janus. We encourage the use of standard containers like the `valarray` template. This template class offers a variety of overloaded operators and mathematical functions, several ways to extract subsets of arrays and is designed to be free of certain types of aliasing which has been considered as a problem of C/C++ for high performance computing [9, section 22.4]. As of the time of writing not all compilers support the `valarray` template. Therefor we have written a simple wrapper `numeric_array` around the standard container `vector`.

4 Application to Parallel FEM

We have written a simple program that can read and refine two-dimensional triangulations, build the element matrices of the Laplace operator for linear triangle elements, and solve the assembled sparse algebraic system with the conjugate gradient method. Diagonal scaling is used as preconditioner.

The triangulations were generated and displayed with the Triangle tool set developed by J.R. Shewchuk [10]. The partitions of the triangulations were generated with the Metis tool set developed by G. Karypis [11].

4.1 Performance Measurements

The program was run on the SR2201 Hitachi massively parallel processor using up to 128 processors. As C++ compiler g++ (version 2.8.1) with the optimization option "-O3" has been used. We also used up to 128 processors of the Linux PC cluster of the Real World Computing Partnership. Here we compiled with the g++ (version eggs-1.1.2). Again we used the option -O3.

On the Hitachi machine we used a vendor supplied MPI whereas on the PC cluster we used a modified MPICH that supports the Myrinet communication network.

For benchmarking the triangulation of a square was used. The triangulation was partitioned using Metis and then regularly refined. This emphasizes that our program can handle irregular triangulations.

The problem mesh consisted of 131,072 triangles and 66,049 nodes. The assembled system matrix had 460,289 entries. In the benchmark only the time for the preconditioned conjugate gradient method was measured. We compared a pure sequential version that does not include any call to parallel communication primitives and a parallel version that was run on 1, 2, 4, . . . , 128 processors.

The runtime (in seconds) of the preconditioned conjugate gradient method is shown in table 4.1. This table also includes the absolute speedup defined as the ratio of the runtime of sequential version to a particular version.

The absolute speedup of 54 when using 128 processors on the SR2201 is quite a good result. On the PC cluster the highest speedup is 16 when using 32 nodes. Here the parallel version is on one processor almost two times slower than the sequential version. This makes the result on the PC cluster look much worse. We

Processing Elements	time (SR2201)	speedup	time (PC-cluster)	speedup
sequential	128.445	1.0	92.9899	1.0
1	140.119	0.917	180.956	0.514
2	68.0119	1.889	80.4036	0.865
4	34.7339	3.698	36.6468	2.537
8	17.6185	7.290	17.8723	5.203
16	9.04789	14.196	9.03195	10.296
32	4.81748	26.662	5.51059	16.875
64	3.27404	39.231	7.42557	12.523
128	2.34185	54.848	22.2499	4.179

Table 2. Absolute speedup of the conjugate gradient method (using diagonal scaling as preconditioner) on the SR2201 Hitachi massively parallel processor and a Linux PC Cluster. All times are in seconds.

think that this is related to better optimization strategies that are implemented in the egcs versions of the gcc.

5 Relations to HPC++ and the POOMA Framework

The Parallel Standard Template Library (PSTL) of HPC++ is a parallel extension of the C++ Standard Template Library [4]. The PSTL provides distributed versions of the STL container classes. These new containers have the prefix `distributed_` in their name and provide beside the standard iterator interface *parallel iterators* which are a generalization of global pointers. The PSTL also includes a multidimensional array class which supports element access via standard array indexing and parallel iterators.

Like their sequential counterparts the distributed containers support insertions and deletions of elements. This is different to our container classes which only supports insertions. Although this sounds like a major restriction it is not since our domains are static search structures (see section 2.1) whose usage pattern is conforming with the requirements of finite sets and relations as they occur in numerical methods.

POOMA [7] is a application-oriented framework that is organized into several abstraction layers and is implemented in ISO-C++. On the lowest level POOMA provides generic containers to model domains and arrays. This is the level on which POOMA and Janus can be compared.

POOMA offers several template classes to represent different kinds of subgrids of Z^n. It is not clear how complicated spatial structures such as finite element meshes can be efficiently represented within this framework.

POOMA takes a sophisticated two layered approach to represent arrays on these domains. On the lower level there is the concept of an *Engine* which provides a common interface for randomly accessing and changing data. How the data are stored and indexed is hidden by the particular engine type. On the higher level there is the template class `Array` that describes a logical rectangular, N-dimensional table of elements. There is no performance penalty in this indirection because the gap between the array and its engine template parameter is resolved at compile time.

As explained in section 3.3 we took a different approach for representing arrays in Janus. We experienced that the explicit decoupling of domain and array types reduced the compilation time considerably. Long compilation times and big executables are still a problem when using template libraries in particular when they contain many closely intertwined template classes and functions.

6 Conclusions and Future Work

In this paper we have presented the C++ template library Janus which offers container classes that are useful for the compact and efficient representation of finite sets and relations. Our containers are models of the Domain and Relation concepts that are refinements of STL concepts.

One key to understand our approach is to realize that a static search structure is usually flexible enough to capture the dynamicity of finite element meshes and sparse matrix graphs. Another fundamental property of our approach is that we consequently use relation types to express communication.

Different to many other C++ libraries for scientific programming Janus provides no general multi-dimensional arrays class. Since domains have random access we can use standard arrays like `valarray` to represent functions on domains. This is more flexible since our domain classes are not restricted to rectangular, grid-like structures.

One constraint in the design of our FEM program was that it should always be possible to built a purely sequential version. Using Janus it is easy to switch between the two implementation modes. Basically the user must replace all classes with prefix `distributed_` against their counterparts with the prefix `complete_`. This change can be localized when using typedefs instead of the "raw" template names.

As an example of using the concepts and classes of Janus we explained it can be used to implement a simple two-dimensional FEM problem. For the highly parallel numerical kernel we achieved an absolute speedup of 54 when using 128 processors of a Hitachi SR2201. On a Linux PC cluster we achieved maximal absolute speedup of 16 when using 32 processors.

Our future research focuses on the following topics.

Provide hash based implementations of domains The implementation of the Janus domain and relation classes rests on *sorted arrays*. We also want to provide implementations that are based on *hash tables*. Together with good hash functions this could accelerate the search functions. Recently there is an interest in using hash based representations of finite element meshes [13]. An advantage of Janus would be that the details of this new domain type could be hidden behind a general interfaces.

***Thawing* of frozen domains** Currently a domain object can only frozen once. Adaptive finite element methods can be represented by using several domain objects. It might be more natural for users if they could *thaw* a frozen domain that represents a triangulation, remove unnecessary and insert new elements, and finally `freeze` the container again.

Multithreading and OpenMP Our classes do currently not support multithreading environments directly. However, using OpenMP [6] this could be done within in the domain and relation classes, mostly transparent to the user.

References

1. J. Gerlach, M. Sato, Y. Ishikawa: *Janus — a C++ Template Library for Parallel Dynamic Mesh Applications*, Proceedings of the Second International Symposium, ISCOPE 98, LNCS 1505, pp. 215–222, Springer Verlag, Berlin, Germany
2. *Users Guide to gperf*, `http://www.gnu.org/manual/gperf-2.7/gperf.html`
3. *Iterative Method Library*, `http://129.6.88.11/iml++/`
4. E. Johnson, D. Gannon, *Programming with the HPC++ Parallel Standard Template Library*, SIAM Conference on Parallel Processing for Scientific Computing, March 1997, Minneapolis, Minnesota
5. D. Musser, *Generic Programming*, `http://www.cs.rpi.edu/~musser/gp/index.html`
6. *The OpenMP Application Programmer Interface* `http://www.openmp.org`
7. J.C. Cunnings et al. *Rapid Application Development and Enhanced Code Interoperability using the POOMA Framework*, Accepted for presentation at SIAM Workshop on Object-Oriented Methods and Code Inter-operability in Scientific and Engineering Computing: `http://www.acl.lanl.gov/pooma/papers/SiamOO98.ps`
8. *The Standard Template Library*, `http://www.sgi.com/Technology/STL/`
9. Bjarne Stroustrup: *The C++ Programming Language, Third Edition*, Addison-Wesley, 1997
10. J.R. Shewchuk: *Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator*, First Workshop on Applied Computational Geometry (Philadelphia, Pennsylvania), pages 124-133, ACM, May 1996
11. G. Karypis, *Metis a Family of Multilevel Partitioning Algorithms* `http://www-users.cs.umn.edu/~karypis/metis/main.shtml`
12. H.Bi, G. Kessler, M. Wilhelmi: *Object-oriented Run-time Support for Data-parallel Applications.* Proc. of the 1998 International Scientific Computing in Object-oriented Parallel Environment Conference ISCOPE'98, LNCS 1505. pp. 175-182. Dec. 8-11, 1998, New Mexico, USA
13. M. Griebel, G. Zumbusch, *Hash-Storage Techniques for Adaptive Multilevel Solvers and their Domain Decomposition Parallelization.*, Proceedings of Domain Decomposition Methods 10, editor: J. Mandel, C. Farhat, X.-C. Cai, Contemporary Mathematics 218, AMS, 1998, pp. 279-286