Approaches to Parallel Generic Programming in the STL Framework

George Fletcher, Sriram Sankaran
Computer Science Department, Indiana University
{gefletch,ssankara}@cs.indiana.edu

July 16, 2003

Abstract

While tremendous progress has been made in developing parallel algorithms, there has not been as much success in developing language support for programming these parallel algorithms. The C++ Standard Template Library (STL) provides an opportunity for extending the concept of generic programming to the parallel realm. This paper discusses the basic requirements for extending STL to provide support for data-parallelism in C++. The ultimate goal is to implement a parallel library that is built within the existing framework of STL and exploits parallelism in existing sequential algorithms and also provides a set of parallel algorithms.

1 Introduction

During the last twenty years, tremendous progress has been made in developing parallel algorithms. Unfortunately, there has not been as much success in developing language support for parallel programming. In this paper we consider the C++ Standard Template Library (STL) as a framework for data-parallel programming. We have chosen the STL because it is a well developed library based on the generic programming paradigm. We first investigate previous attempts to introduce data-parallelism into the generic programming framework and then propose two approaches that we feel overcome some of the conceptual weaknesses of these attempts.

The paper is organized as follows. We begin in section 2 with an overview of the principles of generic programming and the STL (section 2.1), and conclude with a brief discussion of data-parallelism (section 2.2). Next, in section 3, we discuss previous attempts at developing generic data-parallel libraries in C++. Section 4 explores two approaches to generic data-parallel C++ that we have developed. Finally, we provide closing remarks in section 5.

2 Background

We begin by introducing the basic concepts of generic programming and data-parallel languages in this section.

2.1 Generic Programming

Generic programming is founded on the insight that, like data structures, algorithms have natural abstractions [4, 14]. This insight brings to light a new framework for software development in which

algorithms are abstracted away as much as possible from particular implementation details. This is accomplished by abstracting and parameterizing algorithms and the data structures that they operate over. In generic programming parlance, abstracted, parameterized algorithms are called *concepts*. Concepts are *modeled* by the data types for which they are well defined (i.e., by the data types that are naturally parameterized over). Key is the decoupling of an algorithm from any particular primitive type; the algorithm is generalized and parameterized to account for all primitive types for which the concept makes natural sense. This decoupling allows programmers to focus separate attention on the essential operations that an algorithm performs and the essential operations that a datatype that models a particular concept must support.

As a simple example, consider the `maximum(x,y)` function that returns the "greater" of `x` and `y`. A solution for integers would look like:

```
int maximum(int x, int y)
{
  if (x <= y)
    return y;
  else
    return x;
}
```

Of course, the reals and rationals would also need their own `maximum()` functions defined. From a generic perspective, it is clear that the structure of the `maximum()` algorithm will not differ for each of these primitive types. In fact, any type that has the "$\leq$" relation defined on it models the `maximum()` concept. A generic solution for any such type `T` would look like:

```
T maximum(T x, T y)
{
  if (x <= y)
    return y;
  else
    return x;
}
```

This generic solution can be viewed as a template that a compiler can fill in, freeing the programmer from any concerns except the basic modeling requirements of `maximum()`.[1] It is important to note that this single algorithm now replaces a whole group of special case algorithms. This can be viewed as the major goal of the generic programming approach.

The generic programming framework is geared towards providing lasting, reliable, and efficient solutions to the building blocks that frequently occur in large pieces of software. The idea is to develop a well-built generic algorithm for each of these commonly used building blocks. These generic solutions (concepts) can then collectively serve as the core of a reusable and extensible library that supports highly modular programming.

Generic programming has been developed and advocated in large part by David Musser and Alexander Stepanov. Their first attempt to build a generic library was with the language Ada [13]. They extended this work in the C++ language in the form of the Standard Template Library (STL). This is currently the most extensive and well developed library built around generic programming principles [16].

[1] We could also go one step further towards genericity by parameterizing the predicate function. We will discuss how this is done in the STL shortly.

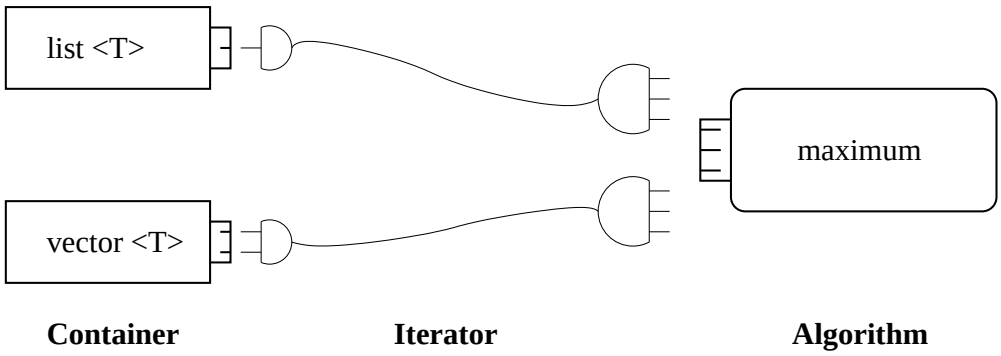

Figure 1: Iterators as generic operators that link algorithms with containers

The STL breaks the generic programming framework down into the following components:

- *Containers.* These are the basic data structures: list, vector, deque, set, multiset, map, and multimap. Each is implemented as a class template parameterized by a primitive type. The programmer specifies the type, such as `int` or `double`, and the compiler fills out the template, obviating the need to have a special class for each container and type combination.
- *Generic Algorithms.* These are the generic concepts, implemented as function templates, and are also filled in by the compiler. They are highly parameterized, and consequently very flexible.
- *Iterators.* The decoupling of generic algorithms from containers is achieved in the STL through iterator objects (see Figure 1). These are basically generalizations of pointers, and are the intermediaries that bring containers and algorithms together. All access and manipulation of a container is done through iterators. There are five categories of iterators: *random access, bidirectional, forward, input, and output.* Each category provides certain operations. For example, forward iterators support increment ($++$), but not decrement ($--$), while bidirectional iterators support both operations. Generic algorithms are mainly parameterized by iterators, and never deal with the details of particular containers.
- *Adaptors.* These objects alter containers, iterators and function objects, providing the means to extend the functionality of these basic STL components. For example, container adaptors can be used to implement a stack with a list or vector container, and iterator adaptors can be used to implement a reverse iterator to traverse a container in reverse order.
- *Allocators.* These objects encapsulate the details of memory management for containers. A custom allocator can be used with any container to implement particular memory allocation/deallocation schemes.
- *Function Objects.* These are objects that wrap up a function (in `operator()`) and are used as parameters to generic algorithm templates. These serve the purpose of function pointers. In the previous `maximum()` example, we could have generalized the test condition in the if-clause by parameterizing the predicate $\leq$.

This structure is highly flexible and modular. It does a good job of fulfilling the generic ideal of not "reinventing the wheel" for each and every possible data structure/algorithm/primitive-type combination.

As an example of using the STL, consider the inner product concept. This algorithm is modeled by vectors and lists of objects for which addition and multiplication are defined (actually the two binary operations can be specified by the programmer via function objects as discussed above), and has the following prototype:

```
template <typename InputIterator1, typename InputIterator2, typename T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T initial_value);
```

A simple example that uses this generic algorithm will illustrate how the STL is used:

```
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>
using namespace std;

int main()
{
  int x[5] = {1, 2, 3, 4, 5};
  int y[5] = {5, 4, 3, 2, 1};

  vector<int> v1(&x[0], &x[5]);
  vector<int> v2(&y[0], &y[5]);

  int out = inner_product(v1.begin(), v1.end(), v2.begin(), 0);

  cout << "result = " << out << endl;
  return 0;
}
```

As expected, the output is:

```
result = 35
```

2.2 Data-Parallel Languages

The two main ways of expressing parallelism in programming languages are dubbed *control* parallelism and *data* parallelism [10]. In a control-parallel language, the programmer has explicit constructs for distributing data and work and for synchronizing processes. A data-parallel language is characterized by support for a simultaneous operation by all nodes in a system on different subsets of a large set of data. Unlike control-parallelism, synchronization is implicit in data-parallel languages. It is general believed that data-parallel languages are natural for expressing solutions to a large class of problems.

We briefly mention a few prominent data-parallel languages. High Performance Fortran (HPF) is a data-parallel variant of Fortran 90 [9]. It is currently in use on the NEC Earth Simulator in Yokohama, Japan. High Performance Java (HPJava) is another data-parallel variant of a standard language [3]. HPJava is currently under development at the Community Grids Lab at Indiana University. C*, an extension of ANSI C, is probably the most mature and well known data-parallel

language. It was originally developed for the Thinking Machine Corporation's CM-1 in the mid-80's [2].

To get a instructive feel for programming in a data-parallel language, let's look at a small program for matrix-matrix multiplication in HPF. This example captures the major aspects of this approach.

```
      program matrix_multiplication
        real A(N,N), B(N,N), C(N,N)
        integer i, j

!HPF$ PROCESSORS nodes(4, 4)
!HPF$ ALIGN A(i,*) WITH C(i,j)
!HPF$ ALIGN B(*,j) WITH C(i,j)
!HPF$ DISTRIBUTE C (BLOCK, BLOCK) ONTO nodes

!HPF$ INDEPENDENT
        do i = 1, N
          do j = 1, N
             C(i,j) = DOT_PRODUCT(A(i,:), B(:,j))
          end do
        end do
      end
```

The `PROCESSORS` directive gives a hint to the compiler about how many nodes are in the virtual system and how they are organized. In this system we have 16 nodes in a 4x4 grid. The `ALIGN` directive suggests how array elements should be co-located for locality benifits. For matrix-matrix multiplication, each element $C(i, j)$ is dependent on the ith row of A and the jth column of B. Hence, we align these arrays with $C(i, j)$. The `DISTRIBUTE` directive instructs how arrays should be distributed in the system (here as two-dimensional blocks). Finally, the `INDEPENDENT` directive informs the compiler that the do-loop that follows has independent iterations that can be performed in parallel.

3 Related Work

There has been a lot of work in the past in providing support for both generic programming principles and data parallelism in C++. This section describes some of these approaches.

The Amelia Vector Template Library (AVTL) [15] provides a data-parallel programming model for C++. It provides a templated distributed vector class, generic vector algorithms, generic function objects, and a memory manager. Contrary to the philosophy of the STL, the entire vector is passed to the generic algorithms rather than passing vector iterators. Iterators are not provided at all; instead functions may be applied to the aggregate as a whole or element-wise. Vectors are distributed across contexts in equal-sized blocks. No mechanism is provided for alternative distributions. Furthermore, AVTL vectors are static, that is, elements cannot be inserted or removed after creation.

pC++ [7, 5] is a data-parallel extension to C++. It provides distributed *collections*. Users can define element-wise functions as well as aggregate functions. Distribution of elements is handled in a way similar to High Performance Fortran [9], except that distribution can be specified at runtime

in pC++. Distribution objects define a mapping of a grid to the actual machine processors and then alignment objects specify the mapping of the collection elements to the grid.

STAPL [6] is a parallel C++ library that was intended to be a superset of STL, and is implemented using simple parallel extensions of C++ for parallel containers and algorithms (called `pContainer`s and `pAlgorithm`s) and an entirely new construct called `pRange` which provides random access to elements in a `pContainer`. Analogous to STL iterators, `pRanges` bind `pContainers` to `pAlgorithms`. Unlike STL iterators, `pRanges` also include a *distributor* for data distribution and a *scheduler* that can generically enforce data dependences in the parallel execution according to execution data dependency graphs (DDGs). STAPL allows STL containers and algorithms to be used together with STAPL `pContainers` and `pAlgorithms`.

The HPC++ Parallel Standard Template Library (PSTL) [11, 12] is closest in philosophy and function to the approach suggested in this paper. HPC++ provides an extension of generic programming, as embodied by STL, into the parallel realm. HPC++ defines a set of standard distributed data-structures, a parallel iterator, and parallel algorithms. The details of the design and implementation of HPC++ are described in section 3.1, and provides the context in which the approaches described later in section 4 are based.

3.1 HPC++ and PSTL

The HPC++ framework describes a C++ library along with compiler directives which support parallel C++ programming. The HPC++ framework consists of the following parts:

- parallel loop directives (to support single-context parallelism),
- a parallel Standard Template Library, and
- a multidimensional array class.

The Parallel Standard Template Library (PSTL) is a parallel extension of the C++ Standard Template Library (STL). Distributed versions of the STL container classes are provided along with parallel algorithms and parallel iterators. HPC++ also includes a multidimensional distributed array container that supports element access via standard array indices as well as parallel random access iterators, allowing use of STL and PSTL algorithms on the array class.

3.1.1 PSTL Run-Time Environment

The PSTL run-time environment (RTE) uses Tulip [1] as the underlying system. Tulip runs on various platforms and provides support for remote member function invocation and load/store operations on remote data. A key feature of Tulip is that the system provides a consistent interface across platforms yet its implementation takes advantage of machine-specific hardware features. For example, on shared memory machines data is fetched or stored via global pointers using `memcpy`, and on distributed-memory machines MPI [8] is used to access remote data. Tulip provides several constructs at run-time to support PSTL.

Global Pointers. Global pointers are pointers to (possibly) remote data that encapsulate not only a pointer to the data, but also location information. Global pointers are used to refer to elements in the distributed container. In particular, the parallel iterator for the distributed containers can be cast into a global pointer to a container element. A global pointer to an object of type T is defined as a templated class:

```
HPCxx_GlobalPtr<T> p;
```

Global pointers can be passed between contexts to allow a processor to read and modify objects on a remote node. Two basic operations are defined on global pointers:

```
// returns a global reference which can be used for
// remote stores and fetches
template <class T>
HPCxx_GlobalRef<T> HPCxx_GlobalPtr::operator *();

// casts to a local pointer.  If the object is remote,
// this returns NULL
template <class T>
HPCxx_GlobalPtr<T>::operator T* ();
```

The result of dereferencing a global pointer is a global reference. The primary operations on global references are:

```
// remote store
template <class T>
T HPCxx_GlobalRef<T>::operator=(const T&& rhs);

//remote fetch
template <class T>
HPCxx_GlobalRef<T>::operator T ();
```

The assignment operator performs a simple assignment if the global reference refers to a local object. Otherwise, a remote store operation must be performed using the facilities provided by the RTE.

Remote data access. Access to remote data via global pointers is provided by the following functions:

```
// remote store
template <class T>
void tulip_Put(HPCxx_GlobalPtr<T> dest, const T* src);

// remote fetch
template <class T>
void tulip_Get(T* dest, HPCxx_GlobalPtr<T> src);
```

Besides these, query functions to determine characteristics of the environment such as number of contexts and other location information are needed as well.

3.1.2 PSTL Components

The Parallel Standard Template Library extends the STL to include distributed containers, parallel iterators, and parallel algorithms. A brief description of these components as they are implemented in PSTL follows.

Distributed Containers
Distributed containers are data structures whose elements are distributed across several contexts.

There are seven distributed containers in the PSTL. These mirror the containers in the basic STL. Each container provides a local and global view through iterators which access local and global elements. The PSTL containers use irregular block distribution across contexts. Each context is assigned a unique ID and contexts are ordered by these IDs when determining placement of element blocks. The elements in a particular context form a contiguous segment of the container's element iteration space. The PSTL containers are dynamic — the size will vary as elements are inserted and/or deleted.

Parallel Iterators
The iterators in PSTL are at the heart of the generic programming approach. They extend the functionality of global pointers, and provide the glue between the containers and the algorithms. Parallel iterators are also used to distinguish between calls to sequential algorithms and parallel algorithms. Parallel iterators have the same hierarchy as iterators in STL. Each container class provides methods of the form:

```
template <class T>
class C {
  ....
  class iterator { ... };
  class pariterator { ... };
  pariterator parbegin();
  pariterator parend();
  iterator begin();
  iterator end();
};
```

The *parbegin()* and *parend()* methods return parallel iterators which point to the beginning and one element past the end, respectively, of the container elements. The begin() and end() methods return local iterators which point to the beginning and one element past the end, respectively, of the local portion of the container elements. In order to facilitate effective use of the parallel algorithms, each container has several functions which modify parallel iterators. These include functions to return iterators to the beginning and end of the local section (if any) of an iteration space defined by two parallel iterators.

Parallel Algorithms
As in the STL, the PSTL algorithms are generic — the arguments to the algorithms are not the containers themselves, but rather iterators which access container elements. There are three types of parallel algorithms in the PSTL:

- STL algorithms with parallel semantics,
- *par_* versions of STL algorithms, and
- *par_* algorithms for standard parallel operations.

The algorithms in the first group retain their STL names, but allow *pariterator* arguments in place of *iterators*. These new algorithms are collective and have parallel semantics. The algorithms in the second group are also versions of the STL algorithms, but *par_* will be prepended to their names. When invoked with parallel iterators, these algorithms are semantically equivalent to the first type of algorithm (and are collective). When invoked with local iterators, these algorithms

are not collective. Execution is local to a particular context, but has parallel semantics (so a loop may be parallelized, for example). The differentiation between the versions of these algorithms is accomplished using iterator tags.

4 Data-Parallelism in the STL

In this section, we discuss the requirements for building data-parallel extensions to the C++ language within STL framework. We emphasize that we are interested in using the framework and components of the STL to introduce parallelism, rather than modifying and extending the library as the approaches discussed in section 3 do. This is an initial step towards the ultimate goal of developing a generic C++ library that supports data-parallelism. We examine two possible approaches. Unlike HPC++, neither of these approaches requires parallel counterparts to be built for each base container in STL. In the first approach, we describe the implementation of *distributed allocators* and *iterator adaptors* for each type of STL container that takes advantage of distributed allocation. The second approach is to build a *container adaptor* that adds a parallel dimension to any container developed using the STL.

Before we discuss these approaches, we briefly mention our underlying system model. This work assumes a MPI based RTE of nodes that cooperate via messages to distribute and access data. This model is a generalization of the RTE discussed in section 3.1. Any environment that can support global pointers and remote data access will be sufficient for our approach to data-parallel programming.

4.1 Distributed Allocator and Iterator Adaptor Approach

Each STL container is parameterized by collection type and allocator. As discussed in section 2, allocators encapsulate the underlying memory model used by the container. In a parallel environment with multiple nodes, we need a way to distribute the contents of a container within the system. Unless otherwise specified by the programmer, the data should be distributed evenly among the available nodes. We can accomplish this using a specialized distributed allocator for each base STL container that allocates chunks of data on each node:

```
Container<T, dist_Container_allocator<T> >
```

This distributed allocator creates objects of the base container class on each node, and maintains a table that holds information about the distribution scheme used. This table consists of location information required to access data using global pointers. However, in order to preserve normal iterator semantics and to access data that has been allocated on different nodes by the distributed allocator, the existing iterator in the STL containers must be wrapped in an iterator adaptor that indexes into the table of global pointers:

```
par_iterator_adaptor<Container<T>::iterator>
```

Figure 2 gives a pictorial representation of the scheme. It should be noted that this approach does not require structural extensions to STL, but rather works well within its framework.

4.2 Distributed-Container Adaptor Approach

As mentioned in section 2, STL provides adaptors that can be used to change the interface of containers. Using this facility, we propose another approach for providing support for data-parallelism.

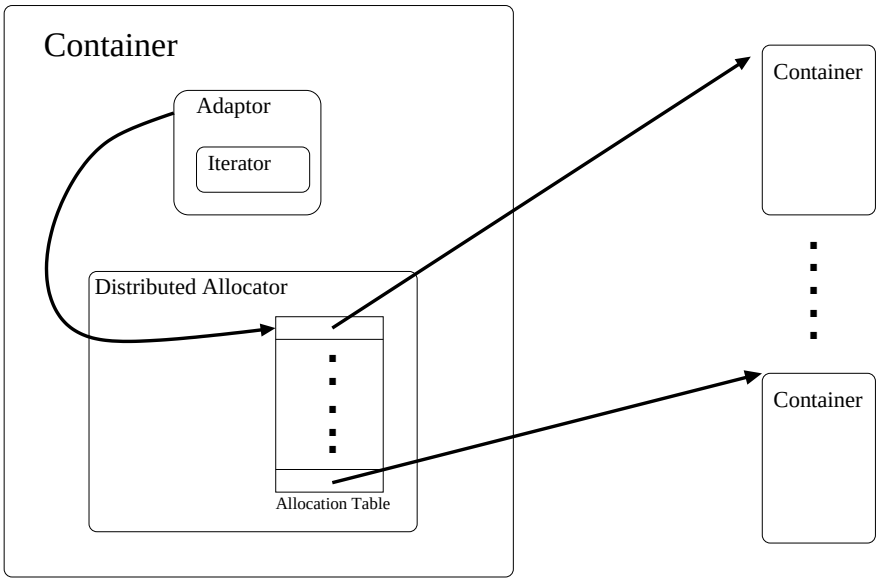

Figure 2: Distributed Allocator and Iterator Adaptor Approach.

In this approach, we define a distributed container adaptor that is parameterized by type and container class:

```
dist_container_adaptor<T, Container<T> >
```

To achieve the same data distribution mentioned above in section 4.1 for the distributed allocator approach, the container adaptor similarly maintains a table of global pointers to the distributed data (standard STL containers at each node). Any operations on the container adaptor's iterator is resolved through this table, as illustrated in Figure 3. The advantage of this approach over the previous one is that we only need a single container adaptor that can be used with all STL containers.

4.3 Future Work

A lot of work remains to be done on these two approaches to introducing data-parallelism into the STL. Most of our work up till now concentrated on coming up with these two new design approaches and evaluating their feasibility. Not much work has been done in working out the implementation details. For example, the support required from the run-time system has not been completely determined. Nor have the implementation details of the container and iterator adaptors and the distributed allocator been worked out. However, it can be seen that the approaches described are certainly workable. Once these approaches are implemented, it will be possible to evaluate them against related work in this field both in terms of performance and in terms of preserving the philosophy of generic programming in general, and STL is particular.

5 Conclusions

In this paper, we undertook a study of data-parallelism in the generic programming paradigm, as embodied in the C++ Standard Template Library. We investigated existing work on introducing

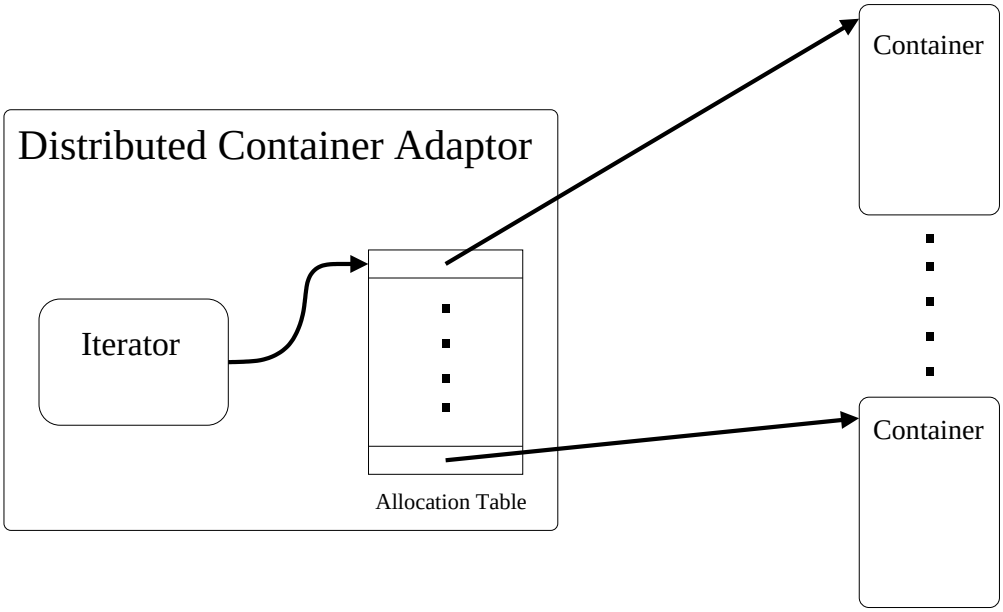

Figure 3: Distributed Container Adaptor Approach.

parallelism in generic programming and noted the shortcomings in many of these approaches. We then proposed two alternate approaches to data-parallelism in the STL which we believe can overcome at least some of these shortcomings. While very little work has been done in proving the validity of the models proposed, there is an indication that these models are both viable and achievable.

We would like to thank Prof. Andrew Lumsdaine for giving us an opportunity to undertake this work, and for the guidance and support provided. We would also like to thank Jeremy Siek and Jaakko Jarvi for their invaluable ideas.

References

[1] P. Beckman and D. Gannon. Tulip: A Portable Run-Time System for Object-Parallel Systems. In *Proc. 10th Int. Parallel Processing Symp (IPPS'96)*, Honolulu, HA, April 1996. IEEE.

[2] P.A. Bigot. pC*: Efficient and Portable Runtime Support for Data-Parallel Languages. Technical Report TR96-8, University of Arizona, Computer Science Dept., 1996.

[3] B. Carpenter and G. Fox. HPJava: A Data Parallel Programming Alternative. *IEEE Computing in Science and Engineering*, pages 60–64, May/June 2003.

[4] J.C. Dehnert and A.A. Stepanov. Fundamentals of Generic Programming. *Generic Programming'98 LNCS*, 1766:1–11, 2000.

[5] F. Bodin et al. Distributed pC++: Basic Ideas for an object parallel language. *Scientific Programming*, 2(3), 1993.

[6] P. An et al. STAPL: A Standard Template Adaptive Parallel C++ Library. In *IWACT*, Bucharest, Romania, July 18-21 2001.

[7] P. Beckman et al. Object-Parallel Programming with pC++. Technical Report CRPC-TR95608, Rice University, 1995.

[8] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, Department of Computer Science, University of Tennessee, 1994.

[9] High Performance Fortran Forum. High Performance Fortran Language Specification, version 1.0. Technical Report CRPC-TR92225, Rice University, 1993.

[10] W.D. Hillis and Jr. G.L. Steele. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[11] E. Johnson and D. Gannon. HPC++: Experiments with the Parallel Standard Template Library. In *International Conference on Supercomputing*, pages 124–131, 1997.

[12] E. Johnson and D. Gannon. Programming with the HPC++ Parallel Standard Template Library. In *Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997.

[13] D.R. Musser and A.A. Stepanov. Generic Programming. *LNCS*, 358:13–25, 1988.

[14] D.R. Musser and A.A. Stepanov. Algorithm-Oriented Generic Libraries. *Software Practice and Experience*, 24(7):623–642, 1994.

[15] T.J. Sheffler. A portable MPI-based parallel vector template library. Technical Report RIACS-TR-95.04, Research Institute for Advanced Computer Science, NASA, March 1995.

[16] A.A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-95-11, HP Laboratories, 1995.