A portable MPI-based parallel vector template library
RIACS Technical Report 95.04

Thomas J. Sheffler [*]

Abstract

This paper discusses the design and implementation of a polymorphic collection library for distributed address-space parallel computers. The library provides a data-parallel programming model for C++ by providing three main components: a single generic collection class, generic algorithms over collections, and generic algebraic combining functions. Collection elements are the fourth component of a program written using the library and may be either of the built-in types of C or of user-defined types. Many ideas are borrowed from the Standard Template Library (STL) of C++, although a restricted programming model is proposed because of the distributed address-space memory model assumed. Whereas the STL provides standard collections and implementations of algorithms for uniprocessors, this paper advocates standardizing interfaces that may be customized for different parallel computers. Just as the STL attempts to increase programmer productivity through code reuse, a similar standard for parallel computers could provide programmers with a standard set of algorithms portable across many different architectures. The efficacy of this approach is verified by examining performance data collected from an initial implementation of the library running on an IBM SP-2 and an Intel Paragon.

1 Introduction

The data-parallel programming paradigm has proven to be popular because of its power and simplicity. While it is not entirely suitable for all parallel applications, a large number of applications are easily expressed in this paradigm. The acceptance of High Performance Fortran (HPF), with its large core of data-parallel array operations, shows that many computer and compiler vendors are committed to providing support for this model in the future [7].

The concept of collections lies at the center of all data-parallel programming languages. In these languages, there are two types of parallelism that can be understood in terms of collections. Simple *elementwise* parallelism is expressed by applying an operation to all of the members of a collection in parallel. *Aggregate* parallelism is expressed as a parallel algorithm defined over an entire collection. Typically, the collections are arrays or vectors, but other collections are possible.

There is a link between collection-oriented programming and object-oriented systems that is often not recognized. Most object-oriented systems provide polymorphic collection classes that manage heterogeneous sets of objects [4]. Traditionally, polymorphism is implemented through a class hierarchy using inheritance (although other mechanisms are possible). Common functionality is provided for a group of

[*] Research Institute for Advanced Computer Science, Mail Stop T27A-1, NASA Ames Research Center, Moffett Field, CA 94035-1000 (sheffler@riacs.edu) The work of this author was supported by the NAS Systems Division via Contract NAS 2-13721 between NASA and the Universities Space Research Association (USRA).

types by “inheriting” the functionality from a base class. All classes derived from this base class then have a minimal set of member functions that a collection class can use to manage its elements.

The Standard Template Library (STL) uses ad-hoc polymorphism and generic classes and functions to provide polymorphic collections. Along with the definition of a small number of collection classes, the STL also provides algorithms on those collections whose implementations run as fast as hand-coded `C` for many applications [11]. Instead of using classes to inherit common functionality, generic functions define operations that the compiler can instantiate for any type. There are two advantages to using generic functions to describe polymorphism instead of inheritance. First, generic functions (if carefully written) may avoid the overhead of calls to member functions, leading to improved performance. Secondly, generic functions can provide new functionality for the built-in types of `C`, which are not classes in `C++` [12]. It is because of these two points that template functions can provide high-level operations on collections of the built-in types and obtain the same performance achieved by hand-coded `C`.

This paper discusses the design and implementation of the Amelia Vector Template Library (AVTL), a polymorphic collection library for distributed address-space parallel computers. Like the STL, it is template based rather than inheritance based. However, because a distributed address-space memory model is assumed, significant restrictions must be placed on the programming model provided by the library. For example, the full generality of the iterators of the STL are not permitted. Instead, a restricted form of access to elements, through `elementwise` functions, provides the necessary safety.

Many collection types exist in data-parallel programming languages. This initial experiment targets only the simplest distributed data type: the vector. Even with only one collection type, there is a significant amount of complexity to be considered. For example, algorithms on vectors often employ algebraic combining functions (*e.g.*, addition in a parallel-prefix algorithm). The library has been carefully designed so that algorithms are generic with respect to all element types and algebraic combining functions. A framework for describing algebraic combining functions is presented that links algebraic combining functions with their identity values, and readily extends to new data types.

The remainder of this paper begins by discussing generic classes and functions, and then introduces the components of the library. Its design emphasizes the orthogonality of *element types*, *collection types*, *algorithms*, and *algebraic combining functions*, and presents a mechanism for the customization of algorithms through function objects. These modify the actions of the algorithms (for both primitive and user-defined types) but do not incur the overhead of a function call, as normal functions do. Examples are used to show the ways in which the components of the library combine, and how they may be extended for user-defined data types. Finally, performance data collected on an IBM SP-2 and an Intel Paragon verify that the initial implementation yields performance comparable to that of hand-coded `C`.

1.1 Algorithmic Templates

Templates are algorithms that may be parameterized by type and function. For example, an algorithm to sort requires knowledge of an element type and comparison function. An algorithm to find transitive closure requires specification of the element type as well as the addition and multiplication functions of the mathematical ring over which to find closure.

A standardized set of algorithms enhances programmer productivity by raising the level of abstraction, while simultaneously providing program portability. Instead of re-targeting an entire program for a new architecture, a standard set of algorithms provided for many architectures ensures the portability of any program written in terms of these algorithms.

Standard function libraries, such as the BLAS [8], are an effort in this direction, but often lack the abilities of polymorphism and function specialization. These capabilities have been lacking in the past because there has not been a widely available programming language that allows the specification of generic algorithms. The C++ function template mechanism provides a good foundation for the encapsulation of the specification of generic algorithms.

1.2 A Case for Standard Parallel Collections

This paper advocates standardizing a set of generic collections and algorithms suitable for distributed address-space parallel computers. To a limited extent, such a library would replace some of the functionality of current data-parallel compilers. Compilers for data-parallel languages, such as those for HPF, are responsible for the instantiation of parallel algorithms on collections. A simple example is the `reduce` function of HPF on arrays. C++ templates provide a way to move this functionality out of the compiler and into a library without sacrificing performance in the way that subroutine libraries often do. By establishing standards for interfaces to parallel collections, it should be possible to experiment with and add new collection types in the future without modification to the underlying compiler.

Of course, many low-level optimizations are beyond the scope of a template library. Such optimizations as loop fusion and array blocking must be handled at a lower level. While many C compilers do not currently implement these optimizations, a growing number are beginning to perform these types of optimizations that have typically been the domain of Fortran compilers [3, 6]. It is reasonable to expect that these optimizations will be commonplace in the C compilers of the next few years. In much the same way the programmers of vector computers write vectorizable code, algorithms in the template library could be written in a scalable style so that compilers can recognize the appropriate optimizations.

Whereas the STL standardizes both interfaces and algorithms, this is not in general possible for distributed address-space parallel computers. Instead, the interfaces may be standardized, but separate implementations may have to be provided for different classes of machine. This initial implementation uses standard C++, and MPI for communication [9], and thus is portable to a wide variety of current distributed address-space parallel computers. However, shared memory multiprocessors and vector multiprocessors present architectures for which an entirely different implementation would be necessary. By standardizing interfaces to functions that have efficient implementations on many architectures, a template library can provide a substrate for the development of portable parallel programs.

1.3 An Initial Implementation

The initial implementation of the library uses MPI (Message Passing Interface) [9] for interprocessor communication, ensuring portability to a wide variety of architectures. C++ templates are used to provide a single generic vector class and many generic algorithms on vectors. Algorithms are parameterized by element types and function objects.

The use of function objects with template algorithms ensures high performance by allowing user defined functions to be compiled in-line, avoiding the overhead of a function call for the application of the function. The compiled code resulting from the instantiation of the template algorithms provide the performance of hand-written C. In addition to simplifying the library design, this capability emphasizes the orthogonality of element types, collection types, algorithms, and algebraic combining functions, and allows them to be combinined in many ways.

Function objects are used uniformly in the library to specify algebraic combining operations for arbitrary data types. For instance, the vector library provides parallel prefix (scan) algorithms for vectors of any homogenous type. For any binary associative operator, $\oplus$, with an identity element $\mathbf{0}$, and a vector a, a scan computes a result b that is defined as

$$\begin{aligned} b_0 &= \mathbf{0} \\ b_i &= b_{i-1} \oplus a_{i-1}. \end{aligned}$$

In the AVTL, there is a single scan algorithmic template and the binary operator and identity element are parameters of the algorithm. Other parallel vector libraries have offered one of two approaches to providing scan functions. To ensure high performance, some provide specialized scan algorithms for a limited number of data types and operators [1]. This approach does not generalize to user defined types. More general libraries have accepted function pointers to allow the user to define any binary associative operator [9]. This approach suffers a performance loss because the repeated invocation of the function may be unacceptably expensive.

Function objects are like function pointers except that the compiler may have complete information about the function so that it can be inlined at compile time. The template-based approach offers the benefits of genericity, efficiency and extensibility. Generic algorithms may be instantiated for any type and function object. Furthermore, users may freely add new element types and combining function objects to extend the vector library with added functionality.

The AVTL comprises four main components that are carefully designed to work together.

1. A memory manager. The memory manager is implemented as a class and has member functions that allocate memory over the available processors in equal sized chunks.

2. A generic distributed collection class: the vector. It is implemented as a template class.

3. Generic algorithms on vectors. These are implemented as template functions. Standard algorithms are elementwise operations, vector permutations, scan and segmented scan operations, reductions, segmented reductions, and combining sends and fetch-and-add communication functions.

4. Generic function objects. These parameterize the vector algorithms to vary the way in which vector elements are combined or fetched.

The library handles homogenous vectors of any fixed-size type. A large number of predefined function objects and algorithms provide standard vector operations on the built-in types of C, but users may easily introduce new element types and function objects.

1.4 An Example

Before delving into the details of the AVTL, a short example will demonstrate some of the features of the library. The vector collection template class is called the pvect (for Parallel Vector). The vector constructor accepts a length argument, and an optional value with which to initialize the elements of the vector. By default, the elements are distributed in equal sized blocks over the available processors.

```
pvect<int>    ones(10, 1);      // length 10, elements set to 1
pvect<double> twos(10, 2.0);    // vector of doubles
```

A vector that enumerates its sites from 0 is called an "index vector." An index vector may be computed from the ones vector by using the scan algorithm with addition as the binary associative operator. The AVTL provides a generic algorithm for scans and a set of standard generic binary associative operators in the form of function objects. These function objects are parameterized by type. An index vector could be created by the following application of the scan algorithm with the addition function object for integers. (Do not be alarmed by the syntax — it becomes familiar quickly.)

```
pvect<int>    index = op_scan(add_op<int>(), ones);
```

A user-defined data type may be used with the AVTL as easily as a builtin type. The following class defines a type that represents a point on the plane in polar coordinates. The class also defines the addition operator for objects of type polar.

```
class polar {
    double theta, mag;
  public:
    polar (double init)  { theta = 0.0; mag = init; }

    polar operator+(polar a)
    {
        polar b;
        double bx = a.mag * cos(a.theta) + this->mag * cos(this->theta);
        double by = a.mag * sin(a.theta) + this->mag * sin(this->theta);
        b.mag   = sqrt(bx*bx + by*by);
        b.theta = arctan(by/bx);
        return b;
     }
};
```

With this new data type and an addition operator, it makes sense to speak of performing an add_scan on vectors of polar elements. Because the addition operator is an inlined member function of class polar, it will be inlined in the instantiation of the op_scan algorithm produced for this type, and the performance of the add_scan function on polar types will be as good as a function hand-written expressly for that purpose. The following call to op_scan will produce the prefix sum of a vector of polar coordinates. The generic combining function, add_op may be applied because addition is defined for the polar type.

```
pvect<polar>   a(100, polar(1.0));  // intialized to 1.0
pvect<polar>   prefixsum = op_scan(add_op<polar>(), a);
```

This is a small example of the extensibility of the library. More interesting examples require an understanding of function objects. In summary, the combination of template algorithms and function objects provide an extensible library with the efficiency of hand written code for any vector element type.

1.5 Organization

The rest of the paper discusses the components of the vector library. Two brief sections introduce the memory manager and the vector template class. The vector class section also describes *segment descriptors* and how they are internally managed. Section 4 describes function objects and why they are important to the library both for expanding its functionality and to ensure high performance. The large number of predefined generic function objects provided by the library make it useful for all of the built-in data types of C++, but it is the extensibility of the library through function objects that makes it interesting.

Section 5 is devoted to describing the element-wise application of functions to conforming collections. Because element-wise operations are often the core of a data-parallel program, it is important that the library provide a convenient way to express them. This section also describes how the library can identify scalar data types in mixed-mode (vector/scalar) calculations.

Section 6 describes the generic algorithms and function objects provided by the library. Section 7 briefly discusses some of the difficulties encountered in interfacing with MPI. This section touches on both the implementation of the messaging passing using MPI, and the problems encountered with interfacing the AVTL template algorithms to the MPI subroutine interface.

Section 8 presents the results of performance tests collected on the Intel Paragon and the IBM SP-2. These results illustrate that the template-based approach yields performance on par with that of hand-coded C. Finally, a concluding section summarizes some of the points made in the paper and suggest directions for future work.

2 Memory Management

The Amelia Vector Template Library AVTL provides a standard integrated memory manager defined by the class `am_mem_mgmt`. A brief description of the function of this class follows.

A request for a vector of a given size is broken into equal sized *chunks* across the available processors. Each chunk maintains an opaque descriptor that records its size, starting index in the allocated vector, beginning address in memory, and ending address in memory (by convention, this is the address immediately following the last element of the chunk). The memory manager returns a pointer to the beginning of the chunk and provides access to the information of the chunk descriptor through access functions.

The memory manager class provides very few member functions. There are methods for allocating and freeing vector memory on the heap, and functions for allocating vector memory on a stack. The stack functions, `push` and `pop` must be nested properly in a correct program. The memory manager also provides methods for printing statistics about the memory use of the program.

Because memory management of distributed vectors is often linked to initialization of the parallel machine, the memory manager assumes the responsibilty of initializing the underlying parallel communication library.

3 The Parallel Vector Collection Class

A parallel vector is an instance of the `pvect` template class, parameterized by an element type. The primary responsibility of the vector class is coordinating the allocation and freeing of vector memory. Each instance of a parallel vector is actually a pointer to a hidden vector descriptor that maintains information about the vector's type, length and location in vector memory. Two or more vectors may share the same descriptor, and thus refer to the same location in vector memory. The constructors and destructors of the class maintain reference counts on vector descriptors to determine when vector memory may be released by the memory manager.

The reference counting scheme minimizes the copying of vector data. When passing vectors as arguments to functions, the only thing that is copied in as an argument or out as a result is a pointer to the descriptor. Vector assignment is likewise defined through the sharing of descriptors. Most vector operations return a new vector so that application of one function does not cause a side effect in another shared vector elsewhere. If a true copy in a new vector is required, the `copy` member function may be used.

There are four constructors for `pvect` vectors. The default constructor does not allocate any vector memory. With a single integer argument, vector memory is allocated for the given number of arguments; an optional second argument specifies an intial value for each element of the vector. The copy constructor for a `pvect` shares the descriptor of the argument and increments the reference count.

The following example illustrates the four different constructors. The destructor for a vector decrements the reference count of the descriptor and frees the allocated vector memory when the count reaches zero.

```
pvect<int>    a;                   // no vector memory allocated
pvect<int>    b(100);              // allocated
pvect<int>    c(100, 5);           // allocated, initialized
pvect<int>    d = c;               // c and d share the same vector memory
```

The other member functions of the `pvect` class are listed below. Some implementations of the `pvect` interface may include other member functions, but these may not necessarily be supported in future releases.

```
template <class T>
class pvect {
    pvect<T> copy();             // produce a copy of the vector
    T        get(int pos);       // retrieve a value from a position
    int      len();              // get the length of the vector
    int      slen();             // if this vector is a segment descriptor,
                                 // return the length of the
                                 // segmented vector

    // These members MODIFY the contents of the vector
    void     replace(int pos, const T val)
                                 // replace the value at a position
    void     dist(const T val)  // distribute a value across the vector
    void     send(pvect<T> vals, pvect<int> positions)
                                 // send values into the positions given
};
```

3.1 Segment Descriptors

Many vector operations accept an additional argument that designates *segments* within the vector. A segment is a contiguous range of sites. Each segment is itself a vector, and segmented vector functions perform a parallel function over all of the segments of a vector simultaneously [2]. For example, a segmented scan computes a recurrence in which the running sum is reset to **0** at the beginning of each segment.

There are many ways to represent segment descriptors. One representation is called the *startbits* descriptor. For a vector of length l, the *startbits* descriptor is a boolean vector of length l with a 1 at the beginning of each segment, and with 0 everywhere else. An alternate representation is the *segment-lengths* descriptor. This is a vector of integers whose sum is l; each element of the *segment-lengths* descriptor gives the length of a segment.

The AVTL uses the *segment-lengths* descriptor variety, whose type is simply `pvect<int>` (there is no special segment descriptor type). Internally, the library may compute an alternate representation (such as the *startbits* form) for use within the algorithm. The library caches any such representation, so that reusing the same vector as a segment descriptor will avoid recomputing the internal form. The cached representation is flushed however if the segment vector is modified in any way.

Users of the library need not concern themselves with the internal caching of segment descriptors, except when evaluating the performance of programs. The first time a vector of integers is used as a segment descriptor additional time may be required to compile and cache the internal segment descriptor. After that, the internal form is used directly.

4 Function Objects

A function object is an object with an `operator()()` defined. In the contexts in which C programmers would expect to pass a pointer to a function in a library subroutine, a `C++` function object is used in a template library. Algorithmic templates expecting function objects may also be used with regular function pointers too. However, function objects offer the advantage that inlined member functions do not incur the overhead of a function call.

4.1 Binary Associative Operators and Identity Values

In the parallel algorithm literature, efficient parallel algorithms are well known for scans and reductions using arbitrary binary associative operators (a "binop", for short) [2, 13]. Most of these algorithms require the specification of the identity element associated with the binary operator. The AVTL adopts the following convention: a binary associative operator is an object with two required member functions.

1. `operator()()` is a member function of two arguments performing the binary associative operation.

2. `identity()` is a member function of no arguments that returns the identity element of the appropriate type for the binary associative operator.

A suitable binary associative operator for the addition of integers is shown below.

```
class add_op_int {
  public:
    int operator()(int a, int b)   { return a + b; }
    int identity()                 { return 0; }
};
```

Binary associative operators are used uniformly throughout the library to parameterize the following algorithmic templates.

1. exclusive scans (segmented, unsegmented)

2. inclusive scans (segmented, unsegmented)

3. reductions (segmented, unsegmented)

4. combining-sends (like the add_scatter of HPF)

A large number of binary associative operators (binops) are predefined as generic template classes. These merely give names to standard elementwise algebraic operations: add, mul, max, min, and, or and xor, fst (return the first arg) and scd (return the second). For example, the addition binop is defined as follows. Because it invokes operator+, this binop is defined for any type for which the addition operator is defined.

```
template <class T> class add_op {
  public:
    T operator()(T a, T b)        { return a + b; }
    T identity()                  { return zero_val<T>(); }
};
```

The preceding template class introduced yet another template class: the identity values. The predefined binops require knowledge of four special values of each type. These are the zero value (identity for addition), the one value (identity for multiplicaiton), the minimum value of the type (identity for maximum) and the maximum value of the type (identity for minimum). The template class definition for the zero value follows.

```
template <class T> class zero_val {
  public:
    operator T { return 0; }
};
```

Whenever any object of type zero_val<T> is used in a context where a value of type T is required, the appropriate value is returned. For most builtin types, an appropriate conversion exists from 0 to the builtin type (integer, character, long double, etc.).

The identity classes for all of the builtin types use template specialization and the values from the standard C include file <limits.h> to pre-define the four identity values for each of the builtin types. Thus, for all of the builtin types of C, appropriate identity values are predefined based on the storage formats of the target architecture.

The structure of the binops and identity classes gives users flexibility about how new types and binops are integrated into the library. For a new type, the user may explicitly create binop function objects with the required members. These may specialize some of the predefined binops (e.g. add_op) for the new type, or may have completely new names. Alternatively, the user may simply provide definitions of the standard C arithmetic operators for the new type, as well as specializations for the identity classes. Then, the generic binop classes may be instantiated for all of the predefined binops. Note that if the binop does not have a standard name, the former approach is required.

4.2 Pseudo Binops

Most combining functions can be described merely by defining an appropriate function object. However, there is one combining operation used in conjuction with a combining send that does not fit this mold: it is the "append" operator, indicated by an app_op<T> function object. This binop may only be used with the combining-send function and causes element values sent to the same site to be placed in contigous sites in the result vector. The implementation of a send-with-append function actually requires an algorithm quite different from that of the other combining sends. A specialized template algorithm is defined for the case when the combining operator is of type app_op<T> This type does not have any member functions, but serves merely as a placeholder. In this way, the illusion of "append" as a binop is preserved, even though a different algorithm is invoked.

5 Elementwise Operators and Scalar Extension

All of the standard C arithmetic operators (+, -, *, /, %, <<, >>, |, &, ˆ) are extended to mean the elementwise application of the operator to the elements of the vectors in parallel. For instance, if a and b are vectors, a+b is their elementwise sum. If the elements of a and b are of differing type, the resulting vector will have the type of the left argument. This is not as general as the standard C type coercion rules, but is a workable solution.

The AVTL does not provide automatic scalar extension, because it is difficult to recognize scalar values with template arguments. Scalar extension is the ability to add a constant to all elements of a vector, for example, without explicitly distributing the constant across a new vector. The user of the AVTL can specify such scalar extension, but only by identifying scalar variables explicitly.

The scalar template class is used to indicate scalar extension for elementwise operators. A shorthand function, called make_scalar, may also be used. It attempts to deduce the type of the scalar from its argument.

The following illustrates two equivalent ways of adding 5.0 to each of the elements of a vector. The first method explicitly creates a scalar of type double, the second uses make_scalar to deduce the type from its argument, 5.0.

```
pvect<double>    a(100, 4.0);
pvect<double>    b = a + scalar<double>(5);
pvect<double>    c = a + make_scalar(5.0);
```

5.1 Comparison Operators

The standard C comparison operators (==, !=, <, >, <=, >=) are similarly extended. A scalar may be the either argument in such a comparison. The result of all comparisons of vectors is of type pvect<int>.

5.2 Assignment Operators

The assignment operators of C have not been redefined to be meaningful for vectors. Most operations in the AVTL produce a new vector as a result. An assignment operator would modify the value of a (potentially shared) vector. While these operators would be useful, at this point the ramifications of their inclusion are not fully understood.

5.3 Elementwise application of arbitrary functions

The elementwise template function applies an arbitrary function object to each of the elements of one or more vectors. Because the type of the value returned by a function object cannot be matched by a template argument, the user must give an argument that is of the type of the result desired.

Five variants of the elementwise function are provided. They apply functions of 1, 2, 3, 4 or 5 arguments to the elements of the corresponding vectors.

```
pvect<T>  result = elementwise(fn, pvect<T>(), a)
pvect<T>  result = elementwise(fn, pvect<T>(), a, b)
pvect<T>  result = elementwise(fn, pvect<T>(), a, b, c)
pvect<T>  result = elementwise(fn, pvect<T>(), a, b, c, d)
pvect<T>  result = elementwise(fn, pvect<T>(), a, b, c, d, e)
```

The function argument to an `elementwise` function may be either a function object or a function pointer. For example, the Unix cosine function may be applied to each of the elements of a vector, producing a vector of the cosines.

```
pvect<double> cosines = elementwise(cos, pvect<double>(), argvector)
```

Function objects used with `elementwise` functions are useful for their inlining capabilities. Assume that `a`, `b` and `c` are three vectors in the following example.

```
pvect<double> result = a * b + c;
```

This expression calculates the elementwise product of `a` and `b`, places it in a temporary vector, adds the elements of `c` and places the result in a new vector. Besides the overhead of allocating and freeing the temporary vectors, this operation suffers from writing the temporary values out to memory. A more efficient solution is to use a function object and the `elementwise` function.

```
class multadd {
  public:
    double operator()(double x, double, y, double z)
      { return x * y + z; }
};

pvect<double> result = elementwise(multadd(), pvect<double>(), a, b, c);
```

The resulting code is certainly less readable, because the multiply-add operation must be written as a separate function. However, when absolute high performance is a necessity, this technique can be used in critical regions of a program. The construct ensures that the layout of the vectors in memory remains hidden, but the use of the function object ensures that the performance meets that of a hand-coded loop. Elementwise functions are also useful when operating on members of vectors of user-defined structures.

5.4 Zipping Vectors

The AVTL borrows the `pair` template type from the STL. A `pair` is parameterized by two types and may hold two values of any type.

```
template <class T1, class T2>
class pair {
  public:
    T1 first;
    T2 second;
    pair(const T1 &x, const T2 &y) : first(x), second(y) { }
};
```

Pairs are useful in many contexts. When using the AVTL, pairs can help to speed some communication operations. For example, if permuting two vectors by the same permutation vector, it may be more efficient to pack them into a single vector of pairs and then to perform the permutation. Of course, a user could define an appropriate structure to hold the pair, and could then load the values into a vector of pairs using the `elementwise` function with a new function object. After the permutation, the pairs would have to be unpacked using another new function object.

This sequence of steps is so frequent that helper functions are included in the AVTL. The function `zip` accepts two vectors of any type as arguments and produces a vector of pairs[1].

```
pvect<double>   a;
pvect<complex>  b;

pvect<pair<double, complex> > c = zip(a, b);
```

A vector of pairs may be separated by using the `unzip` functions. There are two: one returns the first elements, the other returns the second elements.

```
pvect<double>   new_a = unzip1(c);
pvect<complex>  new_b = unzip2(c);
```

To return to the original problem that motivated including the pair type, assume that there are two vectors which are to be permuted the same way. The naive way to accomplish the permutation is to use two calls to the `permute` function.

```
pvect<int>      p;    // The permutation vector
pvect<double>   perm_a = permute(a, p);
pvect<complex>  perm_b = permute(b, p);
```

Using the zip functions, a single `permute` will suffice, but additional data movement will have to be peformed locally. The tradeoff may be beneficial if communication is very expensive (and it usually is). The resulting code is only slightly more ugly than the original.

```
pvect<pair<double, complex> > temp = permute(zip(a, b), p);
pvect<double>    perm_a = unzip1(temp);
pvect<complex>   perm_b = unzip2(temp);
```

[1] Guy Blelloch first named this operation “zip.”

6 Algorithms on Vectors

The AVTL provides a large number of standard algorithms for vectors. Most of these are provided in both segmented and unsegmented variants with function overloading used to differentiate between the two. Rather than list all of the algorithms and their arguments, this section only lists the algorithm names and gives a brief description of their function. The following two tables list the generic permutation and scan algorithms provided by the library.

Generic Permutation Algorithms			
Algorithm	Unsegmented	Segmented	Comment
`permute`	X	X	One-to-one permutation within a vector or segment.
`send`	X	X	Scatter
`cond_send`	X	X	Conditional scatter
`unpermute`	X	X	Backwards permutation
`get`	X	X	Gather
`cond_get`	X	X	Conditional gather

Generic Scans and Reduction Algorithms			
Algorithm	Unsegmented	Segmented	Comment
`op_scan`	X	X	Exclusive scan
`op_iscan`	X	X	Inclusive scan
`op_reduce`	X	X	Reduction

The pre-defined generic binops give names to the arithmetic operators of `C`. As described earlier, the binops have an associated identity value, as listed in the table below. When one wishes to use a new type with the vector template, it is sufficient to define the appropriate operator and to specialize the identity value if the default is not appropriate.

Pre-defined Generic Binop Classes			
Name	Uses	Identity	Comment
`add_op`	`operator+`	`zero_val`	Addition
`mul_op`	`operator*`	`one_val`	Multiplication
`max_op`	`operator>`	`min_val`	Maximum
`min_op`	`operator<`	`max_val`	Minimum
`and_op`	`operator&`	`one_val`	Boolean AND
`or_op`	`operator|`	`zero_val`	Boolean OR
`xor_op`	`operator^`	`zero_val`	Exclusive-Or
`fst_op`		`zero_val`	First (left) argument
`scd_op`		`zero_val`	Second (right) argument
`app_op`			Append arguments (a pseudo binop)

The library provides shorthand names for common variants of the scan and reduction functions. These are merely pre-defined functions that make use of the generic algorithm and a particular generic binop. This table reveals that some of the basic scan functions (such as `index`) are in fact derived from the generic scan algorithm.

Pre-defined Scan Functions			
Algorithm	Unsegmented	Segmented	Comment
`add_scan`	X	X	
`mul_scan`	X	X	
`max_scan`	X	X	
`min_scan`	X	X	
`and_scan`	X	X	
`or_scan`	X	X	
`xor_scan`	X	X	
`copy_scan`	X	X	Copy a value across a vector or segment.
`index`	X	X	Create a vector that enumerates its sites.
`rsh1`	X	X	Right shift a vector by one position.

Pre-defined Reductions			
Algorithm	Unsegmented	Segmented	Comment
`add_reduce`	X	X	
`mul_reduce`	X	X	
`max_reduce`	X	X	
`min_reduce`	X	X	
`and_reduce`	X	X	
`or_reduce`	X	X	
`xor_reduce`	X	X	
`maxloc`	X	X	Find the location of the maximum value in a vector or segment.
`minloc`	X	X	Find the location of the maximum value in a vector or segment.

The library also provides two-phase communication operators that separate the specification of a communication pattern from using that pattern to send data. The first phase is called communication “compilation.” When data is repeatedly sent in the same pattern, the total bandwidth achieved can often be greatly improved by re-using a communication schedule.

Pattern Compilation Functions			
Algorithm	Unsegmented	Segmented	Comment
`permute_comp`	X	X	One-to-one permutation within a vector or segment.
`send_comp`	X	X	Scatter
`cond_send_comp`	X	X	Conditional scatter
`unpermute_comp`	X	X	Backwards permutation
`get_comp`	X	X	Gather
`cond_get_comp`	X	X	Conditional gather

The result of communication compilation is a data-object called a “schedule.” A schedule is used with the `run` function to actually move data. The `run` function may also be parameterized with combining functions to specialize how collisions are handled at destination sites. This is the mechanism used to provide combining sends in the library.

Lastly, the library will provide a number of utility algorithms. These may be implemented on-top of the base algorithms, but direct implementations often run faster. Most of the algorithms below are already implemented, and the others will be soon.

Generic High-Level Functions			
Algorithm	Unsegmented	Segmented	Comment
`rank`	X		Rank the contents of a vector.
`hash_insert`	X		Insert elements into a hash vector.
`hash_find`	X		Find elements in a hash vector.
`append`	X	X	Append vectors or segments.
`dist`	X	X	Distribute a vector.
`subseq`	X	X	Extract a subsequence of a vector.
`shift`	X	X	
`cshift`	X	X	
`pack`	X		Compress a vector using a mask.
`unpack`	X		Decompress a vector using a mask.

7 Interfacing with MPI

For the most part, implementing the vector operations using MPI was straightforward. MPI provides a facility for describing messages of varying data types. The implementation of AVTL packs all messages into buffers and simply sends them as bytes. For example, all `permute` functions are implemented by gathering up data destined for each other processor and then sending it there in a single large message as a stream of bytes.

One difficulty was encountered in the implementation of the scan and reduction algorithms. First of all, MPI only defines inclusive scans because these do not require identity elements. To provide the necessary exclusive scans, the AVTL implements inter-processor scans that provide both the inclusive and exclusive scan values.

The most difficult part of implementing the scans and reductions is in the interface to the combining operations. MPI is a compiled library with generalized scans that are parameterized by a pointer to a combining function. The arguments handed to the combining function are specified by MPI. The AVTL is built on function objects that have different argument lists from the MPI combining functions. Wrapper functions were needed that applied the function object and arranged the arguments in the proper order for MPI. Because these had to generalize to various types and function objects, they had to be template functions.

The interface to the wrapper functions are dictated by MPI. The first two arguments are pointers to the values to be combined, the third argument is a length, and the fourth is an MPI datatype. The uses of the third and fourth arguments could not be modified, while the first two are for user data and must be the same type. As mentioned earlier, the wrapper functions had to be parameterized by both value type and function object. Thus, this two-way parameterization had to be accomplished in the type of a single argument.

The final version of the wrapper functions (all overload the name `am_scan_function` in the library) parameterize the first two arguments (the value arguments) by a nested template class that includes both the value type and function object type. This implies that the function object is included in the messages sent between processors - even though it carries no data! Fortunately, an object with no data members occupies only one byte. However, to maintain alignment, typically 4 or 8 bytes are wasted. These extra bytes are sent by MPI and are never used. This overhead is negligible.

Requesting a pointer to the wrapper function causes the compiler to instantiate it. The use of nested templates in this context confuses many compilers. In particular, the GNU `C++` compiler (versions 2.5.8 through 2.6.2) cannot handle this correctly. Other compilers (such as IBM's xlC) handle the nested templates with no problem. With the increased use of templates and the growing desire for acceptance of the STL, I expect most compilers to handle these constructs in the near future.

8 Performance Tests

This section presents some performance data collected from test programs implemented using the AVTL. The two platforms examined are the IBM SP-2 and the Intel Paragon. The intent is not to compare the two machines (the POWER-2 processor of the SP-2 is much faster than the i860 of the Paragon), but to show that the template based approach yields high performance.

This first implementation of the library uses very simple implementations for each of the vector algorithms. Scans, for example, were implemented by using the inter-processor scan functions provided directly by MPI. Permutations were implemented so that each processor exchanges a single large message with every other processor. This approach was chosen simply to reduce development time. The intent of this library was to experiment with the capabilities of template algorithms and to evaluate whether the compiler could generate efficient code. I will point out the limitations of some of the implementations of the algorithms as they are discussed by comparing the performance obtained to that of a library offering similar functionality: the CVL library for MPI, implemented by Jonathon Hardwick at CMU [5].

The compilers used were `xlC` on the SP-2, and the GNU compiler (`gcc`, version 2.6.0) on the Paragon. Full optimization was enabled for these tests. Sophisticated template usage is currently beyond the capabilities of many `C++` compilers, but the IBM compiler had no problem with any of the constructs of the library. The GNU compiler, however, could not successfully instantiate functions with nested templates. This is a known problem that will be fixed in the future. To circumvent this problem, the few functions that use nested templates (see "MPI Wrapper Functions" above) were flattened one level manually.

Elementwise addition and multiplication operations were timed first by measuring the performance obtained when applying a binary operator to two vectors that are much longer than the available cache of the machine. The rates reported in Table 1 were computed including the time to allocate the result vector,

Table 1: Elementwise performance for a very long vector.

Machine	Addition	Multiplication
SP-2	16 Mflops	16 Mflops
Paragon	2.5 Mflops	2.5 Mflops

and thus reflect rates that could be achieved in a program written using the AVTL. These figures reflect the ability of the compiler to transform the generic elementwise templates with template functions into efficient loops. The rate achieved on the Paragon matches that reported by Hardwick. On the SP-2, I have observed 39 Mflops from the DAXPY mathematical library subroutine, and have been able to achieve 33 Mflops by unrolling the loop four times by hand. I was surprised that the compiler could not achieve the same performance, even with unrolling turned on.

The permutation functions were implemented so that each processor exchanges a single large message with every other processor. Figure 1 shows the achieved bandwidth as a function of vector length on each of the two machines. The SP-2 achieves its best bandwidth for vectors of 128K elements. At this length, the entire vector fits in the rather large cache of the processor. Figure 2 evaluates the scalability of the algorithm and shows the achieved bandwidth as the number of processors increases. While the implementation is straightforward, it suffers from a lack of scalability, as the figure indicates.

At the beginning of the algorithm, each processor must sort the data elements by their destination processor ID. Then, for each other processor, a message is assembled and sent to the other processor. The problem with this approach is that the processors alternately spend long periods of time computing locally, and then all attempt to send large messages at approximately the same time. The network is first unused, and then saturated. Increased asynchrony can help to spread the data traffic out more evenly.

Hardwick noted this problem in his MPI implementation of the CVL library. He chose to send many small messages asynchronously and achieves performance roughly double that of the AVTL for permute operations on the Paragon. In the future, the AVTL may adopt such an algorithm.

The AVTL provides compiled communication algorithms. These separate the specification of a communication pattern with the actual movement of the data. While actual times are not reported here, in the current implementation the compilation phase generally takes as much time as a single `permute`. Thus, if a pattern is reused only twice, it pays to precompile the pattern. Figure 3 shows the bandwidth achieved for both a regular `send` on the SP-2 and the `run` function for a compiled pattern. Figure 4 compares the performance of compiled communication to a permute as the number of processors is increased. The initial spike is probably due to the fact that the data to permute and the compiled communication fit into the cache for very small vector sizes.

Figures 5 and 6 compare a regular permute to the bandwidth achieved in a compiled communication operation. In all cases the achieved bandwidth is approximately larger by a factor of two.

Figures 7 and 8 show the performance of the `add_scan` function on both the IBM SP-2 and the Intel Paragon. I was able to verify that the implementations of MPI on both machines do not use a logarithmic combining tree to implement their scan function, but instead use a linear chain! This fact helps to explain why the performance degrades as severely as it does for large numbers of processors. Hopefully, a better algorithm will be used in future versions of MPI.

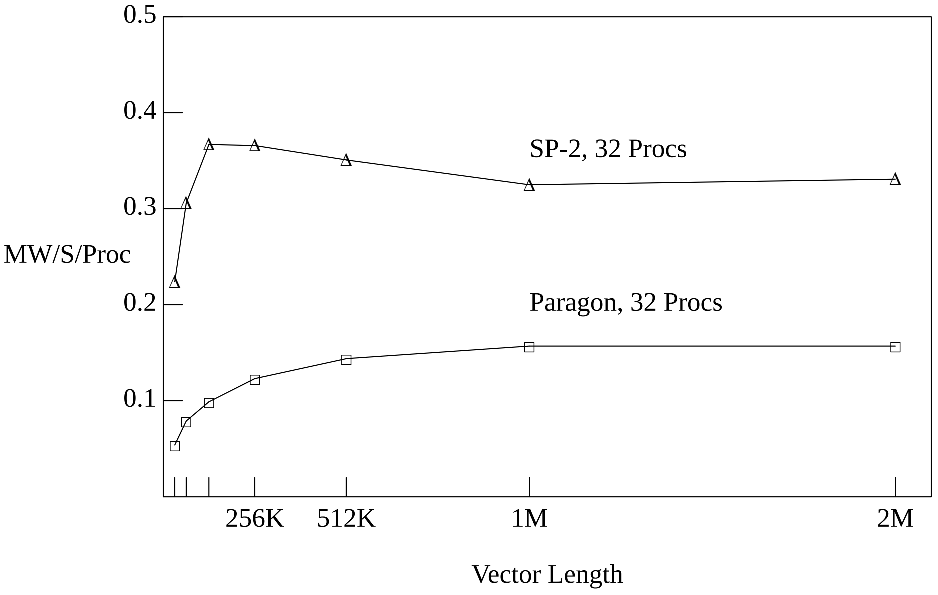

Figure 1: Achieved bandwidth of the `permute` function as a function of vector length.

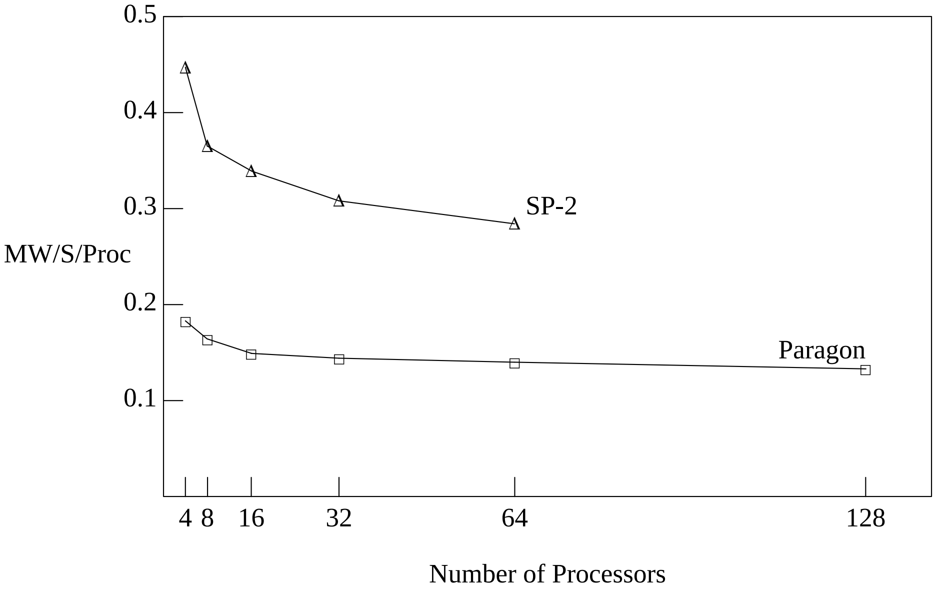

Figure 2: Scalability of the `permute` function for large vectors as the number of processors is increased.

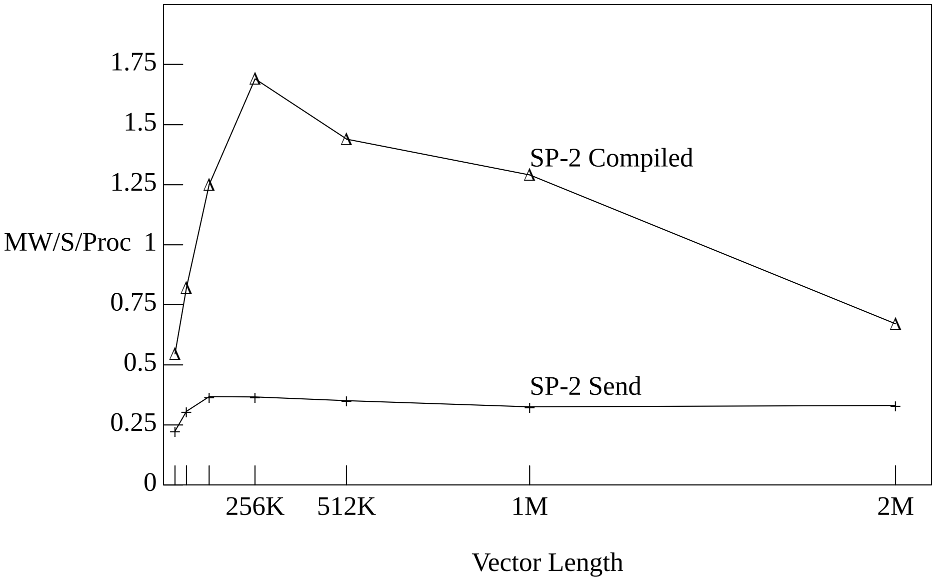

Figure 3: Comparison of the achieved bandwidth of a regular `permute` and a compiled `send` on 32 processors of an IBM SP-2.

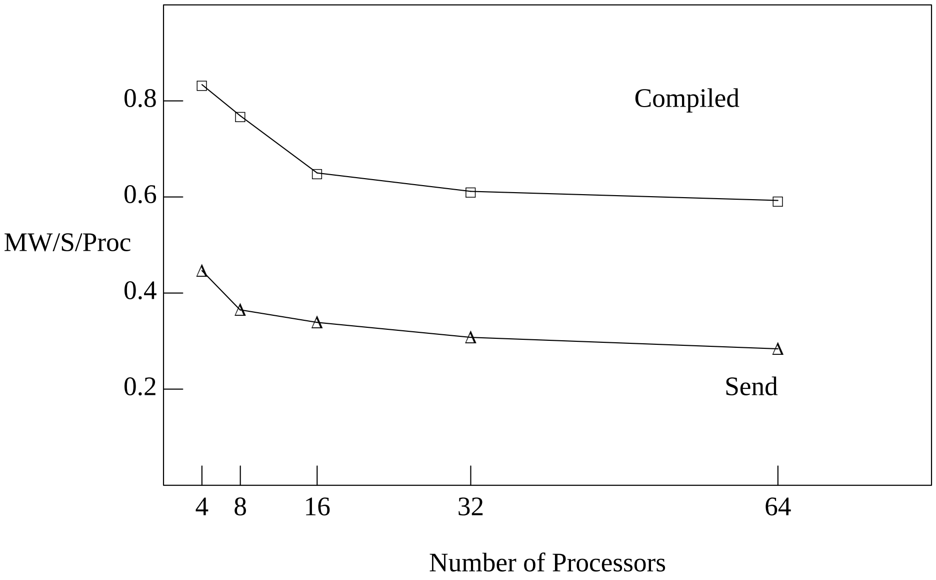

Figure 4: Comparison of the scalability of a regular `permute` and a compiled `send` on the IBM SP-2.

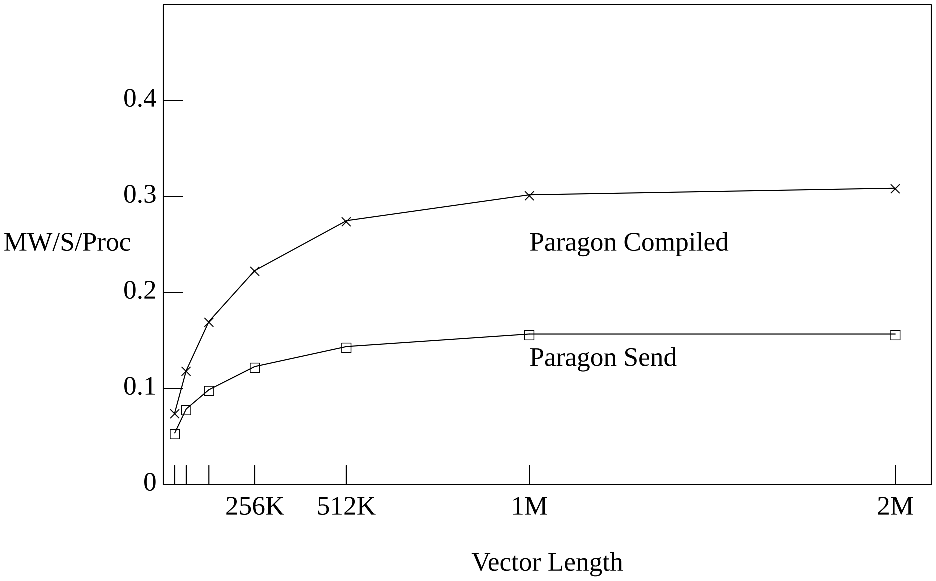

Figure 5: Comparison of the achieved bandwidth of a regular `permute` and a compiled `send` on the Intel Paragon.

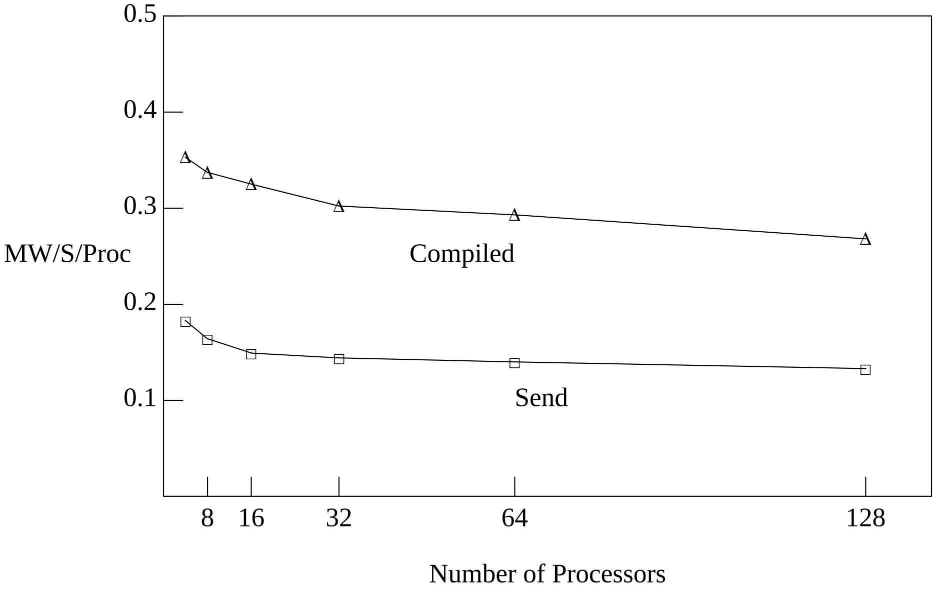

Figure 6: Comparison of the scalability of a regular `permute` and a compiled `send` on the Intel Paragon.

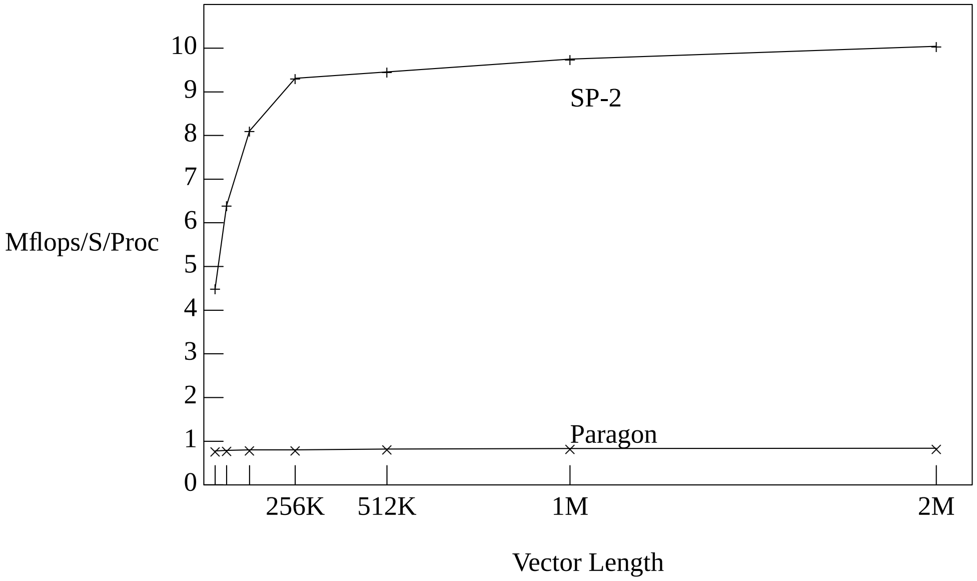

Figure 7: Achieved per-processor performance for an `add_scan` function on 32 processors.

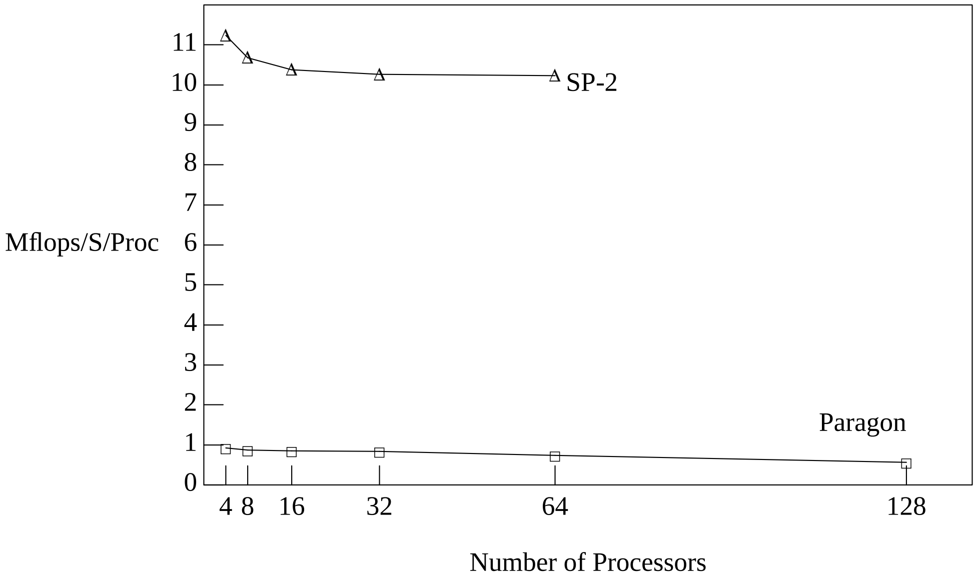

Figure 8: The scalability of the `add_scan` function.

Table 2: Performance of the NAS Conjugate Gradient (CG) benchmark implemented using the AVTL.

Machine	Procs	Normal (sec)	Compiled (sec)
SP-2	16	169.0	57.0
	32	71.5	28.6
Paragon	16	730.4	316.1
	32	320.7	111.6

8.1 An Application

Using the AVTL I wrote a version of the NAS Conjugate Gradient benchmark test and ran it on the SP-2 and the Paragon. Having written the code to use a regular `get` function, it was not difficult to modify it to use compiled communication instead. Table 2 presents the performance obtained by this implementation.

Quite honestly, the performance achieved is not on par with the best implementations provided by the computer vendors for this benchmark. My simple implementation did not attempt to minmize the amount of inter-processor communication in any way. In fact, the structure of the matrix is nearly uniformly random, so that all processors send most of their data off processor. Performance could be improved by rearranging the matrix using a heuristic such as recursive spectral bisection [10]. However, I should point out that the the implementation using the AVTL is only about one page of code (excluding setting up the test matrix) and was coded in about thirty minutes.

9 Conclusions

This paper presented an experiment in the design of a template-based collection library for distributed address space parallel computers. The design of the library stresses the orthogonality of element types, collection types, algorithms, and algebraic combining functions. By carefully differentiating between the roles of each of these the library achieves genericity, efficiency and extensibility to user-defined data types.

The development of quality `C++` compilers has been fueled by the PC and workstation markets. Sophisticated template usage is only now beginning to be supported, largely driven by the desire for the acceptance of the STL. This technology can also be beneficially employed by high-performance parallel computer programmers for the encapsulation of generic parallel algorithms. Just as most MPPs are now using commodity microprocessors whose development was driven by the workstation market (IBM SP-2, Cray T3D, Convex Examplar, Intel Paragon, Thinking Machines CM-5), compiler technology driven by those same large markets should be leveraged to enhance parallel programming productivity.

References

[1] G. Blelloch, S. Chatterjee, J. Sipelstein, and M. Zagha. CVL: a C Vector Library. School of Computer Science, Carnegie Mellon University, 1991.

[2] G. E. Blelloch. *Vector models for data-parallel computing.* MIT Press, 1990.

[3] Cray Research, Inc. *Cray Standard C Programmer's Reference Manual*, SR-2074 4.0 edition.

[4] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison Wesley, Reading, MA, 1983.

[5] J. C. Hardwick. Porting a vector library: a comparison of MPI, Paris, CMMD and PVM. Technical Report CMU-CS-94-200, Carnegie Mellon University, 1994.

[6] IBM. *Optimization and Tuning Guide for Fortran, C and C++*, first edition, 1993.

[7] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, and M. E. Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994.

[8] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Softw.*, 5:308–323, 1979.

[9] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 1994.

[10] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2/3):135–148, 1991.

[11] A. Stepanov and M. Lee. The standard template library. Technical report, Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, October 1994.

[12] B. Stroustrup. *The C++ Programming Language, Second Edition*. Addison-Wesley Publishing Company, 1991.

[13] J. C. Wyllie. *The Complexity of Parallel Computation*. PhD thesis, Cornell University, 1979.

A Code Examples

The examples in this appendix illustrate the simplicity and extensibility of the library. The first example computes Pi by using a simple integration scheme and shows the entire program — including runtime initialization. The second example illustrates the extensibility of the library. A single generic sparse matrix-vector multiplication routine is extended to a blocked version simply by defining two new classes.

A.1 Computing Pi

The first example illustrates a simple program that computes π by integrating $f(x) = \frac{4.0}{x^2+1}$ from 0 to 1. This code makes use of an instance of the class `f_of_x` as a function object (sometimes called a *functor*) with saved state. The constructor for the class produces an integrand function customized for a particular value of h — the width of the rectangle. The function accepts an index value, computes its x coordinate, and returns the area of the rectangle at that coordinate. The rest of the program is straightforward.

```
// Example of a program to compute PI using the AVTL.

#include <stdlib.h>
#include <unistd.h>     // for IBM SP2
#include "pvect.h"      // Amelia Vector Template Lib
```

```
class f_of_x {
    double h;
  public:

    f_of_x(double height) : h(height) { }

    double operator()(int i) {
        double x = (i + 0.5) * h;
        return (4.0 * h) / (x * x + 1.0);
    }
};

double
compute_pi(int n)
{
    double          h = 1.0 / n;
    pvect<int>      i = index(n);
    pvect<double>   rect = elementwise(f_of_x(h), pvect<double>(), i);
    return add_reduce(rect);
}

int
main(int argc, char **argv)
{
    _mem = new am_mem_mgmt(argc, argv); // parallel runtime

    double pi = compute_pi(100000);
    printf("Pi is %f\n", pi);
    delete _mem;                        // terminate runtime
}
```

A.2 Sparse matrix-vector multiplication

This second example shows how a user can write a generic function and extend it to new types. The function mvmult performs a sparse matrix-vector multiply operation for a matrix stored in compressed sparse row format. The elements of each row are stored contiguously along with an integer describing the column of each element. A segment descriptor divides the elements and column labels into rows. The sparse matrix-vector multiplication function follows, along with a code fragment showing an example of its usage where the elements of the matrix and vector are simple floating point values.

```
template <class A, class V>
pvect<V> mvmult(pvect<A> a, pvect<int> cols, pvect<int> seg, pvect<V> v)
    // (a,cols,seg) describe a sparse matrix in compressed sparse row format.
    // v is the vector.
{
    pvect<V> g = get(v, cols);
    pvect<V> p = g * a;
    pvect<V> r = add_reduce(p, seg);
    return r;
}

    // Code fragment of usage
    pvect<double> mata =  ... omitted ...
    pvect<double> veca =  ... omitted ...
    pvect<int> cols =     ... omitted ...
    pvect<int> segs =     ... omitted ...

    pvect<double> resa = mvmult(mata, cols, segs, veca);
```

Many sparse codes can benefit by using blocked algorithms. It is possible to extend the above `mvmult` function to operate with a blocked sparse matrix merely by defining new element types. In this case, the elements of the matrix will be dense blocks, and the elements of the vector will be block vectors. The code that follows shows the declarations of the block matrix and block vector classes, as well as the addition and multiplication operators required. Function `mvmult` is not modified in any way, but automatically extends to the new types.

```
class blockmat {
  public:
    int vals[5][5];
    blockmat() { }
    blockmat(double init)
        // Initialize to scaled identity matrix
        {
            for (int r = 0; r < 5; r++)
                for (int c = 0; c < 5; c++)
                    vals[r][c] = r == c ? init : 0.0;
        }
};

class blockvec {
  public:
    int vals[5];
    blockvec() { }
    blockvec(double init)
        {
            for (int i = 0; i < 5; i++)
                vals[i] = init;
        }
};
```

```
blockvec operator*(blockvec v, blockmat a)
    // Multiply a block vector by a block matrix.
{
    blockvec x(0.0);
    for (int c = 0; c < 5; c++)
        for (int r = 0; r < 5; r++)
            x.vals[c] += a.vals[r][c] * v.vals[r];
    return x;
}

blockvec operator+(blockvec a, blockvec b)
    // Add two block vectors.
{
    blockvec c;
    for (int i = 0; i < 5; i++)
        c.vals[i] = a.vals[i] + b.vals[i];
    return c;
}

    // Code fragment of usage
    pvect<blockmat> matb =  ... omitted ...
    pvect<blockvec> vecb =  ... omitted ...
    pvect<int> cols =       ... omitted ...
    pvect<int> segs =       ... omitted ...

    pvect<blockvec> resb = mvmult(matb, cols, segs, vecb);
```

This call to `mvmult` will be instantiated for a block matrix and a block vector. With the appropriate model of matrix elements, this generic function applies to scalar elements or dense blocks. This short example is a dramatic demonstration of the code reuse that is possible with generic libraries.