HPC++: Experiments with the Parallel Standard Template Library*

Elizabeth Johnson Dennis Gannon
Department of Computer Science, Indiana University
{ejohnson,gannon}@cs.indiana.edu
Peter Beckman
Advanced Computing Laboratory, Los Alamos National Laboratory
beckman@lanl.gov

Abstract

HPC++ is a C++ library and language extension framework that is being developed by the HPC++ consortium as a standard model for portable parallel C++ programming. This paper describes an initial implementation of the HPC++ Parallel Standard Template Library (PSTL) framework. This implementation includes seven distributed containers as well as selected algorithms. We include preliminary performance results from several experiments using the PSTL.

1 Introduction

C++ [20] has become a standard programming language for desktop applications. Increasingly, it is being used in other areas including scientific and engineering applications, and there are dozens of research projects focused on designing parallel extensions for C++ [21].

Several groups have joined to define standard library and language extensions for writing portable, parallel C++ applications. In Europe, the *Europa* consortium [16] has defined a model of parallel C++ computation based on Active Objects [1, 6, 8] and a meta-object protocol derived from the work on reflection in the programming language research community [12, 17]. In Japan, the Real World Computing Partnership has established the MPC++ programming system [11] which provides broad and powerful mechanisms for user-level extensions to a C++ compiler.

In the United States, the HPC++ consortium has focused on extensions to standard C++ class libraries, compiler directives, and a few small language extensions to achieve the goal of portable parallel programming. The consortium is a diverse group, with representatives from industry, academia, and government laboratories [10].

In this paper, we describe an initial implementation of HPC++. This implementation is limited, but it does allow exploration of some of the ideas proposed by the HPC++ consortium. Thus it is an important first step towards defining a framework for portable, parallel C++ programming.

*This work is supported by DARPA under contract DABT63-94-C-0029 and Rome Labs from contract 30602-92-C-0135.

ICS 97 Vienna Austria

2 Overview of HPC++

The current HPC++ framework (Level 1) describes a C++ library along with compiler directives which support parallel C++ programming. A future version (Level 2) will include language extensions for semantics that cannot be expressed by the Level 1 library.

The standard architecture model supported by HPC++ is a system composed of a set of interconnected nodes. Each node is a shared-memory multiprocessor (SMP) and may have several *contexts*, or virtual address spaces. While the implementation described in this paper assumes a homogeneous system, the HPC++ framework will also support heterogeneous systems where nodes may be on physically distinct computers.

Level 1 of the HPC++ framework consists of the following parts:

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

and, in the future,

- a library for distributed active objects,
- an interface to CORBA [9] via IDL mapping, and
- a set of programming and performance analysis tools.

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 parallel algorithms and parallel iterators. These components are discussed in Section 4 below. HPC++ also includes another distributed container – a multidimensional distributed array class based on A++ [18] and LPARX [13]. This array class 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.

Although the last three parts of the HPC++ framework are not yet well-defined, our implementation does include the global pointers and remote function invocation infrastructure necessary to support active objects. These are described in Section 3.4.

3 HPC++ Run-Time Support

Our implementation is separated into two layers – the run-time system (RTS) and the Parallel Standard Template Library. This design provides an opportunity for other developers to substitute their own RTS implementation, provided that it supports a small set of functions and constructs, described in this section. As can be seen from the examples in Section 5, it is possible to do effective parallel programming using only the higher level PSTL components, but the RTS layer components are also available to the applications programmer for use in less structured parallel programming.

While HPC++ supports several execution models, the model for our implementation is Single Program, Multiple Data (SPMD). In this model, an HPC++ program launches a single thread of control in each context. We use Tulip [2], a run-time system developed at Indiana University, as the basis for our RTS layer. This system runs on various machines including the IBM SP-2, SGI Power Challenge, and Cray T3D. It 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.

Using Tulip, we define typed global pointers and a set of functions which can be divided into the following groups: (1) location information, (2) remote store and fetch, and (3) remote function invocation.

A barrier is also included to provide synchronization for operations on distributed containers. The following sections describe the components of the run-time system.

3.1 Global Pointers and Global References

Global pointers are based on the *global* type in languages like CC++ [4], AC [3] and Split-C [7]. 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, an *RTS_put* (see Section 3.3) is invoked to copy *rhs* into the remote object. An analogous operation occurs when the cast operator is used on a global reference. If the global reference refers to a local object, that object is returned. If the reference is to a remote object, an *RTS_get* is done to make a local copy of the remote object value.

This design allows the library user to employ standard C style to read and write remote values.

```
HPCxx_GlobalPtr<float> p;
...
// assign p to a global pointer to a remote object
...
*p = 3.14;          // a remote write
float t = *p + 2;   // a remote read
                    //   followed by a local addition.
```

In this HPC++ implementation, global pointers are primarily used to refer to elements in the distributed container. In particular, the parallel iterator for the distributed containers (described in section 4.1) can be cast into a global pointer to a container element.

3.2 Location Information

Object location is encapsulated by an ObjectFinder object. In Tulip, this is a context number only, but it could include other information such as processor number or machine number which uniquely identifies a location. Query functions are provided to determine locations of objects represented by global pointers and global references.

3.3 Remote Data Access

Access to remote data is supported via global pointers and uses remote store and fetch functions. These functions are:

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

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

In our implementation, these functions utilize the Tulip *tulip_Get* and *tulip_Put* routines, which use methods appropriate to the particular machine type to transfer data. *RTS_put* and *RTS_get* are blocking calls.

3.4 Remote Function Calls

Remote function call support consists of class and function templates for each possible member function arity (up to an implementation-defined limit). Each class template defines a message class. The base class of all message classes, *MessageBase*, is an abstract base class with two virtual functions: *getSize* and *decipher*. These functions are used in sending and unpacking each particular message type. The message objects are similar to the *Mobile Objects* defined in the CHAOS++ run-time library [5].

Each of these message classes is paired with a template function to be invoked by the application when a remote call is needed. This function instantiates a message object in the local context, packs the arguments, and sends the message to the appropriate remote context. A remote action handler in the receiving context calls *decipher* on the received message which, in turn, unpacks the arguments and invokes the specified method on its local object. A global pointer from the calling context is used for storing of the return value by the receiver. For non-blocking calls, a mechanism, similar to the *future* class in ABC++ [22], supports subsequent access to return values from the remote function invocation.

4 Parallel Standard Template Library

One of the major recent changes to the draft C++ standard has been the addition of the Standard Template Library (STL) [19]. The STL has five basic components.

- *Container class templates* provide standard definitions for common aggregate data structures, including *vector*, *list*, *deque*, *set* and *map*.
- *Iterators* generalize the concept of a pointer. There are five basic categories of iterators: *random access*, *bidirectional*, *forward*, *input* and *output*.
- *Generic algorithms* are function templates that allow standard element-wise operations to be applied to containers via iterators.
- *Function objects* are created by wrapping functions with classes that typically have only *operator()* defined. They are used by the generic algorithms in place of function pointers because they provide greater efficiency.
- *Adaptors* are used to modify STL containers, iterators, or function objects. For example, container adaptors are provided to create stacks and queues, and iterator adaptors are provided to create reverse iterators to traverse an iteration space backwards.

The Parallel Standard Template Library extends the STL to include distributed containers, parallel iterators, and parallel algorithms. Currently, our implementation includes distributed versions of all seven STL containers along with parallel iterators for each. In addition, parallel versions of many of the STL algorithms have also been completed. The following sections describe parallel iterators, the distributed versions of STL containers, and the parallel algorithms.

4.1 Parallel Iterators

Parallel iterators extend the functionality of global pointers. In the case of *random access parallel iterators*, the operators *++*, *--*, *+n*,*-n*, and *[i]* allow random access to the entire contents of a distributed container, while the weaker *bidirectional parallel iterators* only provide access via the *++* and *--* operators. In general, each distributed container class *C* has a subclass for the strongest form of parallel iterator that it supports (e.g. random access, forward, or bidirectional).

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 (described in Section 4.3), 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.

4.2 Distributed Container Classes

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. Three of the containers are *sequence* containers, which store elements in a sequential order.

- *distributed_vector* is a one-dimensional array.
- *distributed_deque* is a double-ended queue.
- *distributed_list* is a doubly-linked list.

The other four are *associative* containers. Elements are ordered by a key, which can also be used to retrieve elements from the container.

- *distributed_set* is an ordered set of unique keys.
- *distributed_multiset* is like a *distributed_set* except that duplicate keys are allowed.
- *distributed_map* is an ordered set of unique keys along with associated objects.
- *distributed_multimap* is like a *distributed_map* except that duplicate keys are allowed.

As described in Section 4.1, each container provides a local and global view through iterators which access local and global elements. Iterators for the sequence containers traverse elements in sequence while associative container iterators traverse in the order of the element keys.

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 distribution itself is specified using a *ContainerRatio* object, which may be modified during execution. This object contains information about the proportion of the total elements to be assigned to each context. If no ratio object is specified for a container, block distribution is used – elements are divided evenly among the contexts, with any extra elements assigned to the last context. The creation of a distributed container and the manipulation of its ratio object are collective operations. The ratio object must be identical in each context.

The PSTL containers are dynamic – the size will vary as elements are inserted and/or deleted. For this reason, the distribution of elements in a container is only guaranteed to comply with its ratio object after either container construction or invocation of the *redistribute* method on the container or its ratio object. This compliance remains in effect until the first operation which modifies the size of the container or until the ratio object itself is changed or replaced.

Containers may share a ratio object. Invoking *redistribute* on the ratio object redistributes all containers which share that object. Invoking *redistribute* on the container itself only redistributes that container's elements; the other containers which share the ratio object remain unchanged. Note also that the containers need not be of the same type in order to share a ratio object. A *distributed_vector*, for example, may share a ratio object with a *distributed_list*.

4.3 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. This approach has several advantages. First, the same algorithm can be used for any container which utilizes an iterator of sufficient capability. So, for example, an algorithm which applies a function to each element can be called for any container which provides an iterator capable of the increment (++) operation. Second, algorithms can be easily applied to subranges of elements by passing as arguments to the algorithm the iterators which mark the beginning and one past the end of the subrange. In addition, through the use of user-defined iterator adaptors, subgroups of elements, such as odd- or even- indexed elements or elements with values above a certain threshold, can be accessed.

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. The body of each algorithm consists of a call to another function with an added argument: *iterator_category(first)*. The local or parallel iterator version of the function is called, depending on the iterator tag. Note that this differentiation is done at compile time using function templates.

The following example illustrates these algorithm versions.

```
// create a distributed vector of size 100 and
// initialize each element to 5.0
distributed_vector<double> myVec(100, 5.0);

// sum the local elements in this context
double sum1 = accumulate(myVec.begin(), myVec.end(),
                         0.0);

// sum all elements - collective operation
double sum2 = accumulate(myVec.parbegin(),
                         myVec.parend(), 0.0);

// sum the local elements in this context
//   using single-context parallelism
double sum3 = par_accumulate(myVec.begin(),
                             myVec.end(), 0.0);

// sum all elements - collective operation
double sum4 = par_accumulate(myVec.parbegin(),
                             myVec.parend(), 0.0);
```

In this example, let us assume that 3 contexts are involved. We first create a distributed vector with 100 elements of type double. Since the default distribution is used (no *ContainerRatio* object is specified in the constructor), 33 elements are stored in the first two contexts and 34 are stored in the last. Each element is initialized to 5.0.

The STL algorithm *accumulate* is declared as:

```
template <class Iterator, class T>
T accumulate(Iterator first, Iterator last, T initial)
```

The algorithm sums the elements in the iteration space *[first, last)* using *initial* as the initial value.

The first call to *accumulate* is a non-collective operation and results in the invocation of the STL version of *accumulate* in the calling context. While it does not need to be invoked in each context, it is in this case. Each invocation is, however, completely independent of the other contexts. In contexts 0 and 1, the value of *sum1* is 165.0 (5 * 33). In context 2, the value is 170.0 because there is one extra element.

The second *accumulate* call is differentiated from the first by the use of parallel iterators. It is a collective operation and must be called in all contexts. The local elements are summed in each context and then a global sum is computed using a reduction operation. In all contexts, *sum2* is 500.0.

Because it is invoked with local iterators, the first call to *par_accumulate* is a non-collective operation. Unlike *accumulate*, however, *par_accumulate* with local iterators uses single-context parallelism, if available, to compute the local sum. Lightweight threads are used in a context to accumulate partial sums of the local elements; these are summed to get the total for the local context. The value of *sum3* in each context corresponds to the value of *sum1*; they differ only in the processing involved. Finally, the second call to *par_accumulate* is equivalent to the second call to *accumulate* since behavior of the algorithms is the same when they are called with parallel iterators.

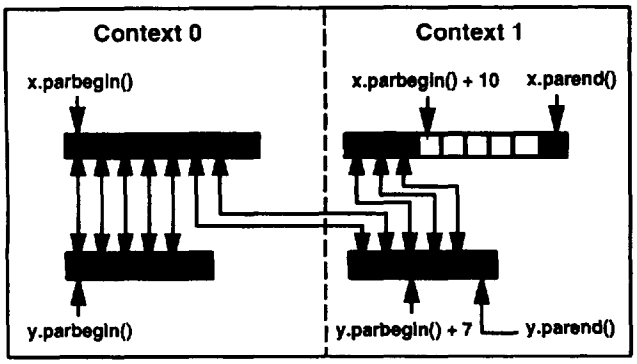
The third group of algorithms consists of special *par_* algorithms such as *par_apply, par_scan*, and *par_reduce*. These are collective operations with parallel semantics. The following have been defined thus far:

- *par_apply*: Applies a function object pointwise to the elements of a set of containers.
- *par_reduction*: Applies a function object pointwise to the elements of a set of containers and then does a reduction on an associative binary operator.
- *par_scan*: Applies a function object pointwise to the elements of a set of containers and then does a parallel prefix operation using an associative binary operation.

As is the case for all PSTL algorithms, the main arguments to these algorithms are iterators which access container elements. When invoked with local iterators, the algorithms are non-collective and use single-context parallelism. When invoked with parallel iterators, the operations are collective and have parallel semantics across contexts. The function object may change only the value of the elements in the space defined by the first two iterators in the argument list. Changes made to elements accessed via any other parallel iterator argument are lost since the function is applied to a temporary local copy of those elements.

For example, the declaration of *par_apply* for a binary function object is:

```
template <class ForwardIterator1,
          class ForwardIterator2,
          class BinaryOperation>
void  par_apply(ForwardIterator1 begin1,
                ForwardIterator1 end1,
                ForwardIterator2 begin2,
                BinaryOperation binop);
```




```
par_apply(x.parbegin(), x.parbegin()+10,
          y.parbegin(), add_to());
```

Figure 1: Example of *par_apply* usage

Here, the first iteration space is defined by *[begin1, end1)*. If this iteration space contains *k* elements, then the second iteration space consists of *[begin2, begin2 + k)*. In applying the binary function, each element in the first iteration space is paired with the corresponding element in the second iteration space.

Consider the following example.

```
class add_to {
public:
   void operator()(int& m, int n) { m += n; }
};

// create vectors
distributed_vector<int> x(15);
distributed_vector<int> y(10);

// initialize element values
do_initialization(x.parbegin(),x.parend());
do_more_initialization(y.parbegin(),y.parend());

// add the first 10 elements of y to the
//  corresponding elements in x
par_apply(x.parbegin(),x.parbegin()+10,y.parbegin(),
          add_to());
```

Figure 1 shows the two vectors, with the elements from the iteration spaces shaded in gray. The processing within *par_apply* in each context is as follows:

1. The iteration space *[x.parbegin(),x.parbegin()+10)* is restricted to its local elements. As shown, in context 0, seven elements are local and in context 1, three elements are local.

2. The portion of *[y.parbegin(),y.parbegin()+10)* which corresponds to these local elements is identified. In context 0, this begins at *y.parbegin()*, while in context 1 it begins at *y.parbegin()+7*.

3. The *operator()* method of *add_to* is applied to each pair of elements in the iteration spaces identified in steps 1 and 2. Paired elements are connected by arrows in the figure. Note that all of the elements in the first iteration space are local, while in the second space elements may be local or remote. In the case of remote elements, a remote fetch is done to get a local copy for the operation. Step 3 is done in parallel if single-context parallelism is supported.

5 Examples

The next two sections describe several examples using the PSTL. First, we show experiments with implementing a partial differential equation (PDE) solver. Second, we describe use of the PSTL to implement an algorithm which finds anagrams in a dictionary. In both examples, note that the parallelism is provided via standard algorithms from the PSTL. The applications programmer must write the code in terms of container iterators in order to use these algorithms, but no extra effort is required in order to add parallelization itself. In addition, the location of elements is transparent to the programmer in terms of writing the code since remote elements are fetched as necessary when the parallel iterator is dereferenced.

5.1 A Fast Poisson Solver

An example of a data-parallel computation that is easy to program with the HPC++ PSTL is a simple Fast Poisson Solver which computes the solution to an elliptic partial differential equation on a two-dimensional grid of size n by n. The computation is very simple in structure. The differential operator is converted to a 5-point finite difference operator and the problem is reduced to solving an algebraic system A*U=F, where F is a matrix of input data and U is another matrix of the same size representing the solution.

The algorithm works as follows:

1. Compute the fast sine transform of each row of F, storing the result in U. The sine transform is computed by means of a complex FFT of size $n/2$.

2. Transpose the matrix U. Each column defines the coefficients in a tridiagonal matrix equation. In parallel, for each column, solve the tridiagonal system of equations and put the solution into the F matrix. Transpose F.

3. Apply the fast sine transform again to each row of F and store the results in U. U now contains the solution to the PDE.

4. To check the result, compute F-A*U. This is done in parallel because A is a simple 5-point finite difference operator.

5. Compute the sum of the squares of the components of F-A*U.

One possible data structure for F and U is a distributed vector of *Row* objects. The class *Row* consists of an array of double precision numbers which can be accessed with the standard array ([i]) operator, a function which returns the array size, and a function that returns the Row position in the vector.

```
class Row{
  public:
    float &operator[](int);
    int size();
    int index(); // position in the vector
};
```

We use the *par_apply* function to compute the sine transforms in parallel to each Row object in the vector. The *par_reduction* function is used to compute the sum of the squares of the residual vector components.

```
double FastPoissionSolver( int n,
        distributed_vector< Row> & U,
        distributed_vector< Row> & F) {
 //apply the sine transforms to each row in F
 // and put result in U
 par_apply(U.parbegin(),U.parend(),F.parbegin(),
          row_ffts());

 //transpose U, solve the tri-diagonal systems
 // of equations, put solution in F, transpose F
 SolveTridiagonalSystems(n, F, U);

 //apply the sine transforms to each row in F
 // again and put result in U
 par_apply(U.parbegin(),U.parend(),F.parbegin(),
          row_ffts());

 // check for errors.  apply the differential operator.
 //   F = A*U
 par_apply(F.parbegin(),F.parend(), DiffOper(U));

 // compute the sums of squares of the differences to
 // get the residual
 double resid = par_reduction(F.parbegin(),F.parend(),
                              plus<double>(), squares(),
                              (double) 0.0);
 return resid;
}
```

The *row_ffts* function object computes the transform on *Row* x (of matrix F here) and puts the result in *Row* y (of U matrix here). We do not show the details of the sine transform function.

The most difficult part is the solution of the system of tridiagonal equations. As was described above, the individual tridiagonal equations are associated with the columns of the matrix. Hence, one solution is to first transpose the matrix so that columns become rows. Then we apply a standard sequential tridiagonal system solver to each row in parallel across the distributed vector of Row objects.

The transpose function object has a copy of the appropriate Row from the source vector. For each element of the destination row, we must access the corresponding (column) element of the source vector Row.

```
class transpose{
  public:
   Row R;
   transpose(distributed_vector<Row> &X, int i){
       distributed_vector<Row>::pariterator r;
       R = X[i]; // R = ith Row of X
       }
   void operator()(Row &y){
       int i = R.index();
       int j = y.index();
       y[i] = R[j];
   }
};
```

The tridiagonal solver takes the following simple form.

```
void SolveTridiagonalSystems(int n,
          distributed_vector<Row> &F,
          distributed_vector<Row> &U  ){
  for (int i=0; i < n  ; i++)
      par_apply(F.parbegin(),F.parend(),transpose(U,i));

  par_apply(U.parbegin(),U.parend(), F.parbegin(),
           solve_tridiag());

  // each iteration fills jth column of F with
  //   jth Row of U
  for (int j=0; j < n  ; j++)
      par_apply(F.parbegin(),F.parend(),transpose(U,j));
}
```

This Fast Poisson Solver was instrumented and run on the SGI Power Challenge and the IBM SP/2. Although the SGI Power Challenge is a shared memory machine, the Tulip runtime system provides a mechanism to run in SPMD mode. The remote fetches and stores, however, do take advantage of the shared memory configuration. Tulip on the IBM SP/2 utilizes the Message Passing Interface (MPI) for remote fetches and stores.

The results, in terms of execution times in seconds for each of the main functions for various numbers of processors and grid size 1024 by 1024 are shown below.

For 10-node SGI Challenge:

	FFTs	Transpose	Tridiagonal	A*U	sum-of-sq	total
P=1	14.94	2.01	1.76	1.77	1.07	21.55
P=2	7.50	1.41	0.88	0.90	0.54	11.23
P=4	3.81	1.32	0.46	0.45	0.27	6.31
P=8	1.96	1.58	0.24	0.24	0.15	4.17

For 24-node IBM SP/2:

	FFTs	Transpose	Tridiagonal	A*U	sum-of-sq	total
P=1	13.26	3.66	1.81	1.73	1.27	21.73
P=2	6.63	5.89	0.92	0.90	0.63	14.97
P=4	3.36	6.02	0.46	0.47	0.32	10.63
P=8	1.68	9.26	0.23	0.29	0.16	11.62
P=16	1.42	71.91	0.24	0.31	0.23	74.11

The degradation of performance apparent in the SP/2 16-node case probably results because only 8 nodes may be reserved for exclusive use on our machine. Jobs using greater than 8 nodes compete for system resources with other processes. In the other cases, all components except the transpose operation scale down very well as the number of processors increases. There are two factors which may explain the timings for the transpose operation. First, all processors iterate through the loop over the rows of the distributed vector in the same order. So all try to fetch a row from processor 0 first, then all try to fetch the next row, and so on. This results in loss of parallelism as one processor tries to accommodate all of the requests. This problem worsens as more processors are added. Providing some means to stagger the row requests may help this problem.

Second, during creation of the transpose function object, an entire row is fetched by each processor for each column. Only the portion of the row matching the locally stored column elements is actually needed. We are working on a version of the program which uses the remote function call infrastructure to fetch partial rows.

Another approach to this problem is to avoid the transpose operation by rewriting the tridiagonal system solver so that it uses a parallel tridiagonal algorithm which operates on *Rows* of tridiagonal systems in sequence.

The algorithm used is called *cyclic reduction*. It is a form of Gaussian elimination with log(n) stages. At stage *i*, all variables in rows *2**(i-1)* are eliminated from the equations for row *2**i*. The algorithm requires a log(n) forward sweep and a log(n) *back-solve* sweep. The basic form of the parallel computation is shown below. (The details of the elimination process and the back-solve are not included here).

```
void
SolveTridiagonalSystems2(int n,
                         distributed_vector<Row> &F,
                         distributed_vector<Row> &U){
     int i,k, s;
     double c;
     CyclicReductionInit();
     for(s = 1; s < n/2; s = 2*s)
```

```
        par_apply(U.parbegin(), U.parend(),
                  factor(s, U));
    for(s = n/2; s >= 1; s = s/2)
        par_apply(U.parbegin(), U.parend(),
                  backsolve(s, U));
    F = U;
}
```

The important property of this method is that each Row object need only access two other Row objects to complete the factorization and backsolve step. The detailed performance numbers for this version of the program are given below.

For 10-node SGI Challenge:

	FFTs	CyclicReduct	A*U	sum-of-squares	total
P=1	15.31	6.89	2.41	1.09	25.71
P=2	7.81	4.31	1.22	0.55	13.89
P=4	3.97	3.17	0.62	0.29	8.04
P=8	1.99	2.53	0.32	0.15	4.98

For 24-node IBM SP/2:

	FFTs	CyclicReduct	A*U	sum-of-squares	total
P=1	13.76	6.37	2.49	1.26	23.88
P=2	6.91	3.96	1.28	0.63	12.78
P=4	3.46	2.69	0.67	0.32	7.14
P=8	1.81	2.17	0.36	0.17	4.51
P=16	1.65	4.05	0.45	0.22	6.37

Except for the cyclic reduction operation, the operations scale down well (with the same problems on 16 nodes as in the other formulation). The cyclic reduction operation involves fetching entire Row objects, so communication latency may play a part here. Also, it should be noted that while the cyclic reduction algorithm is, for the problem size and processor numbers cited here, somewhat faster than the transpose plus parallel apply tridiagonal solver, cyclic reduction is a more complex parallel algorithm that does not scale as well. Consequently, with a large number of processors (P=n), the transpose method may be superior. Furthermore, with an efficient remote member function call, the transpose method will be superior for much smaller values of P.

5.2 An Anagram Group Finder

Our second example uses the *distributed_vector* class and a parallel version of the STL *sort* algorithm. The basic problem is to determine the anagram groups (groups of words which are permutations of each other) in a dictionary. This problem is described in [15] as an example of STL use.

The algorithm is as follows:

1. Read in words from dictionary. For each word, store it along with a key consisting of the word's letters sorted in alphabetical order.
2. Sort words in dictionary by their keys. Anagram groups are now stored consecutively.
3. Find the first two words with matching keys. These words form the beginning of an anagram group.
4. Compare the key with keys of subsequent words until a non-matching key is found.
5. Save the anagram group to a list containing other anagram groups of the same size.
6. If there are more words remaining, repeat steps 3, 4 and 5, beginning with the non-matching word found in step 4.

We store the initial word list from step 1 in a distributed vector. The vector elements are of type PS:

```
class PS {
public:
  char ordered_word[N];
  char word[N];
};
```

where N is an upper limit of the word length. The strings are fixed-size arrays because the current implementation of PSTL does not allow pointers as data members since there is no mechanism for remote stores and fetches to copy the data referenced by the local pointer. Comparison for ordering during the sort uses the following function object:

```
struct FirstLess : public binary_function<PS, PS, bool>{
  bool operator()(const PS& p, const PS& q) const
  {
    if (strcmp(p.ordered_word,q.ordered_word) < 0)
      return true;
    else
      return false;
  }
} firstLess;
```

The dictionary is read in by one processor and inserted into the distributed vector. In order to redistribute the data evenly, a *redistribute* operation is then done on the distributed vector. This moves elements from the single processor context and distributes them to the other processor contexts. The vector is then sorted using the PSTL *sort* algorithm, which is a parallel bitonic sort [14].

The anagram group finding portion of the algorithm is done in each context in a loop over the context's local elements. Step 3 is accomplished using the STL *adjacent_find* algorithm (which finds the first adjacent pair of equivalent elements) along with one extra check which is invoked when the current element is the last local element. In this case, the next (remote) element is retrieved and a check for a match of these keys on the boundary is done.

Step 4 uses the STL *find_if* algorithm to find the next key which does not match the key found in step 3. Again, since anagram groups can overlap more than one context, this checking must continue into the next context's elements if a non-matching key has not yet been found.

In Step 5, an STL map is used to store the newly discovered anagram group. A map consists of keys and their associated values. The key in this case is the anagram group size. The value is a list of parallel iterator pairs, with each pair marking the beginning and end of an anagram group. After processing is complete, a parallel reduction operation is used to sum the list length for each group size over all processors and the results are output.

This implementation was instrumented and run on the SGI Challenge and the IBM SP/2 with the following results. The dictionary consists of 403,200 words.

For 10-node SGI Challenge:

	Input	Redist	Sort	Group_Anagrams	Output	Total
P=1	8.07	0.00	23.22	1.11	0.00	32.41
P=2	8.19	1.24	20.75	0.59	0.50	30.82
P=4	8.18	0.76	14.84	0.30	0.10	24.21
P=8	10.95	1.40	12.87	0.20	0.30	25.74

For 24-node IBM SP/2:

	Input	Redist	Sort	Group_Anagrams	Output	Total
P=1	8.75	0.00	14.47	3.63	0.01	26.87
P=2	8.71	1.84	15.05	1.84	0.37	27.82
P=4	8.73	1.33	11.38	0.94	0.70	23.09
P=8	8.81	1.08	8.16	0.47	1.03	19.57
P=16	8.93	1.62	12.97	0.54	19.25	43.37

Again, we see degradation of performance in the runs with 8 (on the SGI) and 16 nodes (on the IBM SP/2), probably due to competition with other processes. The input step is fairly stable across all runs because words are read in by only one processor and then the redistribute step balances the elements among the processors. We could achieve speed-up on the input step by utilizing parallel I/O, but chose not to do this in these experiments since we are primarily interested in data structure performance. Redistribution does not occur in the 1-processor case, but there is some speed-up when the 2-processor case is compared to 4 and 8. This speed-up occurs because the primary operations are copying the vector segments from the first context and inserting them into the remote vector segment. As the number of processors increases, the size of the segment is reduced, resulting in smaller messages and shorter insert operations. The sort operation also shows speedup, but the major cost of this operation (about 2/3 of the time) is the local sort at the beginning and end of the bitonic sort. We used the HP STL *stable_sort* without modification (HP STL *sort* had a bug which inhibited its use in this program); we will look closely at this implementation to see where improvement might be made. As expected, the portion of the program which finds the anagram groups has very good speed-up. This is because the elements are evenly divided among the contexts and, except for the checks at the boundaries, all work for the local elements can be done in the local context.

6 Conclusion

This paper describes initial experiments with implementing the HPC++ Parallel Standard Template Library. While much work remains to be done, these experiments show the power of the HPC++ framework in terms of the expression of parallelism in applications. The experiments have also raised interesting questions about implementation details and their relationship to performance bottlenecks. In particular, the inefficiencies of remote fetches and other global pointer operations provide ripe ground for exploration. We plan to continue work on the PSTL implementation in the coming months.

References

[1] G. Agha. *Actors.* MIT Press, 1986.

[2] P. Beckman and D. Gannon. Tulip: A portable run-time system for object-parallel systems. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[3] William W. Carlson and Jesse M. Draper. Distributed data access in AC. In *Fifth ACM Sigplan Symposium on Principles and Practices of Parallel Programming*, 1995.

[4] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation, 1993. In *Research Directions in Concurrent Object Oriented Programming*, MIT Press.

[5] C. Chang, A. Sussman, and J. Saltz. Object-oriented runtime support for complex distributed data structures. Technical Report UMIACS-TR-95-35, University of Maryland Institute for Advanced Computer Studies and Department of Computer Science, March 1995.

[6] Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs.* MIT Press, 1993.

[7] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing '93*, 1993.

[8] Wayne Fenton, Balkrishan Ramkumar, Vikram Saletore, Amitabh B. Sinha, and Laxmikant V. Kalè. Supporting machine-independent parallel programming on diverse architectures. In *Proceedings of the 1991 International Conference on Parallel Processing*, 1991.

[9] Object Management Group. The Common Object Request Broker: Architecture and specification, July 1995. Revision 2.0.

[10] The HPC++ Working Group. HPC++ White Papers. Technical Report TR 95633, Center for Research on Parallel Computation, 1995.

[11] Yutaka Ishikawa. Meta-level architecture for extendable C++ draft document. Technical Report TR-94024, Real World Computing Partnership, February 1995.

[12] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1992.

[13] Scott Kohn and Scott Baden. Irregular coarse-grain data parallelism under LPARX. *Journal of Scientific Programming*, 5(3):185–202, Fall 1996.

[14] A. Malony, B. Mohr, P. Beckman, and D. Gannon. Program analysis and tuning tools for a parallel object oriented language: An experiment with the Tau system. In *Proceedings 1994 Los Alamos Workshop on Parallel Performance Tools*, 1994.

[15] David Musser and Atul Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 1996.

[16] The EUROPA Working Group on Parallel C++ Architecture SIG. EC++ - EUROPA Parallel C++ Draft Definition. Unpublished manuscript, 1995.

[17] Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective.* MIT Press, 1993.

[18] Rebecca Parsons and Daniel Quinlan. Run-time recognition of task parallelism within the P++ parallel array class library. In *Proceedings of the Workshop of Scalable Parallel Libraries*, 1993.

[19] Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report HPL-95-11, Hewlett-Packard Laboratories, January 1995.

[20] Bjarne Stroustrup. *The C++ Programming Language, Second Edition.* Addison-Wesley, 1991.

[21] Gregory Wilson and Paul Lu. *Parallel Programming Using C++.* MIT Press, 1996.

[22] Gregory Wilson and William O'Farrell. An introduction to ABC++. Unpublished manuscript, 1995.