HPC++ Whitepapers and Draft Working Documents.

The HPC++ working group.

1 Introduction

In 1994 a group of researchers began a series of discussion about building a common framework for parallel programming with C++. Our goal was to identify common language and library mechanisms that would allow a variety of parallel programming systems and libraries to work together and to make it easy to port applications across a wide range of computing platforms. In this chapter we describe some of the problems addressed by this group as well as some potential solutions.

The group of individuals involved in this discussion include, Scott Baden, Peter Beckman, Siddhartha Chatterjee, Andrew Chien, Janice Cuny, Barbara Chapman, Nikos Chrisochoides, Ian Foster, Dennis Gannon, Dirk Grunwald, Lars Hansen, L.V. Kale, Carl Kesselman, Allen Malony, Piyush Mehrotra, Bernd Mohr, Bill OFarrell, Joel Saltz, Vince Schuster, Alan Sussman, Tom Sheffler, Neelakantan Sundaresan, Steve Tuecke, Dan Quinlan, Greg Wilson, Beata Winnicka, Shelby Yang and Hans Zima. While this group is far from being in total agreement on the details of any plan, the have all contributed to the discussions that shaped this document. To give it a name, we refer to the framework and programming language which we hope will emerge from this process HPC++.

The questions that have been the most frequently discussed are as follows.

- *The Target.* What is the family of target architectures that HPC++ should run on?.
- *The Model.* What parallel programming model or models should HPC++ support? Who will be the users?
- *Language Extensions.* In what ways should the C++ programming language be extended to support parallelism? Are extensions needed? Are they a good thing?
- *The C++ Standards.* C++ has evolved rapidly. The *Standard Template Library* is part of the new C++ standard. How will HPC++ interact with STL?
- *Array Classes.* There is a pressing need to design an open standard for multidimensional array containers that can be used in scientific and engineering computation. How can HPC++ leverage existing work in this area?
- *Interaction with the Standards.* Most parallel programming is done in *Single Program Multiple Data* SPMD style using the standard message passing interface MPI. On the other hand distributed object oriented computing is increasingly based on the

Object Management Groups standard CORBA (Common Object Request Broker Architecture). How can HPC++ be designed so that it works in harmony with these standards?

- *Libraries and Tools.* What are the application libraries that should be provided with HPC++? What tools are needed to help programmers?

The first conclusion that this group reached was that we should attempt factor the problems of designing a common language and framework for parallel C++ programming into two areas: those areas that can be approached by using a C++ library and those areas that require extensions to the C++ language. Consequently, we have adopted a two level design for HPC++.

- Level 1 will be a complete library and set of tools and compiler directives that support parallel applications programming design on all high performance computing platforms that support a standard C++ compiler.

- Level 2 will be a set of language extensions that define the syntax and semantic basis for those parallel programming activities that can not be conveniently describe in the library. This extended language will also provide a foundation for a reference implementation of the implicit parallelism in the Level 1 library.

This report describes many of the preliminary conclusions we have reached in our discussions concerning the design of Level 1 HPC++. The document is divided into four section. In section 2 presents the draft proposal for the compiler directives that we feel are essential for the parallel execution of parallel loops. Section 3 outlines our requirements for a parallel extension to the Standard Template Library. Section 4 describes an extensive array container library intended for use in scientific computation. In the remainder of this section we discuss the target machine and programming models in more detail.

Several important topics that will become part of HPC++ are not discussed in this document. In particular, the definition of the HPC++ interface to active object libraries like ABC++ and Europa EC++ are far from complete. In addition work on an interface between HPC++ and CORBA is still in need of much further discussion. In the area of programming environment and performance tools more remains to be done and much of it must wait until a complete picture of HPC++ emerges.

1.1 Target Systems and Programming Models

There are three types of computing platforms that are of interest to parallel programmers and HPC++ must support programming on each type of system.

- *Shared Memory Multiprocessors (SMPs)* are computing systems with two or more processors which share direct read and write access to a large block of memory. The operating system on an SMP allows multi-threaded execution of an individual process and a scheduling mechanism that allows multiple processors to execute multiple threads concurrently. SMPs are the most common form of parallel computing systems. They range in scale from 2 pentium processors running NT , to mid-sized systems like the 16 processor power challenge, and up to hundreds of processors in the Convex Exemplar or Tera systems.

- *Massively Parallel Processors (MPPs)* consist of a large number (dozens to thousands) of processors connected to a distributed memory system by a high bandwidth switch that supports message passing . In addition, the switch also often provides limited shared memory operations as well as collective operations such as barrier synchronization, reductions and scans. Examples include the Cray T3D, Intel Paragon, IBM SP/2 and the Meiko CS-2. In the future we expect that the individual node computers of an MPP will be SMPs.

- *Networks of Computers* are systems that consist of a set of individual computers connected into a network and used as an ensemble. The individual computers may be of any type including MPPs and SMPs.

There is general agreement that HPC++ should provide technology to support programming in all three of these environments. In order to simplify the discussion we make the following definition.

1.1.1 Programming Models and Language Extension.

The question that has been most often discussed in the context of HPC++ is what, if any, programming model or models should it support. Several alternatives have already been mentioned and many others exist. The most common include

- *Control Parallelism.* Special control constructs like parallel loops, spawn statements, futures and parallel blocks work together with a set of synchronization primitives to provide multithreaded execution of user code. It is usually assumed that there is a shared memory and control begins with a single thread.

- *Data Parallelism.* A single thread of control running in a single virtual address space. Parallel operations are applied to aggregates of data. Parallelism may be nested in the most general model because aggregates may have elements that are parallel aggregates.

- *SPMD.* A Single Program is run on each processor. Based on the identity of the executing processor, each process operates on a different subset of the data. Each process has a private address space. SPMD is often used to emulate flat (non-nested) data parallelism.

- *Object Networks.* A set of objects interact by sending each other messages. The method invocations on different objects can execute concurrently. The object may share an address space, but if not, a mechanism for managing a distributed name space is provided so that object may point to each other across address spaces. There are several way in which the concurrency can be supported in this model. Active Objects are object which each have an associated virtual processor or thread. This is used in C++ libraries like ABC++ and Awesime as well as new languages like Java.

- *Object Oriented Macro Dataflow.* An object network can also be viewed as a demand driven data flow network. As described in the chapter on Mentat this can be a very powerful model.

There are many more programming models. In fact, the choice of a programming model is often driven by the nature of the application. The C++ language is designed to support many different sequential programming models including object oriented, generic programming, and standard imperative style with strong typing. It does not favor one point of view. This is accomplished in C++ by providing a set mechanisms that can be used in a wide variety of different ways. Consequently, the majority of the HPC++ contributors agree that HPC++ should follow the same philosophy. However, for the initial release of the HPC++ level 1 library there will be at least two models that will be well supported.

1. *Shared Memory Loop Level Parallelism.* We define a *context* to be a a process with a single address space and access to one or more processors. When an HPC++ program starts execution in a single context it begins life as a sequential program running in a single thread. However, HPC++ compiler directive and some of the HPC++ library functions allow the program to execution certain loops and STL algorithms in parallel by using the other processors available to the context.

2. *Multi-Context SPMD Parallelism.* In most non-shared memory MPP systems (and SMP clusters) the standard mode of execution is to load the same program object on each SMP node. In this case HPC++ provides a SPMD programming model. A single thread of execution in each context start the program and collective operations and message passing can be used to communicate and synchronize between contexts. The HPC++ loop directives will, in this case, extend to multi-context program and the HPC++ library will provide distributed-parallel STL implementations and a distributed array library.

As the HPC++ committee completes work on a CORBA and active object library, new programming models will be supported.

2 Loop Parallelism

HPC++ provides a method to parallelize well-behaved "for" loops. Compilers can utilize the HPC_INDEPENDENT directive to parallelize the loop for SMP systems as well as for non-SMP parallel systems. Two pragma directives are supplied. Each of these has been defined so as to facilitate SMP loop parallelism in ANSI C as well. It should be noted that this HPC_INDEPENDENT facility can be expanded greatly if need and compiler capability dictate.

The execution model for the HPC_INDEPENDENT loop is SPMD for distributed memory machines; however, execution on an SMP parallel system may differ significantly. Also, even though the execution model is SPMD, each processor may be executing on widely different portions of code. Note that the HPC_INDEPENDENT allows for a capability similar to a parallel switch statement. On an SMP parallel system, one might have a single thread of control before and after loop execution but multiple threads may be used to execute the loop using either static or dynamic scheduling. On a non-SMP parallel system, multiple processors may all operate on different chunks of the loop depending on how objects are partitioned. On hybrid systems containing both SMP and non-SMP processors, both methods might be employed.

The first `#pragma` added to HPC++ is:

```
#pragma HPC\_INDEPENDENT [PRIVATE (variable\_list)] [ON HOME(mapped\_variable)]
```

where variable_list is a comma separated list of one or more variable names that have previously been declared and active at the scope of this `#pragma`.

The first statement following this `#pragma` must be a "for" statement of the form:

```
for ( [<initialization and optional local declaration of  induction
       variables>] [, <other inits>];
        <termination condition of induction variables only>;
        <induction variable modifications> ) {
        <body>
}
```

The HPC_INDEPENDENT pragma asserts that each iteration of the loop is independent of any other iteration and that each iteration of the loop may be executed in any order.

The actual number of iterations of an HPC_INDEPENDENT loop must either be calculable at compile-time or at run-time prior to loop entry.

The optional PRIVATE qualifier asserts that each variable in its variable_list is active prior to loop entry and may be duplicated for each iteration of the loop. PRIVATE variables may not be pointers. The initial value of each PRIVATE variable is carried into each iteration of the loop. The final value of a PRIVATE variable upon completion of the loop is the same as the initial value if the PRIVATE variable is not modified in any iteration of the loop. The final value of a PRIVATE variable is undefined upon completion of the loop if the PRIVATE variable is modified during any iteration of the loop.

The optional ON HOME qualifier is generally useful only when parallelizing for non-SMP parallel systems and indicates to the compiler how to best partition the iteration space of the

”for” loop. The ON HOME’s mapped_variable must refer to a explicitly partitioned array object that is active prior to loop entry. ¡NOTE: Much of the ON HOME wording is expected to be taken from the ON HOME wording for INDEPENDENT DO loops in proposed HPF 2.0. This will be done after getting more explicit description of the partitionment of objects.¿

The <body> of the loop may contain zero (0) or more pragmas of the form:

```
#pragma HPC_REDUCE.
```

The first statement following this pragma must be a statement of the form:

```
reduction_variable := reduction_variable reduction_op expr;
```

or

```
reduction_variable := reduction_function(reduction_variable, expr);
```

where ”reduction_variable” is a scalar or array-element-scalar of a primitive type (e.g. int, double) that has not been used in an enclosing PRIVATE qualifier; where ”reduction_op” is one of the commutative non-overloaded binary operators ”+”, ”-”, ”*” or ”/”; where ”expr” is any conformant C++ expression not referencing ”reduction_variable” and does not contain any side-effects; and where ”reduction_function” is one of ”max” or ”min”.

The HPC_REDUCE pragma facilitates parallelization of primitive reduction operations and must be used to identify each reduction within the <body> of the HPC_INDEPENDENT loop. A reduction statement immediately following the #pragma HPC_REDUCE is the only statement allowed within an HPC_INDEPENDENT loop which may have a carry dependence.

The degree of parallelization of an HPC_INDEPENDENT loop is implementation dependent. Compilers are free to ignore the HPC_INDEPENDENT pragma as well as any of the qualifiers. However, if the loop is parallelized, compilers must process all #pragma REDUCE properly within the enclosing loop.

The <body> may contain additional HPC_INDEPENDENT pragmas (i.e. HPC_INDEPENDENT loops may be nested) but the degree of parallelization is implementation-dependent.

Calls to other functions may be performed during execution of <body> subject to the constraints below.

Constraints:

- C1: Induction variables must not be modified within the <body> of the loop. Induction variables but be declared in a PRIVATE qualifier enclosing the loop or the scope of the induction variable must be declared local to the loop.

- C2: No carry dependence of any variable within the loop is allowed in the loop except for reductions preceded by a ”#pragma REDUCE”. This implies that, except for marked reductions, the value assigned to any variable in one iteration cannot be used in any other iteration.

- C3: Any function called within an HPC_INDEPENDENT loop must not have side-effects and must be re-entrant. Note that this places extra requirements on standard C++ library routines that may be called. HPC_INDEPENDENT loops may be used within these functions but the degree of parallelization is implementation-dependent.

- C4: I/O performed within HPC_INDEPENDENT loops is undefined.
- C5: Any transfer of control to a branch target statement outside the `<body>` is not allowed. Long jumps are prohibited.
- C6: Memory may not be allocated by the user while executing the `<body>` of the loop. NEW may not be used while executing the `<body>` of the loop.
- C7: Activation of exception handlers while executing the `<body>` may result in undefined behavior.
- C8: All induction variables must be of signed or unsigned "int", "long", "short", or "char" or one of the parallel STL iterators.
- C9: Users must be cautious in using the HPC_INDEPENDENT capability. While good parallelizing compilers may be able to often warn the user at compile time or at runtime about interference between loop iterations or about potential race conditions, it is not possible to detect all errors and it is up to the user to ensure that the loop is indeed truly independent between loop iterations.

3 The STL and Parallel Iterators in HPC++.

The C++ Standard Template Library document defines the concept of an iterator to be an object that is used to traverse a data structures in the same way that pointers are used to step through arrays in C. Iterators are the parameters for the algorithms in the STL library.

There are five categories of standard iterators which are each defined by their of behavior with respect to the operators "*", "[ ]", "+", and "++". These categories are *random access*, *bidirectional*, *forward*, *input* and *output*.

In HPC++, we add a new categories of iterator: *parallel random access*. The semantics of parallel iterators are identical to those of the random access iterator with the exception of when they are used to instantiate an algorithm that takes a function object as an argument. In such cases, the implementation of the algorithm is permitted to apply the function objects to elements of the container concurrently. It is the responsibility of the user to ensure that function objects used in conjunction with parallel iterators will function correctly.

To support use of parallel iterators, the following definitions augment in the STL definition for HPC++:

```
// We derive from the non-parallel iterator tag so that algorithms
// that don't care about parallel iterators don't have to worry
// about them.
struct parallel_random_access_iterator_tag :
    public random_access_iterator_tag {};

template <class T, class Distance>
inline parallel_random_access_iterator_tag
iterator_category(const parallel_bidirectional_iterator<T, Distance> &) {
    return parallel_bidirectional_iterator_tag();
}

template <class T, class Distance>
inline parallel_random_access_iterator_tag
iterator_category(const parallel_random_access_iterator<T, Distance> &) {
    return parallel_random_access_iterator_tag();
}

template <class T, class Distance>
inline T* value_type(const parallel_random_access_iterator<T, Distance>&) {
    return (T*)(0);
}

template <class T, class Distance>
inline T* distance_type(const parallel_random_access_iterator<T, Distance>&) {
    return (Distance*)(0);
}
```

As with regular iterator classes, the *iterator_category* function can be used to apply specialized algorithms to parallel iterator. For example, the following example shows how a parallel for each statement can be written. Notice how the iterator category is used to select an appropriate algorithm at compile time:

```
template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f)

{
  return for_each(first, last, f, iterator_category(first));
}

template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f,
  random_access_iterator_tag)
{
  while (first != last) f(*first++);
  return f;
}

// Preserve for_each functionality on other iterators.
template <class InputIterator, class Function>
Function _for_each(InputIterator first, InputIterator last, Function f,
   forward_iterator_tag)
{
  while (first != last) f(*first++);
  return f;
}

// Parallel implementation of for_each.
template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f,
          parallel_iterator_tag)
{
  return _parfor_each(first, last, f, value_type(first));
}

// Parallel iterator using CC++ parfor
template <class InputIterator, class Function, class T>
Function _parfor_each(InputIterator first, InputIterator last, Function f, T
*) {
#pragma INDEPENDENT
  for (T val = *first; first != last; first++, val = *first) f(val);
  return f;
}
```

Define reverse iterator types and insert iterator types.

To make it possible to use standard C arrays with the parallel iterators we define the following iterator adaptor.

```
template <class T, class ptrdif_t> class par_iterator;
```

which can be used as follows.

```
double x[1000], y[1000];
par_iterator<double, long> px(x), py(y);

partial_sum(px, px+1000, py); // parallel prefix operation
```

In addition, we can use parallel iterators as induction variables in parallel loops. For example,

```
double x[1000], y[1000];
par_iterator<double, long> t, px(x), py(y);

#pragma INDEPENDENT
for(t = px; t < px+100; t++, py++)
    *t = 3.14*(*py) + *t;
```

3.1 Parallelizable Containers

The HPC++ version of STL defines the notion of a parallelizable container. A parallelizible container can be located within a single context or distributed. A distributed container consists of more than one context. The data within a context is assumed to organized into a contiguous "slice". Parallelizable containers have the following characteristics:

Characteristic	Notes
X::parallel_iterator	Type of the parallel iterator for class X
X::const_parallel_iterator	Type of the constant parallel iterator for this class
a.begin()	returns an iterator to the "local" section of the context on which the call executes.
a.end()	returns an iterator to end of the local section of the context on which the call executes.
a.pbegin()	returns a parallel iterator to the start of the container
a.pend()	returns a parallel iterator to the end of the container.
a.contexts()	returns the number of contexts contexts
a.distribution()	return a distribution object of some sort?

We can also have parallel reversible containers. These have the following additional characteristics:

Characteristic	Notes
X::parallel_reverse_iterator	Type of the reverse parallel iterator for class X
X::const_parallel_reverse_iterator	
a.prbegin()	returns a parallel reverse iterator to the start of the container
a.prend()	returns a parallel reverse iterator to the end of the container.

In addition to the standard container classes, HPC++ STL defines the following additional template classes:

```
template <class T> class distributed_vector<T>;
template <class T> class distributed_list<T>;
template <class K, class T> class distributed_set<K,T>;
template <class K, class T> class distributed_map<K,T>;
template <class K, class T> class distributed_multimap<K,T>;
template <class K, class T> class distributed_multiset<K,T>;
```

All of these classes take a distribution argument as a constructor argument.

3.2 Distributed Parallel Iterators

In HPC++ we require a distributed parallel iterator to satisfy all the conditions for a parallel random access iterator `p` with the following exceptions. Let `T` be the base type of the container over which `p` iterates.

1. The dereference operator, $*$ cannot be used to cast `p` to `T &`. This is because the iterator may point to a part of the data structure that is not in the local address space. The value of `*p` will depend upon the context. In the simple case where a value of type `T` is required `*p` returns the appropriate value from the remote location. However when used in an assignment operation of the form

   ```
   *p = ....
   ```

   `*p` will be of type `T &` only if `p` is currently pointing to a local memory object. Otherwise, `*p` will be a proxy to a remote reference. The assignment operation will behave as expected in all cases, however any updates to remote memory will not be considered valid until after the following barrier synchronization.

2. The same behavior applies to the `[ ]` operator.

4 Overview of the HPC++ Array Class

Scott Baden, Daniel Quinlan, John Reynders

The purpose of this document is to propose a standard multidimensional array class sufficient for general numerical computation on both serial and parallel computers. It is proposed that such an array class library would form the basis for an evolving array language syntax within HPC++ (so called "level 1 in HPC++.").

This array class supports Fortran-90 style pointwise array operations, some whole array intrinsics, triplet indexing notation, and HPF-style BLOCK array decomposition. User-defined data distributions are supported, as well as a *domain abstraction* that enables information about the distributed structure of data to be represented separately from the data itself. The array class does not support matrix operations, nor grid functions specific to the numerical solution of partial differential equations. The HPC++ array class provides a more minimal set of tools and mechanism to permit development of higher level applications and application specific libraries.

The motivation for using an approach based on a class library is that it is inherently simple and uses the C++ language features as its principle means of implementation. A class library requires no compiler support and thus can work on a wide range of architectures.

The programming model is similar to FORTRAN 90, with HPF-like extensions to support parallel execution. The HPC++ array class library assumes an SPMD simulation of a data parallel model on distributed memory parallel computers. On shared memory multiprocessors the programming model is sequential, but the array operators are implemented to run in parallel. The programming model has proven valuable in permitting source code developed serially to be easily recompiled and run in the parallel environment without change.

As an example of an array operation we show simple 1D Jacobi relaxation on the Laplace equation with zero Dirichlet boundary conditions:

```
   Range I(1,8);  // base and bound of range object
   Array<double> X(10);
   X (I) = 1;
   X(0) = X(9) = 0;
// One iteration of Jacobi relaxation
   X(I) = (X(I+1) + X(I-1)) / 2.0;
```

Many array class libraries have been proposed and implemented, see, for example, the *valarray* proposal.[1] However, to our knowledge, none of these proposals support multidimensional arrays directly or address issues of parallel computation. To provide a concrete implementation of an array class, we will base level-1 HPC++ on the A++/P++ array class libraries developed at Los Alamos, which have been in use for the past two years.[2]

The HPC++ array class will be an efficient array class for numerical work, and will also serve as a container class for other user defined objects. Thus templating of the array class is both a natural part of its C++ design and a feature that will simplify its implementation for

[1]See URL: `http://www.cygnus.com/misc/wp/draft/lib-numerics.html`.

[2]A++ is the serial array class and P++ is the parallel array class with the same interface. More information is available at: `http://www.c3.lanl.gov/~dquinlan/A++P++.html`.

the many primitive types required for numerical work. The HPC++ array class is oriented toward numerical work and may not be the most natural container class for all objects. Other container classes maybe more natural in some contexts.

4.1 Terminology

HPC++ array objects will be given the name: `Array<type>.` Indexing objects specified by a base, bound, and stride are called `Range.` Indexing objects specified as a Cartesian product of Range are called: `Region.` We assume a parallel computing platform consisting of a set of compute nodes, called processors below. Each processor consists of a shared address space and one or more CPUs.

Each of the array class member functions (with one exception) operates as an SPMD collective operation. That is, in SPMD mode, if `foo()` is a member function of the array class, then if `A` is an array, then the operation `A.foo()` is a barrier operation, i.e. one thread in each context must make the same call before any is allowed to return.

4.2 Standard Template Library compliance

The HPC++ array class will have functionality consistent with the Standard Template Library (STL). This means that indexing of HPC++ array objects can alternatively be handled though the use of STL like iterators. This is discussed in greater detail at the end of this section.

4.3 Overloading of indexing operator

HPC++ will overload the operator() to enable indexing of subsections of multidimensional array objects. Overloaded versions of the operator will be provided for all the indexing operators: scalars, Range objects, and Region objects.

In addition HPC++ will overload operator() for objects of type Point, but in this case the access is unsynchronized (NOT and SPMD collective). The value returned in this case is of type `Global_Ref<T>` which is a proxy for a remote reference. This is the only operator in the HPC++ array class that is not a collective operation.

4.4 Overloading of arithmetic operations

Standard array and mixed array-scalar arithmetic operations and math functions (sin, cos, tan, sqrt, etc.) will be provided with an element-wise semantics. These operations are too numerous to detail within this document.

4.5 Deep Copy Constructors

Copy constructors should be deep by default. An optional parameter which defaults to forcing a deep copy is implemented, the use of this parameter permits the specialized use as a shallow copy constructor.

```
Array<T> ( const Array<type> & X , bool DEEP_COPY = TRUE );
```

4.6 Independent base within multidimensional arrays

HPC++ arrays may have an arbitrary origin. Unlike C++ arrays, the origin is not zero-based. This property is encompassed throughout the design of the HPC++'s array facility, e.g. copy constructors and arrays section selections will preserve the base of copied array object within the new array object.

Thus in,

```
Array<type> A(10);
Range I(5,6);
```

the base of `A(I)` is 5, not zero or anything selected by default. Thus in,

```
Range J (6,6);
A(I)(J) is a reference to the 6th element of A.
```

4.7 Multidimensional Index Spaces

HPC++ supports a Region object that is an extension of the one dimensional Range object to higher spatial dimension. Such a Region object may be used to index HPC++ array objects. Shift member functions provide the mechanism for manipulating the Region with stencil-type operations. A set of *Region calculus* operations enables the programmer to express geometric operations like region intersection, region growth, and scaling which are using in managing data motion in irregular applications. *Region calculus* operations must be rank conforming, and only operations that are closed on the set of rectilinear regions are defined. (e.g. There are no union operations since union is not generally closed over the set of Regions.)

```
Array<T> A(100,100);
Range I(2,5), J(4,8);
Region R (I,J);
A(R) = 1;
Region S = R.grow(1);
A(S) = A(S) + 1
```

This implies that the operator() should be defined to accept combinations of Region and Range objects.

Pointwise Regions generalize the `Region` Class. Pointwise Regions generalize the `Region` Class. Operations like intersection, union, set difference, and so on have obvious interpretations, but these have been left out of HPC++ because their implementation depends on the problem structure—compare for example, an irregular Region comprising many isolated points with a square annulus. For now, we disallow Region objects to define more than a single Cartesian product of one dimensional index spaces, since these may We may implement these as either as a Pointwise_Region or as `Array <Region>`.)

A similar and equally natural extension of the indirect addressing for multiple dimensions is also useful. Indirect addressing via the `Pointwise_Region` object is permitted. The result is an array subsection defined only by the points specified in the Pointwise_Region input to the operator(). The linear ordering of the result is preserved, but the elements are renumbered, starting from zero.

```
Array<T> A(10,10);
Array<int> I(3), J(3);
I(0) = J(0) = 1;
I(1) = J(1) = 5;
I(2) = J(2) = 7;
Pointwise_Region R(I,J);
A(R) = 6;   // assignment of 6 to elements A(1,1), A(5,5), A(7,7);
Array<double> B;
B = A(R);  // builds array of length 3  (element indices are renumbered)
```

4.8 Where statement

The only way we know to cleanly implement the `where` statement within the array class is to use a macro to simplify it being turned on and off as an internal state. This will be a part of the HPC++ array class, but we suggest that it be incorporated into level 2 functionality at a later stage of HPC++ development to avoid the use of a macro.

4.8.1 Relationship to indirect addressing

The `where` statement uses a mask which is simply an HPC++ `Array<bool>` and not a separate type. This simplifies the implementation and provides the same functionality for the mask array as provided for other array objects. There is a correspondence to indirect addressing though the

```
Array<int>::Array<int> ( const Array<bool> & X );
```

constructor.[3] Thus a mask array is not a separate type and a conversion facility exists between the mask returned by the `Array<T>` relational operators and the indirection arrays used for indirect addressing.

- *Technical note 1.* The use of the "where" statement also makes the lowest level array operations more complex since each operation must either use on not use a mask type within its inner looping structure.

- *Technical note 2.* Other array classes (including the *valarray* proposal) define a separate mask array type, We recommend that HPC++ not do this. (The implementation and

[3] We will provide a special member function to do the conversion if the type propagation is problematic.

interface is greatly simplified by using the integer array for the index array required in indirect addressing.) If the array class is templated then it might be a superior design to use the "bool" as a separate mask type.

4.9 Interaction with standard C arrays

The HPC++ array class will provide for operations which mix standard C arrays with array objects. The mechanism is limited due to the primitive nature of C arrays (a pointer lacking access to descriptor information). For distributed arrays it is the users responsibility to assure that the C array on each processor is conformable in size with the local part of the HPC++ distributed array. Constructors are provide with HPC++ that allow creation of array objects from pointers to existing data, but these require sufficient additional information to define the global and local array descriptors (local and global size information for each axis is sufficient).

Pointers to the internal data of HPC++ array objects are also accessible to permit the use of HPC++ array objects with existing C or FORTRAN (or HPF) applications.

4.10 Null Arrays

Array objects created without specification of size along any dimension creates a "null" array. Such a "null" array has no data associated with it (i.e. zero length in each axis). Operations on "null" arrays result in "null" arrays. The existence of "null" arrays and this behavior in their operation permit the SPMD/Data Parallel operation on processors not containing data (i.e. on processors not an active part of an array's distribution). Indexing objects can similarly be "null", operations on arrays using "null" indexing objects result in "null" array objects.

4.11 Distributions

HPC++ enables the programmer to define the distribution —or partitioning— of array objects across the nodes of a distributed memory parallel computer via a distribution object:

```
Distribution D;
Array<double> A (100,100,D);
```

The distribution of an array assumes a processor map, which is a logical arrangement of processors onto a one dimensional array. The one dimensional processor array provides an implicit ordering of processors.

```
Processor_Map P();  // default processor map
Processor_Map Q(N); // N processor map
```

Subseting of the Processor_Map permits the restriction of the Processor_Map objects to fewer processors. The mechanism for subsetting is the specification of a contiguous range of an existing Processor_Map object via indexing. More general methods of subsetting processor maps may be added later.

```
Range I(3,7);
Processor_Map P = Q(I);  // P is a subset of processor map Q
```

HPC++ defines an implementation-dependent *default* distribution for each array. The default distribution object is a static object within the HPC++ array class library and can be manipulated to achieve different default distributions. It is expected that the default distribution would be the most commonly used means of defining distributed HPC++ array objects.

Array objects wholly located on a single processor are considered distributed either over a `Processor_Map` containing a single processor, or distributed over a `Processor_Map` containing more than one processor but with a distribution object specifying the partitioning over a single processor.

HPC++ array objects can either be distributed or replicated across processors. Because of the programming model, SPMD simulation of data parallel execution, all processors contain the replicated array. The replication of HPC++ array objects is achieved through the specification of a distribution object that provides replication for all arrays build using it.

```
Distribution D(REPLICATE); // static declaration
Array<float> A(100,100,D); // replicated array
Distribution E();
E.redistribute(BLOCK,BLOCK); // dynamic declaration
Array<float> B(100,100,E);   // distributed array
E.redistribute(REPLICATE);   // dynamic declaration
Array<float> C(100,100,E);   // replicated array
```

Such replicated objects are useful when small array objects are required which would not require parallel operations. Operations on such arrays are done redundantly in parallel on each processor. Such serial arrays are most often trivially small and not worth the overhead of distribution and the communication required for parallel operations. It is the responsibility of the user to guaranteed that replicated arrays have consistent values on all processors.

HPC++ supports two types of distributions:

- *Regular Blocked Distributions.* This consists of the Fortran-D/HPF style distributions BLOCK, and WHOLE. This distribution system is adequate for many applications. It is simple and permits quick compiler analysis. Such methods are already in use within several compilers including HPF and Vienna Fortran. (Note: CYCLIC, BLOCK_CYCLIC, and other sorts of partitioning methods may also be added later on.)

  ```
  Distribution D(BLOCK, WHOLE, P);
  Array<double> X(100, 100, D), Y(100, 100, D);
  D.redistribute(WHOLE,BLOCK);
  Array<double> Z(100, 100, D);
  ```

- *Table Based Distributions.* A more flexible extensible distribution scheme which enables the user or library to manage customized distributions. This method supports a richer set of distributions than the Blocked Distribution, but at a possibly higher cost of table lookup to obtain distribution data.

The table based distribution allows each processor in the processor map to have entries which specify the local Range of the global index space. Access functions are provided to insert and extract information from the table. The table contains one entry for each distribution object and is tied implicitly to a Processor_Map object.

```
Processor_Map P (10);
Distribution D(P);
Region X[10];
for (int i=0; i < P.getLength(); i++)
        {
          int Base  = i * (Size / P.getLength(0));
          int Bound = (i+1) * (Size / P.getLength(0)) - 1;
          D (i) = Region(Range (Base,Bound));
        }
Array<double> X( Size, D), Y( Size, D);
```

The purpose of this two tier approach to partitioning is to provide the user with the flexibility to utilize an improved distribution scheme in applications that can benefit from customization. This is particularly important in irregular applications.

Multiple distribution objects may be created, manipulated, and associated with different HPC++ array objects. We expect the most common use of distribution objects will involve the HPF style distribution features. Libraries that require different distributions should derive new distribution mechanisms from the more general HPC++ distribution objects.

4.11.1 Binding of distributions to array objects

HPC++ distributions are bound to arrays though calls to the virtual **reference** member function. Within HPC++ the distribution object implements **reference** as a deep copy. User defined distributions derived from the HPC++ distribution object have the flexibility to force binding of distributions to HPC++ array object either by reference or value through specification of a new **reference** function within a derived class. To permit greater functionality the **reference** function takes the HPC++ array object as a parameter.

The modification of an HPC++ array object after use with the constructor of an HPC++ array object does not propagate back to previously associated HPC++ array objects. As an example of derivation from an HPC++ distribution object, this functionality may be added to an application specific distribution.

```
template <class T>
 class Better_Distribution : public Distribution {
   public:
      list<Array<T> > Associated_Arrays; // using the STL list container
      Better_Distribution () {};
      Better_Distribution (Processor_Map & P) : Distribution(P) {};
```

```
        virtual reference ( const Array<T> &X )
           { Associated_Arrays.push_back(X); }
        virtual unreference ( const Array<T> &X )
           { Associated_Arrays.remove(X); }
        void redistribute ( ... )
           {
          // back propagate change of distribution to
          // previously associated HPC++ Array<double> objects
             for (list::iterator p = Associated_Arrays.begin();
                          p != Associated_Arrays.end(); p++)
                 p->redistribute( ... );
           }
      };

   Distribution D();
   Array<double> A(100,100,D);  // uses default distribution
   D.redistribute(BLOCK,BLOCK); // does NOT redistribute A
   Array<double> B(100,100,D);  // distributed BLOCK,BLOCK

   Better_Distribution E<double>(BLOCK,WHOLE);
   Array<double> Y(100,100,E);  // distributed BLOCK,WHOLE
   Array<double> Z(100,100,E);  // distributed BLOCK,WHOLE
   E.redistribute(BLOCK,BLOCK); // redistributes Y and Z BLOCK,BLOCK
```

4.11.2 Derivation of specialized distributions

Both user defined Processor_Map objects and user defined Distribution objects maybe derived from the HPC++ classes to provide a mechanism for customized distributions and control of distributions. As an example, multidimensional processor arrays could be derived from the HPC++ Processor_Map object. And more sophisticated distributions which combine the replication and distribution of axes could be derived from the HPC++ Distribution object. Thus HPC++ provides users/library builders with a mechanism to specify more specialized application specific distributions though the object-oriented features of C++.

As a trivial example we can build a specialized distribution object which restricts the partitioning to a single processor.

```
class Single_Processor_Distribution : public Distribution {
// restrict to processor zero
   Single_Processor_Distribution () : Distribution(0) {};
// restrict to first processor of "P"
   Single_Processor_Distribution (Processor_Map & P) :
      Distribution(P(0)) {};
          ...
        };
```

4.11.3 Distributions as an optimization issue

It is important that the semantics of the computations involving arrays be independent of their distribution, modulo roundoff. In other words, the user's code should always give the same answers regardless of how the arrays are distributed or redistributed. On the other hand, performance is likely to be dramatically affected. This forces HPC++ to provide the means to allow array operations even where the operands have different distributions or change their distributions dynamically, as in the following:

```
Distribution D(BLOCK, BLOCK, P);
Distribution E(BLOCK, WHOLE, P);
Array<double> X(100, 100, 100, D);
Array<double> Y(100, 100, 100, E); // Y and Z have different distributions
Array<double> Z(100, 100, 100, D);
Y = Z = 5;          // assignment of 5 to all elements of Y and Z
X = Y + Z;          // array operation (all operands aligned) slower
                    // due to lack of alignment
Y.redistribute(D); // redistributes X according to distribution D
X = Y + Z;          // array operation (Y in not aligned for X or Z)
                    // X,Y, and Z are aligned for faster operation
```

4.11.4 Member functions of HPC++ Distribute

The design of the HPC++ distribution object includes virtual functions to access mappings of processors to arrays and array subsections, and mappings of array and array subsections to processors. Objects derived from the HPC++ Distribute may impose restrictions to provide more efficient access to either the *Regular Blocked Distributions* or the *Table Based Distributions*. The HPC++ Distribute interface includes virtual functions:

```
Array<int> getProcessorsOwningRegion ( const Array<T> & Target_Array ,
                                       const Region & Space );
Region getRegionOwnedByProcessor ( int Processor );
```

4.12 HPC++ Array Iterators

In the case of the HPC++ array class there is a built-in parallel random access array iterator

```
Array<T>::parallel_iterator
```

which is defined in `Array` as the subclass

```
template <class T, class ptrdif_t>
  class parallel_iterator{
      public:
        T& operator[](int);
        T operator[](int) const;
        T& operator *();
```

```
        T   operator *() const;
        parallel_iterator<T, ptrdif_t> & operator ++();
        ... // and the other standard random access iterator operators
        operator Point (); // cast to a point
};
```

and there are two standard predefined iterators *pbegin* and *pend* of this type that are defined for each array. Note that these are 1 dimensional iterator for a multidimensional structure. In that sense, they represent a "space filling curve" for the array object.

The only non-standard iterator operator is the Point cast operator. In the HPC++ array library it is possible to access the elements of an array by an index through a point which is a tuple in an index space. By being able to cast an array iterator back to the array index space we can build functions that can be used for complex data manipulations of multiple arrays. For example, a simple relaxation is given by

```
Array<double> A, B;
Array<double>::parallel_iterator p;
Point n(1,0), e(0,1), q;

  #pragma INDEPENDENT
  for(p = A.pbegin(); p!= A.end(); p++){
      q = (Point) p;
      A(q) = A(q) - 0.25*( B(q+n) + B(q-n) + B(q+e) + B(q-e) )
    }
```

4.12.1 Higher Dimensional Array Iterator

A more natural extension to C array semantics in higher dimensions is as a nested sequence of iterator types. For example,

```
  double x[100][50][20];
```

defines a 3 dimensional structure. If we use the name of the array as an argument to a function expecting an iterator, then the type of that iterator is `(double *)[50][20]` . In other words `*x` or `x[i]` is an array of type `double[50][20]` . And `x[i][j]` is an array of type `double[20]`.

In order for HPC++ array containers to maintain first class status as C++ arrays, they must also provide parallel iterators of higher dimension. The is accomplished with the a family of iterator templates of the following form. There are two classes. One is a class for an iterator of dimension 2 and a parameterized class for dimensions greater than 2.

```
//iterators of dimension greater than 2
template < class T, class ptrdif_t, int k> class parallel_iteratorK{
     public:
        parallel_iteratorK<T, ptrdif_t, k-1>  operator[](int);
        parallel_iteratorK<T, ptrdif_t, k-1>  operator *();
        parallel_iteratorK<T, ptrdif_t, k> & operator ++();
```

```
        ... // other random access iterator forms
}
```

and in the case of dimension 2,

```
template < class T, class ptrdif_t> class parallel_iterator2{
     public:
        parallel_iterator<T, ptrdif_t>  operator[](int);
        parallel_iterator<T, ptrdif_t>  operator *();
        parallel_iterator<T, ptrdif_t> & operator ++();
        ... // other random access iterator forms
}
```

The array class will also contain members pbeginK() and pendK() for each K in the range from 2 to 8. The relation between these is given by

```
  pbeginK()[0] == *pbeginK() == pbeginK-1()
  pbegin2()[0] == *pbegin2() == pbegin()
```

Having the higher dimensional iterators allows the construction of generic functions which will work with either standard C arrays or the HPC++ array containers.

```
template< class vec_iterator, class array_iterator >
void matvec(int n, array_iterator a, vec_iterator x,
             par_vec_iterator y){
   // y = a*x matrix multiply for an n by n matrix.
   #pragma INDEPENDENT
   for(int i = 0; i < n; i++){
        y[i] = 0.0;
        for(int j = 0; j < n; j++)
            y[i] += a[i][j]*x[j];
       }
```

4.12.2 Distributed Parallel Iterators for HPC++ Arrays

A more complex situation involves iterators that are bound to distributed HPC++ array containers which are used in SPMD computations. Let $S \subset Z^n$ be the index space In these cases, the array is allocated to processor contexts by means of a pair of functions as follows. Let $S \subset Z^n$ be the index space of the array. A distributed array defines two maps

$$Context : S- > Z$$

$$LocalAdress : S- > Z$$

where *Distribution* maps array index tuples into integer context identifiers and *LocalAddress* maps tuples to a local address within a context. The distribution is well defined if the map

$$(Context, LocalAddress) : S- > Z \times Z$$

defines a unique value for each index in the array. We refer to the pair $(Context, LocalAddress)$ as a *global address*.

An iterator over a distributed array can be viewed as a map from the integers to a "space filling curve" in the array. Because the distribution of the array over contexts induces a partition of the array, it also then induces a partition of the "space filling curve" traced out by the iterator. Consequently the iterator steps through address spaces as it traces it path through the array.

In the case of the HPC++ array containers and parallel loops we must consider the problem associated with partitioning the work as well as distributing the data. For example, consider the following loop.

```
Distribution D1, D2, D3;
Array<float> X(.., D1), Y(..,D2), Z(..,D3);
Array<float>::parallel_iterator x, y, z;

#pragma HPC_INDEPENDENT
for(x = X.pbegin(), y = Y.pbegin(), z = Z.pbegin();
    x != X.pend(); x++, y++, z++){
         *x = *y + 3.14*(*z);
}
```

In this case, it is the task of the compiler to divide the computation into segments based on the values of the iterators. By taking the number of required iterations `X.pend() - X.pbegin()` we can divide that by the number of processors participating in the computation. In this case the compiler will transform the program above into the following code segment.

```
Distribution D1, D2, D3;
Array<float> X(.., D1), Y(..,D2), Z(..,D3);
Array<float>::parallel_iterator x, y, z;

  int N = (X.pbegin()-X.pend())/total_number_of_processor();
  int first = this_processor_id()*N;
  int last = (this_processor_id()+1)*N;
  x = X.pbegin(); y = Y.pbegin(); z = Z.pbegin();
  for(int i = first; i < last; i++)
    x[i] = y[i] + 3.14*z[i];
  }
  barrier();
```

It should be noted that this translation of the program will work completely independent of the distributions of the array. That is, changing the distribution will not effect the result of the computation. However, it may be very inefficient for distributions of `X` that do not have the i^{th} block of N iterate values in the same address space as the i^th processor. Consequently, to support the ON_HOME(X) directive qualifier, each HPC++ array object should also provide two additional iterators: `begin` and `end` and an integer `local_stride` that describes the stride of the local set of container elements. In this case the transformed program takes the form shown below.

```
Distribution D1, D2, D3;
Array<float> X(.., D1), Y(..,D2), Z(..,D3);
Array<float>::parallel_iterator x, y, z;

  int first = X.begin - X.pbegin;
  int last = X.end - X.pbegin;
  x = X.pbegin(); y = Y.pbegin(); z = Z.pbegin();
  for(int i = first; i < last; i += X.local_stride())
    x[i] = y[i] + 3.14*z[i];
  }
  barrier();
```

However, this translation scheme will only work for arrays whose distribution induces a local iteration sequence that can be describe in terms of a base, bound and stride.

4.13 Notes to implementors

4.13.1 Temporary handling

The lifetime of temporaries generated by the evaluation of long expressions varies greatly from compiler to compiler and there has been no attempt to define any strict rules regarding such details. ANSI C++ rules are vague on this issue. Therefore, it is the responsibility of the array class to properly manage the lifetime of temporaries so that their data does not persist beyond the execution of the statement where they were formed.

4.13.2 Untyped Array Dimensions

The first iteration of the array class design does not include the array dimension within the type system, both for `Array<T>` and Class Region. If it is clear that it must be added then it will be done later. We suggest an array object independent of the dimension. The Maximum dimension is installation dependent (see section on Programming Model). For the purposes of STL, HPC++ array class iterators are provided which are dimension dependent.

4.13.3 BLAS indexing

Alternative indexing object may be derived from the HPC++ Range object to allow specification of (Base,Length,Stride) instead of the Range object's (Base,Bound,Stride).

4.13.4 Maximum Number of Dimensions.

The limitation of the number of dimensions in HPF is seven, the maximum array dimension in HPC++ can be specified at compile time (compile time for the array class), and is hence implementation-dependent. A temporary working limit for HPC++ array dimensions is eight.

4.13.5 Bases of Array Subsections.

Within FORTRAN 90 the base of a reference into an array is the default base for any FORTRAN array (namely "1"). This is because of consistency with FORTRAN 77 which carries no descriptor with its arrays (or subarrays) since the array is assumed to be linear and the pass by reference is implement internally using a pointer to the beginning of the new subsection. Since, the implementation of the HPC++ array class uses descriptors the base of a multidimensional subsection (even with non-unit strides) is preserved.

4.14 HPC++ Array Class Specification

This section is still under construction. It will appear similar to the "valarray" specification in format (see the www page for "valarray")

```
class Range
   {
     public:
     int Base;
     int Bound;
     int Length;
     int Stride;
     Index_Mode_Type Index_Mode;

     static bool Index_Bounds_Checking;
     static void setBoundsCheck ( On_Off_Type On_Off );

     Range ();
     Range ( int Base );
%    Range ( int Base, int Count; int Stride = 1 );
     Range ( int Input_Base, int Input_Bound, int Input_Stride = 1 );
     Range ( const Range & X );
     Range & operator= ( const Range & X );

     Range & operator+= ( int i );
     Range & operator-= ( int i );

     friend Range operator+ ( const Range & Lhs , int i );
     friend Range operator+ ( int i , const Range & Rhs );
     friend Range operator- ( const Range & Lhs , int i );
     friend Range operator- ( int i , const Range & Rhs );

     int getBase() const;
     int getBound() const;
     int getCount() const;
```

```
     int getStride() const;
     Index_Mode_Type getMode() const;
     int length () const;

     Range operator- () const;

     void display( const char* Label = "" ) const;

  // Changes the Range from either the base of the bound side
     void adjustBase  ( int x );
     void adjustBound ( int x );

  // These functions allow the indexing of the Range
  // objects using either scalar or Range objects.
     Range operator() ( int x ) const;
     Range operator() ( const Range & X ) const;
  };


class Point
  {
   public:
   Point();
   Point(const Array<int> ai);
   Point(const Point& rhs);
   Point& operator = (const Point& p);
  ~Point();

   int operator () (const int i);
   Point operator + (const Point p) const;
   Point operator - (const Point p) const;
   Point operator + (const int i) const;
   Point operator - (const int i) const;
   Point operator * (const int i) const;
   Point operator / (const int i) const;
   int operator == (const Point& p) const;
   int operator != (const Point& p) const;
   }

template <class T, class ptrdif_t>
  class Array_parallel_iterator{
      public:
        T& operator[](int);
        T operator[](int) const;
        T& operator *();
```

```
        T   operator *() const;
        Array_parallel_iterator<T, ptrdif_t> & operator ++();
        ... // and the other standard random access iterator operators
            // --, +n, -n,  -, <, >, <=, >=, +=n, -=n, ==, !=, =
        operator Point (); // cast to a point
};

template < class T, class ptrdif_t> class Array_parallel_iterator2{
     public:
        Array_parallel_iterator<T, ptrdif_t>  operator[](int);
        Array_parallel_iterator<T, ptrdif_t>  operator *();
        Array_parallel_iterator<T, ptrdif_t> & operator ++();
        ... // other random access iterator forms:
            // --, +n, -n,  -, <, >, <=, >=, +=n, -=n, ==, !=, =
};

template < class T, class ptrdif_t, int dim> class Array_parallel_iteratorK{
     public:
        Array_parallel_iteratorK<T, ptrdif_t, dim-1>  operator[](int);
        Array_parallel_iteratorK<T, ptrdif_t, dim-1>  operator *();
        Array_parallel_iteratorK<T, ptrdif_t, dim> & operator ++();
        ... // other random access iterator forms:
            // --, +n, -n,  -, <, >, <=, >=, +=n, -=n, ==, !=, =
};

template <class T > class Global_Ref; // a proxy object for remote
  //pointwise references.  This has the same behavior as T & but
  // is implementation defined.

class Region
  {
    public:
     typedef Array_parallel_iterator<Point, ptrdif_t> parallel_iterator;
     parallel_iterator begin(), end(), pbegin(), pend();

    ~Region () {};
     Region ();

     Region ( const Array<Range> & ar );
     Region (const Region& r);
     Region ( const Region & r );
     Region shift(const Region& r, const Point& p)
     Region operator * (const Region& rhs) const
     Region grow(const Region& r, const Point& p);
     Region grow(const Region& r, const int c);
```

```
     Region & operator= ( const Region & r );
     int lower(const int i) const;
     int upper(const int i) const;
     Point& lower();
     Point& upper();
     void setempty();
     int empty() const;
     int size() const;
     int operator == (const Region& rhs) const;
     int operator != (const Region& rhs) const;
     Region& operator *= (const Region& rhs);


template  class RegionIter{
    Point operator[](int) const;
    T   operator *() const;
    RegionIter<T> & operator ++();
};


class Pointwise_Region
   {
    ~Pointwise_Region ();
     Pointwise_Region ();
     Pointwise_Region (const Region& r);
     Pointwise_Region (const Range& r);
     Pointwise_Region ( const Pointwise_Region & D );
     Pointwise_Region shift(const Pointwise_Region& r, const Point& p);
     Pointwise_Region operator * (const Pointwise_Region& rhs) const;
     Pointwise_Region operator + (const Pointwise_Region& rhs) const;
     Pointwise_Region operator - (const Pointwise_Region& rhs) const;
     Region grow(const Pointwise_Region& r, const Point& p);
     Region grow(const Pointwise_Region& r, const int c);
     Pointwise_Region & operator= ( const Pointwise_Region & D );
     void setempty();
     int empty() const;
     int size() const;
     int operator == (const Pointwise_Region& rhs) const;
     int operator != (const Pointwise_Region& rhs) const;
     Pointwise_Region& operator *= (const Pointwise_Region& rhs);
     Pointwise_Region& operator += (const Pointwise_Region& rhs);
     Pointwise_Region& operator -= (const Pointwise_Region& rhs);
   };
```

```
template< class T > Array{
   public:
    typedef Array_parallel_iterator<T, size_t> parallel_iterator;
    typedef Arrap_parallel_iteratorK<T, size_t, 2> parallel_iterator2;
    typedef Arrap_parallel_iteratorK<T, size_t, 3> parallel_iterator3;
    ... // to max dimension size.
    parallel_iterator begin(),  end(),   pbegin(),   pend();
    parallel_iterator2 begin2(), end2(), pbegin2(), pend2();
    parallel_iterator3 begin3(), end3(), pbegin3(), pend3();
    ...

    Array(const Array<T> &, bool Deep_Copy = TRUE);
    Array(size_t);            // 1-D
    Array(size_t, Distribution); // 1-D distributed.
    Array(T*, size_t);        // 1-D reference of T* vector
    Array(Fortran_array, size_t);
    Array(HPF_array);         // alias to an HPF array
    Array(size_t, size_t);            // 2-D
    Array(size_t, size_t, Distribution); // 2-D distributed according to Distribution
    Array(T*, size_t, size_t);        // 2-D reference of T* vector
    Array(Fortran_array, size_t, size_t);
    ...  same for 3, 4 and 5 d.

    T & operator()(int);      // valid only for 1-D.
    T & operator()(int, int); // valid only for 2-D.
    ...  same for 3...8d.

 // Range object indexing operators
    Array<T>  operator()(const Range &); // valid only for 1-D.
    Array<T>  operator()(const Range &, const Range &); // valid only for 2-D.
    ...  same for 3...8d.

 // Indirect indexing using Array<int>
    Array<T>  operator()(Array<int> &); // valid only for 1-D.
    Array<T>  operator()(Array<int> &, Array<int> &); // valid only for 2-D.
    ...  same for 3...8d.
    Array<T>  operator()(Domain);
    Array<T>  operator()(Unstructured_Domain);

 // unsynchronized pointwise access
    global_ref<T> operator ()(Point);

 // standard operators:
    Array<T> operator & += (const Array<T> &);
    ... etc.
```

```
 // global inquiry functions
    int getBase(int Axis);
    int getBound(int Axis);
    int getStride(int Axis);
    int getLength(int Axis);
    Range dimension (int Axis);

 // local inquiry functions
    int getLocalBase(int Axis);
    int getLocalBound(int Axis);
    int getLocalStride(int Axis);
    int getLocalLength(int Axis);

 // shape functions
    redim (int i, int j = 1, int k = 1, int l = 1 );   // Number of parameters depends
                                                       // on the MAX_ARRAY_DIMENSION
    redim (const Range & I,     const Range & J = 1,
           const Range & K = 1, const Range & L = 1 ); // Number of parameters depends
                                                       // on the MAX_ARRAY_DIMENSION
    redim ( const Array<T> & X );

    reshape (int i, int j = 1, int k = 1, int l = 1 ); // Number of parameters depends
                                                       // on the MAX_ARRAY_DIMENSION
    reshape (const Range & I,     const Range & J = 1,
             const Range & K = 1, const Range & L = 1 ); // Number of parameters depends
                                                         // on the MAX_ARRAY_DIMENSION
    reshape ( const Array<T> & X );

 // parallel specific member functions
    ghostBoundaryUpdate();
    partition ( const Partition_Object & X ) const;

 // display if internal data
    display(char* = "") const;

 // display if internal data and descriptor info
    view(char* = "") const;
}

// arithmetic operators for +, -, *, \, %, etc.
template< class T > Array<T> & operator+(const Array<T> &, const Array<T> &);
template< class T > Array<T> & operator+(const T &, const Array<T> &);
template< class T > Array<T> & operator+(const Array<T> &, const T &);
...
```

```
// Transcendental operators sin, cos, tan, acos, asin,
// atan, sinh, cosh, tanh, asinh, acosh, atanh, etc.
template< class T > Array<T> & sin(const Array<T> &);
template< class T > Array<T> & cos(const Array<T> &);
...

// Relational operators for ==, !=, <, >, <=, >=, ||, &&, |, &, ^, etc.
template< class T > Array<T> & operator==(const Array<T> &, const Array<T> &);
template< class T > Array<T> & operator==(const T &, const Array<T> &);
template< class T > Array<T> & operator==(const Array<T> &, const T &);
...

// Min Max functions for different cases
template< class T > T max(const Array<T> &);
template< class T > Array<bool> & max(const Array<T> &, const Array<T> &);
template< class T > Array<bool> & max(const T &, const Array<T> &);
template< class T > Array<bool> & max(const Array<T> &, const T &);
...

class Processor_Map{
  Implementation not sufficiently defined
  ...
};

class Distribution{
  Implementation not sufficiently defined
  ...
};
```