Adlib Reference Manual

D.B. Carpenter

Northeast Parallel Architectures Centre,
Syracuse University,
111 College Place,
Syracuse, New York 13244-410

[In preparation]

Contents

1 The Range class

1.1 Default constructors and copying

```
Range() ;

Range(const Range& x) ;

Range& operator=(const Range& x) ;

Range :: clone(const Range&) ;

~Range() ;
```

The `Range` class is implemented as a handle class. A `Range` object contains a single word which is a pointer to a representation object of type `RangeRep`.

The default constructor for the `Range` class creates an object containing a null pointer.

The copy constructor copies the pointer in the parent object to the new object, and increments a reference count in the representation object.

The assignment operator behaves in a similar way, but also decrements the reference count of (and deletes, if necessary) any representation object referenced by the assignment variable prior to the assignment. The assignment then overwrites that reference.

The `clone` member allocates a new representation object and copies all fields of the parent object into the new object.

The destructor `~Range` decrements the reference count of (and deletes, if necessary) any representation object referenced by the `Range` object.

1.2 Proper constructors

```
Range(const int extent) ;

Range(const int extent, Dimension dim, const Format fmt) ;

Range(const int extent, Dimension dim, const Format fmt,
      const int blockSize, const int layout = ADLIB) ;

Range(const int extent, const int base, const int stride,
      Range parent) ;

Range(const int extent, const int base, Range parent) ;
```

These functions all create new representation objects and initialise handle objects referencing them. In all cases the first argument is the extent of the range.

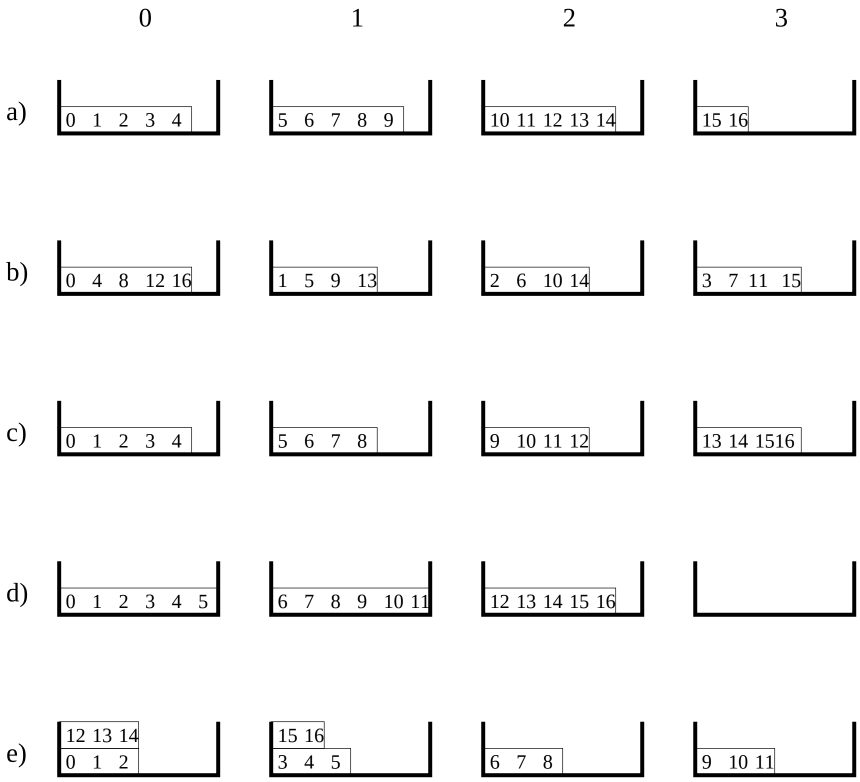

Figure 1: Example distribution formats.

The remaining arguments

The first constructor creates a "collapsed" range. The whole of the range is mapped into the local address space.

The second constructor create a range with block or simple cyclic or "stepped" block distribution format. The `fmt` argument takes one of the values `BLK`, `CYC` or `STP`. Suppose the extent of the range is `n` and the extent of the process dimension is `p`[1].

In *block* distribution contiguous blocks of the range of size `b = [(n + p - 1) / p]` are allocated to processes, starting from the first. When fewer than `b` elements of the range remain, the residue is allocated to the current process. Any further processes receive no elements. This distribution format is illustrated for `n = 17`, `p = 4`

[1] In the current implementation, `p` is the *physical* number of processors in the process dimension. This may or may not be equal to the extent specified in the `Procs` declaration that originally created the dimension. If the specified size of the process array is larger than the number of active processes, the actual value of `p` may be smaller than the specified extent.

in figure ??(a). The constructor call is

```
Range(17, d, BLK)
```

assuming `d` is a process dimension of extent 4.

In simple *cyclic* distribution, the elements of the range are allocated one per process, and the range wraps around when all processes have received an element. This distribution format is illustrated in figure ??(b) for the case

```
Range(17, d, CYC)
```

In *stepped* block distribution the first `n mod p` processes are assigned `b + 1` range elements and the remaining processes are assigned b elements, where `b = [n / p]` . This distribution format is illustrated in figure ??(c) for the case

```
Range(17, d, STP)
```

The third constructor creates a range with block or block cyclic (according to the value of `fmt`) distribution format, and a user-specified block size. The ranges constructed by

```
Range(17, d, BLK, 6)
```

and

```
Range(17, d, CYC, 3)
```

are illustrated in figures ??(d) and ??(e), assuming `d` is a process dimension of extent 4.

The optional `layout` parameter of the last constructor affects the mapping of array elements into memory. It will be discussed later.

The final two forms of the `Range` constructor are used for constructing derived ranges. Base and stride are specified for alignment to some parent range. In the second form the `stride` parameter is omitted and it defaults to 1.

Derived ranges can be aligned to parent ranges of any distribution format. This is a fundamental requirement for consistency of the library. If a new distribution format is to be implemented, one must also implement "affine" alignment to the new distribution format.

1.3 General inquiry functions

```
int size() ;

Dimension dim() ;

Range prm() ;
```

The member `size` returns the extent of the range. The member `dim` returns (a handle object for) the process dimension over which the range is distributed. This will be a null handle for collapsed ranges. The member `prm` returns (a copy of) the primitive range structure from which this range was derived.

1.4 The Subscript record

```
struct Subscript : public Coord {
  int blk ;
  int sub ;
} ;
```

A `Subscript` is implemented as a simple record containing two publically accessible data members in addition to the members of `Coord`. `blk` and `sub` are components of a local subscript. Their precise interpretation depends on the implementation of the parent range, and is not strictly prescribed by the general Adlib framework. The *expected* use is as follows.

- A `Subscript` always defines a location within a parent primitive range (in HPF terms, it is a subscript into a template).
- `sub` is an offset within a local block—for simple distribution formats it is "the" local subscript.
- `blk` is intended for use in "multi-block" distribution formats such as *block cyclic* distribution. It identifies a particular local block, of which there may be several. For simple distribution formats with only a single block per process, the `blk` field is unused.

A subscript record will normally be initialised by a member of the `Range` class. Although its fields are publically accessible, it is not normally necessary to read or write them directly.

1.5 Subscripting a Range object

```
int operator()(Subscript& s) ;

Subscript& operator()(const int i) ;
```

These are address translation operations. They convert between global index values and local subscripts. A global index is an `int` value from `0, ..., n - 1`, where n is the extent of the range. A local subscript is a member of the class `Subscript`.

The operator `operator()(Subscript& s)` takes a local subscript and returns the corresponding global subscript into this range. Its argument must be aligned to an element of the range for this operation to be legal. Typically, the argument was originally created from this range, or from some range derived from this range.

The `operator()(const int i)` takes a global subscript into this range, and returns a reference to a correponding local subscript. For collapsed distribution formats the `dim` field in the returned subscript is null. Because the result of this call is a reference, there is an issue about persistence of the referenced object. In fact there is a `Subscript` object inside the representation of every `Range` object. The subscripting call modifies this object and returns a reference to it. The reference remains valid until the next operation which might change the internal state of the range used to construct it—a subsequent subscripting operation or one of the iterator operations described in the following sections.

In normal usage the transitory nature of the returned reference does not seem to cause problems, and we avoid the overhead of allocating a new record whenever a local subscript is computed in an expression. If a persistent local subscript object is required, use the `Subscript` copy constructor explicitly to create one.

1.6 High level *where* iterators

```
void where() ;
int  test()  ;
void next()  ;
void erewh() ;

operator int() ;
operator Subscript&() ;
```

The four members `where`, `test`, `next` and `erewh` are intended for use in the idiom

```
for(x.where() ; x.test() ; x.next()) {
  ...
} x.erewh() ;
```

or in some strictly equivalent looping construct. Between the calls to `where()` and `erewh()` the range `x` is "activated as an iterator".

The semantics of the *where* construct has been explained in the introductory section of this document. In summary

- A *where* loop over `x` can be executed provided `x` is distributed over a dimension of the active process group, `p`.
- The body of the loop executes once for every element of the range held on the local processor.
- While a range is activated as a *where* iterator, the type conversions `operator int()` and `operator Subscript&()` allow the range to be coerced to the global or local subscript associated with the current iteration. As a rule the *only* appearance of a loop range inside the loop body will be in coercions.

- During an iteration of the loop, the active process group is reduced to `p / x`, assuming the coercion from range to `Subscript` described above.

Two common ambiguities associated with range-to-subscript cooercions need to be resolved.

- *If a range activated as an iterator is passed as an argument to a section constructor, it will, in effect, be cooerced to a local subscript.*

  The alternative would be to intepret the range "uncoerced". This would construct a section whose a process group includes the dimension over which `x` is distributed. That violates the constraint that the process group of a newly created section must be contained in the active process group, because this dimension is deleted from the active process group inside the loop.

- *If a range activated as an iterator is passed to a subscripting operation on another range, it will, in effect, be cooerced to a local subscript.*

  It will occasionally be necessary to subscript a range `y` inside the where construct. There is an ambiguity in the subscripting operations of the previous section. The operation

  ```
  y(x)
  ```

  could be interpretted as

  ```
  y((Subscript&) x)
  ```

  or as

  ```
  y((int) x)
  ```

  The first interpretation is legal if `y` is a parent range of `x`; it yields the global subscript in `y` associated with the current iteration of the loop. This is a likely requirement, because often we often need to perform a distributed loop over some subrange, and may want to access the global index values of the parent range. The second interpretation is legal, but less obviously useful. So the first interpretation is the default.

1.7 Lower level iterators

```
int blockLo()   ;
int blockHi()   ;
int blockStep() ;

int blockCount() ;
```

```
void locBlocksEnum() ;
int  locBlocksTest() ;
void locBlocksNext() ;

void allBlocksEnum() ;
int  allBlocksTest() ;
void allBlocksNext() ;

void blockNext() ;

int  at() ;
void ta() ;

int step(Range x) ;
```

These operations are used for optimizing `where` constructs, and in the implementation of various collective communication operaitons.

Discussion will be deferred to section ??.

1.8 Support for array construction

1.9 Support for section construction

```
Range operator()(const int extent, const int base,
                 const int stride) ;
Range operator()(const int extent, const int base) ;

Range(const Subscript& i) ;

int isScalar() ;
```

The first two operators simply provide syntactic sugar for construction of derived ranges. The expressions

```
x(e, b, s)

x(e, b)
```

are equivalent to

```
Range(e, b, s, x)

Range(e, b, x)
```

They provide a convenient syntax for "triplet subscripts" of array sections.

The third function is a new constructor, used to implement a type conversion which "wraps up" a `Subscript` as a `Range`. This is purely a device to allow scalar subscripts to be passed to the section constructor functions. These functions expect `Range` structures as arguments.

A `Range` object created in this way has very limited functionality. The only member functions that can meaningfully applied to it are `isScalar` and coercion back to a `Subscript`.

The member `isScalar` is used inside section constructor functions to determine whether a subscript argument (formally a `Range`) is to be treated as a proper range, or as a scalar subscript. The two cases in which it should coerced to a `Subscript` are:

1. The original argument really was a `Subscript`, wrapped up in the way described in the preceding paragraphs.
2. The argument was a range structure activated as an iterator.

In these two situations `isScalar` returns a non-zero value. Otherwise it returns zero.

2 The `Stride` class

This section describes the public interface of the `Stride` class.

The `Stride` is a support class for construction of distributed arrays. It provides a mapping from local subscripts within a range to offsets in memory. The most primitive distributed array—a distributed vector—can be parameterized by three entities

1. A `Range` object, describing the range of subscripts for the vector.
2. A base address in memory
3. A `Stride` object, describing the layout of array elements in memory.

A `Stride` object typically has a rather simple structure—for some kinds of array range it may contain a single integer—the physical memory displacement between consecutive array elements. In general, for multi-block array ranges or array ranges with strided alignments, it may involve a few more parameters.

It follows that the representation of a `Stride` object depends on the representation of the associated `Range` object for the array.

2.1 Default constructors and copying

```
Stride() ;

Stride(const Stride& x) ;
```

```
Stride& operator=(const Stride& x) ;

~Stride() ;
```

The `Stride` class is implemented as a handle class. A `Stride` object contains a single word which is a pointer to a representation object of type `StrideRep`. The `StrideRep` class is at the base of a hierarchy of representations. In general each subclass of `RangeRep` may be associated with its own variant of `StrideRep`.

The default constructor for the `Stride` class creates an object containing a null pointer.

The copy constructor copies the pointer in the parent object to the new object, and increments a reference count in the representation object.

The assignemnt operator behaves similarly, but will also decrement the reference count of (and delete, if necessary) any representation object referenced by the assignment variable prior to the assignment. The assignment then overwrites that reference.

The destructor `~Stride` decrements the reference count of (and deletes, if necessary) any representation object referenced by the `Range` object.

2.2 Proper constructors

```
Stride Range :: vector(int* dataLen) ;

Stride Range :: promote(const int oldStr) ;
```

(These "constructors" are implemented as member functions of the `Range` class; logically they are constructors for the `Stride` class.) The first constructor is the "normal" constructor for a `Stride` object. Its parameter is a "value-result" argument. Before the call the integer referenced by `dataLen` should be set to the size of a individual item or element in the vector to be parameterized. On exit, it is overwritten with the total amount of space which must be allocated locally to hold the vector, in the same units as the input value.

The `vector` constructor does not, and can not, sensibly, allocate any memory for the array. However it *does* define how much memory is to be allocated. An important property of the constructor is that it should direct every processor to allocate the same amount of space. Some processors may hold fewer elements from the index range, but all processors allocate space enough to hold the maximum possible of number of elements.

This has a couple of useful consequences, one being that all processes holding portions of a multi-dimensional distributed array contain identical stride structures. This facilitates access to remotely held array elements.

A typical usage of the constructor is in defining the layout for a multi-dimensional array of rank `rank` and type `T`. Suppose the ranges for the array are defined in a vector of `Range` structures called `rngs`. We need to initialize a corresponding vector of `Stride` structures called `strs`, then allocate space for the elements in a `char*` variable called `data`. This could be done as follows

```
int size = sizeof(T) ;
for(int i = 0 ; i < rank ; i++)
  strs [i] = (rngs [i]).vector(&size) ;
data = new char [size] ;
```

This leads to a layout in which the first dimension is "most-rapidly-varying" in memory (as in Fortran). The reverse convention can be obtained by reversing the order of the calls to the constructor.

The second constructor, `promote`, is more specialized. It is used in array promotions which give a distributed view of replicated, sequential arrays. The argument is the memory stride of the original sequential array.

[MOVE (PART OF) THE ABOVE INTO THE SECTION ABOUT 'Range' SUPPORT FOR CONSTRUCTION OF ARRAYS].

2.3 Memory offsets

```
int offset(Subscript& subs) ;

int step(Range x) ;
```

The principal operation on a `Stride` object, apart from its constructor, is the computation of the memory offset for a given `Subscript`. The argument of `offset` must be aligned with some element of the range used to construct the stride. Its result is the memory offset generated by the subscript, in the same units used for the `dataLen` argument of the constructor.

The `step` member is akin to the `step` member of the `Range` class. It is designed for the optimisation of *where* constructs. We will give a small example of manipulation of a distributed vector in a *where* construct.

Suppose we create a vector parametrised by range `x`, stride `vec` and base address `data` as follows

```
float* data ;
int size = 1 ;
Stride vec = x.vector(&size) ;
data = new float [size] ;
```

This vector might be initialised in the following *where* construct

```
for(x.where() ; x.test() ; x.next()) {
  data [vec.offset(x)] = x * x ;
} x.erewh() ;
```

We want to optimise this loop along the lines described in section ??. The first stage is to transform the loop to

```
x.at() ;
for(x.locBlocksEnum() ; x.locBlocksTest() ; x.locBlocksNext()) {
  const int l = x.blockLo() ;
  const int u = x.blockHi()  ;
  const int s = x.blockStep()  ;
  int i ;
  for(i = l ; i <= u ; i += s) {
    data [vec.offset(x)] = i * i ;
    x.blockNext() ;
  }
}
x.ta() ;
```

Now, using the `Stride :: step` member and the fact that the coercion of `x` to an integer will just yield `i`, the inner loop can be optimized to

```
  int data_vec = data + vec.offset(x) ;
  const int step_vec = vec.step(x) ;
  for(i = l ; i <= u ; i += s) {
    *data_vec = i * i ;
    data_vec += step_vec ;
  }
```

The call to `x.blockNext()` was eliminated; `x` is no longer used inside the loop and its state needn't be updated. The outer calls to `at` and `ta` can also be eliminated, because there are no function calls in the loop which could depend on the state of the active process group.

Note that in normal usage *the iterator argument of* `step` *is identical to, or is a range derived from* the range used to construct the subscripted `Stride`.

3 The `RangeRep` class

The `Range` class contains just one data member—a pointer to a representation object from the class `RangeRep`. Most of the member functions of `Range` are inline functions which dispatch the required operation to the representation object. The definition of the `Range` class looks something like this:

```
class Range {
public :
  ... constructors

  // Inquiries

  int size() {return rep->size() ;}

  Dimension* dim() {return rep->dim() ; }
```

```
  // Subscripting

  int operator()(Subscript& s) {
    return rep->subs(&s) ;
  }

  Subscript& operator()(const int i) {
    return *rep->subs(i) ;
  }

  operator int() {return rep->subs() ; }
  operator Subscript&() {return *rep ; }

  // `where' iterators

  void where() {rep->where() ; }
  int  test()  {return rep->test() ; }
  void next()  {rep->next() ; }
  void erewh() {rep->erewh() ; }

  // Lower level iterations...

  void locBlocksEnum() {rep->locBlocksEnum() ; }
  int  locBlocksTest() {return rep->locBlocksTest() ; }
  void locBlocksNext() {rep->locBlocksNext() ; }

  void allBlocksEnum() {rep->allBlocksEnum() ; }
  int  allBlocksTest() {return rep->allBlocksTest() ; }
  void allBlocksNext() {rep->allBlocksNext() ; }

  int blockBase() {return rep->blockLo() ; }
  int blockCount() {return rep->blockCount() ; }
  int blockLo() {return rep->blockLo() ; }
  int blockHi() {return rep->blockHi() ; }
  int blockStep() {return rep->blockStep() ; }

  void blockNext() {rep->blockNext() ; }

  int step(Range x) {return rep->step(x.rep) ; }

  int at() {return rep->at() ; }
  void ta() {rep->ta() ; }

  int isScalar() {return rep->isScalar() ; }

  ...
private :
```

```
  RangeRep* rep ;
} ;
```

The class `RangeRep` is an abstract class at the base of a hierarchy. Its concrete subclasses realise the various distribution and alignment formats supported for Adlib `Range` objects. The `Range` constructors, omitted above, will invoke constructors for the derived classes.

Several of the member functions of `RangeRep` are defined as (inline) functions on the base class. These include the inquiry functions and the *where* iterators. The remaining functions are virtual; they must be defined within the concrete subclasses.

The definition of the `RangeRep` base class looks something like this:

```
class RangeRep : public Subscript {
friend class Range ;
friend class DerivedStrideRep ;          // Accesses 'count'
public :
  RangeRep() : scalar(0) {}
  RangeRep(const Subscript& i) : Subscript(i), scalar(1) {}

  virtual ~RangeRep() {}

  int size() const {return extent ; }

  int subs() const {return i ; }
  int isScalar() const {return scalar ; }

  // 'where' iterators

  void where() ;
  int  test()  ;
  void next()  ;
  void erewh() ;

  int at() ;
  void ta() ;

  ... virtual functions
protected :
  int extent ;
private :
  int i ;
  int scalar ;

  int count ;
} ;
```

As promised, the representation of a range includes a subscript record—`RangeRep` is derived from `Subscript` (this explains why the coercion operator `operator Range :: Subscript&()` just returns a reference to the representation object).

The value of the private member `scalar` defines the result of `isScalar`. The private member `i` is updated by the *where* iterators, and defines the result of the coercion `operator Range :: int()` throug `RangeRep :: subs()`. The private member `count` is a reference count for the representation object: it is updated by the constructor and destructor members of the handle class.

The member functions `at` and `ta` are defined by

```
inline int RangeRep :: at() {
  scalar = 1 ;
  return Subscript :: at() ;
}

inline void RangeRep :: ta() {
  Subscript :: ta() ;
  scalar = 0 ;
}
```

and the *where* iterators by

```
inline void RangeRep :: where() {
  at() ;
  locBlocksEnum() ;
  i = blockLo() ;
}

inline int RangeRep :: test() {
  return locBlocksTest() ;
}

inline void RangeRep :: next() {
  blockNext() ;
  i += blockStep() ;
  if(i > blockHi()) {
    locBlocksNext() ;
    i = blockLo() ;
  }
}

inline void RangeRep :: erewh() {
  ta() ;
}
```

The only data member of `RangeRep` relevant or accessible to derived classes is the protected member `extent`, the extent of the range.

The remainder of this section describes the virtual members of `RangeRep`, which must be defined in the derived classes.

3.1 Subscripting a `RangeRep` object

```
virtual Subscript* subs(const int i) ;

virtual int subs(Subscript* s) ;
```

These are the implementation-level counterparts of the address-translation operators on the `Range` class. The convert a global index to a `Subscript` and vice versa.

3.2 Lower level iterators

```
virtual int blockLo()   ;
virtual int blockHi()   ;
virtual int blockStep() ;

virtual int blockCount() ;

virtual void locBlocksEnum() ;
virtual int  locBlocksTest() ;
virtual void locBlocksNext() ;

virtual void allBlocksEnum() ;
virtual int  allBlocksTest() ;
virtual void allBlocksNext() ;

virtual void blockNext() ;

virtual void blockStepPrim() ;

virtual int step(RangeRep* x) ;
```

These are the implementation-level counterparts of the "lower level iterator" functions on the `Range` class.

The function `blockStepPrim` has no counterpart in the higher level `Range` class. It is similar to `blockStep` but returns the step in the *primitive* range. In other words satisfies

```
blockStepPrim() == s * blockStep()
```

where `s` is the alignment stride of the range, or one for a primitive range. This function is needed to implement the `step` members of `RangeRep` and `StrideRep`. We could have provided an inquiry function to return the alignment stride of a range, and used the above formula directly. It happens that the strides of external ranges are only ever used in this combination, so we save a function call by putting this computation inside `blockStepPrim`.

As an example, the implementation of `step` on a concrete `RangeRep` class describing a primitive range is invariably just

```
int step(RangeRep* x) {
  return x->blockStepPrim() ;
}
```

whereas on a `RangeRep` which has an alignment stride `stride` it is

```
int step(RangeRep* x) {
  return x->blockStepPrim() / stride ;
}
```

Remember that the iterator argument of `step` is normally a subrange of the subscripted range. Consequently `x.stride` is a multiple of `stride`. Since `x->blockStepPrim()` is certainly a multiple of `x.stride`, the division above yields a whole-number result.

3.3 Miscellaneous constructor functions

```
virtual RangeRep* alignedRangeRep(const int extent,
                                  const int base,
                                  const int stride) ;

virtual StrideRep* vector(int* dataLen) ;
```

The function `alignedRangeRep` creates a derived range. The implementation of the `Range` constructor for a derived range is just

```
Range(const int extent, const int base, const int stride,
      Range parent) {
  rep = parent.rep->alignedRangeRep(extent, base, stride) ;
  rep->count = 1 ;
}
```

(As well as creating the `RangeRep` object, this constructor initializes its reference count).

The function `vector` is the representation-level counterpart of the `Range :: vector`. It creates a `StrideRep` object to describe the memory layout of a vector distributed over the range.

3.4 The `RangeRep` hierarchy

The `RangeRep` hierarchy as currently implemented is illustrated in figure ??.

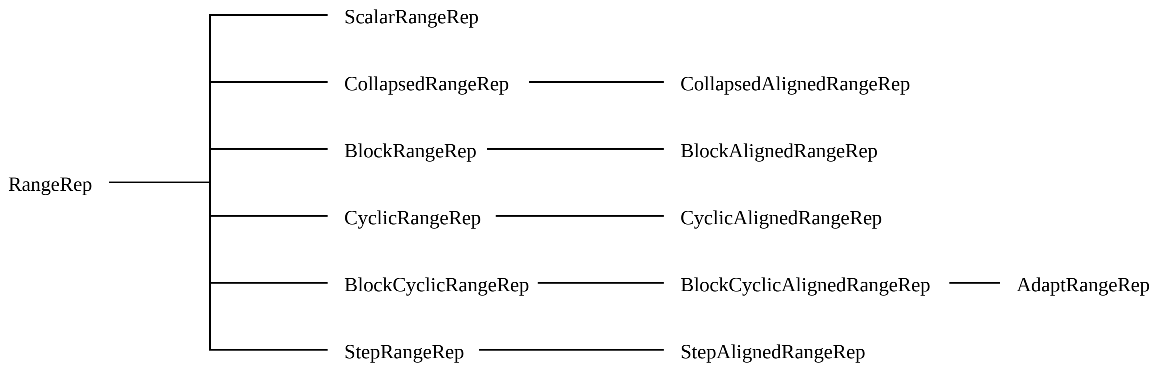

Figure 2: The `RangeRep` hierarchy.

RangeRep base class is a degenerate case. Its only direct use (other than as a base class) is to implement the Range wrapper for subscripts, returned by the promotion from Subscript to Range. This conversion is provided to support scalar subscripting in array sections.

The five immediate subclasses of RangeRep represent the five distribution formats described in section ??. Each of these classes in turn has a subclass representing a range aligned to the directly distributed ranges.

Finally, the subclass BlockCyclicAlignedRangeRep has a further subclass: ADAPTRangeRep. This subclass is used in the the SHPF translation system. The ADAPT translation tool lays out memory for aligned block-cyclic arrays in a slightly different way to the default Adlib scheme. The new subclass overrides the vector member, which defines this memory arrangement. This example illustrates how one can sometimes use inheritance to tailor Adlib to different applications. The StepRangeRep branch is another example—it was introduced to support a C* translator.

4 The StrideRep class

The Stride class contains just one data member—a pointer to a representation object from the class StrideRep. The member functions of Stride are inline functions which dispatch the required operation to the representation object. The definition of the Stride class looks like this:

```
class Stride {
public :
  Stride() : rep(0) {}

  ... copy constructor, assignment, destructor

  int offset(Subscript& subs) {
    return rep ? rep->offset(&subs) : 0 ;
  }
  int step(Range x) {
    return rep ? rep->step(x.rep) : 0 ;
  }
private :
  StrideRep* rep ;
} ;
```

The default constructor creates a null stride. The offset member returns 0 for a null stride, or dispatches the call to the representation object of type StrideRep for a non-null stride. Non-null strides are created by members of the Range class with which the stride is associated.

Null strides are used in certain array promotions, to simulate replication of values in particular dimensions.

4.1 Memory offsets

```
int offset(Subscript* subs) ;

int step(RangeRep* x) ;
```

These are representation-level counterparts of the `offset` and `step` members of the `Stride` class.

4.2 The `StrideRep` hierarchy

The `StrideRep` hierarchy as currently implemented is illustrated in figure ??.

There is a `StrideRep` subclass associated most (but not all) of the `RangeRep` subclasses (in some cases the `StrideRep` associated with a parent `RangeRep` may be adequate).

`PromoteStrideRep` is a special case. It is the representation object for a stride returned by the `Range :: promote` "constructor".

5 Optimization

All supported distribution formats have the property that the local segment of a range consists of zero or more blocks of elements. A block contains a set of regularly spaced global index values. If a "current block" is defined (see below), the members `blockLo`, `blockHi` and `blockStep` define the global index sub-range for this block, as follows. Suppose

```
l = x.blockLo()
u = x.blockHi()
s = x.blockStep()
```

then the global index values in the current block of `x` are

```
l, l + s, l + 2s, ..., u
```

The function `blockCount` will returns the value `(u - l)/s + 1`.

The result of `blockCount()` is always *greater than zero*. If a particular processor contains no elements from a range, this is flagged by the absence of blocks from the processor, not by the presence of empty blocks.

There are two sets of iterator functions for enumerating the blocks of a range: the "`locBlock`" iterators and the "`allBlock`" iterators. As the names suggest, the `locBlock` functions enumerate all locally held blocks of a range. The `allBlock` functions enumerate *all* blocks of a range, including blocks held non-locally. They are intended for use in the idioms

```
for(x.locBlocksEnum() ; x.locBlocksTest() ; x.locBlocksNext()) {
  ...
}
```

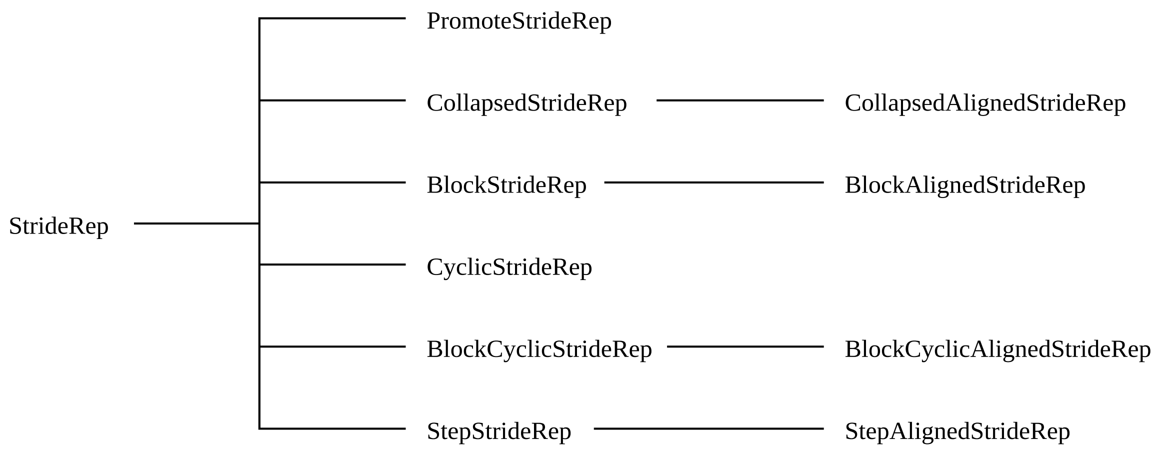

Figure 3: The `StrideRep` hierarchy.

```
for(x.allBlocksEnum() ; x.allBlocksTest() ; x.allBlocksNext()) {
  ...
}
```

or in some strictly equivalent looping constructs. During an iteration of the loops, one particular block is current. Its parameters can be determined by the inquiry functions introduced above.

The iterator functions set the `Subscript` record in the range representation to point to the first element of the block. A coercion of the range to a `Subscript&` returns this value unless it is explicitly changed inside the loop body. To support implementation of *where* iterators the function `blockNext` shifts internal `Subscript` state one place up in the block.

Note that the coercion of a `Range` to an `int` is not defined inside the low-level iterators. Definition of this coercion is peculiar to *where* iterators.

Two guarantees are provided on the order of enumeration of blocks:

1. For a particular range structure, `allBlocks` loops enumerate blocks in the same order on every processor.
2. The blocks for an individual processor are interleaved in a `allBlocks` enumeration in the same order in which they are appear in a `locBlocks` enumeration on that processor.

The second guarantee *does not* require that the blocks for a single processor are consecutive in a `allBlocks` enumeration. Also note that this guarantee is only contentful for multi-block distribution formats—ie formats where a single processor may contain more than one block

The `locBlocks` iterators enable optimisation of `where` constructs. The fragment

```
for(x.where() ; x.test() ; x.next()) {
  ...
} x.erewh() ;
```

can be replaced by

```
x.at() ;
for(x.locBlocksEnum() ; x.locBlocksTest() ; x.locBlocksNext()) {
  const int l = x.blockLo()   ;
  const int u = x.blockHi()   ;
  const int s = x.blockStep() ;
  for(int i = l ; i <= u ; i += s) {
    ...
    x.blockNext() ;
  }
}
x.ta() ;
```

This transformation only requires one change in the elided code. Because the low-level iterators do not define the coercion of the range to `int`, any such coercion should be replaced by the value `i` of the inner loop index.

The `at` and `ta` members effect the changes on the active process group necessary to implement a `where` construct properly, and the other operations have been described above.

This is still not a fully optimized translation approach. More substantial gains are possible by further alterations to the body of the loop. Coercions to `Subscript` can be eliminated according to context. As a simple example suppose the body of the loop contains the expression `y(x)`, where `y` is some parent range of `x`. The member function `step` returns the step of global index values for this range associated with the current block of `y`. So the inner loop can be replaced by

```
int i_y = y(x) ;
const int step_y = y.step(x) ;
for(i = b, j = 0 ; j < e ; i += s, j++) {
  ...
  i_y += step_y ;
  x.blockNext() ;
}
```

with appearances of `y(x)` in the elided code replaced by just `i_y`.

We note that in normal usage *the iterator argument of* `step` *is a range derived from* the subscripted range.

In later sections similar techniques will be introduced to eliminate *all* explicit uses of `Subscript` in the inner loop, enabling highly efficient translation of the `where` construct.

Other optimisations are possible:

- If it is known that a particular range does not have a multi-block distribution, the enumeration of the local blocks can be simplified to

  ```
  x.locBlocksEnum() ;
  if(x.locBlocksTest()) {
    ...
  }
  ```

  This saves at most one call to `locBlocksNext` and one call to `locBlocksTest`. The conditional is still required to allow for the case where the local processor holds no blocks. This optimization may be sensible in automatically generated code.

- If the body of the loop contains no call to functions which might access the active process group descriptor, the calls to the `at` and `ta` members can be omitted.