Chapter 5

Communication schedules

The type of the elements must be a POD type[1].

The constructor *does not* create copies of the frame objects referenced by `dst` and `src`—the pointer values are simply copied. It is the caller's responsibility to ensure that frame objects passed to schedule constructors persist as long as the constructed schedule may be used.

But, schedules never cache pointers to local data segments. The rationale for caching pointers to eg *Frame* objects but never to local data segments is ...

All communication schedules should be assumed to read the static variable `apg`, defining the active process group. The active process group in effect when a schedule is executed must be the same as the one in effect when it was created.

[1] *Plain Ol' Data* type, as defined in the draft ANSII C++ standard.

5.1 class Remap

A *remap schedule* is a communication schedule for copying the elements of one distributed array (the source array) to another (the destination array). The source and destination must have the same shape and same element-type. No relation between the mapping of the two arrays is assumed. A remap schedule is described by a collective object with local components of class `Remap`. The public interface of the `Remap` class is

```
class Frame ;

class Remap {
public :
  Remap(const Frame* dst, const Frame* src, const int len) ;

  void execute(void* dstDat, void* srcDat) ;
private :
  ...
} ;
```

5.1.1 Constructor

`Remap(const Frame* dst, const Frame* src, const int len)`

Create a schedule for copying data from a source array described by frame `*src` to a destination array described by `*dst`. `len` specifies the size of the array elements in bytes.

When the schedule is created the destination array must be accessible and the source array must be value-accessible.

5.1.2 Methods

`execute(void* dstDat, void* srcDat)`

Execute the schedule. The arguments are the base addresses for the local segments of the destination and source arrays. These should point to vectors of locally held elements.

A schedule may be executed once or many times. In principle the `execute` member may be passed different base addresses in different calls, so long as the associated arrays are described by frames structurally identical (shape, mapping *and* strides) to the ones passed to the constructor.

In-place updates are not allowed. The pairs `*dst`, `dstDat` and `*src`, `srcDat` must define non-overlapping arrays.

In general a schedule can only be executed when the active process group is the same as the one in effect at the time the schedule was created.

5.2 class `Shift`

A *shift schedule* is a communication schedule for shifting the elements of one distributed array (the source array) along a particular dimension, placing the result in another (the destination array). The source and destination must have the same shape and same element-type, and they must have a certain alignment relation. A shift schedule is described by a collective object with local components of class `Shift`. The public interface of the `Shift` class is

```
class Frame ;
enum Mode {CYCL, EDGE} ;

class Shift {
public :
  Shift(const Frame* dst, const Frame* src, const int len,
        const int shift, const int dim, const Mode mode) ;

  void execute(void* dstDat, void* srcDat) ;
private :
  ...
} ;
```

5.2.1 Constructor

```
Shift(const Frame* dst, const Frame* src, const int len,
      const int shift, const int dim, const Mode mode)
```

The source array *must be value-aligned with* the destination array (see section 4.5). `len` specifies the size of the array elements in bytes. The shift amount, positive or negative, is given by `shift`. The value of `dim` is in the range $0, \ldots, R-1$ where R is the rank of the source array. It selects the array dimension in which the shift occurs. The flag `mode` defines whether the shift is cyclic or "end-off".

When the schedule is created the destination array must be accessible and the source array must be value-accessible.

5.2.2 Methods

```
execute(void* dstDat, void* srcDat)
```

Execute the schedule. The arguments are the base addresses for the local segments of the destination and source arrays.

On exit, if the shift is cyclic, the value of

$$\texttt{dst}\ [x_0, \ldots, x_{\texttt{dim}}, \ldots, x_{R-1}]$$

is

$$\texttt{src}\ [x_0, \ldots, x_{\texttt{dim}} + \texttt{shift} \mod N, \ldots, x_{R-1}]$$

where N is the extent of dimension `dim`. If the shift is end-off, the exit value of the `dst` element is

$$\texttt{src}\ [x_0, \ldots, x_{\texttt{dim}} + \texttt{shift}, \ldots, x_{R-1}]$$

if $x_{\texttt{dim}} + \texttt{shift}$ is in the range $0, \ldots, N-1$, or unchanged from the entry value, if not.

When the schedule is executed the destination array must be accessible and the source array must be value-accessible.

In-place updates are not allowed. The pairs `*dst`, `dstDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.3 class Skew

A *skew schedule* is a communication schedule for performing a skewed shift—a shift where the shift amount is itself an array—in a particular dimension of one distributed array (the source array), placing the result in another (the destination array). The source and destination must have the same shape and same element-type, and they must have a certain alignment relation. A skew schedule is described by a collective object with local components of class `Skew`. The public interface of the `Skew` class is

```
class Frame ;
enum Mode {CYCL, EDGE} ;

class Skew {
public :
  Skew(const Frame* dst, const Frame* src, const int len,
       const Frame* shf, int* shfDat, const int dim,
       const Mode mode) ;

  void execute(void* dstDat, void* srcDat) ;
private :
  ...
} ;
```

5.3.1 Constructor

```
Skew(const Frame* dst, const Frame* src, const int len,
     const Frame* shf, int* shfDat,
     const int dim, const Mode mode)
```

The source array *must be value-aligned with* the destination array (see section 4.5). The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array. It selects the dimension in which the shifts occur. The shape of the shift array parametrized by `*shf` and `shfDat` is obtained from the shape of the destination array by deleting dimension `dim`. The shift array should be value-aligned with the destination array, with replicated alignment over dimension `dim`. The flag `mode` defines whether the shifts are cyclic or "end-off".

It is the caller's responsibility to ensure that associated objects persist as long as the constructed schedule may be used.

When the schedule is created the destination array must be accessible and the source array must be value-accessible.

5.3.2 Methods

`execute(void* dstDat, void* srcDat)`

Execute the schedule. The arguments are the base addresses for the local segments of the destination and source arrays.

The description of the exit value of `dst` is identical to the description after execution of a `Shift` schedule (see section 5.2), except that the constant `shift` is replaced by

`shf` $[x_0, \ldots, x_{\mathtt{dim}-1}, x_{\mathtt{dim}+1}, \ldots, x_{R-1}]$

In-place updates are not allowed. The pairs `*dst`, `dstDat` and `*src`, `srcDat` must define non-overlapping arrays.

When the schedule is executed the destination array must be accessible and the source array must be value-accessible.

5.4 class `MultiShift`

A *multishift schedule* is a communication schedule for shifting the elements of one distributed array (the source array) along some or all of its dimensions concurrently, placing the result in a destination array. The source and destination must have the same shape and same element-type, and they must have a certain alignment relation. A multishift schedule is described by a collective object with local components of class `MultiShift`. The public interface of the `MultiShift` class is

```
class Frame ;
enum Mode {CYCL, EDGE} ;

class MultiShift {
public :
  MultiShift(const Frame* dst, const Frame* src, const int len,
             int shift [], Mode mode []) ;

  void execute(void* dstDat, void* srcDat) ;
private :
  ...
} ;
```

5.4.1 Constructor

```
MultiShift(const Frame* dst, const Frame* src, const int len,
           int shift [], Mode mode [])
```

The source array *must be value-aligned with* the destination array (see section 4.5). `len` specifies the size of the array elements in bytes. Vectors `shift` and `mode` have extent R—the rank of the source array. The shift amount in dimension r, positive or negative, in dimension r is given by `shift` [r]. The flag `mode` [r] defines whether the shift is cyclic or "end-off". If the array is to be shifted only in a subset of its dimensions, this can achieved by setting elements of `shift` for the other dimensions to zero

When the schedule is created the destination array must be accessible and the source array must be value-accessible.

5.4.2 Methods

```
execute(void* dstDat, void* srcDat)
```

Execute the schedule. The arguments are the base addresses for the local segments of the destination and source arrays.

Equivalent in effect (not in implementation) to successive execution of R schedules of the form

```
Shift(dst, src, len, r, shift [r], mode [r])
```

for r in the range $0, \ldots, R-1$ (see section 5.2).

When the schedule is executed the destination array must be accessible and the source array must be value-accessible.

In-place updates are not allowed. The pairs `*dst`, `dstDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.5 class `WriteHalo`

A *write-halo schedule* is a communication schedule for filling *overlap regions* or *ghost cells* surrounding the local segment of a distributed array. A write-halo schedule is described by a collective object with local components of class `WriteHalo`. The public interface of the `WriteHalo` class is

```
class Frame ;

class WriteHalo {
public :
  WriteHalo(const Frame* src, const int len,
            const int wlo [], const int whi []) ;

  ~WriteHalo() ;

  void execute(void* base) ;
private :
  ...
}
```

5.5.1 Constructor and destructor

```
WriteHalo(const Frame* src, const int len,
          const int wlo [], const int whi [])
```

`len` specifies the size of the array elements in bytes. Vectors `wlo` and `wlo` have extent R—the rank of the array. These vectors define the halo of ghost cells updated by the schedule. The upper and lower widths in dimension r are given by `wlo` $[r]$ and `wlo` $[r]$. These values are non-negative, and can only be non-zero if array `src` actually has suitable ghost extensions in the dimension concerned.

Suppose either `wlo` $[r]$ and `wlo` $[r]$ is non-zero, and the range of `src` in dimension r is `x`, and the array stride is `u`. If the template of the parent range of `u`[2] was created by a call such as

$$\texttt{BlockRange(z, B, LAY\_EXTENDED, wlo}_{\text{act}}\texttt{, whi}_{\text{act}}\texttt{)}$$

and `x` has alignment stride s, the inequalities

$$s \times \texttt{whi}\ [r] \leq \texttt{whi}_{\text{act}}$$
$$s \times \texttt{wlo}\ [r] \leq \texttt{wlo}_{\text{act}}$$

[2] The careful wording is needed to allow for the fact that the array could be a section, or it could have non-trivial alignment, or both.

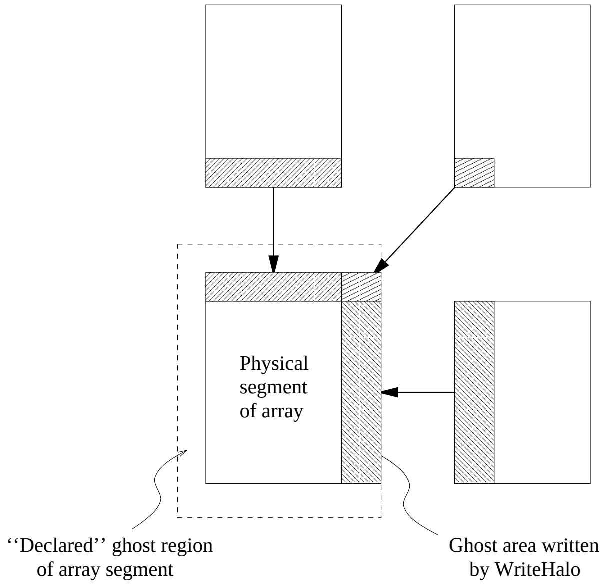

Figure 5.1: Illustration of the effect of executing a write-halo schedule.

must hold. The widths defined in the `WriteHalo` construtor are in terms of the *global subscript* of the associated range of the array, *not* in terms of *template* subscript.

When the schedule is created the array must be accessible. The vectors `wlo` and `whi` must have identical values in all members of the active process group.

[Will add a `mode` *vector to this constructor, analogous to* `MultiShift`*, to support use of ghost regions in problems with cyclic boundary conditions.]*

`~WriteHalo()`

Destructor.

5.5.2 Methods

`execute(void* base)`

Execute the schedule. The argument is the base addresses of the local segment of the array.

We distinguish between the locally held *physical segment* of an array and the surrounding *ghost region*, which is used to cache local copies of remote elements. The effect of this operation is to overwrite a portion of the ghost region—a halo of extent defined by the `wlo`, `whi` vectors of the constructor—with values from processes holding the corresponding elements in their physical segments. The operation is visualized in figure 5.1. Note that, so long as it fits in the ghost area *allocated* for the array, there is no restriction on the width of the halo region. In particular, the width of the halo region may be larger than the width of the neighbouring physical segment, in which case values will be fetched from next-nearest neighbours, and so on.

5.6 class Gather

A *gather schedule* is a communication schedule for collecting an arbitrary set of values from one distributed array (the source array) into the elements of another (the destination array). The selected set of elements is defined by a vector of *subscript arrays*, with an optional *mask array*. A gather schedule is described by a collective object with local components of class `Gather`. The public interface of the `Gather` class is

```
class Frame ;

class Gather {
public :
  Gather(const Frame* dst, const Frame* src, const int len,
         const Frame* subs [], int* subsData [],
         const Frame* msk, int* mskData) ;

  void execute(void* dstDat, void* srcDat) ;
private :
  ...
} ;
```

5.6.1 Constructor

```
Gather(const Frame* dst, const Frame* src, const int len,
       const Frame* subs [], int* subsData [],
       const Frame* msk, int* mskData)
```

The source and destination arrays must have the same element-type. `len` specifies the size of the array elements in bytes. The vectors `subs` and `subsData` have extent R—the rank of the *source* array. The subscript arrays defined by the `*subs` [r], `subsData` [r] pairs must be value-aligned to the destination array. If `msk` is non-null, the mask array defined by the `*msk`, `mskData` pair must be value-aligned to the destination array.

5.6.2 Methods

```
execute(void* dstDat, void* srcDat)
```

Execute the schedule. The arguments are the base addresses for the local segments of the destination and source arrays.

On exit, the value of

`dst` [$x_0, \ldots, x_{S-1}$]

is

src [**subs**$_0$ [$x_0, \ldots, x_{S-1}$], ..., **subs**$_{R-1}$ [$x_0, \ldots, x_{S-1}$]]

unless **msk** was non-null and

msk [$x_0, \ldots, x_{S-1}$]

was zero, in which case the exit value of the `dst` element is unchanged from the entry value. S is the rank of the destination array.

5.7 class Scatter

A *scatter schedule* is a communication schedule for distributing values from one distributed array (the source array) into elements of another another (the destination array) in an arbitrary way. The target set of elements is defined by a vector of *subscript arrays*, with an optional *mask array*. A scatter schedule is described by a collective object with local components of class `Scatter`. The public interface of the `Scatter` class is

```
class Frame ;

class Scatter {
public :
  Scatter(const Frame* src, const Frame* dst, const int len,
          const Frame* subs [], int* subsData [],
          const Frame* msk, int* mskData) ;

  void execute(void* srcDat, void* dstDat) ;
private :
  ...
} ;
```

5.7.1 Constructor

```
Scatter(const Frame* src, const Frame* dst, const int len,
        const Frame* subs [], int* subsData [],
        const Frame* msk, int* mskData)
```

The source and destination arrays must have the same element-type. `len` specifies the size of the array elements in bytes. The vectors `subs` and `subsData` have extent R—the rank of the *destination* array. The subscript arrays defined by the `*subs [`r`]`, `subsData [`r`]` pairs must be value-aligned to the source array. If `msk` is non-null, the mask array defined by the `*msk`, `mskData` pair must be value-aligned to the source array.

5.7.2 Methods

```
execute(void* srcDat, void* dstDat)
```

Execute the schedule. The arguments are the base addresses for the local segments of the source and destination arrays.

If `msk` was non-null, the effect is like executing the conditional assignment

```
if(msk [x_0,...,x_{S-1}])
  dst [subs_0 [x_0,...,x_{S-1}], ..., subs_{R-1} [x_0,...,x_{S-1}]] =
                          src [x_0,...,x_{S-1}]
```

for every tuple $[x_0, \ldots, x_{S-1}]$ of global subscripts of the source array. These assignments are executed in an undefined order. If `msk` is null, the assignment is unconditional.

5.8 class VecGather

A *vector-subscript gather schedule* is a communication schedule for collecting a set of values from one distributed array (the source array) into the elements of another array of the same rank (the destination array). The selected set of elements is defined by a vector of rank-1 arrays—the *subscript arrays*. This schedule can be regarded as an optimized special case of the general gather schedule of section 5.6. It is functionally equivalent to certain Fortran-90 array assignments. A vector-subscript gather schedule is described by a collective object with local components of class `VecGather`. The public interface of the `VecGather` class is

```
class Frame ;

class VecGather {
public :
  VecGather(const Frame* dst, const Frame* src, const int len,
            const Frame* subs [], int* subsData []) ;

  void execute(void* dstDat, void* srcDat) ;
private :
  ...
} ;
```

5.8.1 Constructor

```
VecGather(const Frame* dst, const Frame* src, const int len,
          const Frame* subs [], int* subsData [])
```

The source and destination arrays must have the same element-type. `len` specifies the size of the array elements in bytes. The vectors `subs` and `subsData` have extent R—the rank of the source and destination arrays. If `subs [`r`]` is non-null, the pair `*subs [`r`]`, `subsData [`r`]` defines a one-dimensional subscript array. The subscript array must be value-aligned to the destination array, *with replicated alignment in all dimensions except* r.

5.8.2 Methods

```
execute(void* dstDat, void* srcDat)
```

Execute the schedule. The arguments are the base addresses for the local segments of the destination and source arrays.

On exit, the value of

`dst [`$x_0, \ldots, x_{S-1}$`]`

is

$\texttt{src [subs}_0\ \texttt{[}x_0\texttt{], ..., subs}_{R-1}\ \texttt{[}x_{R-1}\texttt{]]}$

If `subs [`r`]` is null, the value of the expression $\texttt{subs}_r\ \texttt{[}x_r\texttt{]}$ in this formula is defined to be x_r.

5.9 class VecScatter

A *vector-subscript scatter schedule* is a communication schedule for distributing values from one distributed array (the source array) into elements of another (the destination array). The target set of elements are defined by a vector of rank-1 arrays—the *subscript arrays.* This schedule can be regarded as an optimized special case of the general scatter schedule of section 5.7. It is functionally similar to certain Fortran-90 array assignments—minus the constraints that enforce determinism in Fortran. A vector-subscript scatter schedule is described by a collective object with local components of class `VecScatter`. The public interface of the `VecScatter` class is

```
class Frame ;

class VecScatter {
public :
  VecScatter(const Frame* src, const Frame* dst, const int len,
             const Frame* subs [], int* subsData []) ;

  void execute(void* srcDat, void* dstDat) ;
private :
  ...
} ;
```

5.9.1 Constructor

```
VecScatter(const Frame* src, const Frame* dst, const int len,
           const Frame* subs [], int* subsData [])
```

The source and destination arrays must have the same element-type. `len` specifies the size of the array elements in bytes. The vectors `subs` and `subsData` have extent R—the rank of the source and destination arrays. If `subs` [r] is non-null, the pair `*subs` [r], `subsData` [r] defines a one-dimensional subscript array. The subscript array must be value-aligned to the source array, *with replicated alignment in all dimensions except* r.

5.9.2 Methods

```
execute(void* srcDat, void* dstDat)
```

Execute the schedule. The arguments are the base addresses for the local segments of the source and destination arrays.

The effect is like executing the assignment

`dst` [`subs`$_0$ [x_0], ..., `subs`$_{R-1}$ [x_{R-1}]] = `src` [$x_0, \ldots, x_{R-1}$]

for every tuple, $[x_0, \ldots, x_{R-1}]$, of global subscripts of the source array. These assignments are executed in an undefined order. If `subs [`r`]` is null, the value of the expression `subs`$_r$ `[`x_r`]` in the left-hand-side of the assignment is defined to be x_r.

5.10 class `Reshape`

[There are are existing schedules for this F90 intrinsic, but their interface is under revision.]

5.11 class Sum

A *sum schedule* is a communication schedule for adding together all elements of a distributed array (the source array). A sum schedule is described by a collective object with local components of class `Sum`. The public interface of the `Sum` class is

```
class Frame ;

template<class T>
class Sum {
public :
  Sum(const Frame* _src) ;

  void execute(T* _res, T* _srcBase) ;
private :
  ...
} ;
```

5.11.1 Constructor

`Sum(const Frame* _src)`

The source array is described by the frame `*_src`. It will have elements of type `T`, which should be a POD type. A conventional binary addition operator, `+`, must be defined for objects of type `T`, the assignment operator `+=` must operate in a compatible way, and the symbol 0 must be convertible to type `T`, yielding a zero of the addition operation. All the standard arithmetic types of C++ satisfy these requirements for `T`.

5.11.2 Methods

`execute(T* _res, T* _srcBase)`

Execute the schedule. The arguments are the address of the result and the base address of the local segment of the source array. Executing the schedule adds together all elements of the array in an unspecified order. The result value, written to `*_res`, is broadcast to all members of the active process group. This must be specified in the static variable `apg`.

5.12 class SumMsk

A *masked sum schedule* is a communication schedule for adding together all elements of a distributed array (the source array) under the control of a mask array. A masked sum schedule is described by a collective object with local components of class `SumMsk`. The public interface of the `SumMsk` class is

```
class Frame ;

template<class T>
class SumMsk {
public :
  SumMsk(const Frame* src, const Frame* msk) ;

  void execute(T* _res, T* _srcBase, int* _mskBase) ;
private :
  ...
} ;
```

5.12.1 Constructor

`SumMsk(const Frame* src, const Frame* msk)`

The source array is described by the frame `*src`. It will have elements of type `T`, which should be a POD type. A conventional binary addition operator, `+`, must be defined for objects of type `T`, the assignment operator `+=` must operate in the conventional way, and the symbol 0 must be convertible to type `T`, yielding a zero of the addition operation. All the standard arithmetic types of C++ satisfy these requirements for `T`. The mask array is described by the frame `*msk`. It should have the same shape as the source array, and *be value-aligned with* the source array (see section 4.5).

5.12.2 Methods

`execute(T* _res, T* _srcBase, int* _mskBase)`

Execute the schedule. The arguments are the address of the result and the base address of the local segment of the source array and the base address of the local segment of the mask array. Executing the schedule adds together all elements of the array for which the corresponding element of the mask array is non-zero, in an unspecified order. The result value, written to `*_res`, is broadcast to all members of the active process group. This must be specified in the static variable `apg`.

5.13 class Product

5.14 class ProductMsk

5.15 class Maxval

5.16 class MaxvalMsk

5.17 class Minval

5.18 class MinvalMsk

5.19 class All

5.20 class Any

5.21 class Count

5.22 class DotProduct

5.23 class BoolDotProduct

5.24 class `Maxloc`

A *maximum location schedule* is a communication schedule for finding the location of the largest element of a distributed array. A maximum location schedule is described by a collective object with local components of class `Maxloc`. The public interface of the `Maxloc` class is

```
class Frame ;

template<class T>
class Maxloc {
public :
  Maxloc(const Frame* _src) : Locate_<T, T>(_src) {}

  void execute(T* _val, int _pos [], T* _srcBase) ;
private :
  ...
} ;
```

5.24.1 Constructor

`Maxloc(const Frame* _src)`

The source array is described by the frame `*_src`. It will have elements of type `T`, which should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostneg` must be overloaded with the entry point

```
void mostneg(T* res) ;
```

which sets `*res` to the smallest allowed value of type `T`. The Adlib header files define `mostneg` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

5.24.2 Methods

`execute(T* _val, int _pos [], T* _srcBase)`

Execute the schedule. The value of largest element in the array is written to `*_val`. The global subscripts of the first occurrence of this value in the array are written to the vector `_pos`. The value of `_srcBase` should be the base address of the local segment of the source array.

The size of the `_pos` vector must be equal to the rank of the source array. The values written to this vector are the global subscripts within the array of the first occurence of the maximum value. If the maximum value occurs more than once in the array, “first occurence” is defined by ordering the set of global subscripts with first subscript *least* significant (Fortran-like array-element ordering).

The result values written to `*_val` and `_pos` are broadcast to all members of the active process group, which must be specified in the static variable `apg`.

5.25 class `MaxlocMsk`

A *masked maximum location schedule* is a communication schedule for finding the location of the largest element of a distributed array, under the control of a mask array. A masked maximum location schedule is described by a collective object with local components of class `MaxlocMsk`. The public interface of the `Maxloc` class is

```
class Frame ;

template<class T>
class MaxlocMsk {
public :
  Maxloc(const Frame* _src, const Frame* _msk) ;

  void execute(T* _val, int _pos [], T* _srcBase, int* _mskBase) ;
private :
  ...
} ;
```

5.25.1 Constructor

`MaxlocMsk(const Frame* _src, const Frame* _msk)`

The source array is described by the frame `*_src`. It will have elements of type `T`, which should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostneg` must be overloaded with the entry point

```
void mostneg(T* res) ;
```

which sets `*res` to the smallest allowed value of type `T`. The Adlib header files define `mostneg` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

The mask array is described by the frame `*msk`. It should have the same shape as the source array, and *be value-aligned with* the source array (see section 4.5).

5.25.2 Methods

`execute(T* _val, int _pos [], T* _srcBase, int* _mskBase)`

Execute the schedule. The value of largest element in the array for which the corresponding element of the mask array is non-zero is written to `*_val`. The global subscripts of the first occurrence of this value in the array are written to the vector _pos. The values of _srcBase and _mskBase should be the base addresses of the local segments of the source array and the mask array.

The size of the _pos vector must be equal to the rank of the source array. The values written to this vector are the global subscripts within the array of the first occurence of the masked maximum value. If the maximum value occurs more than once in the unmasked part of the array, "first occurence" is defined by ordering the set of global subscripts with first subscript *least* significant (Fortran-like array-element ordering).

The result values written to *_val and _pos are broadcast to all members of the active process group, which must be specified in the static variable apg.

5.26 class `Minloc`

5.27 class `MinlocMsk`

5.28 class SumDim

A *dimension sum schedule* is a communication schedule for summing the elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension sum schedule is described by a collective object with local components of class `SumDim`. The public interface of the `SumDim` class is

```
class Frame ;

template<class T>
class SumDim {
public :
  SumDim(const Frame* _res, const Frame* _src, const int _dim) ;

  void execute(T* _resBase, T* _srcBase) ;
private :
  ...
} ;
```

5.28.1 Constructor

`SumDim(const Frame* _res, const Frame* _src, const int _dim)`

The source array is described by the frame `*_src` and the result array is described by the frame `*_res`. They will both have elements of type `T`, which should be a POD type. A conventional binary addition operator, `+`, must be defined for objects of type `T`, the assignment operator `+=` must operate in a compatible way, and the symbol 0 must be convertible to type `T`, yielding a zero of the addition operation. All the standard arithmetic types of C++ satisfy these requirements for `T`.

The reduction occurs in dimension `_dim`. The value of `_dim` is in the range $0, \ldots, R-1$ where R is the rank of the source array. The result array must be aligned to the source array, *with replicated alignment in dimension* `_dim`.

5.28.2 Methods

`execute(T* _resBase, T* _srcBase)`

Execute the schedule. The arguments are the base addresses of the local segments of the result and source array. On exit, the value of

`res` $[x_0, \ldots, x_{\mathtt{dim-1}}, x_{\mathtt{dim+1}}, \ldots, x_{R-1}]$

is

$$\sum_{x_{\texttt{dim}}=0}^{N-1} \texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

where N is the extent of the source array in dimension `_dim`. The sum is performed in an unspecified order.

5.29 class SumDimMsk

A *masked dimension sum schedule* is a communication schedule for summing, under the control of mask, the elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A masked dimension sum schedule is described by a collective object with local components of class `SumDimMsk`. The public interface of the `SumDimMsk` class is

```
class Frame ;

template<class T>
class SumDimMsk {
public :
  SumDimMsk(const Frame* _res, const Frame* _src, const int _dim,
            const Frame* _msk) ;

  void execute(T* _resBase, T* _srcBase, int* _mskBase) ;
private :
  ...
} ;
```

5.29.1 Constructor

```
SumDimMsk(const Frame* _res, const Frame* _src, const int _dim,
          const Frame* _msk)
```

The source array is described by the frame `*_src` and the result array is described by the frame `*_res`. They will both have elements of type `T`, which should be a POD type. A conventional binary addition operator, `+`, must be defined for objects of type `T`, the assignment operator `+=` must operate in a compatible way, and the symbol 0 must be convertible to type `T`, yielding a zero of the addition operation. All the standard arithmetic types of C++ satisfy these requirements for `T`.

The reduction occurs in dimension `_dim`. The value of `_dim` is in the range $0, \ldots, R-1$ where R is the rank of the source array. The mask array is described by the frame `*_msk`.

The result array must be aligned to the source array, *with replicated alignment in dimension* `_dim`. The mask array must be value-aligned with the source array.

5.29.2 Methods

```
execute(T* _resBase, T* _srcBase, int* _mskBase)
```

Execute the schedule. The arguments are the base addresses of the local segments of the result, source and mask array. On exit, the value of

$$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is

$$\sum_{\substack{x_{\texttt{dim}} = 0 \\ \texttt{msk}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, \\ x_{\texttt{dim+1}}, \ldots, x_{R-1}] \neq 0}}^{N-1} \texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

where N is the extent of the source array in dimension `_dim`. The sum is performed in an unspecified order.

5.30 class `ProductDim`

5.31 class `ProductDimMsk`

5.32 class `MaxvalDim`

5.33 class `MaxvalDimMsk`

5.34 class `MinvalDim`

5.35 class `MinvalDimMsk`

5.36 class `AllDim`

5.37 class `AnyDim`

5.38 class `CountDim`

5.39 class MaxlocDim

A *dimension maximum location schedule* is a communication schedule for searching for the largest elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension maximum location schedule is described by a collective object with local components of class `MaxlocDim`. The public interface of the `MaxlocDim` class is

```
class Frame ;

template<class T>
class MaxlocDim {
public :
  MaxlocDim(const Frame* _val, const Frame* _pos,
            const Frame* _src, const int _dim) ;

  void execute(T* _valBase, int* _posBase, T* _srcBase) ;
private :
  ...
} ;
```

5.39.1 Constructor

```
MaxlocDim(const Frame* _val, const Frame* _pos,
          const Frame* _src, const int _dim)
```

The source array is described by the frame `*_src` and the array of maximum values is described by the frame `*_val`. They will have elements of type `T`, which should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostneg` must be overloaded with the entry point

```
void mostneg(T* res) ;
```

which sets `*res` to the smallest allowed value of type `T`. The Adlib header files define `mostneg` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

The array of maximum locations is described by the frame `*_pos`.

The search occurs in dimension `_dim`. The value of `_dim` is in the range $0, \ldots, R-1$ where R is the rank of the source array. The two result arrays (`*_val` and `*_pos`) must be aligned to the source array, *with replicated alignment in dimension* `_dim`.

5.39.2 Methods

`execute(T* _valBase, int* _posBase, T* _srcBase)`

Execute the schedule. The arguments are the base addresses of the local segments of the maximum value, maximum location and source array. On exit, the value of

`val` $[x_0, \ldots, x_{\mathtt{dim-1}}, x_{\mathtt{dim+1}}, \ldots, x_{R-1}]$

is the maximum value of

`src` $[x_0, \ldots, x_{\mathtt{dim-1}}, x_{\mathtt{dim}}, x_{\mathtt{dim+1}}, \ldots, x_{R-1}]$

across allowed values of $x_{\mathtt{dim}}$. The value of

`pos` $[x_0, \ldots, x_{\mathtt{dim-1}}, x_{\mathtt{dim+1}}, \ldots, x_{R-1}]$

is the smallest $x_{\mathtt{dim}}$ value at which this maximum occurs.

5.40 class MaxlocDimMsk

A *masked dimension maximum location schedule* is a communication schedule for searching, under the control of a mask, for the largest elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A masked dimension maximum location schedule is described by a collective object with local components of class `MaxlocDimMsk`. The public interface of the `MaxlocDimMsk` class is

```
class Frame ;

template<class T>
class MaxlocDimMsk {
public :
  MaxlocDimMsk(const Frame* _val, const Frame* _pos,
               const Frame* _src, const int _dim, const Frame* _msk) ;

  void execute(T* _valBase, int* _posBase, T* _srcBase, int* _mskBase) ;
private :
  ...
} ;
```

5.40.1 Constructor

```
MaxlocDimMsk(const Frame* _val, const Frame* _pos,
             const Frame* _src, const int _dim,
             const Frame* _msk)
```

The source array is described by the frame `*_src` and the array of maximum values is described by the frame `*_val`. They will have elements of type `T`, which should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostneg` must be overloaded with the entry point

```
void mostneg(T* res) ;
```

which sets `*res` to the smallest allowed value of type `T`. The Adlib header files define `mostneg` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

The array of maximum locations is described by the frame `*_pos`.

The search occurs in dimension `_dim`. The value of `_dim` is in the range $0, \ldots, R-1$ where R is the rank of the source array. The mask array is described by the frame `*_msk`.

The two result arrays (`*_val` and `*_pos`) must be aligned to the source array, *with replicated alignment in dimension* `_dim`. The mask array must be value-aligned with the source array.

5.40.2 Methods

`execute(T* _valBase, int* _posBase, T* _srcBase, int* _mskBase)`

Execute the schedule. The arguments are the base addresses of the local segments of the maximum value, maximum location, source and mask array. On exit, the value of

$\texttt{val}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

is the maximum value of

$\texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

for which

$$\texttt{msk}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}] \neq 0$$

over the allowed values of $x_{\texttt{dim}}$. The value of

$\texttt{pos}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

is the smallest $x_{\texttt{dim}}$ value at which this maximum occurs.

5.41 class `MinlocDim`

5.42 class `MinlocDimMsk`