NPAC PCRC Runtime Kernel (Adlib) Definition
[DRAFT]

Bryan Carpenter, Guansong Zhang and Yuhong Wen

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

July 12, 2002
# Contents

1 Introduction ................................................................. 9

2 Process Groups ............................................................. 11
  2.1 General Definitions ................................................... 11
  2.2 class Procs ............................................................ 13
     2.2.1 Constructors and destructors ................................. 13
     2.2.2 Methods ......................................................... 14
     2.2.3 Examples ....................................................... 15
  2.3 class Dimension ....................................................... 16
     2.3.1 Constructors and destructors ................................. 16
     2.3.2 Methods ......................................................... 16
     2.3.3 Operators ....................................................... 17
     2.3.4 Examples ....................................................... 17
     2.3.5 Implementation notes ......................................... 17
  2.4 class DimensionSet .................................................. 18
     2.4.1 Constructors and destructors ................................. 18
     2.4.2 Methods ......................................................... 18
     2.4.3 Operators ....................................................... 19
     2.4.4 Examples ....................................................... 20
     2.4.5 Implementation notes ......................................... 20
  2.5 class Group ........................................................... 21
     2.5.1 Constructors and destructors ................................. 21
     2.5.2 Methods ......................................................... 22
     2.5.3 Examples ....................................................... 24
     2.5.4 Implementation notes ......................................... 24
  2.6 Group apg .............................................................. 25

3 Ranges .............................................................................. 27
  3.1 General Definitions ................................................... 27
  3.2 struct Location ......................................................... 31
     3.2.1 Public data fields .............................................. 31
     3.2.2 Related operators .............................................. 31
  3.3 struct Block ............................................................ 33
     3.3.1 Public data fields .............................................. 33
3.4 class Range ........................................ 34
  3.4.1 Constructors and destructors .................. 34
  3.4.2 Methods ...................................... 34
  3.4.3 Operators .................................... 38
3.5 class CollapsedRange .......................... 39
  3.5.1 Constructors ................................ 39
3.6 class DimRange ............................. 40
  3.6.1 Constructors ................................ 40
3.7 class BlockRange .......................... 41
  3.7.1 Constructors ................................ 41
3.8 class CyclicRange ........................ 42
  3.8.1 Constructors ................................ 42
3.9 class BlockCyclicRange ....................... 43
  3.9.1 Constructors ................................ 43
3.10 class IrregRange .......................... 44
  3.10.1 Constructors .............................. 44

4 Arrays .................................................. 45
  4.1 struct Map ...................................... 46
    4.1.1 Constructors and destructors .............. 46
    4.1.2 Methods .................................. 46
  4.2 struct DAD ...................................... 48
    4.2.1 Public data fields ......................... 48
    4.2.2 Constructors ................................ 49
    4.2.3 Methods .................................. 49
  4.3 Examples ...................................... 51
  4.4 Array shape .................................... 52
  4.5 Alignment ..................................... 53
  4.6 Accessibility .................................. 53

5 Communication schedules ......................... 55
  5.1 General features of schedules .................... 55
    5.1.1 Value restrictions ......................... 56
    5.1.2 Type restrictions .......................... 56
    5.1.3 Shape restrictions ......................... 57
    5.1.4 Alignment restrictions ...................... 57
    5.1.5 Accessibility restrictions ................. 57
    5.1.6 Argument lifetime ........................ 57
    5.1.7 Effect ..................................... 58
    5.1.8 Replicated data ............................ 58
    5.1.9 Overlap restrictions ....................... 58
  5.2 class Remap ..................................... 59
    5.2.1 Constructor ................................ 59
    5.2.2 Method .................................... 59
  5.3 class Shift ..................................... 61
    5.3.1 Constructor ................................ 61
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.2</td>
<td>Method</td>
</tr>
<tr>
<td>5.4</td>
<td>class Skew</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Constructor</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Method</td>
</tr>
<tr>
<td>5.5</td>
<td>class MultiShift</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Constructor</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Method</td>
</tr>
<tr>
<td>5.6</td>
<td>class WriteHalo</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Constructors</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Method</td>
</tr>
<tr>
<td>5.7</td>
<td>class Gather</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Constructor</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Methods</td>
</tr>
<tr>
<td>5.8</td>
<td>class Scatter</td>
</tr>
<tr>
<td>5.8.1</td>
<td>Constructor</td>
</tr>
<tr>
<td>5.8.2</td>
<td>Methods</td>
</tr>
<tr>
<td>5.9</td>
<td>class VecGather</td>
</tr>
<tr>
<td>5.9.1</td>
<td>Constructor</td>
</tr>
<tr>
<td>5.9.2</td>
<td>Methods</td>
</tr>
<tr>
<td>5.10</td>
<td>class VecGatherMask</td>
</tr>
<tr>
<td>5.10.1</td>
<td>Constructor</td>
</tr>
<tr>
<td>5.10.2</td>
<td>Methods</td>
</tr>
<tr>
<td>5.11</td>
<td>class VecScatter</td>
</tr>
<tr>
<td>5.11.1</td>
<td>Constructor</td>
</tr>
<tr>
<td>5.11.2</td>
<td>Method</td>
</tr>
<tr>
<td>5.12</td>
<td>class VecScatterMask</td>
</tr>
<tr>
<td>5.12.1</td>
<td>Constructor</td>
</tr>
<tr>
<td>5.12.2</td>
<td>Method</td>
</tr>
<tr>
<td>5.13</td>
<td>class ScatterComb</td>
</tr>
<tr>
<td>5.14</td>
<td>class Reshape</td>
</tr>
<tr>
<td>5.15</td>
<td>class Sum</td>
</tr>
<tr>
<td>5.15.1</td>
<td>Constructor</td>
</tr>
<tr>
<td>5.15.2</td>
<td>Method</td>
</tr>
<tr>
<td>5.16</td>
<td>class SumMask</td>
</tr>
<tr>
<td>5.16.1</td>
<td>Constructor</td>
</tr>
<tr>
<td>5.16.2</td>
<td>Method</td>
</tr>
<tr>
<td>5.17</td>
<td>class Product</td>
</tr>
<tr>
<td>5.17.1</td>
<td>Constructor</td>
</tr>
<tr>
<td>5.17.2</td>
<td>Method</td>
</tr>
<tr>
<td>5.18</td>
<td>class ProductMask</td>
</tr>
<tr>
<td>5.18.1</td>
<td>Constructor</td>
</tr>
<tr>
<td>5.18.2</td>
<td>Method</td>
</tr>
<tr>
<td>5.19</td>
<td>class Maxval</td>
</tr>
<tr>
<td>5.19.1</td>
<td>Constructor</td>
</tr>
<tr>
<td>5.19.2</td>
<td>Method</td>
</tr>
<tr>
<td>5.20</td>
<td>class MaxvalMask</td>
</tr>
</tbody>
</table>

62 63 66 67 71 73 74 75 76 77 78 80 81 82 83 85 86 88 89 90 90 92 92 93 94 94 94 96
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.20.1</td>
<td>Constructor</td>
<td>96</td>
</tr>
<tr>
<td>5.20.2</td>
<td>Method</td>
<td>97</td>
</tr>
<tr>
<td>5.21</td>
<td>class Minval</td>
<td>98</td>
</tr>
<tr>
<td>5.21.1</td>
<td>Constructor</td>
<td>98</td>
</tr>
<tr>
<td>5.21.2</td>
<td>Method</td>
<td>98</td>
</tr>
<tr>
<td>5.22</td>
<td>class MinvalMask</td>
<td>100</td>
</tr>
<tr>
<td>5.22.1</td>
<td>Constructor</td>
<td>100</td>
</tr>
<tr>
<td>5.22.2</td>
<td>Method</td>
<td>101</td>
</tr>
<tr>
<td>5.23</td>
<td>class All</td>
<td>102</td>
</tr>
<tr>
<td>5.23.1</td>
<td>Constructor</td>
<td>102</td>
</tr>
<tr>
<td>5.23.2</td>
<td>Method</td>
<td>102</td>
</tr>
<tr>
<td>5.24</td>
<td>class Any</td>
<td>103</td>
</tr>
<tr>
<td>5.24.1</td>
<td>Constructor</td>
<td>103</td>
</tr>
<tr>
<td>5.24.2</td>
<td>Method</td>
<td>103</td>
</tr>
<tr>
<td>5.25</td>
<td>class Count</td>
<td>104</td>
</tr>
<tr>
<td>5.25.1</td>
<td>Constructor</td>
<td>104</td>
</tr>
<tr>
<td>5.25.2</td>
<td>Method</td>
<td>104</td>
</tr>
<tr>
<td>5.26</td>
<td>class DotProduct</td>
<td>105</td>
</tr>
<tr>
<td>5.26.1</td>
<td>Constructor</td>
<td>105</td>
</tr>
<tr>
<td>5.26.2</td>
<td>Method</td>
<td>106</td>
</tr>
<tr>
<td>5.27</td>
<td>class BoolDotProduct</td>
<td>107</td>
</tr>
<tr>
<td>5.27.1</td>
<td>Constructor</td>
<td>107</td>
</tr>
<tr>
<td>5.27.2</td>
<td>Method</td>
<td>107</td>
</tr>
<tr>
<td>5.28</td>
<td>class Maxloc</td>
<td>109</td>
</tr>
<tr>
<td>5.28.1</td>
<td>Constructor</td>
<td>109</td>
</tr>
<tr>
<td>5.28.2</td>
<td>Method</td>
<td>109</td>
</tr>
<tr>
<td>5.29</td>
<td>class MaxlocMask</td>
<td>111</td>
</tr>
<tr>
<td>5.29.1</td>
<td>Constructor</td>
<td>111</td>
</tr>
<tr>
<td>5.29.2</td>
<td>Method</td>
<td>112</td>
</tr>
<tr>
<td>5.30</td>
<td>class Minloc</td>
<td>113</td>
</tr>
<tr>
<td>5.30.1</td>
<td>Constructor</td>
<td>113</td>
</tr>
<tr>
<td>5.30.2</td>
<td>Method</td>
<td>113</td>
</tr>
<tr>
<td>5.31</td>
<td>class MinlocMask</td>
<td>115</td>
</tr>
<tr>
<td>5.31.1</td>
<td>Constructor</td>
<td>115</td>
</tr>
<tr>
<td>5.31.2</td>
<td>Method</td>
<td>116</td>
</tr>
<tr>
<td>5.32</td>
<td>class SumDim</td>
<td>117</td>
</tr>
<tr>
<td>5.32.1</td>
<td>Constructor</td>
<td>117</td>
</tr>
<tr>
<td>5.32.2</td>
<td>Method</td>
<td>118</td>
</tr>
<tr>
<td>5.33</td>
<td>class SumDimMask</td>
<td>119</td>
</tr>
<tr>
<td>5.33.1</td>
<td>Constructor</td>
<td>119</td>
</tr>
<tr>
<td>5.33.2</td>
<td>Method</td>
<td>120</td>
</tr>
<tr>
<td>5.34</td>
<td>class ProductDim</td>
<td>121</td>
</tr>
<tr>
<td>5.34.1</td>
<td>Constructor</td>
<td>121</td>
</tr>
<tr>
<td>5.34.2</td>
<td>Method</td>
<td>122</td>
</tr>
<tr>
<td>5.35</td>
<td>class ProductDimMask</td>
<td>123</td>
</tr>
<tr>
<td>5.35.1</td>
<td>Constructor</td>
<td>123</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>5.35.2</td>
<td>Method</td>
<td>124</td>
</tr>
<tr>
<td>5.36</td>
<td>class MaxValDim</td>
<td>125</td>
</tr>
<tr>
<td>5.36.1</td>
<td>Constructor</td>
<td>125</td>
</tr>
<tr>
<td>5.36.2</td>
<td>Method</td>
<td>126</td>
</tr>
<tr>
<td>5.37</td>
<td>class MaxValDimMask</td>
<td>127</td>
</tr>
<tr>
<td>5.37.1</td>
<td>Constructor</td>
<td>127</td>
</tr>
<tr>
<td>5.37.2</td>
<td>Method</td>
<td>128</td>
</tr>
<tr>
<td>5.38</td>
<td>class MinValDim</td>
<td>129</td>
</tr>
<tr>
<td>5.38.1</td>
<td>Constructor</td>
<td>129</td>
</tr>
<tr>
<td>5.38.2</td>
<td>Method</td>
<td>130</td>
</tr>
<tr>
<td>5.39</td>
<td>class MinValDimMask</td>
<td>131</td>
</tr>
<tr>
<td>5.39.1</td>
<td>Constructor</td>
<td>131</td>
</tr>
<tr>
<td>5.39.2</td>
<td>Method</td>
<td>132</td>
</tr>
<tr>
<td>5.40</td>
<td>class AllDim</td>
<td>133</td>
</tr>
<tr>
<td>5.40.1</td>
<td>Constructor</td>
<td>133</td>
</tr>
<tr>
<td>5.40.2</td>
<td>Method</td>
<td>134</td>
</tr>
<tr>
<td>5.41</td>
<td>class AnyDim</td>
<td>135</td>
</tr>
<tr>
<td>5.41.1</td>
<td>Constructor</td>
<td>135</td>
</tr>
<tr>
<td>5.41.2</td>
<td>Method</td>
<td>136</td>
</tr>
<tr>
<td>5.42</td>
<td>class CountDim</td>
<td>137</td>
</tr>
<tr>
<td>5.42.1</td>
<td>Constructor</td>
<td>137</td>
</tr>
<tr>
<td>5.42.2</td>
<td>Method</td>
<td>138</td>
</tr>
<tr>
<td>5.43</td>
<td>class MaxLocDim</td>
<td>139</td>
</tr>
<tr>
<td>5.43.1</td>
<td>Constructor</td>
<td>139</td>
</tr>
<tr>
<td>5.43.2</td>
<td>Method</td>
<td>140</td>
</tr>
<tr>
<td>5.44</td>
<td>class MaxLocDimMask</td>
<td>141</td>
</tr>
<tr>
<td>5.44.1</td>
<td>Constructor</td>
<td>141</td>
</tr>
<tr>
<td>5.44.2</td>
<td>Method</td>
<td>142</td>
</tr>
<tr>
<td>5.45</td>
<td>class MinLocDim</td>
<td>144</td>
</tr>
<tr>
<td>5.45.1</td>
<td>Constructor</td>
<td>144</td>
</tr>
<tr>
<td>5.45.2</td>
<td>Method</td>
<td>145</td>
</tr>
<tr>
<td>5.46</td>
<td>class MinLocDimMask</td>
<td>146</td>
</tr>
<tr>
<td>5.46.1</td>
<td>Constructor</td>
<td>146</td>
</tr>
<tr>
<td>5.46.2</td>
<td>Method</td>
<td>147</td>
</tr>
</tbody>
</table>

6 The ad++ interface 149

7 Distributed loops 151

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>class Index</td>
<td>153</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Constructor</td>
<td>153</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Methods</td>
<td>153</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Operators</td>
<td>154</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Translation of overall construct by macro expansion</td>
<td>154</td>
</tr>
<tr>
<td>7.2</td>
<td>class LocBlocksIndex</td>
<td>156</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Constructor</td>
<td>156</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Methods</td>
<td>156</td>
</tr>
</tbody>
</table>
7.2.3 Translation of overall construct using LocBlocksIndex ... 157
7.3 Translation using the kernel range .............................. 164
7.4 Access to ghost regions ........................................... 172
7.5 class AllBlocksIndex .............................................. 174
    7.5.1 Constructor ................................................. 174
    7.5.2 Methods ...................................................... 174
    7.5.3 Use of AllBlockIndex in the communication library .. 174

8 Implementation of the communication schedules .................. 177
Chapter 1

Introduction

The library defined in this document is designed as a common runtime kernel for manipulating HPF-style regular distributed arrays. It is not expected that the kernel interface described here will be used directly by an application programmer. It may not even appear directly in code generated by a compiler. Additional interface code, built on top of the kernel, will probably be needed.

One example of a user-level interface to the kernel is the \textit{ad++} interface. This is a set of C++ header files which define type-secure template classes—\textit{container classes}—representing distributed arrays, various macros implementing \textit{distributed control} constructs, and template functions for performing transformations on distributed arrays. The latter functions fulfil a role something like the array transformational intrinsics of Fortran 90. In particular, they abstract communication.

Compiler-level interfaces, on the other hand, provide any additional run-time structure assumed by the distributed array model of the source language, and make the kernel callable from the target language of the compiler. In principle this target language could be machine code, but historically the library has been used by source-to-source translators targeting Fortran 77 or Fortran 90. So existing “compiler-level” interfaces to the kernel are actually Fortran bindings. A Java interface is under development.

The kernel itself provides:

- A distributed array descriptor (DAD), implemented as a series of C++ classes.
- A large set of \textit{communication schedules} implementing Fortran-90 like array transformational functions and other basic communication operations.
- As part of the DAD—support for “distributed control”. In practise what this means is support for “distributed loops”: support for translation of parallel loops whose index range is distributed over some processor set.

This document is not supposed to be read in isolation. The underlying model
of data distribution should be understood, for example by learning the user-level interface to the library, before attempting direct use of the kernel.

The next three chapters define the components of the Adlib array descriptor—groups, ranges, then the array descriptor record itself. Chapter 5 describes the communication library. This will followed by a chapter on the ad++ interface, then a chapter about using the Adlib descriptor technology in translation of distributed loops. A final chapter will discuss the implementation of the communication schedules.
Chapter 2

Process Groups

The Adlib idea of a process group is analogous to the MPI idea of group. The Adlib version is more specialized. Adlib groups are frequently created and modified in the runtime. For efficiency it is important to provide a specialized lightweight representation—one cannot afford to import the general group technology of MPI. For example, Adlib needs the structure of multi-dimensional grids without the overheads involved in building an MPI Cartesian communicator.

The most direct analogue of the MPI Cartesian communicator is the Adlib process array, represented by a Procs object. Even in Adlib, constructing a process array is a relatively expensive procedure. It involves initializing several sub-structures. The general group object of Adlib—class Group—has a much more streamlined implementation. Group is essentially a handle class. Its objects contain a pointer to a parent Procs object plus a few extra words defining a subset of that Cartesian grid. Group objects are created, copied, changed, and discarded freely within the library code. They can be handled with similarly free rein in user code.

2.1 General Definitions

Initial process group: The set processes on which an Adlib program is initiated.

Active process group: The set of processes notionally sharing the local thread of control. This may be the initial process group or it may be an arbitrary subset of the initial process group. When an Adlib program starts the active process group is the initial process group.

The logical active process group may be changed by operations that cause some processors to skip a block of code (leaving the remaining processors active for that block), or cause the current active process group to be partitioned into a set of smaller groups, with each group acting independently for some time.
Every process executing an Adlib program must maintain a static data structure, \texttt{apg}, describing the current active process group. This variable is read by collective functions in the library to determine which processes are cooperating in the operation. If the logical active process group changes, the variable \texttt{apg} must be updated (see section 2.6) before invoking a collective operation. During the library initialization procedure, \texttt{apg} is set to a value describing the initial process group.

Collective object: A collection of similar objects, one in every process from a particular group, which can also be viewed as a single logical entity.

Collective operation: An operation executed cooperatively by all members of the active process group. Typically this will take the form of a call to the same function with the same arguments. In this context two arguments are “the same” if they are simple expressions with the same value, or are references to local components of the same collective object.

A collective operation may or may not involve synchronization between members of the active process group. In general the programmer should work on the (worst case) assumption that synchronization is implied. If any member of the active process group fails to engage in a collective operation that others are executing, or executes collective operations in a different order to other members of the group, the program is liable to fail.

Examples of collective operations are constructors for collective objects and collective communication operations.

Process array: Some subset of the initial process group, organized in a Cartesian grid.

Coordinate: Dimensions of a process array are labelled by \textit{coordinates}.

Process group: In general, either a process array, or some slice of a process array defined by uniquely fixing the coordinate in one or more of its dimensions.

Process id: An identifier relative to a particular process group\textsuperscript{1}. If the size of the group is \( P \), the id is in the range \( 0 \ldots P - 1 \). Relative to a one-dimensional process array, the value of the process id coincides with the value of the unique coordinate. The id relative to the initial process group is also called the \textit{absolute process id}.

\textsuperscript{1}Called a \textit{rank} in MPI. Following Fortran, we will reserve “rank” for the dimensionality of arrays or process arrays.
2.2 class Procs

A process array is a set of processes organized in a multi-dimensional grid. A process array is described by a collective object with local components of class Procs. The public interface of the Procs class is

```cpp
class Dimension;
class DimensionSet;

class Procs {
public:
    Procs(const int _rank, int* _n, int* subIds = 0);
    ~Procs() ;

    int rank() const ;
    Dimension dim(const int r) const ;

    DimensionSet dims() const ;

    int member() const ;

    int size() const ;

    int id() const ;

    int id_abs(const int id) const ;

    int lead_abs() const ;

private:
    Procs() ;
    Procs(const Procs& p) ;

...;
};

2.2.1 Constructors and destructors

Procs(const int _rank, int* _shape, int* ids = 0)

The normal constructor. Initializes a Procs object describing a process grid of rank _rank and shape _shape. The rank is a non-negative integer. If it is zero the new group is a scalar “grid” containing a single process. The shape is a vector of _rank positive integer extents. The size, \( P \), of the new grid is the product of the elements of _shape. This must be less than or equal to the size of the active process group.

If the optional argument ids is specified, this should be a vector of \( P \) distinct values, each in the range \( 0, \ldots, A - 1 \). A is the size of the group defined by the current value of apg. It defines a mapping of the new process group into the
active process group. If the ids argument is omitted, the library selects an arbitrary mapping. In either case if \( P \) is less than \( A \), some members of the active process group are not identified with any member of the new grid. At most one member of the new grid may be mapped to each process of the active group.

This constructor is a collective operation, and calls to it should obey the normal rules for collective operations.

**Procs()**

The default constructor is *private*. Creating an uninitialized *Procs* object is not allowed in normal code.

**Procs(const Procs& p)**

The copy constructor is *private*. Copying a *Procs* object is not allowed in normal code.

"Procs()"

Destructor.

### 2.2.2 Methods

**rank()**

An inquiry function returning the rank (dimensionality) of the grid.

**dim(const int r)**

An inquiry function returning an object that describes the \( r \)th dimension of the grid. The argument \( r \) is in the range \( 0, \ldots, R - 1 \) where \( R \) is the rank of the grid.

**dims()**

An inquiry function returning an object that describes the complete set of dimensions of the grid.

**member()**

An inquiry function which returns 1 if a process from the grid is mapped to the local process, and 0 if not. In other words, a boolean function that returns true iff this process is a member of the grid.

**size()**

An inquiry function returning the total number of processes, \( P \), in the grid.
id()

An inquiry function that returns the id of the local process relative to this grid. Its value is defined if and only if the local process is a member of the grid, in which case the result is in the range $0, \ldots, P - 1$.

id_abs(const int id)

A conversion function from id relative to this grid to id relative to the initial process group—i.e., to absolute process id.

lead_abs()

The absolute id of the lead process of the grid—equivalent to id_abs(0).

2.2.3 Examples

```c
int shp [1] = {4} ;
Procs p(1, shp) ;
```

Creates a one-dimensional process array representing a linear set of 4 processes. The active process group must contain at least 4 processes when the declaration of p is encountered.

```c
int shp [4] = {2, 2, 2, 2} ;
Procs h(4, shp) ;
```

Creates a process array representing a 4-dimensional binary hypercube. The active process group must contain at least 16 processes when the declaration of h is encountered.

```c
int shp [2] = {2, 3}, ids [6] = {0, 1, 3, 6, 10, 15} ;
Procs q(2, shp, ids) ;
```

Creates a process array representing a 2 by 3 grid with a user-defined mapping to members of the active process group. The active process group must contain at least 16 processes when the declaration of q is encountered (because ids includes values up to 15).

```c
Procs s(0, 0) ;
```

Creates scalar process grid containing a single process.
2.3 class Dimension

A *process dimension* is a dimension of a process array. A process dimension is described by a collective object with local components of class Dimension. The public interface of the Dimension class is

class Dimension {
 public:
  Dimension();
  int size() const;
  int crd() const;
  int str_prc() const;
  operator int() const;

 private:
  ...
};

2.3.1 Constructors and destructors

Dimension()

The default constructor. Initializes a Dimension object describing a *collapsed dimension*. This is a degenerate, “internal” dimension, not associated with a process array.

2.3.2 Methods

size()

An inquiry function returning the extent, \(N\), of the dimension. For a dimension of a process array, this is the extent specified in the shape vector for that array. For a collapsed dimension, \(N\) defined to be 1.

crd()

An inquiry function returning an integer coordinate for the local process relative to this process dimension. For a process array dimension the result value is only defined if the local process is a member of the parent array. It is then in the range \(0, \ldots, N - 1\), where \(N\) is the extent of the dimension. For a collapsed dimension, the result is 0.

str_prc()

An inquiry function defining the stride in process id (relative to the parent array) associated with this dimension. Undefined for a collapsed dimension.
2.3.3 Operators

operator int()

Returns a non-zero value if the dimension belongs to a process array. Returns 0 if the dimension is collapsed. [Replace this with a dims member?]

2.3.4 Examples

    int shp [2] = {2, 3} ;
    Procs p(2, shp) ;

    Dimension d = p.dim(0), e = p.dim(1) ;
    Dimension f ;

Create a process array p representing a 2 by 3 grid. The active process group must contain at least 6 processes when the declaration of p is encountered. Set d to represent the first dimension of p and e to represent the second. Set f to a collapsed dimension.

The inquiries d.size(), e.size() and f.size() return 2, 3 and 1 respectively. The inquiries d.crd() and e.crd() will return values in the ranges {0,1} and {0,1,2} respectively if the inquiry p.member() returns value 1. They return unspecified values if p.member() returns 0. The inquiry f.crd() will return 0.

2.3.5 Implementation notes

Dimension is currently implemented as a simple, non-reference-counting handle class. If it is a process array dimension it contains a pointer to an object of type DimensionRep associated with the parent Procs structure. If it is collapsed dimension it contains a null pointer.
2.4 class DimensionSet

A *dimension set* is some subset of the dimensions of a particular process array. A dimension set is described by a collective object [?] with local components of class DimensionSet. The public interface of the DimensionSet class is

```cpp
class DimensionSet {
public:
    DimensionSet() ;
    int member(Dimension d) const ;
    int isEqual(DimensionSet ds) const ;
    int contains(DimensionSet ds) const ;
    operator int() const ;
    DimensionSet operator+(DimensionSet ds) const ;
    DimensionSet operator-(DimensionSet ds) const ;
    DimensionSet& operator+=(DimensionSet ds) ;
    DimensionSet& operator-=(DimensionSet ds) ;
    DimensionSet operator+(Dimension d) const ;
    DimensionSet operator-(Dimension d) const ;
    DimensionSet& operator+=(Dimension d) ;
    DimensionSet& operator-=(Dimension d) ;

private :
    ... ;
} ;
```

2.4.1 Constructors and destructors

DimensionSet()

The default constructor. Initializes a DimensionSet object to an empty set.

2.4.2 Methods

member(Dimension d)

Inquiry function. Returns a non-zero result if d is a member of this set, and 0 if it is not.

isEqual(DimensionSet ds)
Returns a non-zero result if this set is equal to ds and 0 if it is not.

contains(DimensionSet ds)

Returns a non-zero result if this set contains ds and 0 if it does not.

2.4.3 Operators

operator int()

Returns a non-zero value if the set is non-empty, or 0 if it is empty.

operator+(DimensionSet ds)

Create a new set containing the union of this set with ds. If both input sets are non-empty, they must contain dimensions from the same process array.

operator-(DimensionSet ds)

Create a new set by subtracting ds from this set. If both input sets are non-empty, they must contain dimensions from the same process array.

operator+=(DimensionSet ds)

Replace this set by its union with ds. If both input sets are non-empty, they must contain dimensions from the same process array.

operator-=(DimensionSet ds)

Subtract ds from this set. If both input sets are non-empty, they must contain dimensions from the same process array.

operator+(Dimension d)

Create a new set by adding a dimension to this set. If the set is non-empty before this operation d must be collapsed or a dimension from the same process array as the current elements. Adding a collapsed dimension, or one already in the set, does not change the set.

operator-(Dimension d)

Create a new set by removing d from this set. If the set is non-empty before this operation d must be collapsed or a dimension from the same process array as the current elements. Removing a collapsed dimension, or one not in the set, does not change the set.

operator+=(Dimension d)

19
Add a dimension to this set. If the set is non-empty before this operation \( d \) must be collapsed or a dimension from the same process array as the current elements. Adding a collapsed dimension, or one already in the set, does not change the set.

```cpp
operator-=(Dimension d)
```

Remove a dimension from this set. If the set is non-empty before this operation \( d \) must be collapsed or a dimension from the same process array as the current elements. Removing a collapsed dimension, or one not in the set, does not change the set.

### 2.4.4 Examples

```cpp
int shp [3] = {2, 2, 2} ;
Procs p(3, shp) ;

DimensionSet ds ;
ds += p.dim(0) ;
ds += p.dim(2) ;
```

Create a process array \( p \) representing a 3 dimensional binary hypercube. Create \( ds \) as an empty dimension set, then add the first and third dimension of \( p \) to it.

### 2.4.5 Implementation notes

`DimensionSet` is currently realized as a simple, one-word record containing a `long` integer used as a bitmask. All the members can be implemented as short inline functions using bitwise operations—this class is very lightweight.

This implementation imposes the Adlib kernel’s only limitation on dimensionality—the number of process dimensions in a process array should normally be less than the number of bits in a `long` integer. Unless, for unknown reasons, an application involves a process array with many dimensions of unit extent, this constraint is not really stronger than the tacit assumption that all processes can be distinctly labelled by a single integer. On a 32-bit computer, for example, this implementation theoretically supports binary hypercubes of up to about 4 billion processors...
2.5 class Group

A process group is described by an object of class Group. The public interface of the Group class is

class Procs;
class DimensionSet;

class Group {
  public:
    Group();
    Group(const Procs& p);
    const Procs* prc() const;
    DimensionSet dims() const;
    int isEqual(const Group& p) const;
    int contains(const Group& p) const;
    int member() const;
    int size() const;
    int id() const;
    int id_prp(const int id) const;
    int id_abs(const int id) const;
    int lead_prp() const;
    int lead_abs() const;
    void restrict(Dimension d, const int coord);
    void restrict(Dimension d);
    void restrict(DimensionSet ds);

  private:
    ...
};

2.5.1 Constructors and destructors

Group()

The default constructor. Creates an uninitialized Group object.

Group(const Procs& p)

Conversion constructor. Initializes a Group object representing all processes in process array p. By definition p is the parent process array of the constructed
group. The new group object can subsequently be reduced to a subset of p by using the restrict members below.

2.5.2 Methods

prc()

An inquiry function returning a pointer to the local Procs object associated with the parent process array.

dims()

An inquiry function returning the effective dimension set of this group. This is a subset of the dimensions of the parent process array,

isEqual(const Group& p)

A comparison operation that returns a non-zero value if p is equal to this group. The definition of equality used here is that groups are equal if they share the same parent process array\(^2\), their dimension sets are equal, and their lead processes are the same.

contains(const Group& p)

A comparison operation that returns a non-zero value if this group contains p. The definition of “contains” used here is as follows: q contains p if

1. p is equal to q according to the definition above, or

2. p represents a whole process array, and q was the active process group when that process array was created, or

3. p can be obtained by applying a restrict method to q (see below), or

4. q contains some group r and r contains p—the relation is transitive.

This definition of contains is stricter than the intuitive definition based on comparison of the sets of processes in the groups. It can often be computed more directly.

member()

An inquiry function that returns a non-zero value if the local process is a member of the group, and 0 if not.

size()

An inquiry function returning the total number of processes, \( P \), in the group.

\(^2\)Their parent arrays must be described by the same object, not merely be structurally similar.
id()

An inquiry function that returns the id of the local process relative to this group. Its value is defined if and only if the local process is a member of the group, in which case it is in the range 0,...,P - 1.

id_prc(const int id)

A conversion function from id relative to this group to id relative to the parent process array.

id_abs(const int id)

A conversion function from id relative to this group to id relative to the initial process group—ie, to absolute process id. Equivalent to prc() -> id_abs(id_prc(id))

lead_prc()

The id relative to the parent process array of the lead process of this group—equivalent to id_prc(0).

lead_abs()

The absolute id of the lead process of the group—equivalent to id_abs(0).

restrict(Dimension d, const int coord)

Reduce the process group by restricting the coordinate in d to the value coord. The operation removes d from dims(). It is ill-defined if d is not a member of dims() beforehand. It is a null operation if d is collapsed.

restrict(Dimension d)

Reduce the process group by restricting the coordinate in d to the value associate with the local process, equivalent to restrict(d, d.crd()). The operation is ill-defined if the local process is not a member of the parent process array, or if d is not a member of dims() beforehand. It is a null operation if d is collapsed.

restrict(DimensionSet ds)

Reduce the process group by restricting the coordinate in all dimensions from ds to the values associate with the local process. Equivalent to applying restrict(d) for every member, d, of ds. The operation is ill-defined if the local process is not a member of the parent process array, or if ds is not a subset of dims() beforehand.
2.5.3 Examples

```c
int shp[2] = {2, 2};
Procs p(2, shp);

Group q = p;
q.restrict(p.dim(0), 0);

Group r = q;
r.restrict(p.dim(1), 1);
```

These groups are illustrated in figure 2.1. The dimension set of q contains just \( p.dim(1) \), and dimension set of \( r \) is empty.

2.5.4 Implementation notes

Group can be realized as a simple three-word record containing a pointer to the parent Procs object, a DimensionSet (one word) representing the set of effective dimensions, and an integer specifying the id relative to the parent process array of the lead process of the group. These are the values returned by \( \text{prc, dims} \) and \( \text{lead.prc} \) respectively.
2.6 Group apg

The static variable apg is accessed by most of the collective operations in the library. Its value should reflect the currently effective active progress group. Its declaration is

```
extern Group apg;
```

When an Adlib program starts the value of apg represents the initial process group. If collective operations are to function properly there is an onus on the programmer to maintain the value of apg consistently. In this section three common idiomatic ways of modifying the active process group are discussed.

The first idiom involves a conditional construct which restricts the set of processes performing a block of code to some group, p. Membership of the local process in this group is determined by calling p.member(), as follows:

```
if(p.member()) {
    ...
}
```

If it is necessary to invoke collective library operations inside the construct, apg must be reset appropriately. Assuming the construct appears at the point where p is a subset of the active process group, the logical active process group inside the construct will be the whole of p. Appropriate manipulations of apg are

```
Group apgSave = apg;
if(p.member()) {
    apg = p;
    ...
}
apg = apgSave;
```

The only subtlety is the need to save the old value of apg so that it can be restored on completion of the construct. With these manipulations of the apg variable we can safely invoke collective operations before, during and after the conditional construct. This idiom will sometimes be called an on construct. In the ad++ interface to Adlib macros ON/NO are defined to allow the whole of the fragment above to be written as

```
ON(p) {
    ...
} NO(p);
```

A second common idiom for modifying apg occurs when the logical active process group is partitioned by breaking up one of its process dimensions. Suppose the current active process group is a multi-dimensional array of processes. One of its process dimensions is d. Suppose we need to perform operations collectively across dimensions orthogonal to d, but independently for each value of the d coordinate. In this case the value of apg should be temporarily changed as follows

```
```
Group apgSave = apg;
apg.restrict(d);
...
apg = apgSave;

In the ad++ interface, for example, this kind of manipulation of apg occurs in the \texttt{AT/TA} construct and \texttt{OVERALL/ALLOVER} distributed loop.

The preceding idiom allows for regular partitioning of the active process group across one of its dimensions. A different technique can be used to achieve arbitrary partitioning. Suppose each of $A_1, \ldots, A_n$ is a set of processes, and together they partition the current active process group, $A$. For each $i$ in $1, \ldots, n$, $a_i$ is a vector of ids (relative to $A$) for the processes in $A_i$, and $r_i$ and $s_i$ define a rank and a shape for a grid of size $|A_i|$. Now, if the local process is a member of $A_i$, the following constructor call

\texttt{Procs p(r_i, s_i, a_i)}

creates $n$ logically distinct collective objects representing non-overlapping process arrays. An \texttt{on} construct using \texttt{p} will now partition the active process group as required. For example, suppose the current active process group has 5 processes. We can partition these temporarily into groups of 2 and 3 as follows

\begin{verbatim}
int shp [1], *ids ;
if(apg.id() < 2) {
    shp [0] = 2 ;
    ids = {0, 1} ;
} else {
    shp [0] = 3 ;
    ids = {2, 3, 4} ;
}
Procs p(1, shp, ids) ;

Group apgSave = apg ;
if(p.member()) {
    apg = p ;
    ...
}  
apg = apgSave ;
\end{verbatim}

This use of the \texttt{Procs} constructor departs slightly from usual rules for collective operations, because when the constructor is called the value of \texttt{apg} is the original active process group, but its arguments take different values in sectors of that group which will later be in separate partitions.

Notice that the idioms we have introduced in this section can be nested freely. As usual, the only subtlety is in ensuring that old values of the active process group are restored properly when the constructs complete. This can be achieved either by using a suitable series of automatic variables (like \texttt{apgSave}), or by using an explicit stack of group objects.
Chapter 3

Ranges

An Adlib range is a map from an integer interval \(0, \ldots, N - 1\) into a process dimension. Each value, or *global subscript*, in the interval is mapped to a particular process coordinate. While the library does *not* support completely general, irregular mappings from global subscript to process coordinate, it does provide many of the most popular options (*block*, *cyclic*, *block-cyclic*, etc). In particular the allowed mappings support all distribution formats and intra-dimensional alignments of HPF 1.0, and the irregular block distribution format of HPF 2.0.

Adlib currently supports three kinds of range, distinguished by an integer called the *level* of the range. The simplest distribution formats of HPF (including simple block and simple cyclic distribution) are described by *level 1 ranges*. Block-cyclic distribution format is described by a *level 2 range*. For completeness, Adlib adds *level 0 ranges*. These are ranges that describe an unadorned process dimension, or a subrange of a process dimension.

3.1 General Definitions

**Level:** A non-negative integer, characteristic of any range.

**Primitive range:** The range of a process dimension. The global subscripts are simply the coordinates of the process dimension.

A primitive range has level 0.

**Subrange:** A range defined as a (strided) subinterval of some parent range.

A subrange retains an *alignment* relation to the parent range—the mapping of a subrange element to the process dimension is the same as that of the corresponding element of the parent range. Subranges can be used to implement the general alignment options of HPF.

The level of a subrange is the same as the level of the parent range.

**Template range:** A range that is not a subrange of any range except itself.

Template ranges thus include primitive ranges, block-distributed ranges
with (in HPF terms) identity alignment, and cyclic-distributed ranges with identity alignment.

Template ranges can be used to represent the dimensions of an HPF template.

Parent template range: Any range is a subrange of its unique parent template.

Kernel range: Any range of level higher than 0 is distributed over some kernel range.

The level of a range’s kernel one less than range’s own level. For example, the kernel of a range with HPF-like block distribution or simple cyclic distribution is a primitive range—a process-dimension range. The kernel of a range with HPF-like block-cyclic distribution, on the other hand, is a range with simple cyclic distribution.

Global subscript: The primary subscript associated with a range. A value from the interval \(0, \ldots, \text{size}() - 1\). Sometimes just called the subscript.

Template subscript: For any element of a range, the associated global subscript in the parent template range.

Kernel subscript: For any element of a range, the associated global subscript in the kernel range (for level 1 ranges this is the process coordinate).

Block: The section of a range associated with a particular fixed value of the kernel subscript (for level 1 ranges, the section of the range mapped to a particular process coordinate). Any element of the range lies in a particular block. Note that elements of a block need not have contiguous global subscripts—in cyclic distribution format, adjacent elements of the block have template subscript differing by the extent of the kernel.

Shell subscript: For any element of a range, a subscript identifying the position of the element within its own block. In general there is a one-to-one mapping between between legal kernel-subscript/shell-subscript pairs and legal template subscripts.
Figure 3.1: Illustration of various subscripts for a block-distributed subrange.
Figure 3.2: Illustration of various subscripts for a cyclic-distributed subrange with stride 2.
3.2 struct Location

A record that bundles together a process dimension, a coordinate, and any shell subscripts implied by a particular global subscript value in a particular range.

The interface of the Location struct is

```cpp
struct Location {
    Dimension dim ;
    int crd ;
    int sub ;
    int blk ;
};

Group operator/(const Group& p, const Location& i) ;
Group operator/(const Proc& p, const Location& i) ;
```

The Location record does not include the global subscript itself. This means that a Location record constructed from a subrange is identical to a Location record representing the aligned element of the parent range, although global subscript relative to the two ranges are generally different.

3.2.1 Public data fields

dim

The dim field should define the process dimension of the parent range.

crd

The crd field should define the coordinate in dim of processes holding the location.

sub

The shell subscript associated with the location. Undefined if the range has level 0.

blk

Used only for level 2 ranges. This field specifies the block that contains the location. Technically it is a shell subscript in the immediate kernel of the parent range (that kernel being a level 1 range).

3.2.2 Related operators

operator/(const Group& p, const Location& i)

Create a new group formed by restricting this group by i. The inline definition is
Group operator/(const Group& p, const Location& i) {
    Group result(p);
    result.restrict(i.dim, i.crd);
    return result;
}

operator/(const Procs& p, const Location& i)

Create a new group formed by restricting this process array by i. The inline
definition is

Group operator/(const Procs& p, const Location& i) {
    Group result(p);
    result.restrict(i.dim, i.crd);
    return result;
}
3.3 struct Block

A record parametrizing an individual block of a range. It includes size (count) of the block, together with global and shell subscript bases and steps.

The interface of the Block struct is

```c
struct Block {
  int count ;

  int glb_bas, glb_stp ;
  int sub_bas, sub_stp ;
} ;
```

3.3.1 Public data fields

count

The number of elements in the block.

glb_bas

The global subscript of the first element in the block.

glb_stp

The increment of the global subscript between adjacent elements in the block. For most distribution formats this will be 1, but we have to allow for cyclically distributed ranges, where blocks are generally strided sections of the parent range.

sub_bas

The shell subscript of the first element in the block.

sub_stp

The increment of the shell subscript between adjacent elements in the block. Again, we have to allow for ranges with strided alignment.
3.4  class Range

A range is a mapping from an integer interval to a process dimension. A range is described by a collective object with local components of class Range. The public interface of the Range class is given in figure 3.3.

The constructors of the Range class itself do not implement any specific distribution formats. These are provided by the constructors for a series of derived classes.

3.4.1  Constructors and destructors

Range()

The default constructor. Creates a null (uninitialized) range object. The only members that can legally be applied to such an object are the assignment operator=(const Range& x) and the copy constructor.

Range(const Range& x)

Copy constructor. Range is implemented as a reference-counted handle class to an object of type RangeRep. The copy constructor copies the reference and increments the reference count in the representation object.

~Range()

Destructor. If the range is non-null, the representation object’s reference count is decremented. If this reduces the count to 0, the representation object is deleted.

3.4.2  Methods

size()

An inquiry function returning the extent of the range.

dim()

An inquiry function returning the underlying process dimension. For a collapsed range this is a collapsed dimension.

tem()

An inquiry function returning the parent template range.

bas()

An inquiry function returning the alignment base of this range in the parent template range. Element 0 of this range corresponds to element bas() in the template range.
class Range
{
public:
    Range();
    Range(const Range& x);
    ~Range();

    Range& operator=(const Range& x);

    int size() const;
    Dimension dim() const;

    Range tem() const;
    int bas() const;
    int str() const;

    Range ker() const;
    Range shell() const;

    Format format() const;

    int loExtension() const;
    int hiExtension() const;

    int isAligned(const Range& x);

    Range subrng(const int extent, const int base,
                 const int stride = 1) const;

    void location(Location* loc, const int glb) const;

    void crds(int* crd_count, int* crd_bas, int* crd_str) const;
    void crds(int* crd_count, int* crd_bas, int* crd_str,
              const int count, const int base,
              const int stride = 1) const;

    void block(Block* blk, const int crd) const;
    void block(Block* blk, const int crd,
               const int count, const int base,
               const int stride = 1) const;

    int volume() const;
    int offset(const Location& loc) const;

    Location operator()(const int k);
};

Figure 3.3: Public interface of the Range class.
str()
An inquiry function returning the alignment stride of this range relative to the parent template range. Adjacent elements of this range correspond to elements separated by `str()` in the template range. The result may be positive or negative (not zero).

ker()
An inquiry function returning the kernel range. In general this is a range of level one lower than this range. Result is undefined for level 0 ranges.

shell()
An inquiry function returning the shell range. This is a collapsed range large enough to describe any block of the parent template range. Result is undefined for level 0 ranges.

format()
An inquiry function returning a code that specifies the distribution format of this range. The result is a member of the enumeration type:

```c
enum Format {DIST_PRIMITIVE, DIST_COLLAPPED,
             DIST_BLOCK, DIST_CYCLIC, DIST_BLOCK_CYCLIC,
             DIST_IRRED, ...} ;
```

loExtension()
An inquiry function returning the width of the lower ghost extension of this range, or zero if there is no such extension. For a subrange, the returned value is the width of the ghost region for the parent range.

hiExtension()
An inquiry function returning the width of the upper ghost extension of this range, or zero if there is no such extension. For a subrange, the returned value is the width of the ghost region for the parent range.

isAligned(const Range& x)
A comparison operation that returns a non-zero value if `x` is alignment equivalent with this range. The definition of alignment equivalence used here is that ranges are aligned if their parent template ranges are aligned, and they have the same extent and alignment base and stride. Template ranges are aligned if they are distributed over the same dimension of the same process array\(^1\), and distribution parameters such as the distribution format and local block sizes are the same. Alignment equivalence does not require local memory layout parameters (in particular ghost extension widths) to be identical.

\(^1\)The process arrays must be described by the same object, not merely structurally similar process arrays.
subrng(const int extent, const int base, const int stride)

Returns a Range object representing a subrange of this range. The size of the subrange is extent and its elements are labelled 0,...,extent-1. The subrange includes elements

    base, base + stride, base + 2 * stride, ...
    base + (extent - 1) * stride

from the parent. The elements of the subrange are mapped to the underlying process dimension in the same way as the corresponding elements of the parent range (they are aligned with the corresponding elements of the parent). The value of stride must be positive or negative (not zero). The values of base and base + (extent - 1) * stride must be in the range 0,...,size()-1.

If the stride actual argument is omitted, it defaults to 1. If the base actual argument is also omitted, it defaults to 0.

If this range (the parent) is itself a subrange, the alignment parameters base and stride for the new subrange are composed with those of the parent, to give a simple map to the common template range.

location(Location* loc, const int glb)

Overwrites *loc with a location record corresponding to the element of the range with global subscript glb.

crds(int* crd_count, int* crd_bas, int* crd_str)

Returns the range of kernel subscripts for which this range has non-empty blocks. The kernel subscripts (usually equivalent to coordinates) are

    *crd_bas, *crd_bas + *crd_str, *crd_bas + 2 * crd_str, ...
    *crd_bas + (*crd_count - 1) * crd_str

crds(int* crd_count, int* crd_bas, int* crd_str,
     const int count, const int base, const int stride)

Defines the range of kernel subscripts for which some subrange has non-empty blocks. Equivalent to

    subrng(count, base, stride).crds(crd_count, crd_bas, crd_str)

block(Block* blk, const int crd)

Given a kernel subscript value, crd, writes the parameters of the associated block in *blk. The value of crd must be in the range defined by a call to crds.

The value written to blk->glb_bas is the smallest value of the global subscript contained in the block. The corresponding value of the shell subscript overwrites blk->sub_bas. The number of active elements in the block overwrites blk->count (in certain cases this may be zero).

The difference in the values of global and shell subscripts between adjacent active elements in the block overwrite blk->glb_str and blk->sub_str respectively.
block(Block* blk, const int crd,
    const int count, const int base, const int stride)

Given a kernel subscript value, crd, writes the parameters of the block associated with some subrange. Equivalent to

    subrng(count, base, stride), block(blk, crd)

The value of crd must be in the range defined by a call to crds with the same subrange parameters.

volume()

A bound on the number of elements associated with any single process for an array allocated with this range. This member is used directly to control the allocation of memory for elements of distributed arrays.

offset(const Location& loc)

Returns the total offset for the location loc.

For ranges with ghost extensions, note that the computed offset is from the start of the locally owned elements. It excludes the displacement (of size 10Extension()) from the start of allocated memory to the start of the locally owned elements.

3.4.3 Operators

operator()(const int glb)

Returns a Location object associated with global subscript value glb. Equivalent to

    Location operator()(const int glb) {
        Location res;
        location(&res, glb);
        return res;
    }

operator=(const Range& r)

Assignment operator. Copies the reference in handle r and increments the reference count in the representation object. Decrements the reference count of (and deletes, if necessary) any representation object referenced by the assignment variable prior to the assignment.
3.5 class CollapsedRange

The class CollapsedRange is a subclass of Range describing \textit{collapsed} (sequential) ranges. The \texttt{format} inquiry returns \texttt{DIST\_COLLAPSED} for a CollapsedRange or any subrange.

The public interface of the CollapsedRange class is

```cpp
class CollapsedRange : public Range {
  public:
    CollapsedRange(const int extent);
}
```

3.5.1 Constructors

CollapsedRange(const int extent)

Creates a \textit{collapsed} template range. This is a range of size \texttt{extent} mapped entirely to the local process. The \texttt{dim()} inquiry applied to a collapsed range returns a collapsed dimension. The level of a collapsed range is 1. The \texttt{ker()} inquiry will return a primitive range representing a collapsed dimension\textsuperscript{2}.

\textsuperscript{2}This is different to a null range.
3.6 class DimRange

The class DimRange is a subclass of Range describing primitive ranges. The format inquiry returns DIST_PRIMITIVE for a DimRange or any subrange.

The public interface of the DimRange class is

```cpp
class DimRange : public Range {
  public:
    DimRange(Dimension dim);
};
```

3.6.1 Constructors

DimRange(Dimension dim)

Creates a level 0 template range (a primitive range) describing dim. The size of this range is dim.size(). The dim() inquiry applied to the constructed range returns the constructor argument. The ker() inquiry is undefined.
3.7 class BlockRange

The class BlockRange is a subclass of Range describing uniform block-distributed ranges. The format inquiry returns DIST_BLOCK for a BlockRange or any subrange.

The public interface of the BlockRange class is

```cpp
class BlockRange : public Range {
    public:
        BlockRange(const int extent, Dimension dim);
        BlockRange(const int extent, Dimension dim,
                    const int wlo, const int whi);
}
```

3.7.1 Constructors

BlockRange(const int extent, Dimension dim)

Creates a block-distributed range of extent `extent` and kernel `dim`. The block size is

```
(extent + dim.size() - 1) / dim.size()
```

The constructed range is a template range.

BlockRange(const int extent, Dimension dim,
           const int wlo, const int whi)

Similar to the constructor above, but arrays constructed with this range (or its subranges) have ghost regions of width `wlo`, `whi` at the upper and lower edges of each block of the range. These values will be returned by the `loExtension()`, `hiExtension()` inquiries applied the constructed range.
3.8 class CyclicRange

The class CyclicRange is a subclass of Range describing *cyclically-distributed* ranges. The format inquiry returns DIST_CYCLIC for a CyclicRange or any subrange.

The public interface of the CyclicRange class is

```cpp
class CyclicRange : public Range {
  public:
    CyclicRange(const int extent, Dimension dim);
}
```

3.8.1 Constructors

*CyclicRange(const int extent, Dimension dim)*

Creates a cyclically-distributed range of extent `extent` and kernel `dim`. The number of cycles is

\[(\text{extent} + \text{dim}.\text{size}() - 1) / \text{dim}.\text{size}()\]

The constructed range is a template range.
3.9 class BlockCyclicRange

The class BlockCyclicRange is a subclass of Range describing block-cyclically-distributed ranges. The format inquiry returns DIST\_BLOCK\_CYCLIC for a BlockCyclicRange (or any subrange). The lev inquiry returns the value 2.

The public interface of the BlockCyclicRange class is

```cpp
class BlockCyclicRange : public Range {
  public:
    BlockCyclicRange(const int extent, Dimension dim,
                     const int blockSize);
}
```

3.9.1 Constructors

BlockCyclicRange(const int extent, Dimension dim,
                  const int blockSize)

Creates a block-cyclically-distributed range of extent extent distributed cyclically over dimension dim with block-size blockSize. The constructed block-cyclic range is a template range. The number of blocks is

```
numBlocks = (extent + blockSize - 1) / blockSize
```

The kernel of the range is a simple cyclically-distributed template range with number of cycles given by

```
numCycles = (numBlocks + dim.size() - 1) / dim.size()
```

The extent of the kernel range is thus

```
numCycles * dim.size()
```

The detailed structure of a block-cyclic range can be discovered with the aid of the lev and ker inquiries.
3.10 class IrregRange

The class IrregRange is a subclass of Range describing irregular block-distributed ranges. The format inquiry returns DIST_IRREG for an IrregRange or any subrange.

The public interface of the IrregRange class is

```cpp
class IrregRange : public Range {
public:
    IrregRange(Dimension dim, int* blocks) ;
};
```

3.10.1 Constructors

IrregRange(Dimension dim, int blocks [])

Creates a irregular block-distributed with kernel dim. The array blocks should have dim.size() elements containing non-negative integers. These define the block size associated with each process. The extent of the range is

\[
\text{dim.size()} \sum_{i=0}^{blocks[i]}
\]

The constructed range is a template range.
Chapter 4

Arrays

An Adlib array is a rectangular distributed array of any rank, whose elements are partitioned or replicated across some set of processes. The layout of the array is described by an object of type DAD which contains

- The rank, \( R \) of the array (i.e., its number of dimensions). \( R \) is greater than or equal to zero.
- The process group over which the elements of the array are distributed.
- A vector of \( R \) dimension map objects, describing the shape of the array, and the mapping of its index space into the dimensions of the process group, and the memory strides in the locally-held segment of the array.

The definition of the array is completed by a local array, or a pointer to a base address in the memory of each process, where the elements are actually stored. The type of the array elements is determined by the type of the local array segment—the DAD itself is blind to element type.

Note that the Adlib kernel makes no intrinsic assumptions about the ordering of array elements in local memory. For example, when a new distributed array is created, the dimension map vectors can set up to describe situations where either the first or the last dimension is “most rapidly varying” in memory\(^1\). Similarly, no assumption is made that array elements fill a contiguous region of memory. In Fortran 90 terms, a DAD record can describe an arbitrary regular section of some parent array.

\(^1\)There is a caveat—although all functions in the library will operate correctly regardless of majority, many of them are optimized to perform best on the assumption of first dimension most rapidly varying—the Fortran convention.
4.1 struct Map

A dimension map is part of a distributed array descriptor. It is a map from a Location to an offset in a local data segment. Dimension maps are associated with particular parent ranges. They also incorporate a memory stride.

The public interface of the Map class is

```cpp
struct Map {
    Map() ;
    Map(Range _range, const int _stride) ;

    Map ker() const ;
    Map shell() const ;

    Range rng() const ;
    int str() const ;

    int offset(const Location& i) const ;

    Range range ;
    int stride ;
} ;
```

4.1.1 Constructors and destructors

Map()

The default constructor. Creates a null (uninitialized) dimension map object.

Map(Range _range, const int _stride)

Create a dimension map with parent range _range and memory stride _stride.

4.1.2 Methods

ker()

An inquiry function returning the kernel dimension map. This is a dimension map associated with the kernel of the parent range. Undefined for dimension maps with level 0 parent ranges.

Equivalent to

```cpp
Map ker() const {
    return Map(rng().ker(), rng().shell().volume() * str()) ;
}
```

shell()

An inquiry function returning the shell dimension map. This is a dimension map associated with the shell of the parent range. Undefined for dimension maps with level 0 parent ranges.

Equivalent to
Map shell() const {
    return Map(rng().shell(), str()) ;
}

str()

An inquiry function returning the stride used in creation of this dimension map. To avoid confusion, note well that this is a memory stride, and this concept is completely unrelated to the alignment stride, returned by the str member of Range.

offset(const Location& i)

Returns the offset in local array segment produced by location i.
    Equivalent to
    
    int offset(const Location& i) const {
        return str() * rng().offset(i) ;
    }
4.2 struct DAD

A distributed array descriptor describes the layout of the elements of a distributed array. An array descriptor is represented by a collective object with local components of class DAD. The public interface of the DAD class is

```c
struct DAD {
    DAD(const int _rank, const Group& _group, Map* _maps) ;
    DAD(const int _rank, const Group& _group) ;
    DAD() ;
    int rnk() const ;
    const Group &grp() const ;
    const Range rng(const int r) const ;
    int str(const int r) const ;
    const Map &map(const int r) const ;
    DimensionSet sig() const ;
    int rank ;
    Group group ;
    Map* maps ;
} ;
```

4.2.1 Public data fields

rank

The rank, \( r \), of the array. Greater than or equal to zero.

group

The group over which the array is distributed.

maps

The vector of dimension maps of the array. Size of the vector should be \( r \). Distributed ranges in the maps held in this vector should be distributed over distinct (orthogonal) dimensions of group.
4.2.2 Constructors

DAD(const int _rank, const Group& _group, Map* _maps)

The normal constructor. Initializes all fields of the DAD object, simply copying the values passed as arguments. The constructor does not allocate any new vector internally. The maps pointer is simply copied. It is the caller's responsibility to ensure that associated vector persists as long as the constructed DAD record.

DAD(const int _rank, const Group& _group)

Simplified constructor. The maps field is left undefined. Since it is publically accessible, it can be initialized later.

DAD()

Default constructor. All fields are left undefined.

4.2.3 Methods

rpk()

An inquiry function returning the rank of the array (the value of rank).

grp()

An inquiry function returning the group over which the array is distributed (the value of group).

rng(const int r)

An inquiry function returning the rth range of the array—the value of map(r).rng().

str(const int r)

An inquiry function returning the rth memory stride of the array—the value of map(r).str().

map(const int r)

An inquiry function returning the rth dimension map of the array—the value of maps [r].

DimensionSet sig()

Return the signature of the array. The signature is the set of process dimensions over which the ranges of the array are distributed. Equivalent to
DimensionSet sig() const {
    DimensionSet signature;
    for (int r = 0; r < rank; r++)
        signature += rng(r).dim();
    return signature;
}

One common use for the sig member is to compute the set of process dimensions over which the array is replicated. This is given by

    grp().dims() - sig()
4.3 Examples

Suppose p is a group and x and y are ranges distributed over dimensions of this group. We want to create a new distributed array of floating point numbers described by these parameters. A typical procedure would be

```c
Map [2] a_maps ; int size = 1 ;

a_maps [0] = Map(x, size) ; size *= x.volume() ;
a_maps [1] = Map(y, size) ; size *= y.volume() ;

DAD a Frm(2, p, a_maps) ;
float* a_dat = new float [size] ;
```

New dimension maps are created in the vector maps. This code implements an array with first dimension “most-rapidly-varying” in memory². After the two multiplications by Range :: volume the value accumulated in size is the total volume of the local array segment. The group and map vector are installed in a Frm and the local data segment is allocated in a_dat.

Now suppose i and j are Location objects with parent ranges x and y respectively. We want to access the array element associated with these subscripts. A typical procedure would be

```c
Location i(x(11)), j(y(12)) ;...
if(i.dim.crd() == i.crd)
  if(j.dim.crd() == j.crd) {
    ...
    a_dat [a Frm.map(0).offset(i) + a Frm.map(1).offset(j)]
    ...
  }
```

First we test if the element is held on the local process by looking at the coordinate field in the subscript objects³, then the offset member is applied to the map objects to find the offsets in the local data segment. Compared with the sort of address computations involved in accessing elements of sequential arrays, free use of calls to member functions incurs a large, frequently intolerable, performance penalty. General techniques for eliminating these penalties are discussed in section 7.

Suppose we want to set up a description of a two-dimensional section of a, with subranges of x and y (something like a(11:u1, 12:u2) in Fortran).

²To implement an array with last dimension most-rapidly varying one would simply reverse the order of construction of the dimension maps.

³An outer test may be needed, to see if the local process is a member of p (ie, of a Frm.grp())
Map [2] b_maps;

b_maps[0] = Map(x.subrng(n1, 11), a.str(0));
b_maps[1] = Map(y.subrng(n2, 12), a.str(1));

DAD b_frm(2, p, b_maps);
float* b_dat = a_dat;

The memory strides and local data segment are copied from the original array.
(Here n1 is u1 - 11 + 1 and n2 is u2 - 12 + 1.)

If, instead, we wanted to set up a description of a one dimensional section of a, with global subscript 11 in the first dimension, retaining the whole of the second dimension (something like a(11, : ) in Fortran) we could write

Map [1] c_maps;

c_maps[0] = a.map(1);

Location i(x(11));

DAD c_frm(1, p / i, c_maps);
float* c_dat = a_dat + a_frm.map(0).offset(i);

No new map objects need be created: the dimension map of the unsubscripted dimension is copied from the original array descriptor. The process group over which the section is distributed is p / i. The offset for the first dimension is added to the base address of the original array. One can access element j of the section c by

if(c_frm.grp().member())
  if(j, dim, crd() == j, crd) {
    ...
    c_dat[c_frm.map(0).offset(j)]
    ...
  }

The two examples given above easily generalize to allow construction any Fortran-90 style regular section of any distributed array.

4.4 Array shape

As in Fortran, the shape of an array, a, is defined as the vector of extents of its ranges, ie (a. rng(0). size(), ... , a. rng(R-1). size()), where R is the rank of the array.

---

4The offset pointer value is only valid on processes that hold part of the section, but it does no harm to do the addition on other processes. The value b_dat should never be used on processes where it is not valid—ie on processes outside the group p / i.
4.5 Alignment

An array, a is \textit{aligned with} an array b if they are distributed over the equivalent
process groups and their ranges are all equivalent:

\[
\begin{align*}
    a.\text{grp}(0) & \approx b.\text{grp}(0) \\
    a.\text{rng}(0) & \approx b.\text{rng}(0) \\
    & \vdots \\
    a.\text{rng}(R-1) & \approx b.\text{rng}(R-1)
\end{align*}
\]

The equivalence relations can be assumed to be the ones defined by the members
\texttt{Group :: isEqual} and \texttt{Range :: isAligned}. Informally, two groups or two
ranges are equivalent if they are \textit{structurally equivalent}. The informal meaning
of array alignment is that corresponding elements of the two arrays are stored
on the same process, or replicated over the same group of processes.

The array, a is aligned with b \textit{with replicated-alignment} in some dimensions
if the groups are equivalent, and the ranges of a can be paired with equivalent
ranges of b by omitting the ranges of b associated with the specified dimensions.

4.6 Accessibility

An array, a, is \textit{accessible} at a particular point in program execution if it is
distributed over a group contained in the active process group:

\[
a.\text{grp}(0) \subseteq \text{apg}
\]

The contains relation can be assumed to be the one defined by the member
\texttt{Group :: contains}. Informally this means that all copies of all elements of
the array are available within the set of processes sharing the current thread of
control.

\footnote{A special feature is that alignment equivalence of ranges does not imply that the ranges
must have the same layout options. So, for example, two arrays may be aligned even if one
has ghost extensions and the other doesn't.}
Chapter 5

Communication schedules

This chapter defines the Adlib communication library. Currently all communication functions in Adlib take the form of collective transformations on distributed arrays. These transformations are implemented in terms communication schedules. Each kind of transformation has an associated class of schedules. Specific instances of these schedules, involving particular data arrays and particular parameters, are created as objects from the classes concerned. Executing a schedule initiates the communications required to implement the transformation. A single schedule may be executed many times, repeating the same communication pattern.

Future versions of Adlib may support other communication paradigms. A convenient extension would allow primitives for one-sided communication. These would permit direct read or write access to remote patches of distributed arrays.

5.1 General features of schedules

Typically the communication schedules described in this chapter have only three public members: a constructor, a destructor, and an execute member.

Usually the constructor is passed all the detailed information describing how the input and output data is organized and how it is to be transformed. This includes the array descriptors and any parameters of the transformation. The constructor may perform extensive processing on these arguments to convert them into a simplified list of data movements. In some cases this processing may involve exchange of information—ie, communication—between the active processes. A schedule constructor should always be treated as a collective operation. As usual with arguments of collective operations, every member of the active process group must pass consistent arguments to the schedule constructor call. In particular this means that any simple value passed to a schedule constructor (any argument that is not a local component of a collective object such as a DAD, or a vector of local array elements associated with a DAD) must
have \textit{identical} values in every process.

Local segment addresses of input and output arrays—the vectors where individual array elements are stored—are usually \textit{not} passed to the schedule constructor. Instead these addresses are passed to the \texttt{execute} member. One practical reason for not storing the data pointers in the schedule itself is that these vectors are typically allocated by the user, outside control of the library. The user’s program may be written in a programming language other than C++. Cacheing pointers to memory allocated and managed outside the library itself can cause problems in some programming environments—especially in garbage-collected languages such as Java. Another advantage of specifying the data pointers at execution time rather than schedule construction time is that in principle it allows the same schedule to be used with different arrays, providing corresponding arrays have identical DADs.

The \texttt{execute} member nearly always involves communication. It should of course be treated as a collective operation, executed by all members of the active process group.

Most schedule constructors will access the \texttt{apg} variable. The state of this variable should accurately describe the set of processes involved in creating the schedule. The active process group at the point of execution of a schedule should be the same as the active process group at its point of creation.

The following sections describe the interfaces of the schedule classes. The choice of transformations in the current schedule library has been strongly influenced by HPF. Apart from a handful of generic operations like \texttt{Remap}, \texttt{Gather}, \texttt{Scatter} and \texttt{WriteHalo}, most of the schedules are designed to support the specific array syntax and array transformational intrinsics of Fortran 90\footnote{An exception is the \texttt{MultiShift} schedule, which was actually introduced to support the array syntax of C*}. Focusing on Fortran provides a concrete standard with respect to which some kind of completeness can be achieved.

In the interfaces given in the following sections, only the constructor and the \texttt{execute} members will be exhibited. Base classes (public or otherwise), members of base classes, destructors, etc, will be suppressed. Subsections describing the constructors detail any restrictions the arguments must satisfy. Subsections describing the \texttt{execute} members specify the effects of the schedules. Below we briefly discuss various terms and notations used, following the subheadings used in the schedule definitions.

\subsection*{5.1.1 Value restrictions}

These are simply restrictions on the input values of data, such as constraints ensuring values used as subscripts are in the required bounds.

\subsection*{5.1.2 Type restrictions}

These are restrictions on the types of array elements. Typically if an array’s elements are to be communicated, they must have \texttt{POD} type. The idea of a
POD type is defined in the ANSI C++ standard. Informally it is any type that can be copied to a byte array by standard operations such as \texttt{memcpy}.

Schedules that perform arithmetic operations or comparisons will impose further restrictions on the types of the array elements. In the current library all such schedules are template classes, parametrized by the element type. The template argument can only be instantiated to a type that supports a suitable set of arithmetic operations.

5.1.3 Shape restrictions

Restrictions on the shape of the array arguments, such as the requirement that a particular pair of arrays passed to the constructor should have the same shape.

5.1.4 Alignment restrictions

Many of the schedules in the library assume some alignment relations (see section 4.5) between their array arguments. Although these are usually natural restrictions from the point of view of the parallel implementation, they can sometimes take programmers by surprise, or appear unnecessarily complex.

For example, it is required that the source array for a \texttt{Shift} is aligned with the destination array. This sometimes surprises people, although it probably shouldn't. Historically, an essential feature of the shift operation is that it can be implemented very efficiently by simple nearest neighbour communications. The library could easily have been defined to implement shift without the alignment constraint, but then implementation would be essentially the same as the more complex \texttt{Remap} operation, and the main point of providing \texttt{Shift} is that it is a simpler, relatively light-weight operation. If versions of the library functions without alignment restrictions are needed, they can always be constructed by combining the constrained operation with \texttt{Remap} operations.

Because the alignment constraints implied by particular implementations of the schedules can be quite complex, the restrictions are sometimes simplified slightly in the following sections. The conditions given are always sufficient, but sometimes weaker conditions would have been adequate. For simplicity, we restrict ourselves to the vocabulary defined in section 4.5.

5.1.5 Accessibility restrictions

Accessibility restrictions are needed to ensure that copies of array elements are available inside the group of processes that execute a schedule. Access to elements stored outside the active process group is not allowed by the collective communication paradigm currently implemented by Adlib.

5.1.6 Argument lifetime

This usually refers to the situation where a pointer to a \texttt{DAD} is passed to a schedule constructor. To avoid the overhead of copying the \texttt{DAD}, the schedule
often saves a reference to the existing DAD object. The programmer must then
ensure that the DAD is not deleted during the lifetime of the schedule.

Deleting a DAD before completing communications involving the associated
array would be fairly unusual in practise, so these “argument lifetime” restric-
tions are not expected to be troublesome.

5.1.7 Effect

In describing the effect of schedules, array subscripting notation will often be
used informally. In this context, the subscripting should always be understood
in terms of global subscripts to abstract global arrays, without reference to the
distributed nature of the actual arrays.

5.1.8 Replicated data

By definition, an array is replicated over a particular process dimension if the
dimension appears in its process group but not its signature (ie, the array has
no range distributed over the dimension concerned).

As a rule it is good practise for programmers to maintain the same values
in all copies of an element of a replicated array. If all arrays input to the
communication schedules meet this requirement, it is guaranteed that those
output do. This is not an absolute requirement on arrays passed to schedules,
and the sections on individual schedules discuss the effect of defaulting on this
rule.

5.1.9 Overlap restrictions

In general the library does not allow in-place updates. No array written by
a communication schedule should overlap with an array read by the schedule.
The sections on individual to schedules give the specific restrictions.
5.2 class Remap

A *remap schedule* is a communication schedule for copying the elements of one distributed array to another. The source and destination must have the same shape and same element-type, but no relation between the mapping of the two arrays is required. If the target array has a replicated mapping, the remap operation implements a broadcast.

A remap schedule is described by a collective object with local components of class *Remap*. The public interface of the *Remap* class is

```cpp
class Remap {
public:
    Remap(const DAD* dst, const DAD* src, const int len) ;

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

5.2.1 Constructor

*Remap(const DAD* dst, const DAD* src, const int len)*

The source array is described by the DAD *src and the destination array is described by the DAD *dst*. *len* specifies the size of each array element, in bytes.

*Type restrictions*: The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size *len* bytes.

*Shape restrictions*: The source and destination array must have the same shape (see section 4.4).

*Accessibility restrictions*: The source and destination arrays must be accessible (see section 4.6).

*Argument lifetime*: The dst and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.2.2 Method

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

Arguments are the base addresses for local segments of the destination and source arrays. They should point to vectors of the locally held elements.
**Effect:** Copy the elements of the source array to the corresponding elements of the destination array.

**Replicated data:** If the source array has replicated mapping, the value for a particular element is taken from *one* of its copies. If the destination array has replicated mapping, identical values are broadcast to *every* copy of the element.

**Overlap restrictions:** In-place updates are not allowed. The pairs *dst*, dstDat and *src*, srcDat must define non-overlapping arrays.
5.3 class Shift

A shift schedule is a communication schedule for shifting the elements of a distributed array along one of its dimensions, placing the result in another array. The source and destination 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

```cpp
class DAD;
enum Mode {CYCL, EDGE, NONE};

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

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

5.3.1 Constructor

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

The source array is described by the DAD *src and the destination array is described by the DAD *dst. len specifies the size of each array element, in bytes. The shift amount, which may be negative, is given by shift. The dim argument selects the array dimension in which the shift occurs. The flag mode specifies the type of shift. It takes one of the values CYCL, EDGE or NONE.

**Value restrictions:** The value of dim must be in the range 0, \ldots, R−1 where R is the rank of the source array.

**Type restrictions:** The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size len bytes.

**Shape restrictions:** The source and destination array must have the same shape (see section 4.4).

**Alignment restrictions:** The source array must be aligned with the destination array (see section 4.5).
**Accessibility restrictions:** The source and destination arrays must be accessible (see section 4.6).

**Argument lifetime:** The dst and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

### 5.3.2 Method

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

Arguments are the base addresses for local segments of the destination and source arrays. They should point to vectors of the locally held elements.

**Effect:** On exit, if mode is CYCL, the value of

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

is

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

where \( N \) is the extent of dimension \( \text{dim} \). If mode is EDGE, the exit value of the dst element is

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

if \( x_{\text{dim}} + \text{shift} \) is in the range \( 0, \ldots, N - 1 \), or *unchanged from the entry value*,

if not. If mode is NONE executing the schedule has no effect.

**Replicated data:** If the arrays have replicated mapping, values for individual copies of the destination are generally taken from the nearest copy of the corresponding source array element. The definition of “nearest” is implementation dependent. This schedule does not implement a broadcast—consistent replication of copies in the destination array relies on consistency of copies of the source array.

**Overlap restrictions:** In-place updates are not allowed. The pairs *dst*, dstDat and *src*, srcDat must define non-overlapping arrays.
5.4 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 a distributed array placing the result in another 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

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

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

5.4.1 Constructor

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

The source array is described by the DAD *src and the destination array is described by the DAD *dst. len specifies the size of each array element, in bytes. The array of shift amounts (any of which may be negative), is described by the DAD shf. The base address for the local segment of this array is shfDat. The shift-amount array should have rank one less than the source array. The dim argument selects the array dimension in which the shift occurs. The flag mode specifies the type of shift. It takes one of the values CYCL, EDGE or NONE.

**Value restrictions:** The value of dim must be in the range 0,...,R–1 where R is the rank of the source array.

**Type restrictions:** The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size len bytes.

**Shape restrictions:** The source and destination array must have the same shape (see section 4.4). The shape of the shift array must be obtained from the shape of the source array by deleting dimension dim.
Alignment restrictions: The source array must be aligned with the destination array. The shift-amount array should be aligned with the destination array, with replicated alignment over dimension \( \text{dim} \) (see section 4.5).

Accessibility restrictions: The source array, the shift-amount array, and the destination array must be accessible (see section 4.6).

Argument lifetime: The \( \text{dst} \) and \( \text{src} \) arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule. (There are no such requirements for the shift-amount array).

5.4.2 Method

\texttt{execute(void* dstDat, void* srcDat)}

Arguments are the base addresses for local segments of the destination and source arrays. They should point to vectors of the locally held elements.

Effect: The description of the exit value of \( \text{dst} \) is identical to the description after execution of a \texttt{Shift} schedule (see section 5.3), except that the constant \texttt{shift} is replaced by

\[
\text{shf}[x_0, \ldots, x_{\text{dim}-1}, x_{\text{dim}+1}, \ldots, x_{R-1}]
\]

Replicated data: If the arrays have replicated mapping, values for individual copies of the destination are generally taken from the nearest copy of the corresponding source array element. The definition of “nearest” is implementation dependent. This schedule does not implement a broadcast—consistent replication of copies in the destination array relies on consistency of copies of the source array.

Overlap restrictions: In-place updates are not allowed. The pairs \( \ast \text{dst}, \text{dstDat} \) and \( \ast \text{src}, \text{srcDat} \) must define non-overlapping arrays.
5.5 class MultiShift

A multishift schedule is a communication schedule for shifting the elements of a distributed array along some or all of its dimensions concurrently, placing the result in another 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 DAD;
enum Mode {CYCL, EDGE, NONE};

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

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

5.5.1 Constructor

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

The source array is described by the DAD *src and the destination array is described by the DAD *dst. len specifies the size of each array element, in bytes. Vectors shift and mode have extent $R$—the rank of the source array. The shift amount, which may be negative, in dimension $r$ is given by shift $[r]$. The flag mode $[r]$ specifies the type of shift: It takes one of the values CYCL, EDGE or NONE.

Value restrictions: The value of dim must be in the range $0, \ldots, R-1$ where $R$ is the rank of the source array.

Type restrictions: The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size len bytes.

Shape restrictions: The source and destination array must have the same shape (see section 4.4).

Alignment restrictions: The source array must be aligned with the destination array (see section 4.5).
Accessibility restrictions: The source and destination arrays must be accessible (see section 4.6).

Argument lifetime: The dst and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.5.2 Method

eexecute(void* dstDat, void* srcDat)

Arguments are the base addresses for local segments of the destination and source arrays. They should point to vectors of the locally held elements.

Effect: 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,...,R - 1 (see section 5.3).

Replicated data: If the arrays have replicated mapping, values for individual copies of the destination are generally taken from the nearest copy of the corresponding source array element. The definition of “nearest” is implementation dependent. This schedule does not implement a broadcast—consistent replication of copies in the destination array relies on consistency of copies of the source array.

Overlap restrictions: In-place updates are not allowed. The pairs *dst, dstDat and *src, srcDat must define non-overlapping arrays.
5.6 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

```cpp
class DAD;
enum Mode {CYCL, EDGE, NONE};
class WriteHalo {
public:
    WriteHalo(const DAD* src, const int len,
               const int wlo [], const int whi [],
               const Mode [] mode);
    WriteHalo(const DAD* src, const int len);
    void execute(void* srcDat);
private:
    ...
};
```

5.6.1 Constructors

WriteHalo(const DAD* src, const int len,
          const int wlo [], const int whi [],
          const Mode [] mode)
WriteHalo(const DAD* src, const int len)

The array is described by the DAD *src. len specifies the size of each array element, in bytes. In the first form, the vectors wlo, whi and mode have extent R—the rank of the array.

Vectors wlo and wlo define the halo of ghost cells updated by the schedule. The upper and lower widths in dimension r are given by wlo [r] and whi [r]. These values are non-negative, and can only be non-zero if array src actually has suitable ghost extensions in the dimension concerned. More specifically, if the array src was created using a range with ghost extensions wlo_act, whi_act as its rth dimension, i.e

\[
\begin{align*}
wlo_{act} &= src.rng(r).loExtension() \\
whi_{act} &= src.rng(r).hiExtension()
\end{align*}
\]

it is required that

\[
\begin{align*}
whi [r] &\leq whi_{act} \\
wlo [r] &\leq wlo_{act}
\end{align*}
\]

67
The situation is complicated if the array is a section, or some other array with non-trivial alignment stride. In practice it is unusual to construct WriteHalo schedules for such arrays, but for completeness we describe the constraints that apply in that case. First, by definition, the ghost extensions of a subrange are those of its template range. Now, suppose the array of which src is a section has ghost extensions \(\text{name}_{act}, \text{name}_{act}\). If the alignment stride of \(\text{src.rng}(r)\) is \(s\), the required constraints are

\[
\begin{align*}
\text{name}_{act} & \leq \text{name}_{act} \\
\text{name}_{act} & \leq \text{name}_{act}
\end{align*}
\]

The point to note is that the widths defined in the WriteHalo construtor are expressed in terms of the global subscript of the range of the array, whereas the ghost extensions of the array are measured in units of template range subscripts.

The vector \text{mode} defines how ghost cells are updated in each dimension— including how the cells at the extremes of the array are updated. Its elements take value CYCL, EDGE or NONE.

Note that (as usual for ordinary data arguments of collective constructors) the vectors \text{name}, \text{name} and \text{mode} must have identical values in all members of the active process group.

The second form of constructor defines a schedule in which whole of the array ghost region is updated using EDGE mode:

\[
\begin{align*}
\text{name}[r] & = \text{src.rng}(r).\text{name}(r) \\
\text{name}[r] & = \text{src.rng}(r).\text{name}(r) \\
\text{mode}[r] & = \text{EDGE}
\end{align*}
\]

In this case the behaviour of the constructed schedule is ill-defined if the relevant ranges have non-unit alignment stride.

Type restrictions: The elements of array must be a POD type (see section 5.1.2) of size \text{name} bytes.

Accessibility restrictions: The array must be accessible (see section 4.6).

Argument lifetime: The src argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.6.2 Method

\text{execute(\text{void* srcDat})}

Argument is the base addresses for the local segment of the array. It should point to a vector of the locally held elements.
Figure 5.1: Illustration of the effect of executing a write-halo schedule.

**Effect:** 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.

If the value of the mode element for a dimension is EDGE, ghost cells past the extreme ends of the array range are not updated by the write-halo operation. If the value is CYCL, those cells are updated assuming cyclic wraparound\(^2\). If the value is NONE, there is no updating at all of the ghost cells associated with this dimension.

**Replicated data:** If the array has replicated mapping, values for individual copies of the ghost cell are generally taken from the nearest copy of the corresponding physical array element. The definition of "nearest" is implementation dependent. This schedule does not implement a broadcast—consistent replication of copies in the final state of the array relies on consistency of copies in the

---

\(^2\)This option may lead to odd effects if the source array is actually a section with a triplet subscript. The updated cells, past the ends of the section, may actually reside in the physical segment of the parent array.
initial state of the array.
5.7 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

```cpp
class DAD;

class Gather {
    public:
        Gather(const DAD* dst, const DAD* src, const int len,
               const DAD* subs [], const int* subsDat [],
               const DAD* msk, const int* mskDat);

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

5.7.1 Constructor

```cpp
Gather(const DAD* dst, const DAD* src, const int len,
       const DAD* subs [], const int* subsDat [],
       const DAD* msk, const int* mskDat)
```

The source array is described by the DAD *src and the destination array is described by the DAD *dst. 

len specifies the size of each array element, in bytes. The vectors subs and subsDat have extent R—the rank of the source array. The subscript arrays are defined by the pairs *subs [r], subsDat [r]. If msk is non-null, the pair *msk, mskDat defines a mask array.

**Value restrictions:** All elements of the rth subscript array must be in the range 0,...,N−1 where N is the extent of the source array in its rth dimension.

**Type restrictions:** The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size len bytes.

**Shape restrictions:** The destination array, all subscript arrays, and the mask array, if defined, must have the same shape (see section 4.4).
Alignment restrictions: All subscript arrays and the mask array, if defined, must be aligned with the destination array (see section 4.5).

Accessibility restrictions: The source and subscript arrays, the mask array, if defined, and the destination array must all be accessible (see section 4.6).

Argument lifetime: The dst and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule. (There are no such requirements for the subscript and mask arrays).

5.7.2 Methods

execute(void* dstDat, void* srcDat)

Arguments are the base addresses for local segments of the destination and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

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

is

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

unless the mask array is defined and

\[ \text{mask} [x_0, \ldots, x_{S-1}] \]

was zero, in which case the exit value of the dst element is unchanged from the entry value. Here \( \text{subs}_r \) denote the \( r \)th subscript array and \( S \) is the rank of the destination array.

Replicated data: If the source array has replicated mapping, the value for a particular element is taken from one of its copies. If the destination array has replicated mapping, identical values are broadcast to every copy of each element.

Overlap restrictions: In-place updates are not allowed. The pairs *dst, dstDat and *src, srcDat must define non-overlapping arrays.
5.8 class Scatter

A scatter schedule is a communication schedule for scattering 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

```cpp
class Scatter {
    public:
        Scatter(const DAD* src, const DAD* dst, const int len,
                const DAD* subs [], const int* subsDat [],
                const DAD* msk, const int* mskDat);

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

5.8.1 Constructor

Scatter(const DAD* src, const DAD* dst, const int len,
         const DAD* subs [], const int* subsDat [],
         const DAD* msk, const int* mskDat)

The source array is described by the DAD *src and the destination array is described by the DAD *dst. len specifies the size of each array element, in bytes. The vectors subs and subsDat have extent R—the rank of the destination array. The subscript arrays are defined by the pairs *subs [r], subsDat [r]. If msk is non-null, the pair *msk, mskDat defines a mask array.

Note well that the source and destination arguments of Scatter members are reversed relative to the conventions for other communication schedules. This is to emphasize the symmetry with Gather.

Value restrictions: All elements of the rth subscript array must be in the range 0, . . . , N − 1 where N is the extent of the destination array in its rth dimension.

Type restrictions: The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size len bytes.

Shape restrictions: The source array, all subscript arrays, and the mask array, if defined, must have the same shape (see section 4.4).
Alignment restrictions: All subscript arrays and the mask array, if defined, must be aligned with the source array (see section 4.5).

Accessibility restrictions: The source and subscript arrays, the mask array, if defined, and the destination array must be all accessible (see section 4.6).

Argument lifetime: The dst and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule. (There are no such requirements for the subscript and mask arrays).

5.8.2 Methods

execute(void* srcDat, void* dstDat)

Arguments are the base addresses for local segments of the source and destination arrays. They should point to vectors of the locally held elements.

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

\[
\text{if}(\text{msk} \ [x_0, \ldots, x_{S-1}]) \\
\text{dst} \ [\text{subs}_0 \ [x_0, \ldots, x_{S-1}], \ldots, \text{subs}_{S-1} \ [x_0, \ldots, x_{S-1}]] = \\
\text{src} \ [x_0, \ldots, x_{S-1}]
\]

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 a null pointer, the assignment is unconditional. Here subs\(_r\) denote the \(r\)th subscript array.

Replicated data: If the source array has replicated mapping, the value for a particular element is taken from one of its copies. If the destination array has replicated mapping, identical values are broadcast to every copy of each element.

Overlap restrictions: In-place updates are not allowed. The pairs \(*\text{dst}, \text{dstDat}\) and \(*\text{src}, \text{srcDat}\) must define non-overlapping arrays.
5.9 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 5.7. It is functionally equivalent to certain Fortran-90 array assignments involving vector subscripts. 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

```cpp
class DAD ;

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

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

5.9.1 Constructor

VecGather(const DAD* dst, const DAD* src, const int len,
           const DAD* subs [], const int* subsDat [])

The source array is described by the DAD *src and the destination array is described by the DAD *dst. *len specifies the size of each array element, in bytes. The source and destination arrays have the same rank, *R*. The vectors *subs and subsDat have extent *R*. If *subs [*r]* is non-null, the pair *subs [*r]*, subsDat [*r]* defines the *r*th subscript array.

**Value restrictions:** If the *r*th subscript array is defined, all its elements must be in the range 0, ..., *N* - 1 where *N* is the extent of the source array in its *r*th dimension.

**Type restrictions:** The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size *len* bytes.

**Shape restrictions:** If the *r*th subscript array is defined it must be rank-1 and its extent must be *M*, where *M* is the extent of the destination array in its *r*th dimension.

If the *r*th subscript array is *undefined* (because *subs [*r]* is null) the destination and source arrays must have the same extent in their *r*th dimensions.
Alignment restrictions: If defined, the rth subscript array must be aligned to the destination array, with replicated alignment in all dimensions except r (see section 4.5).

Accessibility restrictions: The source and subscript arrays and the destination array must all be accessible (see section 4.6).

Argument lifetime: The dst and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule. (There are no such requirements for the subscript arrays.)

5.9.2 Methods

execute(void* dstDat, void* srcDat)

Arguments are the base addresses for local segments of the destination and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

$$\text{dst} [x_0, \ldots, x_{n-1}]$$

is

$$\text{src} [\text{sub}_0 [x_0], \ldots, \text{sub}_{n-1} [x_{n-1}]]$$

Here sub$_r$ denotes the rth subscript array. If sub$_r$ is null, the value of the expression sub$_r$ [x$_r$] in this formula is defined to be $x_r$ (hence if all subscripts are undefined, this operation is equivalent to a Remap operation).

Replicated data: If the source array has replicated mapping, the value for a particular element is taken from one of its copies. If the destination array has replicated mapping, identical values are broadcast to every copy of each element.

Overlap restrictions: In-place updates are not allowed. The pairs *dst, dstDat and *src, srcDat must define non-overlapping arrays.
5.10 class VecGatherMsk

A masked 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—and controlled by a mask array. This schedule can be regarded as an optimized special case of the general gather schedule of section 5.7. It is functionally equivalent to certain Fortran-90 array assignments involving vector subscripts and executed inside WHERE statements. A masked vector-subscript gather schedule is described by a collective object with local components of class VecGatherMsk. The public interface of the VecGatherMsk class is

    class DAD;

    class VecGatherMsk
    {
      public:
        VecGatherMsk(const DAD* dst, const DAD* src, const int len,
                      const DAD* subs [], const int* subsDat [],
                      const DAD* msk, const int* mskDat);

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

      private:
        ...
    }

5.10.1 Constructor

VecGatherMsk(const DAD* dst, const DAD* src, const int len,
             const DAD* subs [], const int* subsDat [],
             const DAD* msk, const int* mskDat)

The source array is described by the DAD *src and the destination array is described by the DAD *dst. len specifies the size of each array element, in bytes. The source and destination arrays have the same rank, R. The vectors subs and subsDat have extent R. If subs [r] is non-null, the pair *subs [r], subsDat [r] defines the rth subscript array. The pair *msk, mskDat defines a mask array.

Value restrictions: If the rth subscript array is defined, all its elements must be in the range 0, ..., N - 1 where N is the extent of the source array in its rth dimension, unless the subscript element is unused because of masking. (In other words, this restriction need not apply for elements

    subs. [x]

such that

    msk [x_0, ..., x_{R-1}]

is zero for all allowed values of x_0, ..., x_{r-1}, x_{r+1}, ..., x_{R-1}.)

77
**Type restrictions:** The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size `len` bytes.

**Shape restrictions:** If the `r`th subscript array is defined it must be rank-1 and its extent must be `M`, where `M` is the extent of the destination array in its `r`th dimension.

If the `r`th subscript array is `undefined` (because `subs [r]` is null) the destination and source arrays must have the same extent in their `r`th dimensions.

The mask array must have the same shape as the destination array (see section 4.4).

**Alignment restrictions:** If defined, the `r`th subscript array must be aligned to the destination array, with replicated alignment in all dimensions except `r`. The mask array must be aligned with the destination array (see section 4.5).

**Accessibility restrictions:** The source, subscript, mask array, and the destination array must all be accessible (see section 4.6).

**Argument lifetime:** The `dst` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule. (There are no such requirements for the subscript and mask arrays).

### 5.10.2 Methods

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

Arguments are the base addresses for local segments of the destination and source arrays. They should point to vectors of the locally held elements.

**Effect:** On exit, the value of

```c
dst [x_0, ..., x_{N-1}]
```

is

```c
src [subs_0 [x_0], ..., subs_{R-1} [x_{R-1}]]
```

unless

```c
mask [x_0, ..., x_{N-1}]
```

was zero, in which case the exit value of the `dst` element is unchanged from the entry value. Here `subs` denote the `r`th subscript array. If `subs [r]` is null, the value of the expression `subs_r [x_r]` in this formula is defined to be `x_r` (hence if all subscripts are undefined, this operation is equivalent to a `Remap` operation).
**Replicated data:** If the source array has replicated mapping, the value for a particular element is taken from *one* of its copies. If the destination array has replicated mapping, identical values are broadcast to *every* copy of each element.

**Overlap restrictions:** In-place updates are not allowed. The pairs *dst*, `dstDat` and *src*, `srcDat` must define non-overlapping arrays.
5.11 class VecScatter

A vector-subscript scatter schedule is a communication schedule for scattering 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 5.8. It is functionally similar to certain Fortran-90 array assignments involving vector subscripts—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

```cpp
class DAD ;

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

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

5.11.1 Constructor

VecScatter(const DAD* src, const DAD* dst, const int len,
           const DAD* subs [], const int* subsDat [])

The source array is described by the DAD *src and the destination array is described by the DAD *dst. len specifies the size of each array element, in bytes. The source and destination arrays have the same rank, R. The vectors subs and subsDat have extent R. If subs [r] is non-null, the pair *subs [r], subsDat [r] defines the rth subscript array.

Note well that the source and destination arguments of VecScatter members are reversed relative to the conventions for other communication schedules. This is to emphasize the symmetry with VecGather.

Value restrictions: If the rth subscript array is defined, all its elements must be in the range 0, ..., N – 1 where N is the extent of the destination array in its rth dimension.

Type restrictions: The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size len bytes.
**Shape restrictions:** If the rth subscript array is defined it must be rank-1 and its extent must be \( M \), where \( M \) is the extent of the source array in its rth dimension.

If the rth subscript array is *undefined* (because \( \text{subs} [r] \) is null) the destination and source arrays must have the same extent in their rth dimensions.

**Alignment restrictions:** If defined, the rth subscript array must be aligned to the source array, with *replicated alignment in all dimensions except r* (see section 4.5).

**Accessibility restrictions:** The source and subscript arrays and the destination array must all be accessible (see section 4.6).

**Argument lifetime:** The dst and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule. (There are no such requirements for the subscript arrays.)

### 5.11.2 Method

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

Arguments are the base addresses for local segments of the source and destination arrays. They should point to vectors of the locally held elements.

**Effect:** The effect is like executing the assignment

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

for every tuple, \((x_0, \ldots, x_{R-1})\), of global subscripts of the source array. These assignments are executed in an undefined order. Here \( \text{subs}_r \) denotes the rth subscript array. If \( \text{subs} \ [r] \) is null, the value of the expression \( \text{subs}_r \ [x_r] \) in the left-hand-side of the assignment is defined to be \( x_r \) (hence if all subscripts are undefined, this operation is equivalent to a \textbf{Remap} operation).

**Replicated data:** If the source array has replicated mapping, the value for a particular element is taken from *one* of its copies. If the destination array has replicated mapping, identical values are broadcast to *every* copy of each element.

**Overlap restrictions:** In-place updates are not allowed. The pairs \(*\text{dst}, \text{dstDat} \) and \(*\text{src}, \text{srcDat} \) must define non-overlapping arrays.
5.12 class VecScatterMs

A *masked vector-subscript scatter schedule* is a communication schedule for scattering 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 5.8. It is functionally similar to certain Fortran-90 array assignments involving vector subscripts and executed inside WHERE statements—minus the constraints that enforce determinism in Fortran. A vector-subscript scatter schedule is described by a collective object with local components of class VecScatterMs. The public interface of the VecScatterMs class is

```cpp
class DAD;

class VecScatterMs {
public:
  VecScatterMs(const DAD* src, const DAD* dst, const int len,
                const DAD* subs [], const int* subsDat [],
                const DAD* msk, const int* mskDat);

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

5.12.1 Constructor

VecScatterMs(const DAD* src, const DAD* dst, const int len,
             const DAD* subs [], const int* subsDat [],
             const DAD* msk, const int* mskDat)

The source array is described by the DAD *src and the destination array is described by the DAD *dst. len specifies the size of each array element, in bytes. The source and destination arrays have the same rank, R. The vectors subs and subsDat have extent R. If subs [r] is non-null, the pair *sub [r], subsDat [r] defines the rth subscript array. The pair *msk, mskDat defines a mask array.

*Note:* the source and destination arguments of VecScatterMs members are reversed relative to the conventions for most communication schedules. This is to emphasize the symmetry with VecGatherMs.

**Value restrictions:** If the rth subscript array is defined, all its elements must be in the range 0,...,N - 1 where N is the extent of the destination array in its rth dimension, unless the subscript element is unused because of masking. (In other words, this restriction need not apply for elements

\[ \text{subs}_r [x_r] \]

such that
mask \([x_0, \ldots, x_{R-1}]\)

is zero for all allowed values of \(x_0, \ldots, x_{r-1}, x_{r+1}, \ldots, x_{R-1}\).

**Type restrictions:** The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size len bytes.

**Shape restrictions:** If the \(r\)th subscript array is defined it must be rank-1 and its extent must be \(M\), where \(M\) is the extent of the source array in its \(r\)th dimension.

If the \(r\)th subscript array is *undefined* (because \(\text{subs} \ [r]\) is null) the destination and source arrays must have the same extent in their \(r\)th dimensions.

The mask array must have the same shape as the source array (see section 4.4).

**Alignment restrictions:** If defined, the \(r\)th subscript array must be aligned to the source array, with replicated alignment in all dimensions except \(r\). The mask array *must be aligned with* the source array (see section 4.5).

**Accessibility restrictions:** The source, subscript, mask arrays, and destination array must all be accessible (see section 4.6).

**Argument lifetime:** The \(\text{dst}\) and \(\text{src}\) arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule. (There are no such requirements for the subscript and mask arrays).

### 5.12.2 Method

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

Arguments are the base addresses for local segments of the source and destination arrays. They should point to vectors of the locally held elements.

**Effect:** The effect is like executing the conditional assignment

\[
\text{if(mask } [x_0, \ldots, x_{S-1}])
\]
\[
\text{dst } [\text{subs}_0 \ [x_0], \ldots, \text{subs}_{R-1} \ [x_{R-1}]] = \text{src } [x_0, \ldots, x_{R-1}]
\]

for every tuple, \((x_0, \ldots, x_{R-1})\), of global subscripts of the source array. These assignments are executed in an undefined order. Here \(\text{subs}_r\) denote the \(r\)th subscript array. If \(\text{subs} \ [r]\) is null, the value of the expression \(\text{subs}_r \ [x_r]\) in the left-hand-side of the assignment is defined to be \(x_r\).
**Replicated data:** If the source array has replicated mapping, the value for a particular element is taken from one of its copies. If the destination array has replicated mapping, identical values are broadcast to every copy of each element.

**Overlap restrictions:** In-place updates are not allowed. The pairs *dst*, dstDat and *src*, srcDat must define non-overlapping arrays.
5.13 class ScatterComb

[There are existing schedules for combining scatter, but their interface is under revision.]

5.14 class Reshape

[There are existing schedules for this F90 intrinsic, but their interface is under revision.]
5.15 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

```cpp
class DAD ;

template<class T>
class Sum {
public :
    Sum(const DAD* src) ;

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

### 5.15.1 Constructor

*Sum(const DAD* src)*

The source array is described by the *DAD* *src*. It will have elements of type *T*.

**Type restrictions:** *T* should be a POD type (see section 5.1.2). 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 standard arithmetic types of C++ satisfy these requirements for *T*.

**Accessibility restrictions:** The source array must be accessible (see section 4.6).

**Argument lifetime:** The *src* argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

### 5.15.2 Method

*execute(T* res, T* srcDat)*

The arguments are the address to which the result should be written and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.
**Effect:** 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.

**Replicated data:** If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of “nearest” is implementation dependent.
5.16 class SumMask

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 `SumMask`. The public interface of the `SumMask` class is

```cpp
class DAD;

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

    void execute(T* res, T* srcDat, int* mskDat);

private:
    ...
};
```

5.16.1 Constructor

`SumMask(const DAD* src, const DAD* msk)`

The source array is described by the DAD *src. It will have elements of type T. The mask array is described by the DAD *msk.

**Type restrictions:** T should be a POD type (see section 5.1.2). 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 standard arithmetic types of C++ satisfy these requirements for T.

**Shape restrictions:** The mask array must have the same shape as the source array (see section 4.4).

**Alignment restrictions:** The mask array must be aligned with the source array (see section 4.5).

**Accessibility restrictions:** The source array and the mask array must be accessible (see section 4.6).

**Argument lifetime:** The src and msk arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.
5.16.2 Method

\textbf{execute(T* res, T* srcDat, int* mskDat)}

The arguments are the address to which the result should be written and the base addresses of the local segments of the source and mask arrays. The latter should point to vectors of the locally held elements.

\textbf{Effect:} Executing the schedule adds together all elements of the array for which the corresponding element of the mask array is non-zero. The addition is performed in an unspecified order. The result value, written to \texttt{*res}, is broadcast to all members of the active process group.

\textbf{Replicated data:} If the source or mask array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of “nearest” is implementation dependent.
5.17 class Product

A product schedule is a communication schedule for multiplying together all elements of a distributed array (the source array). A product schedule is described by a collective object with local components of class Product. The public interface of the Product class is

class DAD ;

template<class T>
class Product {
public :
    Product(const DAD* src) ;
    
    void execute(T* res, T* srcDat) ;
private :
    ...
} ;

5.17.1 Constructor

Product(const DAD* src)

The source array is described by the DAD *src. It will have elements of type T.

Type restrictions: T should be a POD type (see section 5.1.2). A conventional binary multiplication operator, *, must be defined for objects of type T, the assignment operator * must operate in a compatible way, and the symbol 1 must be convertible to type T, yielding a unit of the multiplication operation.

All standard arithmetic types of C++ satisfy these requirements for T.

Accessibility restrictions: The source array must be accessible (see section 4.6).

Argument lifetime: The src argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.17.2 Method

execute(T* res, T* srcDat)

The arguments are the address to which the result should be written and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.
**Effect:** Executing the schedule multiplies 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.

**Replicated data:** If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of “nearest” is implementation dependent.
5.18 class ProductMsk

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

```cpp
class DAD;

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

    void execute(T* res, T* srcDat, int* mskDat);
private:
    ...}
};
```

5.18.1 Constructor

`ProductMsk(const DAD* src, const DAD* msk)`

The source array is described by the `DAD* src`. It will have elements of type `T`. The mask array is described by the `DAD* msk`.

**Type restrictions:** `T` should be a POD type (see section 5.1.2). A conventional binary multiplication operator, `*`, must be defined for objects of type `T`, the assignment operator `*=` must operate in a compatible way, and the symbol `1` must be convertible to type `T`, yielding a unit of the multiplication operation.

All standard arithmetic types of C++ satisfy these requirements for `T`.

**Shape restrictions:** The mask array must have the same shape as the source array (see section 4.4).

**Alignment restrictions:** The mask array must be aligned with the source array (see section 4.5).

**Accessibility restrictions:** The source array and the mask array must be accessible. (see section 4.6).

**Argument lifetime:** The `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.
5.18.2 Method

\texttt{execute(T* res, T* srcDat, int* mskDat)}

The arguments are the address to which the result should be written and the base addresses of the local segments of the source and mask arrays. The latter should point to vectors of the locally held elements.

\textbf{Effect:} Executing the schedule multiplies together all elements of the array for which the corresponding element of the mask array is non-zero. The multiplication is performed in an unspecified order. The result value, written to \texttt{*res}, is broadcast to all members of the active process group.

\textbf{Replicated data:} If the source or mask array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of “nearest” is implementation dependent.
5.19 class Maxval

A maximum value schedule is a communication schedule for finding the largest element of a distributed array (the source array). A maximum value schedule is described by a collective object with local components of class Maxval. The public interface of the Maxval class is

```cpp
class DAD;

template<class T>
class Maxval {
    public:
        Maxval(const DAD* src);
        void execute(T* res, T* srcDat);
    private:
        ...
};
```

5.19.1 Constructor

Maxval(const DAD* src)

The source array is described by the DAD *src. It will have elements of type T.

**Type restrictions**: T should be a POD type. Conventional binary comparison operators, > and <, must be defined for objects of type T, and a function mostneg must be overloaded with the entry point

```cpp
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.

**Accessibility restrictions**: The source array must be accessible (see section 4.6).

**Argument lifetime**: The src argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.19.2 Method

execute(T* res, T* srcDat)

The arguments are the address to which the result should be written and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.
**Effect:** Executing the schedule finds the largest element of the array. The maximum value, written to *res*, is broadcast to all members of the active process group.

**Replicated data:** If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of “nearest” is implementation dependent.
5.20 class MaxvalMsk

A masked maximum value schedule is a communication schedule for finding the largest element of a distributed array (the source array) under the control of a mask array. A masked maximum value schedule is described by a collective object with local components of class MaxvalMsk. The public interface of the MaxvalMsk class is

    class DAD;

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

        void execute(T* res, T* srcDat, int* maskDat);
    private:
        ...
};

5.20.1 Constructor

MaxvalMsk(const DAD* src, const DAD* msk)

The source array is described by the DAD *src. It will have elements of type T. The mask array is described by the DAD *msk.

Type restrictions: T should be a POD type. Conventional binary comparison 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.

Shape restrictions: The mask array must have the same shape as the source array (see section 4.4).

Alignment restrictions: The mask array must be aligned with the source array (see section 4.5).

Accessibility restrictions: The source array and the mask array must be accessible. (see section 4.6).

Argument lifetime: The src and msk arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.
5.20.2 Method

execute(T* res, T* srcDat, int* mskDat)

The arguments are the address to which the result should be written and the base addresses of the local segments of the source and mask arrays. The latter should point to vectors of the locally held elements.

Effect: Executing the schedule finds the largest element of the array for which the corresponding element of the mask array is non-zero. The maximum value, written to *res, is broadcast to all members of the active process group.

Replicated data: If the source or mask array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of “nearest” is implementation dependent.
5.21 class Minval

A minimum value schedule is a communication schedule for finding the smallest element of a distributed array (the source array). A minimum value schedule is described by a collective object with local components of class Minval. The public interface of the Minval class is

```cpp
class DAD;

template<class T>
class Minval {
  public:
    Minval(const DAD* src);

    void execute(T* res, T* srcDat);

  private:
    ...

};
```

5.21.1 Constructor

Minval(const DAD* src)

The source array is described by the DAD *src. It will have elements of type T.

**Type restrictions:** T should be a POD type. Conventional binary comparison operators, > and <, must be defined for objects of type T, and a function `mostpos` must be overloaded with the entry point

```cpp
void mostpos(T* res);
```

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

**Accessibility restrictions:** The source array must be accessible (see section 4.6).

**Argument lifetime:** The src argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.21.2 Method

execute(T* res, T* srcDat)

The arguments are the address to which the result should be written and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.
**Effect:** Executing the schedule finds the smallest element of the array. The minimum value, written to `res`, is broadcast to all members of the active process group.

**Replicated data:** If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of “nearest” is implementation dependent.
5.22 class MinvalMsk

A masked minimum schedule is a communication schedule for finding the smallest element of a distributed array (the source array) under the control of a mask array. A masked minimum schedule is described by a collective object with local components of class MinvalMsk. The public interface of the MinvalMsk class is

```cpp
class MinvalMsk {
  public:
    MinvalMsk(const DAD* src, const DAD* msk);  
    void execute(T* res, T* srcDat, int* mskDat);  
  private:
    ...  
};
```

5.22.1 Constructor

MinvalMsk(const DAD* src, const DAD* msk)

The source array is described by the DAD *src. It will have elements of type T. The mask array is described by the DAD *msk.

**Type restrictions:** T should be a POD type. Conventional binary comparison operators, > and <, must be defined for objects of type T, and a function mostpos must be overloaded with the entry point

```cpp
void mostpos(T* res);  
```

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

**Shape restrictions:** The mask array must have the same shape as the source array (see section 4.4).

**Alignment restrictions:** The mask array must be aligned with the source array (see section 4.5).

**Accessibility restrictions:** The source array and the mask array must be accessible. (see section 4.6).

**Argument lifetime:** The src and msk arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.
5.22.2 Method

`execute(T* res, T* srcDat, int* mskDat)`

The arguments are the address to which the result should be written and the base addresses of the local segments of the source and mask arrays. The latter should point to vectors of the locally held elements.

Effect: Executing the schedule finds the smallest element of the array for which the corresponding element of the mask array is non-zero. The minimum value, written to *res, is broadcast to all members of the active process group.

Replicated data: If the source or mask array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of “nearest” is implementation dependent.
5.23 class All

A all schedule is a communication schedule for computing the logical conjunction of the elements of a distributed array of boolean values (the source array). An all schedule is described by a collective object with local components of class All. The public interface of the All class is

```cpp
class DAD;

class All {
  public:
    All(const DAD* src);

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

5.23.1 Constructor

All(const DAD* src)

The source array is described by the DAD *src. It will have elements of type int.

Accessibility restrictions: The source array must be accessible (see section 4.6).

Argument lifetime: The src argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.23.2 Method

execute(int* res, int* srcDat)

The arguments are the address to which the result should be written and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.

Effect: Executing the schedule forms the logical conjunction (boolean and) of the elements of the array. The result value, written to *res, is broadcast to all members of the active process group.

Replicated data: If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.
5.24 class Any

A *any schedule* is a communication schedule for computing the logical disjunction of the elements of a distributed array of boolean values (the source array). An any schedule is described by a collective object with local components of class *Any*. The public interface of the *Any* class is

```cpp
class DAD;

class Any {
public:
    Any(const DAD* src);
    
    void execute(int* res, int* srcDat);
private:
    ...
};
```

5.24.1 Constructor

*Any(const DAD* src)*

The source array is described by the DAD *src*. It will have elements of type int.

**Accessibility restrictions:** The source array must be accessible (see section 4.6).

**Argument lifetime:** The src argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.24.2 Method

*execute(int* res, int* srcDat)*

The arguments are the address to which the result should be written and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.

**Effect:** Executing the schedule forms the logical disjunction (boolean or) of the elements of the array. The result value, written to *res*, is broadcast to all members of the active process group.

**Replicated data:** If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of “nearest” is implementation dependent.
5.25 class Count

A count schedule is a communication schedule for counting the number of true
elements in a distributed array of boolean values (the source array). A count
schedule is described by a collective object with local components of class Count.
The public interface of the Count class is

class DAD ;

class Count {
  public :
    Count(const DAD* src) ;

    void execute(int* res, int* srcDat) ;
  private :
    ...
} ;

5.25.1 Constructor

Count(const DAD* src)

The source array is described by the DAD *src. It will have elements of type
int.

Accessibility restrictions:  The source array must be accessible (see section
4.6).

Argument lifetime:  The src argument is stored in the schedule as a refer-
ence. The associated object must persist for the lifetime of the constructed
schedule.

5.25.2 Method

execute(int* res, int* srcDat)

The arguments are the address to which the result should be written and the
base address of the local segment of the source array. The latter should point
to a vector of the locally held elements.

Effect:  Executing the schedule returns the number of true (non-zero) elements
of the array. The result value, written to *res, is broadcast to all members of
the active process group.

Replicated data:  If the source array has replicated mapping, values for par-
ticular elements are generally taken from the nearest copy. The definition of
"nearest" is implementation dependent.
5.26 class DotProduct

A *dot product schedule* is a communication schedule for computing the dot product of two distributed arrays (the source arrays). A dot product schedule is described by a collective object with local components of class *DotProduct*. The public interface of the *DotProduct* class is

```cpp
class DAD ;

template<class S, class T, class U>
class DotProduct {
    public :
        DotProduct(const DAD* src1, const DAD* src2) ;

        void execute(S* res, T* src1Dat, U* src2Dat) ;
    private :
        ... ;
};
```

5.26.1 Constructor

*DotProduct*(const DAD* src1, const DAD* src2)

The source arrays is described by the DADs *src1* and *src2*. They will have elements of type *T* and *U* respectively. The result will have type *S*.

**Type restrictions:** *S* should be a POD type (see section 5.1.2). A conventional binary multiplication operator, *, must be defined to multiply objects of type *T* with objects of type *U*. The assignment operator *=* must be defined on objects of type *S*. This operator must be able to accept an RHS operand of type *S*, and must also be able to accept an operand produced by multiplying together *T* and *U* (if this is has type different to *S*). The symbol 0 must be convertible to type *S*, yielding a unit of the addition operation.

All standard arithmetic types of C++ satisfy these requirements, provided the product of a *T* with a *U* is convertible to an *S*.

**Shape restrictions:** The two source arrays must have the same shape (see section 4.4).

**Alignment restrictions:** The source arrays must be aligned with one another (see section 4.5).

**Accessibility restrictions:** The source arrays must be accessible (see section 4.6).

**Argument lifetime:** The *src1* and *src2* arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.
5.26.2 Method

```c
void execute(S* res, T* src1Dat, U* src2Dat) ;
```

The arguments are the address to which the result should be written and the base addresses of the local segments of the source arrays. The latter should point to vectors of the locally held elements.

**Effect:** Executing the schedule multiplies together corresponding elements of the source arrays, in pairs, then adds together all pairwise products. The addition occurs in an unspecified order. The result value, written to *res, is broadcast to all members of the active process group.

**Replicated data:** If the source arrays have replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.
5.27  class BoolDotProduct

A boolean dot product schedule is a communication schedule for computing the boolean dot product of two distributed arrays of boolean values (the source arrays). A boolean dot product schedule is described by a collective object with local components of class BoolDotProduct. The public interface of the BoolDotProduct class is

class DAD ;
class BoolDotProduct {
  public :
    BoolDotProduct(const DAD* src1, const DAD* src2);

    void execute(int* res, int* src1Dat, int* src2Dat);
  private :
    ...
} ;

5.27.1 Constructor

BoolDotProduct(const DAD* src1, const DAD* src2)

The source arrays is described by the DADs *src1 and *src2.

Shape restrictions: The two source arrays must have the same shape (see section 4.4).

Alignment restrictions: The source arrays must be aligned with one another (see section 4.5).

Accessibility restrictions: The source arrays must be accessible (see section 4.6).

Argument lifetime: The src1 and src2 arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.27.2 Method

void execute(int* res, int* src1Dat, int* src2Dat) ;

The arguments are the address to which the result should be written and the base addresses of the local segments of the source arrays. The latter should point to vectors of the locally held elements.
**Effect:** Executing the schedule computes logical and of corresponding elements of the source arrays, in pairs, then computes logical or of all the pairwise conjunctions. The result value, written to `*res`, is broadcast to all members of the active process group.

**Replicated data:** If the source arrays have replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of “nearest” is implementation dependent.
5.28 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

```c
class DAD;

template<class T>
class Maxloc {
    public:
        Maxloc(const DAD* src);

        void execute(T* val, int pos[], T* srcDat);

    private:
        ...
};
```

5.28.1 Constructor

Maxloc(const DAD* src)

The source array is described by the DAD *src*. It will have elements of type T.

**Type restrictions:** T should be a POD type. Conventional binary comparison operators, > and <, must be defined for objects of type T, and a function *mostneg* must be overloaded with the entry point

```c
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.

**Accessibility restrictions:** The source array must be accessible. (see section 4.6).

**Argument lifetime:** The src argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.28.2 Method

execute(T* val, int pos [], T* srcDat)

The arguments are the addresses to which the result value and location should be written, and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.

The size of the pos vector must be equal to the rank of the source array.
**Effect:** The value of largest element in the array is written to *val*. The global subscripts of the first occurrence of this element are written to the vector *pos*. If the maximum value occurs more than once in the array, “first occurrence” 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.

**Replicated data:** If the source array has replicated mapping, values for a particular element are generally taken from the nearest copy. The definition of “nearest” is implementation dependent.
5.29 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
MaxlocMsk class is

```cpp
class DAD;

template<class T>
class MaxlocMsk {
public:
    Maxloc(const DAD* src, const DAD* msk);

    void execute(T* val, int pos [], T* srcDat, int* mskDat);
private:
    ...
};
```

5.29.1 Constructor

MaxlocMsk(const DAD* src, const DAD* msk)

The source array is described by the DAD *src. It will have elements of type
T. The mask array is described by the DAD *msk.

**Type restrictions:** T should be a POD type. Conventional binary compar-
ison operators, > and <, must be defined for objects of type T, and a function
mostneg must be overloaded with the entry point

```cpp
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.

**Shape restrictions:** The mask array must have the same shape as the source
array (see section 4.4).

**Alignment restrictions:** The mask array must be aligned with the source
array (see section 4.5).

**Accessibility restrictions:** The source array and the mask array must be
accessible. (see section 4.6).

**Argument lifetime:** The src and msk arguments are stored in the sched-
ule as references. The associated objects must persist for the lifetime of the
constructed schedule.

111
5.29.2 Method

`execute(T* val, int pos [], T* srcDat, int* mskDat)`

The arguments are the addresses to which the result value and location should be written, and the base addresses of the local segments of the source and mask arrays. The latter should point to vectors of the locally held elements.

The size of the `pos` vector must be equal to the rank of the source array.

**Effect:** 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 element are written to the vector `pos`. If the maximum value occurs more than once in the unmasked part of the array, “first occurrence” 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.

**Replicated data:** If the source or mask array has replicated mapping, values for a particular element are generally taken from the nearest copy. The definition of “nearest” is implementation dependent.
5.30 class Minloc

A *minimum location schedule* is a communication schedule for finding the location of the smallest element of a distributed array. A minimum location schedule is described by a collective object with local components of class Minloc. The public interface of the Minloc class is

```cpp
class DAD;
template<class T>
class Minloc {
    public:
        Minloc(const DAD* src);
        void execute(T* val, int pos[], T* srcDat);
    private:
        ...
};
```

5.30.1 Constructor

Minloc(const DAD* src)

The source array is described by the DAD *src*. It will have elements of type T.

Type restrictions: T should be a POD type. Conventional binary comparison operators, > and <, must be defined for objects of type T, and a function mostpos must be overloaded with the entry point

```cpp
void mostpos(T* res);
```

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

Accessibility restrictions: The source array must be accessible. (see section 4.6).

Argument lifetime: The src argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.30.2 Method

execute(T* val, int pos[], T* srcDat)

The arguments are the addresses to which the result value and location should be written, and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements. The size of the pos vector must be equal to the rank of the source array.
**Effect:** The value of smallest element in the array is written to *val*. The global subscripts of the first occurrence of this element are written to the vector *pos*. If the minimum value occurs more than once in the array, “first occurrence” 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.

**Replicated data:** If the source array has replicated mapping, values for a particular element are generally taken from the nearest copy. The definition of “nearest” is implementation dependent.
5.31  class MinlocMsk

A masked minimum location schedule is a communication schedule for finding the location of the smallest element of a distributed array, under the control of a mask array. A masked minimum location schedule is described by a collective object with local components of class MinlocMsk. The public interface of the MinlocMsk class is

```cpp
class DAD;

template<class T>
class MinlocMsk {
    public:
        Minloc(const DAD* src, const DAD* msk);

        void execute(T* val, int pos[], T* srcDat, int* mskDat);
    private:
        ...
};
```

5.31.1  Constructor

MinlocMsk(const DAD* src, const DAD* msk)

The source array is described by the DAD *src. It will have elements of type T. The mask array is described by the DAD *msk.

**Type restrictions:** T should be a POD type. Conventional binary comparison operators, > and <, must be defined for objects of type T, and a function `mostpos` must be overloaded with the entry point

```cpp
void mostpos(T* res);
```

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

**Shape restrictions:** The mask array must have the same shape as the source array (see section 4.4).

**Alignment restrictions:** The mask array must be aligned with the source array (see section 4.5).

**Accessibility restrictions:** The source array and the mask array must be accessible. (see section 4.6).

**Argument lifetime:** The `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.
5.31.2 Method

\texttt{execute(T* val, int pos [], T* srcDat, int* mskDat)}

The arguments are the addresses to which the result value and location should be written, and the base addresses of the local segments of the source and mask arrays. The latter should point to vectors of the locally held elements.

The size of the \texttt{pos} vector must be equal to the rank of the source array.

\textbf{Effect: } The value of smallest element in the array for which the corresponding element of the mask array is non-zero is written to \texttt{*val}. The global subscripts of the first occurrence of this element are written to the vector \texttt{pos}. If the minimum value occurs more than once in the unmasked part of the array, “first occurrence” is defined by ordering the set of global subscripts with first subscript \textit{least significant} (Fortran-like array-element ordering).

The result values written to \texttt{*val} and \texttt{pos} are broadcast to all members of the active process group.

\textbf{Replicated data: } If the source or mask array has replicated mapping, values for a particular element are generally taken from the nearest copy. The definition of “nearest” is implementation dependent.
5.32 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

```c++
class DAD;

template<class T>
class SumDim {
  public:
    SumDim(const DAD* res, const DAD* src, const int dim);
    
    void execute(T* resDat, T* srcDat);
  private:
    ...
};
```

5.32.1 Constructor

**SumDim(const DAD* res, const DAD* src, const int dim)**

The source array is described by the **DAD* src** and the result array is described by the **DAD* res**. They will both have elements of type **T**. The reduction occurs in dimension **dim**.

**Value restrictions:** The value of **dim** must be in the range **0, ..., R - 1** where **R** is the rank of the source array.

**Type restrictions:** **T** should be a POD type (see section 5.1.2). 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 standard arithmetic types of C++ satisfy these requirements for **T**.

**Shape restrictions:** The shape of the result array must be obtained from the shape of the source array by deleting dimension **dim** (see section 4.4).

**Alignment restrictions:** The result array must be aligned to the source array, with replicated alignment in dimension **dim** (see section 4.5).

**Accessibility restrictions:** The source array and the result array must be accessible (see section 4.6).
Argument lifetime: The res and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.32.2 Method
execute(T* resDat, T* srcDat)

Arguments are the base addresses for local segments of the result and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

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

is

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

where \( N \) is the extent of the source array in dimension \( \text{dim} \). The sums are performed in an unspecified order.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension \( \text{dim} \) of the source array.

Replicated data: If the source array has replicated mapping, values for particular contributions to the sums are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pairs *res, resDat and *src, srcDat must define non-overlapping arrays.
5.33 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

```cpp
class DAD;

template<class T>
class SumDimMsk {
public:
    SumDimMsk(const DAD* res, const DAD* src, const int dim,
               const DAD* msk);

    void execute(T* resDat, T* srcDat, int* mskDat);
private:
    ...
};
```

5.33.1 Constructor

SumDimMsk(const DAD* res, const DAD* src, const int dim, 
           const DAD* msk)

The source array is described by the DAD *src and the result array is described by the DAD *res. They will both have elements of type T. The reduction occurs in dimension dim. The mask array is described by the DAD *msk.

**Value restrictions:** The value of dim must be in the range 0,...,R−1 where R is the rank of the source array.

**Type restrictions:** T should be a POD type (see section 5.1.2). 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 standard arithmetic types of C++ satisfy these requirements for T.

**Shape restrictions:** The mask array must be the same shape as the source array. The shape of the result array must be obtained from the shape of the source array by deleting dimension dim (see section 4.4).

**Alignment restrictions:** The mask array must be aligned with the source array. The result array must be aligned to the source array, *with replicated alignment in dimension* dim (see section 4.5).
**Accessibility restrictions:** The source and mask arrays and the result array must be accessible (see section 4.6).

**Argument lifetime:** The `res`, `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

### 5.33.2 Method

**execute(T* resDat, T* srcDat, int* mskDat)**

Arguments are the base addresses for local segments of the result, source and mask arrays. They should point to vectors of the locally held elements.

**Effect:** On exit, the value of

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

is

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

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

where \(N\) is the extent of the source array in dimension \(\text{dim}\). The sum is performed in an unspecified order.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension \(\text{dim}\) of the source array.

**Replicated data:** If the source array has replicated mapping, values for particular contributions to the sums are generally taken from the nearest copy of the source element. The definition of “nearest” is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

**Overlap restrictions:** In-place updates are not allowed. The pair `*res, resDat` must define an array that has no overlap with the arrays defined by either of the pairs `*src, srcDat` and `*msk, mskDat`
5.34 class ProductDim

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

A dimension product schedule is described by a collective object with local components of class ProductDim. The public interface of the ProductDim class is

class DAD;

template<class T>
class ProductDim {
    public:
        ProductDim(const DAD* res, const DAD* src, const int dim);
        void execute(T* resDat, T* srcDat);
    private:
        ...
};

5.34.1 Constructor

ProductDim(const DAD* res, const DAD* src, const int dim)

The source array is described by the DAD*src and the result array is described by the DAD*res. They will both have elements of type T. The reduction occurs in dimension dim.

Value restrictions: The value of dim must be in the range 0,...,R−1 where R is the rank of the source array.

Type restrictions: T should be a POD type (see section 5.1.2). A conventional binary multiplication operator, *, must be defined for objects of type T, the assignment operator *= must operate in a compatible way, and the symbol 1 must be convertible to type T, yielding a unit of the multiplication operation.

All standard arithmetic types of C++ satisfy these requirements for T.

Shape restrictions: The shape of the result array must be obtained from the shape of the source array by deleting dimension dim (see section 4.4).

Alignment restrictions: The result array must be aligned to the source array, with replicated alignment in dimension dim (see section 4.5).

Accessibility restrictions: The source array and the result array must be accessible (see section 4.6).
Argument lifetime: The res and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.34.2 Method
execute(T* resDat, T* srcDat)

Arguments are the base addresses for local segments of the result and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of
res \([x_0, \ldots, x_{\text{dim}-1}, x_{\text{dim}+1}, \ldots, x_{R-1}]\)

is
\[
\prod_{x_{\text{dim}}=0}^{N-1} \text{src} \left[ x_0, \ldots, x_{\text{dim}-1}, x_{\text{dim}}, x_{\text{dim}+1}, \ldots, x_{R-1} \right]
\]

where \(N\) is the extent of the source array in dimension \(\text{dim}\). The product is computed in an unspecified order.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension \(\text{dim}\) of the source array.

Replicated data: If the source array has replicated mapping, values for particular contributions to the product are generally taken from the nearest copy of the source element. The definition of “nearest” is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pairs *res, resDat and *src, srcDat must define non-overlapping arrays.
5.35 class ProductDimMsk

A masked dimension product schedule is a communication schedule for multiplying together, 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 product schedule is described by a collective object with local components of class ProductDimMsk. The public interface of the ProductDimMsk class is

```cpp
class ProductDimMsk {
public:
  ProductDimMsk(const DAD* res, const DAD* src, const int dim,
                 const DAD* msk);

  void execute(T* resDat, T* srcDat, int* mskDat);
private:
  ...
};
```

5.35.1 Constructor

`ProductDimMsk(const DAD* res, const DAD* src, const int dim,
                 const DAD* msk)`

The source array is described by the DAD *src and the result array is described by the DAD *res. They will both have elements of type T. The reduction occurs in dimension dim. The mask array is described by the DAD *msk.

**Value restrictions:** The value of dim must be in the range 0, \ldots, R - 1 where R is the rank of the source array.

**Type restrictions:** T should be a POD type (see section 5.1.2). A conventional binary multiplication operator, *, must be defined for objects of type T, the assignment operator *= must operate in a compatible way, and the symbol 1 must be convertible to type T, yielding a unit of the multiplication operation. All standard arithmetic types of C++ satisfy these requirements for T.

**Shape restrictions:** The mask array must be the same shape as the source array. The shape of the result array must be obtained from the shape of the source array by deleting dimension dim (see section 4.4).

**Alignment restrictions:** The mask array must be aligned with the source array. The result array must be aligned to the source array, with replicated alignment in dimension dim (see section 4.5).
Accessibility restrictions: The source and mask arrays and the result array must be accessible (see section 4.6).

Argument lifetime: The res, src and msk arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.35.2 Method
execute(T* resDat, T* srcDat, int* mskDat)

Arguments are the base addresses for local segments of the result, source and mask arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

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

is

\[
\prod_{x_{\text{dim}} = 0}^{N-1} \text{src} [x_0, \ldots, x_{\text{dim}-1}, x_{\text{dim}}, x_{\text{dim}+1}, \ldots, x_{R-1}]
\]

where \( N \) is the extent of the source array in dimension \( \text{dim} \). The product is computed in an unspecified order.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension \( \text{dim} \) of the source array.

Replicated data: If the source array has replicated mapping, values for particular contributions to the product are generally taken from the nearest copy of the source element. The definition of “nearest” is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pair *res, resDat must define an array that has no overlap with the arrays defined by either of the pairs *src, srcDat and *msk, mskDat
5.36  class MaxvalDim

A *dimension maximum value schedule* is a communication schedule for finding
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 value schedule is described by a collective object with
local components of class MaxvalDim. The public interface of the MaxvalDim
class is

```cpp
class DAD;

template<class T>
class MaxvalDim {
public:
    MaxvalDim(const DAD* res, const DAD* src, const int dim);

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

5.36.1  Constructor

MaxvalDim(const DAD* res, const DAD* src, const int dim)

The source array is described by the DAD *src and the result array is described
by the DAD *res. They will both have elements of type T. The reduction occurs
in dimension dim.

**Value restrictions:** The value of dim must be in the range 0, ..., R – 1 where
R is the rank of the source array.

**Type restrictions:** T should be a POD type. Conventional binary compar-
sion operators, > and <, must be defined for objects of type T, and a function
`mostneg` must be overloaded with the entry point

```cpp
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.

**Shape restrictions:** The shape of the result array must be obtained from the
shape of the source array by deleting dimension dim (see section 4.4).

**Alignment restrictions:** The result array must be aligned to the source
array, *with replicated alignment in dimension* dim (see section 4.5).

125
**Accessibility restrictions:** The source array and the result array must be accessible (see section 4.6).

**Argument lifetime:** The res and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

### 5.36.2 Method

```cpp
execute(T* resDat, T* srcDat)
```

Arguments are the base addresses for local segments of the result and source arrays. They should point to vectors of the locally held elements.

**Effect:** On exit, the value of

- `res [x_0, ..., x_{dim-1}, x_{dim+1}, ..., x_{N-1}]`

is the maximum value of

- `src [x_0, ..., x_{dim-1}, x_{dim}, x_{dim+1}, ..., x_{N-1}]`

across allowed values of `x_{dim}`.

As implied by the replicated alignment of the result array, the results are broadcast in the process dimension associated with dimension `dim` of the source array.

**Replicated data:** If the source array has replicated mapping, values of particular elements for comparison are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

**Overlap restrictions:** In-place updates are not allowed. The pairs `*res, resDat` and `*src, srcDat` must define non-overlapping arrays.
5.37 class MaxvalDimMsk

A *masked dimension maximum value schedule* is a communication schedule for finding, under the control of mask, 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 value schedule is described by a collective object with local components of class MaxvalDimMsk. The public interface of the MaxvalDimMsk class is

```cpp
class DAD;

template<class T>
class MaxvalDimMsk {
public:
    MaxvalDimMsk(const DAD* res, const DAD* src, const int dim,
                 const DAD* msk);

    void execute(T* resDat, T* srcDat, int* mskDat);
private:
    ...
};
```

5.37.1 Constructor

MaxvalDimMsk(const DAD* res, const DAD* src, const int dim,
              const DAD* msk)

The source array is described by the DAD *src and the result array is described by the DAD *res. They will both have elements of type T. The reduction occurs in dimension dim. The mask array is described by the DAD *msk.

**Value restrictions:** The value of dim must be in the range 0, ..., R−1 where R is the rank of the source array.

**Type restrictions:** T should be a POD type. Conventional binary comparison operators, > and <, must be defined for objects of type T, and a function mostneg must be overloaded with the entry point

```cpp
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.

**Shape restrictions:** The mask array must be the same shape as the source array. The shape of the result array must be obtained from the shape of the source array by deleting dimension dim (see section 4.4).
Alignment restrictions: The mask array must be aligned with the source array. The result array must be aligned to the source array, with replicated alignment in dimension \( \text{dim} \) (see section 4.5).

Accessibility restrictions: The source and mask arrays and the result array must be accessible (see section 4.6).

Argument lifetime: The \( \text{res} \), \( \text{src} \) and \( \text{msk} \) arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.37.2 Method

\[
\text{execute}(T \ast \text{resDat}, T \ast \text{srcDat}, \text{int} \ast \text{mskDat})
\]

Arguments are the base addresses for local segments of the result, source and mask arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

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

is the maximum value of

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

for which

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

over the allowed values of \( x_{\text{dim}} \).

As implied by the replicated alignment of the result array, the results are broadcast in the process dimension associated with dimension \( \text{dim} \) of the source array.

Replicated data: If the source array has replicated mapping, values for particular elements for comparison are generally taken from the nearest copy of the source element. The definition of “nearest” is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pair \( *\text{res}, \text{resDat} \) must define an array that has no overlap with the arrays defined by either of the pairs \( *\text{src}, \text{srcDat} \) and \( *\text{msk}, \text{mskDat} \).
5.38 class MinvalDim

A dimension minimum value schedule is a communication schedule for finding the smallest elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension minimum value schedule is described by a collective object with local components of class MinvalDim. The public interface of the MinvalDim class is

```cpp
class DAD;

template<class T>
class MinvalDim {
  public:
    MinvalDim(const DAD* res, const DAD* src, const int dim);
    void execute(T* resDat, T* srcDat);
  private :
    ...;
};
```

5.38.1 Constructor

MinvalDim(const DAD* res, const DAD* src, const int dim)

The source array is described by the DAD *src and the result array is described by the DAD *res. They will both have elements of type T. The reduction occurs in dimension dim.

Value restrictions: The value of dim must be in the range 0, ..., R - 1 where R is the rank of the source array.

Type restrictions: T should be a POD type. Conventional binary comparison operators, > and <, must be defined for objects of type T, and a function mostpos must be overloaded with the entry point

```cpp
void mostpos(T* res);
```

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

Shape restrictions: The shape of the result array must be obtained from the shape of the source array by deleting dimension dim (see section 4.4).

Alignment restrictions: The result array must be aligned to the source array, with replicated alignment in dimension dim (see section 4.5).
**Accessibility restrictions:** The source array and the result array must be accessible (see section 4.6).

**Argument lifetime:** The `res` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

### 5.38.2 Method

```cpp
evaluate(T* resDat, T* srcDat)
```

Arguments are the base addresses for local segments of the result and source arrays. They should point to vectors of the locally held elements.

**Effect:** On exit, the value of

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

is the minimum value of

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

across allowed values of `x_{\text{dim}}`.

As implied by the replicated alignment of the result array, the results are broadcast in the process dimension associated with dimension `dim` of the source array.

**Replicated data:** If the source array has replicated mapping, values of particular elements for comparison are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

**Overlap restrictions:** In-place updates are not allowed. The pairs `*res`, `resDat` and `*src`, `srcDat` must define non-overlapping arrays.
5.39 class MinvalDimMsk

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

A masked dimension minimum value schedule is described by a collective object with local components of class MinvalDimMsk. The public interface of the MinvalDimMsk class is

```cpp
class DAD;

template<class T>
class MinvalDimMsk {
  public:
    MinvalDimMsk(const DAD* res, const DAD* src, const int dim,
                  const DAD* msk);

    void execute(T* resDat, T* srcDat, int* mskDat);
  private:
    ...
};
```

5.39.1 Constructor

MinvalDimMsk(const DAD* res, const DAD* src, const int dim,
              const DAD* msk)

The source array is described by the DAD *src and the result array is described by the DAD *res. They will both have elements of type T. The reduction occurs in dimension dim. The mask array is described by the DAD *msk.

Value restrictions: The value of dim must be in the range 0, ..., R−1 where R is the rank of the source array.

Type restrictions: T should be a POD type. Conventional binary comparison operators, > and <, must be defined for objects of type T, and a function mostpos must be overloaded with the entry point

```cpp
void mostpos(T* res);
```

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

Shape restrictions: The mask array must be the same shape as the source array. The shape of the result array must be obtained from the shape of the source array by deleting dimension dim (see section 4.4).
Alignment restrictions: The mask array must be aligned with the source array. The result array must be aligned to the source array, with replicated alignment in dimension dim (see section 4.5).

Accessibility restrictions: The source and mask arrays and the result array must be accessible (see section 4.6).

Argument lifetime: The res, src and mask arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.39.2 Method
execute(T* resDat, T* srcDat, int* maskDat)

Arguments are the base addresses for local segments of the result, source and mask arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of
\[
\text{res}[x_0, \ldots, x_{\text{dim}-1}, x_{\text{dim}+1}, \ldots, x_{R-1}]
\]
is the minimum value of
\[
\text{src}[x_0, \ldots, x_{\text{dim}-1}, x_{\text{dim}}, x_{\text{dim}+1}, \ldots, x_{R-1}]
\]
for which
\[
\text{mask}[x_0, \ldots, x_{\text{dim}-1}, x_{\text{dim}}, x_{\text{dim}+1}, \ldots, x_{R-1}] \neq 0
\]
over the allowed values of \(x_{\text{dim}}\).

As implied by the replicated alignment of the result array, the results are broadcast in the process dimension associated with dimension dim of the source array.

Replicated data: If the source array has replicated mapping, values for particular elements for comparison are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pair *res, resDat must define an array that has no overlap with the arrays defined by either of the pairs *src, srcDat and *mask, maskDat
5.40  class AllDim

A *dimension all schedule* is a communication schedule for computing the logical conjunction of the elements of a distributed array of boolean along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension all schedule is described by a collective object with local components of class AllDim. The public interface of the AllDim class is

```cpp
class DAD;

class AllDim {
  public:
    AllDim(const DAD* res, const DAD* src, const int dim);

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

5.40.1  Constructor

AllDim(const DAD* res, const DAD* src, const int dim)

The source array is described by the DAD *src and the result array is described by the DAD *res. They will both have elements of type int. The reduction occurs in dimension dim.

**Value restrictions:** The value of dim must be in the range 0, ..., R−1 where R is the rank of the source array.

**Shape restrictions:** The shape of the result array must be obtained from the shape of the source array by deleting dimension dim (see section 4.4).

**Alignment restrictions:** The result array must be aligned to the source array, with replicated alignment in dimension dim (see section 4.5).

**Accessibility restrictions:** The source array and the result array must be accessible (see section 4.6).

**Argument lifetime:** The res and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.
5.40.2 Method

\texttt{execute(int* resDat, int* srcDat)}

Arguments are the base addresses for local segments of the result and source arrays. They should point to vectors of the locally held elements.

**Effect:** On exit, the value of
\[
\text{res} [x_0, \ldots, x_{\text{dim}-1}, x_{\text{dim}+1}, \ldots, x_{N-1}]
\]
is true if
\[
\text{src} [x_0, \ldots, x_{\text{dim}-1}, x_{\text{dim}}, x_{\text{dim}+1}, \ldots, x_{N-1}]
\]
is true (non-zero) for all allowed values of \(x_{\text{dim}}\).

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension \(\text{dim}\) of the source array.

**Replicated data:** If the source array has replicated mapping, values for particular contributions to the conjunctions are generally taken from the nearest copy of the source element. The definition of “nearest” is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

**Overlap restrictions:** In-place updates are not allowed. The pairs \(*\text{res}, \text{resDat}\) and \(*\text{src}, \text{srcDat}\) must define non-overlapping arrays.
5.41 class AnyDim

A dimension any schedule is a communication schedule for computing the logical disjunction of the elements of a distributed array of boolean values along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension any schedule is described by a collective object with local components of class AnyDim. The public interface of the AnyDim class is

class DAD;
class AnyDim {
public:
    AnyDim(const DAD* res, const DAD* src, const int dim);

    void execute(int* resDat, int* srcDat);
private:
    ...
};

5.41.1 Constructor

AnyDim(const DAD* res, const DAD* src, const int dim)

The source array is described by the DAD* src and the result array is described by the DAD* res. They will both have elements of type int. The reduction occurs in dimension dim.

Value restrictions: The value of dim must be in the range 0, ..., R - 1 where R is the rank of the source array.

Shape restrictions: The shape of the result array must be obtained from the shape of the source array by deleting dimension dim (see section 4.4).

Alignment restrictions: The result array must be aligned to the source array, with replicated alignment in dimension dim (see section 4.5).

Accessibility restrictions: The source array and the result array must be accessible (see section 4.6).

Argument lifetime: The res and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.
5.41.2 Method

\texttt{execute(int* resDat, int* srcDat)}

Arguments are the base addresses for local segments of the result and source arrays. They should point to vectors of the locally held elements.

\textbf{Effect:} On exit, the value of
\[
\text{res} \ [x_0, \ldots, x_{\text{dim}-1}, x_{\text{dim}+1}, \ldots, x_{N-1}]
\]
is true if
\[
\text{src} \ [x_0, \ldots, x_{\text{dim}-1}, x_{\text{dim}}, x_{\text{dim}+1}, \ldots, x_{N-1}]
\]
is true (non-zero) for any allowed values of \( x_{\text{dim}} \).

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension \texttt{dim} of the source array.

\textbf{Replicated data:} If the source array has replicated mapping, values for particular contributions to the disjunctions are generally taken from the nearest copy of the source element. The definition of “nearest” is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

\textbf{Overlap restrictions:} In-place updates are not allowed. The pairs \texttt{*res, resDat} and \texttt{*src, srcDat} must define non-overlapping arrays.
5.42 class CountDim

A dimension count schedule is a communication schedule for counting the number of true elements of a distributed array of boolean values along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension count schedule is described by a collective object with local components of class CountDim. The public interface of the CountDim class is

class DAD ;

class CountDim {
public:
    CountDim(const DAD* res, const DAD* src, const int dim) ;

    void execute(int* resDat, int* srcDat) ;
private :
    ...
};

5.42.1 Constructor

CountDim(const DAD* res, const DAD* src, const int dim)

The source array is described by the DAD *src and the result array is described by the DAD *res. They will both have elements of type int. The reduction occurs in dimension dim.

Value restrictions: The value of dim must be in the range $0, \ldots, R - 1$ where $R$ is the rank of the source array.

Shape restrictions: The shape of the result array must be obtained from the shape of the source array by deleting dimension dim (see section 4.4).

Alignment restrictions: The result array must be aligned to the source array, with replicated alignment in dimension dim (see section 4.5).

Accessibility restrictions: The source array and the result array must be accessible (see section 4.6).

Argument lifetime: The res and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.
5.42.2 Method

execute(int* resDat, int* srcDat)

Arguments are the base addresses for local segments of the result and source arrays. They should point to vectors of the locally held elements.

**Effect:** On exit, the value of

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

is the number of true (non-zero) elements

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

over allowed values of \(x_{\text{dim}}\).

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension \(\text{dim}\) of the source array.

**Replicated data:** If the source array has replicated mapping, values for particular contributions to the count are generally taken from the nearest copy of the source element. The definition of “nearest” is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

**Overlap restrictions:** In-place updates are not allowed. The pairs \(*\text{res}\), \text{resDat}\) and \(*\text{src}\), \text{srcDat}\) must define non-overlapping arrays.
5.43 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

```cpp
class DAD;

template<class T>
class MaxlocDim {
  public:
    MaxlocDim(const DAD* val, const DAD* pos,
               const DAD* src, const int dim);

    void execute(T* valDat, int* posDat, T* srcDat);
  private:
    ...
};
```

5.43.1 Constructor

MaxlocDim(const DAD* val, const DAD* pos,
          const DAD* src, const int dim)

The source array is described by the DAD *src and the array of maximum values is described by the DAD *val. They will both have elements of type T. The array of maximum locations is described by the DAD *pos. The search occurs in dimension dim.

**Value restrictions:** The value of dim must be in the range 0, ..., R – 1 where R is the rank of the source array.

**Type restrictions:** T should be a POD type. Conventional binary comparison operators, > and <, must be defined for objects of type T, and a function mostneg must be overloaded with the entry point

```cpp
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.

**Shape restrictions:** The shape of the two result arrays (*val and *pos) must be obtained from the shape of the source array by deleting dimension dim (see section 4.4).
Alignment restrictions: The two result arrays (*val and *pos) must be aligned to the source array, with replicated alignment in dimension dim (see section 4.5).

Accessibility restrictions: The source array and the result arrays must be accessible (see section 4.6).

Argument lifetime: The val, loc and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.43.2 Method

execute(T* valDat, int* posDat, T* srcDat)

Arguments are the base addresses for local segments of the result value, result position, and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

\[ val[\ldots, x_{\text{dim}-1}, x_{\text{dim}+1}, \ldots, x_{R-1}] \]

is the maximum value of

\[ src[\ldots, x_{\text{dim}-1}, x_{\text{dim}}, x_{\text{dim}+1}, \ldots, x_{R-1}] \]

across allowed values of \( x_{\text{dim}} \). The value of

\[ pos[\ldots, x_{\text{dim}-1}, x_{\text{dim}+1}, \ldots, x_{R-1}] \]

is the smallest \( x_{\text{dim}} \) value at which this maximum occurs.

As implied by the replicated alignment of the result arrays, the results are broadcast in the process dimension associated with dimension dim of the source array.

Replicated data: If the source array has replicated mapping, values of particular elements for comparison are generally taken from the nearest copy of the source element. The definition of “nearest” is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. Neither of the arrays defined by the pairs *val, valDat or *pos, posDat may have any overlap with the array defined by the pair *src, srcDat.
5.44 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

```cpp
class DAD;

template<class T>
class MaxlocDimMsk {
public:
    MaxlocDimMsk(const DAD* val, const DAD* pos,
                   const DAD* src, const int dim, const DAD* msk);

    void execute(T* valDat, int* posDat, T* srcDat, int* mskDat);

private:
    ...
};
```

5.44.1 Constructor

```cpp
MaxlocDimMsk(const DAD* val, const DAD* pos,
              const DAD* src, const int dim,
              const DAD* msk)
```

The source array is described by the DAD *src and the array of maximum values is described by the DAD *val. They will both have elements of type T. The array of maximum locations is described by the DAD *pos. The search occurs in dimension dim. The mask array is described by the DAD *msk.

**Value restrictions:** The value of dim must be in the range 0,...,R - 1 where R is the rank of the source array.

**Type restrictions:** T should be a POD type. Conventional binary comparison operators, > and <, must be defined for objects of type T, and a function `mostneg` must be overloaded with the entry point

```cpp
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.
**Shape restrictions:** The shape of the two result arrays (*val and *pos) must be obtained from the shape of the source array by deleting dimension \( \text{dim} \). The mask array must be the same shape as the source array (see section 4.4).

**Alignment restrictions:** The two result arrays (*val and *pos) must be aligned to the source array, with replicated alignment in dimension \( \text{dim} \). The mask array must be aligned with the source array (see section 4.5).

**Accessibility restrictions:** The source and mask arrays and the result arrays must be accessible (see section 4.6).

**Argument lifetime:** The val, loc, src and msk arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

### 5.44.2 Method

\[
\text{execute}(\text{T* valDat, int* posDat, T* srcDat, int* mskDat})
\]

Arguments are the base addresses for local segments of the result value, result position, source, and mask arrays. They should point to vectors of the locally held elements.

**Effect:** On exit, the value of

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

is the maximum value of

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

for which

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

over the allowed values of \( x_{\text{dim}} \). The value of

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

is the smallest \( x_{\text{dim}} \) value at which this maximum occurs.

As implied by the replicated alignment of the result arrays, the results are broadcast in the process dimension associated with dimension \( \text{dim} \) of the source array.

**Replicated data:** If the source array has replicated mapping, values for particular elements for comparison are generally taken from the nearest copy of the source element. The definition of “nearest” is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.
Overlap restrictions: In-place updates are not allowed. Neither of the arrays defined by the pairs *val, valDat or *pos, posDat may have any overlap with either of the arrays defined by the pairs *src, srcDat and *msk, mskDat
5.45 class MinlocDim

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

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

```cpp
class DAD;

template<class T>
class MinlocDim {
  public:
    MinlocDim(const DAD* val, const DAD* pos,
               const DAD* src, const int dim);

    void execute(T* valDat, int* posDat, T* srcDat);
  private:
    ...
  };
```

5.45.1 Constructor

`MinlocDim(const DAD* val, const DAD* pos,
           const DAD* src, const int dim)`

The source array is described by the DAD *src and the array of minimum values is described by the DAD *val. They will both have elements of type T. The array of minimum locations is described by the DAD *pos. The search occurs in dimension dim.

**Value restrictions:** The value of dim must be in the range 0, . . . , $R - 1$ where $R$ is the rank of the source array.

**Type restrictions:** T should be a POD type. Conventional binary comparison operators, $>$ and $<$, must be defined for objects of type T, and a function mostpos must be overloaded with the entry point

```cpp
void mostpos(T* res);
```

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

**Shape restrictions:** The shape of the two result arrays (*val and *pos) must be must be obtained from the shape of the source array by deleting dimension dim (see section 4.4).
Alignment restrictions: The two result arrays (*val and *pos) must be aligned to the source array, with replicated alignment in dimension dim (see section 4.5).

Accessibility restrictions: The source array and the result arrays must be accessible (see section 4.6).

Argument lifetime: The val, loc and src arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.45.2 Method
execute(T* valDat, int* posDat, T* srcDat)

Arguments are the base addresses for local segments of the result value, result position, and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

val [x0, ..., xdim-1, xdim+1, ..., xN-1]

is the minimum value of

src [x0, ..., xdim-1, xdim, xdim+1, ..., xN-1]

across allowed values of xdim. The value of

pos [x0, ..., xdim-1, xdim+1, ..., xN-1]

is the smallest xdim value at which this minimum occurs.

As implied by the replicated alignment of the result arrays, the results are broadcast in the process dimension associated with dimension dim of the source array.

Replicated data: If the source array has replicated mapping, values of particular elements for comparison are generally taken from the nearest copy of the source element. The definition of “nearest” is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. Neither of the arrays defined by the pairs *val, valDat or *pos, posDat may have any overlap with the array defined by the pair *src, srcDat.
5.46  class MinlocDimMs

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

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

    class DAD;
    template<class T>
    class MinlocDimMs {
    public:
      MinlocDimMs(const DAD* val, const DAD* pos,
                   const DAD* src, const int dim, const DAD* msk) ;

      void execute(T* valDat, int* posDat, T* srcDat, int* mskDat) ;
    private :
      ...  
    } ;

5.46.1  Constructor

MinlocDimMs(const DAD* val, const DAD* pos,  
             const DAD* src, const int dim,  
             const DAD* msk)

The source array is described by the DAD *src and the array of minimum values is described by the DAD *val. They will both have elements of type T. The array of minimum locations is described by the DAD *pos. The search occurs in dimension dim. The mask array is described by the DAD *msk.

**Value restrictions:** The value of dim must be in the range 0,\ldots,R-1 where R is the rank of the source array.

**Type restrictions:** T should be a POD type. Conventional binary comparison operators, > and <, must be defined for objects of type T, and a function mostpos must be overloaded with the entry point

    void mostpos(T* res) ;

which sets *res to the largest allowed value of type T. The Adlib header files define mostpos for all the standard arithmetic types of C++, so these types satisfy all the requirements for T.
**Shape restrictions:** The shape of the two result arrays (*val and *pos) must be obtained from the shape of the source array by deleting dimension *dim*. The mask array must be the same shape as the source array (see section 4.4).

**Alignment restrictions:** 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 aligned with the source array (see section 4.5).

**Accessibility restrictions:** The source and mask arrays and the result arrays must be accessible (see section 4.6).

**Argument lifetime:** The val, loc, src and msk arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

### 5.46.2 Method

execute(T* valDat, int* posDat, T* srcDat, int* mskDat)

Arguments are the base addresses for local segments of the result value, result position, source, and mask arrays. They should point to vectors of the locally held elements.

**Effect:** On exit, the value of

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

is the minimum value of

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

for which

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

over the allowed values of *x*<sub>dim</sub>. The value of

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

is the smallest *x*<sub>dim</sub> value at which this minimum occurs.

As implied by the replicated alignment of the result arrays, the results are broadcast in the process dimension associated with dimension *dim* of the source array.

**Replicated data:** If the source array has replicated mapping, values for particular elements for comparison are generally taken from the nearest copy of the source element. The definition of “nearest” is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.
Overlap restrictions: In-place updates are not allowed. Neither of the arrays defined by the pairs *val, valDat or *pos, posDat may have any overlap with either of the arrays defined by the pairs *src, srcDat and *msk, mskDat
Chapter 6

The ad++ interface
Chapter 7

Distributed loops

In this chapter we discuss various ways to use the Adlib run-time technology in translation of distributed loops—loops whose ranges is partitioned across the active process group. Typically such loops are used to access and modify the data in distributed arrays. For definiteness, we work in the context of the ad++ interface. The techniques can be adapted to other interfaces to the kernel library.

In ad++, the general overall construct is a distributed, parallel loop. It is parametrized by an Index object which maintains local loop state. If x is a range the overall construct has the syntax.

    Index i(x);
    OVERALL(i) {
        ...
    } ALLOVER(i);

If x has extent N, this construct can be compared to the sequential loop

    int i;
    for(i = 0 ; i < N ; i++) {
        ...
    }

The difference is that in the overall construct the N instances of the body of the loop will be partitioned across the set of active processes, following the mapping of x.

The Index class is a subclass of Location. Within an overall construct parametrized by an Index i, the Subscript component of i is set to the local subscript for the current iteration. So i can be used as an array subscript, as in

    Array1<float> c(x);
    Index i(x);
    OVERALL(i) {
        c(i) = ...
    } ALLOVER(i);
The general overall construct has an effect on the active process group as described in section 2.6. If a construct parametrized by \( i \) appears in the context of an active process group \( p \), the body of the construct executes in the context of an active process group \( p / i \) (recall that Index is a subclass of Location, so this expression is well-formed.) The parent range of \( i \) must be distributed over a dimension of \( p \).

Combining these features, we can give a more complete example

```c
Array2<float> a(x, y);
Array1<float> b(y);
...
Index i(x), j(y);
OVERALL(i) {
    OVERALL(j) {
        a(i, j) = 2 * b(j) + i;
    } ALLOVER(j);
} ALLOVER(i);
```

To each element of \( a \), this assigns an expression computed from the aligned value of \( b \) and the global subscript in \( x \). The latter is obtained from \( i \) by applying the overloaded conversion operator \( \text{int}() \). All data accesses through legal subscripting operations are local. If a non-local array element was required, it would take a specific call to a member of the communication library to access it.

The remainder of this chapter discusses several schemes for translating the distributed loop. The first scheme uses the Adlib Index class directly. The “translation” is the trivial one, using only the C macro preprocessor to replace the OVERALL and ALLOVER “keywords”. The second scheme uses another auxiliary (iterator) class from the library—LocBlocksIndex. The translation is still relatively straightforward and has the advantage of being independent of the level of the parametric range. The LocBlocksIndex mechanism is quite efficient, and is used extensively in the implementation of the Adlib communication library. Finally we describe a scheme which works directly in terms of the members of Range class, without introducing any auxiliary iterator class.
7.1 class Index

An iterator class, maintaining loop state for simple enumerations of the elements of a range.

The Index class is derived from the LocBlocksIndex class (see section 7.2) which is in turn derived from Location. These classes are used extensively in Adlib code that has to deal with ranges whose level is not known in advance. Their use can be avoided if the level of the ranges is known at compile time, using code transformations described in section 7.3.

The public interface of the Index class is

class Index : public LocBlocksIndex {
    public :
        Index(Range x) ;

        void begin() ;
        void next() ;

        operator int() { return idx ; }
    } ;

7.1.1 Constructor

Index(Range x)

Create an index object for enumerating the elements of range x.

7.1.2 Methods

begin()

Used to implement the general overall construct—a distributed loop. Begins an enumeration of the elements of the range mapped to the local process. On exit from this member, the Location component of the index represents the first element of the local segment of the range\(^1\).

If i is the loop index, the idiom for the loop is

for(i.begin() ; i.test() ; i.next())

The member test() is inherited from LocBlocksIndex (see section 7.2).

next()

Move to next element in enumeration.

\(^1\)Here “first element” simply means first in the enumeration. It does not necessarily mean the local element with the smallest global subscript.
7.1.3 Operators

operator int()

Returns the global subscript value for the current iteration.

7.1.4 Translation of overall construct by macro expansion

The most naïve translation of the overall construct is by using the standard C preprocessor to expand the macros

```c
#define OVERALL(i) for(apgStack.push(&apg), apg.restrict(i.dim), \ 
     i.begin() ; i.test() ; i.next())

#define ALLOVER(i) apgStack.pop(&apg)
```

Effectively, the loop

```c
OVERALL(i) {
...
} ALLOVER(i) ;
```

becomes

```c
apgStack.push(&apg) ;
apg.restrict(i.dim) ;
for(i.begin() ; i.test() ; i.next()) {
...
}
apgStack.pop(&apg) ;
```

We moved the apg manipulations (see section 2.6) outside the for construct to improve readability. The begin(), test() and next() members of Index are used in the for loop to enumerate the local elements of the range.

This translation is trivial to implement, but it is very inefficient. First we have the overhead of the calls to the iterator members of Index in every iteration. Secondly (probably even more seriously) every array or range subscripting operation in the body of the loop involves calls to member functions. The overhead of all these library calls can easily downgrade performance by an order of magnitude or more relative to a comparable sequential for loop.

There are two related classes Index2 and Index3, with constructors

```c
Index2 :: Index2(Range x, const int count, const int base) ;

Index3 :: Index3(Range x,
                 const int count, const int base, const int stride) ;
```

Essentially the constructed index objects are equivalent to

```c
Index(x.subrng(count, base))

Index(x.subrng(count, base, stride))
```
respectively. The only qualification relates to the global index returned by the int conversion: for Index2 and Index3 this index is relative to the range x, not the subrange.
7.2 class LocBlocksIndex

An iterator class, maintaining loop state for enumerations of the locally held blocks of a range.

The public interface of the LocBlocksIndex class is

class LocBlocksIndex : public Location, public Block {
public:
    LocBlocksIndex(Range _x) ;

    void beginLocBlk() ;
    void nextLocBlk() ;

    int test() ;
};

7.2.1 Constructor

LocBlocksIndex(Range x)

Create an index object for enumerating the local blocks of range x.

7.2.2 Methods

beginLocBlk()

Beginning an enumeration of the blocks of the range mapped to the local process. On exit from this member, the Block component of the Index contains the parameters of the first (non-empty) block of the locally held segment of the range, and the Location component represents the first element (smallest global subscript) in that block. “First block” simply means first in the enumeration—it does not guarantee that the block contains smaller subscripts than later blocks.

If i is the loop index, the idiom for the loop is

    for(i.beginLocBlk() ; i.test() ; i.nextLocBlk())

nextLocBlk()

Move to next (non-empty) block in the enumeration, updating the Block and Location components. Usage is illustrated in the idiomatic example above.

test()

After an initial call to beginLocBlock and zero or more subsequent calls to nextLocBlock, result of this member is non-zero if there are still blocks waiting to be enumerated. Usage is illustrated in the idiomatic example above.
7.2.3 Translation of overall construct using LocBlocksIndex

The class LocBlocksIndex is a superclass of Index that provides members to enumerate local blocks of a range, rather than individual local elements. Library functions return base addresses of the array sections associated with these blocks. Elements within the block are then enumerated with a simple, efficient for loop, computing offsets from base addresses using linear expressions. The performance-critical inner loops can be compiled with high efficiency. If the block size is large enough, most of the cost of the library calls is amortized.

LocBlocksIndex is a subclass of Location and of Block. The outer level in the translation of an overall construct now looks something like

LocBlocksIndex i(x);
apgStack.push(&apg);
apg.restrict(i.dim);
for(i.beginLocBlk() ; i.test() ; i.nextLocBlk()) {
    ... deal with block 'i'
}
apgStack.pop(&apg);

In this translation scheme we have an outer loop enumerating the locally held blocks of the range. In general this loop is needed because Adlib supports higher-level distribution formats like block-cyclic. These allow multiple blocks of a single range to reside on the same processor. In a context where it is known in advance that the local process holds a single block (for example, if the range involved has level 1), the translation scheme given in the next section allows further optimizations.

In general code to “deal with block i”, takes the form

    ... precompute some bases and increments for block

    for(int l = 0 ; l < i.count ; l++) {
        ...
    }

Before filling the details of this code, we need to put the source code into a normalized form. The body of an overall construct parametrized by an index i may use i in several contexts:

1. As a subscript in a local subscripting operation.

2. In an expression such as (int) i used to obtain the global subscript of the current iteration relative to the index range.

3. As a scalar subscript in a section subscripting operation.

4. In a group restriction operation of the form p / i, where p is some group.
As described in chapter 4, subscripting operations on arrays can be replaced with lower-level operations on the `Map` and `Group` classes. For example, if `a` is an array of `float`, the reference

```plaintext
... a(i, j) ...
```
can be replaced with

```plaintext
float* a_dat = a.dat();
...
... a_dat[a.map(0).offset(i) + a.map(1).offset(j)] ...
```

Of course the inquiries `dat()` and `map()` can be lifted outside any loop. Similarly, the section construction

```plaintext
... a.sect(i, y) ...
```
can be replaced with

```plaintext
float* a_dat = a.dat();
...
... Section1<float>(y, a.grp() / i,
    a.map(1), a_dat + a.map(0).offset(i)) ...
```

By applying these transformations, the set of uses of `i` inside the loop can be reduced to four cases

1. As an argument of `Map :: offset`.
2. In `(int) i`, yielding the global subscript.
3. In a group restriction operation of the form `p / i`.

We will deal with these cases in turn. [Replace following enumeration and associated figures with straight descriptive text.]

**Offset computation.** Within a particular block the expression `u.offset(i)` can be rewritten as a linear function of the inner induction variable. The base for this function is given by the value of `offset` for the first `Location` of the range, which is the `Location` component of `i`. Its increment is returned by the member `Map :: step`. Figure 7.1 illustrates the translation. The call to `offset` has been removed from the inner loop. A good compiler will generate very efficient code for the linear expression that replaces it.

**Global subscript computation** is slightly simpler. The base and increment for this expression are already contained in the `Block` component of `i`, computed by its iterator members. Figure 7.2 illustrates the translation.

**Group restriction** is translated trivially, as illustrated in figure 7.3.

Now, consider this example from the preamble to this chapter:
SOURCE:

    Range x ;
    Map u ;

    Index i(x) ;
    OVERALL(i) {
        ... u.offset(i) ...
    } ALLOVER(i) ;

OUTPUT:

    Range x ;
    Map u ;

    LocBlocksIndex i(x) ;

    apgStack.push(&apg);
    apg.restrict(i, dim);
    for(i.beginLocBlk() ; i.test() ; i.nextLocBlk()) {
        int u_bas = u.offset(i);
        int u_stp = u.stp() * i.sub_stp;

        for(int l = 0 ; l < i.count() ; l++) {
            ... u_bas + u_stp * l ...
        }
    }
    apgStack.pop(&apg);

Figure 7.1: Translation of offset computation.
Array2<

float> a(x, y);
Array1<

float> b(y);
...
Index i(x), j(y);
OVERALL(i) {
  OVERALL(j) {
    a(i, j) = 2 * b(j) + i;
  } ALLOVER(j);
} ALLOVER(i);

It can be normalized to the form

Array2<

float> a(x, y);
Array1<

float> b(y);
...
float* a_dat = a.dat();
float* b_dat = b.dat();

Index i(x), j(y);
OVERALL(i) {
  OVERALL(j) {
    a_dat[a.map(0).offset(i) + a.map(1).offset(j)] =
    b_dat[b.map(0).offset(j)] + (int) i;
  } ALLOVER(j);
} ALLOVER(i);

A translation of the loop nest is given in figure 7.4.

Note that in this example the manipulations of apg could have been omitted,
because there are no collective operations inside the loop that depend on the
state of apg.

As a final straightforward optimization, when we have perfectly nested overall
constructs, the loop nesting can be changed to put all intra-block loops
innermost. In that case the inner loops become

for (int l = 0; l < i.count; l++) {
  for (int m = 0; m < j.count; m++) {
    a_dat[a_off0_bas + a_off0_stp * l +
      a_off1_bas + a_off1_stp * m] =
    b_dat[b_off0_bas + b_off0_stp * m] +
      i_glob_bas + i.glob_stp * l;
  }
}

All subscript expressions are linear in the loop induction variables, and we expect
very good code generation from these loops.

The translation scheme described in this section is summarized in figure 7.5.
SOURCE:

Range x;

Index i(x);
OVERALL(i) {
... (int) i ...
} ALLOVER(i);

OUTPUT:

Range x;

LocBlocksIndex i(x);

apgStack.push($apg);
apg.restrict(i.dim);
for(i.beginLocBlk(); i.test(); i.nextLocBlk()) {
for(int l = 0; l < i.count; l++) {
... i.glb_bas + i.glb_stp * l ...
}
}
apgStack.pop($apg);

Figure 7.2: Translation of global subscript computation.
SOURCE:

Range x;

Index i(x);
OVERALL(i) {
    ... p / i ...
} ALLOVER(i);

OUTPUT:

Range x;
LocBlocksIndex i(x);

apgStack.push(&apg);
apg.restrict(i.dim);
for(i.beginLocBlk() ; i.test() ; i.nextLocBlk()) {
    for(int l = 0 ; l < i.count ; l++) {
        ... p / i ...
    }
}
apgStack.pop(&apg);

Figure 7.3: Translation for computation of group restriction.

LocBlocksIndex i(x), j(y);
apgStack.push(&apg);
apg.restrict(i.dim);
for(i.beginLocBlk() ; i.test() ; i.nextLocBlk()) {
    int a_off0_base = a->map(i).offset(i);
    int a_off0stp = a->map(i).str() * i.substp;

    for(int l = 0 ; l < i.count ; l++) {
        apgStack.push(&apg);
apg.restrict(j.dim);
        for(j.beginLocBlk() ; j.test() ; j.nextLocBlk()) {
            int a_off1_base = a->map(j).offset(j);
            int a_off1stp = a->map(j).str() * j.substp;

            int b_off0_base = b->map(i).offset(j);
            int b_off0stp = b->map(j).str() * j.substp;

            for(int m = 0 ; m < j.count ; m++) {
                a->data[a_off0_base + a_off0stp + l +
                         a_off1_base + a_off1stp + m] =
                b->data[b_off0_base + b_off0stp + m] +
                i->glob_base + i->glob_stp + l;
            }
        }
apgStack.pop(&apg);
    }
}
apgStack.pop(&apg);

Figure 7.4: Translation of example.
SOURCE:

Range x;
Map u;
Group p;

Index i(x);
OVERALL(i) {
    ... u.offset(i) ...
    (int) i ...
    p / i ...
} ALLOVER(i);

OUTPUT:

Range x;
Map u;
Group p;

LocBlocksIndex i(x);

apgStack.push(&apg);
apg.restrict(i.dim);
for(i.beginLocBlk(); i.test(); i.nextLocBlk()) {
    int u_bas = u.offset(i);
    int u_stp = u.stp() * i.sub_stp;

    for(int l = 0; l < i.count; l++) {
        ... u_bas + u_stp * l ...
        i.glb_bas + i.glb_stp * l ...
        p / i ...
    }
} apgStack.pop(&apg);

Figure 7.5: Summary of LocBlocksIndex-based translation scheme for overall construct.
7.3 Translation using the kernel range

Internally, all the iterator classes depend on the block member of the kernel range. In this section we show how to use this member directly for translation of overall constructs. The scheme given here effectively inlines the iterator members used in the previous translations. It also inlines the offset member of the Map class, in terms of lower level disp and step members.

To apply the translation scheme described in this section we need some compile-time knowledge about the level of the range involved. The scheme works recursively by expanding a overall construct in the source program in terms of a overall construct for a range one level lower. The procedure can (if desired) be continued recursively. After one or two stages we get down to a overall construct for a level-0 range. This base case has a different, simpler translation. For this scheme to be effective we must at least know in advance if the original parametric range has level 0 (ie, it already represents the base case). Ideally we should know the exact level, as a compile-time constant.

Before applying the translation proper, the input program should be normalized as discussed in the previous section. Rather than discuss translation of the three uses of Index individually, we will go straight to the combined summary form, in the style of figure 7.5. For a parametric range of level greater than zero, the translation summary is given in figure 7.6.

The outer loop is a overall construct parametrized by x.ker(). The block member of x initialize variables describing the block selected by the current value of the kernel subscript. These variables correspond exactly with the fields of the Block component of LocBlocksIndex (and, of course, this is exactly how they are computed in the implementation of that class).

The offset operation is expanded in terms of disp and step operations, and an offset for a kernel range. If necessary the transformation can be applied recursively to eliminate the offset function altogether.

The rest of the translation closely follows that of the previous section.

If the parametric range has level 0 (it is a process dimension, or a subrange of a process dimension) the summary is given in figure 7.7.

One stage of recursion applied to the example of the previous section gives the translation in figure 7.8. If ranges x and y both have level 1 application of the rule for translating level 0 constructs then gives the translation in figure 7.9. The function inRange can be defined by

```cpp
inline int inRange(const int val,
    const int count, const int base, const int stride) {
    const int offset = val - base ;
    const int pos     = offset / stride ;
    return pos >= 0 && pos < count && pos * stride == offset ;
}
```

It tests whether a particular value is in a triplet-defined interval. Finally, figure 7.10 gives an optimized form on the assumption that the original ranges were level 1. We can remove the local conditional because every active process must contain an element of the kernel, and replace the value i1 initialized by local
SOURCE:

Range x;
Map u;
Group p;

Index i(x);
OVERALL(i) {
    ... u.offset(i) ...
    (int) i ... 
    p / i ...
} ALLOVER(i);

OUTPUT:

Range x;
Map u;
Group p;

int crd_count, crd_bas, crd_stp;
x.crdx(&crd_count, &crd_bas, &crd_stp);

Index3 i1(x, ker(), crd_count, crd_bas, crd_stp);
OVERALL(i1) {
    Block i;
    x.block((int) i1, &i);

    int uBas = u.str() * i.sub_bas + u.ker().offset(i1);
    int uStp = u.str() * i.sub_stp;

    for(int l = 0; l < i_count; l++) {
        ... uBas + uStp * l ... 
        i.glB_bas + i.glb_stp * l ... 
        p / i1 ...
    }
} ALLOVER(i1);

Figure 7.6: Summary of recursive translation scheme for overall construct with level greater than 0.
SOURCE:

Range x;
Map u;
Group p;

Index i(x);
OVERALL(i) {
    ... u.offset(i) ...
    (int) i ...
    p / i ...
} ALLOVER(i);

OUTPUT:

Range x;
Map u;
Group p;

Dimension d = x.dim();

apgStack.push(&apg);
apg.restrict(d);

const int crd = d.crd();
const int str = x.str();
const int offset = crd - x.bas();
const int pos = offset / str;
if(pos >= 0 & pos < x.size() & pos * str == offset) {
    ... 0 ...
    pos ...
    p.restrict(d) ...
}
apgStack.pop(&apg);

Figure 7.7: Summary of translation for overall construct of level 0.
with \texttt{d.crd()}. Three other optimizations do not depend on the assumption of simplicity: additions of 0 are constant-folded away, \texttt{x.ker().dim()} is replaced with \texttt{x.dim()}, and the apg manipulations are removed, because the loop body contains no calls to collective operations.

Because level 1 ranges are an important case, figure 7.11 summarizes the translation of the overall construct for level 1 parametric ranges.
Indent \il{s.ker()}, \jl{y.ker()};
OVERALL\il{i} \{ 
  Block \il{i}; 
  s.block((int) \il{i}, \&i) ;
  
  int a_off0_base = a.map0().str() * i.sub_base + a.map0().ker().offset(\il{i}) ;
  int a_off0_stop = a.map0().str() * i.sub_stop ;
  
  for(int l = 0 ; l < i.count ; l++) { 
    OVERALL\jl{j} \{ 
      Block \jl{j} ; 
      y.block((int) \jl{j}, \&j) ;
      
      int a_off1_base = a.map1().str() * j.sub_base + a.map0().ker().offset(\jl{j}) ;
      int a_off1_stop = a.map1().str() * j.sub_stop ;
      
      int b_off0_base = b.map0().str() * i.sub_base + b.map0().ker().offset(\il{i}) ;
      int b_off0_stop = b.map0().str() * i.sub_stop ;
      
      for(int m = 0 ; m < j.count ; m++) { 
        a.data[a_off0_base + a_off1_base + m] =
        a.data[a_off0_base + a_off1_base + m] +
        b.data[b_off0_base + b_off0_base + m] +
        i.glob_base + i.glob_stop + l ;
      }
    }
  }
\} OVERALL\il{i} ;
\}
\} OVERALL\il{i} ;

Figure 7.8: Translation of example. Pass 1, assuming ranges \( x \) and \( y \) have level greater than zero.
Figure 7.9: Translation of example. Pass 2, assuming ranges x and y have level 1, so their kernels are level 0.
\begin{verbatim}
Dimension d = x.dim();

int crd_count, crd_bas, crd_mtp;
x.crd(&crd_count, &crd_bas, &crd_mtp);

comet int crd = d.crd();
if(isRange(crd, crd_count, crd_bas, crd_mtp)) {
  Block i;
  x.block(crd, &i);
  int a_off0_bas = a.map(i).str() * i.mub_bas + 0;
  int a_off0_mtp = a.map(i).str() * i.mub_mtp;
  Dimension a = y.ker().dim();
  int crd_count, crd_bas, crd_mtp;
y.crd(&crd_count, &crd_bas, &crd_mtp);
  comet int crd = d.crd();
  if(isRange(crd, crd_count, crd_bas, crd_mtp)) {
    Block j;
    y.block(crd, &j);
    int a_off1_bas = a.map(j).str() * j.mub_bas + 0;
    int a_off1_mtp = a.map(j).str() * j.mub_mtp;
    int b_off0_bas = b.map(i).str() * i.mub_bas + 0;
    int b_off0_mtp = b.map(i).str() * i.mub_mtp;
    for(int l = 0; l < i.count; l++) {
      for(int m = 0; m < j.count; m++) {
        a.dat[a_off0_bas + a_off0_mtp * l +
              a_off1_bas + a_off1_mtp * m] =
        b.dat[b_off0_bas + b_off0_mtp * l +
              b_off1_bas + b_off1_mtp * m] +
        i.glb_bas + i.glb_mtp * l;
      }
    }
  }
}
\end{verbatim}

Figure 7.10: Translation of example. Optimizations assuming ranges \( x \) and \( y \)
are level 1.
**SOURCE:**

```c
Range x ;
Map u ;
Group p ;

Index i(x) ;
OVERALL(i) {
    ... u.offset(i) ... 
    (int) i ... 
    p / i ... 
} ALLOVER(i) ;
```

**OUTPUT:**

```c
Range x ;
Map u ;
Group p ;

Dimension d = x.dim() ;
apgStack.push(&apg) ;
apg.restrict(d) ;

int crd_count, crdBas, crdstp ;
x.crds(&crd_count, &crdBas, &crdstp) ;

const int crd = d.crd() ;
if(inRange(crd, crd_count, crdBas, crdstp)) {
    Block i ;
    x.block(crd, &i) ;

    int uBas = u.str() * i.subBas ;
    int ustp = u.str() * i.substp ;

    for(int l = 0 ; l < i.count ; l++) {
        ... uBas + ustp * l ... 
        i.globBas + i.globstp * l ... 
        p.restrict(d) ... 
    }
}
apgStack.pop(&apg) ;
```

Figure 7.11: Summary of translation for *overall* construct with level 1 x.
7.4 Access to ghost regions

If an array appearing in a overall construct has ghost regions, the subscripts of the array may be shifted by some amount without violating the restriction that all array element accesses are local. In ad++ this can be expressed by an idiom like

```cpp
Array1<float> b(x), c(z);

Index i(x);
OVERALL(i) {
    b(i) = c(i + 1) + c(i - 1);
} ALLOVER(i);
```

This assumes that the array c has ghost regions of extent one or more on both sides of its “physical” segments. The particular usage illustrated here is very useful in “stencil” computations. The addition and subtraction operators must be overloaded on the Index class to support this usage.

For definiteness, the translation scheme in figure 7.12 uses the blockIndex-based style of translation, but the same techniques can be carried over directly to the recursive style.

A subtle point to note in the definition of the overloaded + and - operators is that the shift amount is measured relative to the underlying template range. This is slightly inconvenient if the array range has a non-trivial alignment stride. Because this is assumed to be unusual in typical stencil updates, and because alternatives that automatically take account of the alignment stride add complexity, the current definition of the shift was adopted as a pragmatic compromise. The problem can be overcome by using the str inquiry, if necessary. If x (i.e., c.rng()) may have a non-unit alignment stride, the assignment in the fragment above should be replaced by

```cpp
b(i) = c(i + x.str()) + c(i - x.str());
```
SOURCE:

Range x ;
Map u ;

Index i(x) ;
OVERALL(i) {
  ... u.offset(i + s) ...
} ALLOVER(i) ;

OUTPUT:

Range x ;
Map u ;

LocBlocksIndex i(x) ;

apgStack.push(&apg) ;
apg.restrict(i.dim) ;
for(i.beginLocBlk() ; i.test() ; i.nextLocBlk()) {
  int u_bas = u.offset(i) + s * u.str() ;
  int u_stp = u.str() * i.sub_stp ;

  for(int l = 0 ; l < i.count ; l++) {
    ... u_bas + u_stp * l ...
  }
}
apgStack.pop(&apg) ;

Figure 7.12: Translation of (constant) shifted offset computation.
7.5 class AllBlocksIndex

An iterator class, maintaining loop state for enumerations of all the blocks of a range.

The public interface of the AllBlocksIndex class is

class AllBlocksIndex : public LocBlocksIndex {
public:
   AllBlocksIndex(Range _x) ;

   void beginAllBlk() ;
   void nextAllBlk() ;
} ;

7.5.1 Constructor

AllBlocksIndex(Range x)

Create an index object for enumerating all blocks of range x.

7.5.2 Methods

beginAllBlk()

 Begins an enumeration of all blocks of the range. On exit from this member, 
the Block component of the Index contains the parameters of the first (non-
empty) block of the range, and the Location component represents the first 
element (smallest global subscript) in that block. “First block” simply means 
first in the enumeration—it does not guarantee that the block contains smaller 
subscripts than later blocks. 

If i is the loop index, the idiom for the loop is

   for(i.beginAllBlk() ; i.test() ; i.nextAllBlk())
   
The member test() is inherited from LocBlocksIndex (see section 7.2).

nextAllBlk()

Move to next (non-empty) block in the enumeration, updating the Block and 
Location components.

7.5.3 Use of AllBlocksIndex in the communication library

The run-time technology introduced in this chapter for translation of distributed 
loops is used extensively in the implementation of the collective communication 
library. In that context, blocks of ranges are usually more important than 
individual elements, so use of the LocBlocksIndex iterator is common.

174
Besides enumeration of locally held blocks, it is sometimes necessary to enumerate all blocks of a range—local and non-local. For example, a data remapping operation defines a map between the elements of two ranges with different distribution formats. A local block in one range maps to a specific subrange of the other. But in general this subrange does not correspond to a single block of the target range—it may be distributed over several processes, divided into a number of blocks. If the original block of data is to be moved to the target array, it must be split into a corresponding number of data blocks, each sent to the appropriate destination.

All blocks enumeration resembles the local blocks enumeration given in section 7.2, but the loop is started using the beginAllBlk() member of AllBlocksIndex.

```cpp
AllBlocksIndex i(x) ;
... for(i.beginAllBlk() ; i.test() ; i.nextAllBlk()) {
  ...
}
```

In each iteration, the crd field inherited from the Location component of i gives the process coordinate of the remote block, while the fields of the Block component define the subscript range. If it is necessary to compute memory offsets in the remote process, this can be done using offset and related members of Map in exactly the same way as for a local block.

There is no guarantee that enumeration of the blocks occurs in an obvious sequential order. It is guaranteed that the subsequence of blocks associated with a single process is enumerated in the same order as for local blocks enumeration using LocBlocksIndex—this guarantee is often exploited to ensure message blocks are sent and received in the same order [need clarification on this point of ordering].
Chapter 8

Implementation of the communication schedules
Index

active process group, 11
collective object, 12
collective operation, 12
contains
    member of Dimension, 19
    member of Group, 22
coordinate, 12
crd
    member of Dimension, 16
dim
    member of Procs, 14
Dimension class, 16
dimension set, 18
    representation of, 20
DimensionSet class, 18
dims
    member of Group, 22
    member of Procs, 14
Group class, 21
id
    member of Group, 22
    member of Procs, 14
id_abs
    member of Group, 23
    member of Procs, 15
id_prc
    member of Group, 23
initial process group, 11
isEqual
    member of Dimension, 18
    member of Group, 22
lead_abs
    member of Group, 23
    member of Procs, 15
lead_prc
    member of Group, 23
member
    member of Dimension, 18
    member of Group, 22
    member of Procs, 14
prc
    member of Group, 22
process array, 12, 13
process dimension, 14, 16
process group, 12, 21
    active, 11
    initial, 11
process id, 12
Procs class, 13
rank
    of process array, 14
restrict
    member of Group, 23
rnk
    member of Procs, 14
size
    member of Dimension, 16
    member of Group, 22
    member of Procs, 14
str_prc
    member of Dimension, 16