Publication date: 2014-08-04

VSIPL++ Parallel

Version 1.3

OMG Document Number: formal/2014-08-07

Normative reference: http://www.omg.org/spec/VSIPL++

1. Use of specification - terms, conditions & notices

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

2. Licenses

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

3. Patents

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

4. General use restrictions

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means -- graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems -- without permission of the copyright owner.

5. Disclaimer of warranty

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.

Foreword

The VSIPL++ Library provides C++ classes and functions for writing embedded signal processing applications designed to run on one or more processors. VSIPL++ contains

- containers such as vectors, matrixes, and tensors,
- mathematical operations such as addition and matrix multiplication on these containers,
- complex numbers and random numbers,
- various linear algebra operations including solvers using LU, QR, and singular value decomposition methods, and
- signal processing classes and functions including fast Fourier transforms, convolutions, correlations, FIR filters, and IIR filters.

The VSIPL++ specification has been split into two overlapping documents, covering uniprocessor execution and multi-processor execution. Uniprocessor execution is specified in the VSIPL++ 1.3, while distributed execution is specified here.

This distributed specification is significantly shorter than the uniprocessor specification because the library has been carefully designed to support running the same programs in either single or multiple processor modes. This specification contains means to specify how a container's contents are distributed among available processors. Mathematical operations and other operations on those containers need not be re-specified because the toolkit automatically ensures data is moved to where it is used so that valid serial and parallel programs have the same effect.

Please visit the High Performance Embedded Computing Software Initiative webpage [http://www.hpec-si.org] for more information on VSIPL++ and for reference implementations.

1. Introduction [intro]

1.1. Scope [intro.scope]

1 This document specifies requirements for implementations of the VSIPL++ Library suitable for a multi-processor environment.

1.2. Normative references [intro.refs]

1 This document “VSIPL++ Parallel Specification (1.2)” is part of the VSIPL++ Specification and is incorporated via reference into the “VSIPL++ Specification (1.2)”.

2 [*Note:* This document and the other VSIPL++ specification documents use the same clause notation even though the clause and sub-clause numbers may differ.]

Header *<vsip/support.hpp>* synopsis

```
namespace vsip
{
  // map and processor types
  typedef implementationdefined processor_type;
  typedef signed-version-of-processor_type processor_difference_type;
  enum distribution_type { block, cyclic, whole, other};

  // functions
  length_type num_processors() VSIP_NOTHROW;
  const_Vector<processor_type, implementationdefined > processor_set()
    VSIP_THROW((std::bad_alloc));
  processor_type local_processor() VSIP_NOTHROW;
  index_type local_processor_index() VSIP_NOTHROW;

  // constants
  processor_type const no_processor;
  index_type const no_subblock;
}
```

2.1. Types [support.types]

2.1.1. Parallel types [support.types.par]

1
```
typedef implementation-defined processor_type;
```

Value:

processor_type is an integral type that represents a particular processor. A *processor* is an execution context with associated memory capably of computation.

2
```
typedef signed-version-of-processor_type processor_difference_type;
```

Value:

This type indicates a difference between two processor_type types. An instance can also be used to increment a processor_type instance. [*Note:* This type is not very useful to the user but is required to satisfy constant iterator requirements in [view.map].]

2.2. Functions [support.functions]

1
```
length_type num_processors() VSIP_NOTHROW;
```

Returns:

The total number of processors executing the data-parallel program.

2
```
const_Vector<processor_type, implementation-defined>
processor_set() VSIP_THROW((std::bad_alloc));
```

Returns:
The set of processors executing the data-parallel program.

3
```
processor_type local_processor() VSIP_NOTHROW;
```

Returns:
The local processor.

4
```
index_type local_processor_index() VSIP_NOTHROW;
```

Returns:
The index of the local processor in the processor set.
(`processor_set().get(local_processor_index()) == local_processor()`).

2.3. Constants [support.constants]

1 A processor_type of value no_processor is used to indicate no valid processor.

2 An index_type of value no_subblock is used to indicate no valid subblock.

1 This clause describes components that VSIPL++ programs can use to describe the mapping of data to multiple processors. A map is an interface that describes how data stored in blocks can be distributed over multiple processors. *Map*, *Local_map*, and *Replicated_map* classes satisfy this interface.

2 [*Note:* As noted in [support], a *processor* is an execution context with associated memory capable of computation. Multiple processors can together compute a single program's values.]

3.1. Map requirements [map.map]

1 Every *map* specifies how data stored in blocks can be distributed over multiple processors.

2 Applying a map to a VSIPL++ block yields a set of disjoint *subblocks*, whose union contains all the block's indices. Each subblock is an ordered set of indices. A *patch* is a maximal subset of a subblock with contiguous indices.

3 The map defines a relation between subblocks and processors in the map's *processor set*. The number of subblocks must be less than or equal to the number of processors, i.e. if map is an object of a class with the map interface, then `map.num_subblocks() <= map.num_processors()`. Each processor can have either 0 or 1 subblocks.

4 In enum distribution_type, block indicates that contiguous values will be placed in the same subblock. cyclic indicates that contiguous values will be distributed in a round-robin manner among the subblocks. whole indicates that all values will be placed in a single subblock. other indicates that values will be distributed in some other fashion.

5 `cyclic_contiguity` indicates the number of contiguous values to consider as a unit when distributing in a round-robin manner. cyclic with a `cyclic_contiguity` value greater than one is commonly called "block-cyclic."

6 A map is said to be x*-dimensional* (where x is a positive integer) if it specifies distributions for all dimensions less than x.

7 Every map shall satisfy the requirements in Table 3.1, "Map requirements". In Table 3.1, "Map requirements", x denotes a Dim-dimensional map object of type M, d is a dimension_type value, sb is an index_type value indicating a subblock, and pr is a processor_type value.

Table 3.1. Map requirements

expression	return type	assertion/note pre/post-condition
`m.num_subblocks()`	length_type	
`m.num_processors()`	length_type	
`m.processor_set()`	const_Vector< processor_type, *implementation-defined*>	
`m.distribution(d)`	distribution_type	d < Dim
`m.num_subblocks(d)`	length_type	pre: d < Dim
		post: greater than zero
`m.cyclic_contiguity(d)`	length_type	d < Dim
`m.subblock()`	index_type	

`m.subblock(pr)`	index_type	pr in processor_set
`M::processor_iterator`	*unspecified*	
`m.processors_begin(sb)`	processor_iterator	0 <= sb < num_subblocks()
`m.processors_end(sb)`	processor_iterator	0 <= sb < num_subblocks()

8 `m.num_subblocks()` returns the total number of subblocks in the map.

9 `m.num_processors()` returns the total number of processors in the map's processor set.

10 `m.processor_set()` returns the map's processor set.

11 `m.distribution(d)` returns the type of distribution for dimension d.

12 `m.num_subblocks(d)` returns the total number of subblocks when the map is projected onto dimension d.

13 `m.cyclic_contiguity(d)` yields a positive value if and only if `m.distribution(d) == cyclic`. Otherwise it yields 0.

14 `m.subblock()` returns the subblock held by the local processor, or no_subblock if no subblock is held.

15 `m.subblock(pr)` returns the subblock held by processor pr, or no_subblock if no subblock is held.

16 `M::processor_iterator` must satisfy random access iterator requirements (ISO14882, lib.random.access.iterators) and constant iterator requirements (ISO14882, lib.iterator.requirements).

17 `m.processors_begin(sb)` returns an iterator referring to the first processor in the sequence of processors that have subblock sb. m.processors_end(sb) returns an iterator which is the past-the-end value for the same sequence. If the sequence is empty (that is, no processors store subblock sb), then m.processors_begin(sb) == m.processors_end(sb).

18 If sb = m.subblock(pr) is valid, then pr is in the sequence denoted by m.processors_begin(sb) and m.processors_end(sb).

19 If pr is in the sequence denoted by `m.processors_begin(sb)` and `m.processors_end(sb)`, then `sb = m.subblock(pr)`.

3.2. Block-Cyclic Maps [map.blockcyclic]

1 The Map class is a map that distributes VSIPL++ blocks with block, cyclic, blockcyclic, or whole distributions in each dimension.

2 The distribution of each dimension in a Map is described by a distribution class Block_dist, Cyclic_dist, or Whole_dist.

Header *<vsip/map.hpp>* synopsis

```
namespace vsip
{

  // Data distributions.
  class Block_dist;
  class Cyclic_dist;
  class Whole_dist;

  // Block-cyclic map class.
```

```
  template <typename Dim0 = Block_dist,
            ...,
            typename Dimn = Block_dist>
    // exactly VSIP_MAX_DIMENSION template parameters
  class Map;

  class Local_map;

  template <dimension_type Dim>
  class Replicated_map;
}
```

3.2.1. Data Distribution Classes [map.blockcyclic.distribute]

1 The distribution classes Block_dist, Cyclic_dist, and Whole_dist describe how a single dimension is distributed.

```
namespace vsip
{
  class Block_dist
  {
  public:
    // constructor, destructor, copies, and assignments
    Block_dist(length_type num_subblocks = 1) VSIP_NOTHROW;
    Block_dist(Block_dist const&) VSIP_NOTHROW;
    Block_dist& operator=(Block_dist const&);
    ~Block_dist() VSIP_NOTHROW;

    // accessors
    distribution_type distribution()      const VSIP_NOTHROW;
    length_type       num_subblocks()     const VSIP_NOTHROW;
    length_type       cyclic_contiguity() const VSIP_NOTHROW;
  };

  class Cyclic_dist
  {
  public:
    // constructor, destructor, copies, and assignments
    Cyclic_dist(length_type num_subblocks = 1, length_type contiguity = 1) VSIP_NOTHROW;
    Cyclic_dist(Cyclic_dist const&) VSIP_NOTHROW;
    Cyclic_dist& operator=(Cyclic_dist const&);
    ~Cyclic_dist() VSIP_NOTHROW;

    // accessors
    distribution_type distribution()      const VSIP_NOTHROW;
    length_type       num_subblocks()     const VSIP_NOTHROW;
    length_type       cyclic_contiguity() const VSIP_NOTHROW;
  };

  class Whole_dist
  {
  public:
    // constructor, destructor, copies, and assignments
    Whole_dist(length_type num_subblocks = 1) VSIP_NOTHROW;
    Whole_dist(Whole_dist const&) VSIP_NOTHROW;
    Whole_dist& operator=(Whole_dist const&);
    ~Whole_dist() VSIP_NOTHROW;

    // accessors
    distribution_type distribution()      const VSIP_NOTHROW;
    length_type       num_subblocks()     const VSIP_NOTHROW;
    length_type       cyclic_contiguity() const VSIP_NOTHROW;
  };
}
```

3.2.2. Block_dist data distributions [map.blockcyclic.blockdistribution]

1 A Block_dist object indicates a block data distribution. Applying a block distribution to a one-dimensional domain yields approximately equally-sized subblocks of contiguous indices. More precisely, if the domain has n indices and the Block_dist object specifies s subblocks, then index i is in subblock $\lfloor i / \lceil n/s \rceil \rfloor$. Since subblocks have contiguous indices, each subblock has exactly one patch.

2
```
Block_dist(length_type num_subblocks = 1) VSIP_NOTHROW;
```

Requires:
num_subblocks > 0.

Effects:
Constructs an object representing a block data distribution with num_subblocks subblocks.

Postconditions:
`this->num_subblocks() == num_subblocks`.

3
```
distribution_type distribution() const VSIP_NOTHROW;
```

Returns:
block.

4
```
length_type num_subblocks() const VSIP_NOTHROW;
```

Returns:
The number of subblocks in the distribution.

5
```
length_type cyclic_contiguity() const VSIP_NOTHROW;
```

Returns:
0

Note:
This function does not make sense for block distributions but is provided so cyclic_contiguity can be called for any data distribution object.

3.2.3. Cyclic_dist data distributions [map.blockcyclic.cyclicdistribution]

1 A Cyclic_dist object indicates a cyclic data distribution. Conceptually, applying a cyclic distribution to a one-dimensional domain divides the domain into contiguous patches which are then distributed in a round-robin fashion among the subblocks. More precisely, if the Cyclic_dist object specifies s subblocks and a contiguity of c, then index i is in patch $\lfloor i/c \rfloor$ and subblock $\lfloor i/c \rfloor$.

2
```
Cyclic_dist(length_type num_subblocks = 1, length_type contiguity = 1) VSIP_NOTHROW;
```

Requires:
num_subblocks > 0. contiguity > 0.

Effects:
Constructs an object representing a cyclic data distribution with num_subblocks subblocks and patches having contiguity indices.

Postconditions:
this->num_subblocks() == num_subblocks this->cyclic_contiguity() == contiguity.

3
```
distribution_type distribution() const VSIP_NOTHROW;
```

Returns:
cyclic.

4
```
length_type num_subblocks() const VSIP_NOTHROW;
```

Returns:
The number of subblocks in the distribution.

5
```
length_type cyclic_contiguity() const VSIP_NOTHROW;
```

Returns:
The number of contiguous indices in a patch of the distribution.

3.2.4. Whole_dist data distributions [map.blockcyclic.wholedistribution]

1 A Whole_dist object indicates a whole data distribution. Applying a whole distribution to a one-dimensional domain yields one contiguous subblock that contains all the indices. Since the subblock has contiguous indices, it has exactly one patch.

2
```
Whole_dist(length_type num_subblocks = 1) VSIP_NOTHROW;
```

Requires:
num_subblocks == 1.

Effects:
Constructs an object representing a whole data distribution.

Note:
Even though num_subblocks must always be 1, this constructor allows the Map constructor taking a number of subblocks in each dimension to work with Whole_dist.

3
```
distribution_type distribution() const VSIP_NOTHROW;
```

Returns:
whole.

4
```
length_type num_subblocks() const VSIP_NOTHROW;
```

Returns:
1.

5
```
length_type cyclic_contiguity() const VSIP_NOTHROW;
```

Returns:
0.

Note:

This function does not make sense for whole distributions but is provided so cyclic_contiguity can be called for any data distribution object.

3.2.5. Map map [map.blockcyclic.mapclass]

```
namespace vsip
{

  template <typename Dim0 = Block_dist, ...,
            typename Dimn = Block_dist>
            // exactly VSIP_MAX_DIMENSION template parameters
  class Map
  {
    // Compile-time typedefs.
  public:
    typedef unspecified processor_iterator;

    // Constructors, destructor, and copy constructor.
    Map(Dim0 const& = Dim0(), ...,
        Dimn const& = Dimn())
      VSIP_NOTHROW;
    Map(const_Vector<processor_type>,
        Dim0 const& = Dim0(), ...,
        Dimn const& = Dimn())
      VSIP_NOTHROW;
    Map(Map const&) VSIP_NOTHROW;
    ~Map() VSIP_NOTHROW;

    // assignment
    Map &operator=(Map const&);

    distribution_type distribution(dimension_type) const VSIP_NOTHROW;
    length_type num_subblocks(dimension_type) const VSIP_NOTHROW;
    length_type cyclic_contiguity(dimension_type) const VSIP_NOTHROW;

    length_type num_subblocks() const VSIP_NOTHROW;
    length_type num_processors() const VSIP_NOTHROW;

    // subblock and processor access
    index_type subblock(processor_type) const VSIP_NOTHROW;
    index_type subblock() const VSIP_NOTHROW;

    const_Vector<processor_type, implementationdefined>
    processor_set() const VSIP_THROW(std::bad_alloc);

    processor_iterator processor_begin(index_type) const VSIP_NOTHROW;
    processor_iterator processor_end  (index_type) const VSIP_NOTHROW;
  };
}
```

3.2.6. Template Parameters [map.blockcyclic.template]

1 There must be exactly VSIP_MAX_DIMENSION template parameters, each having a default type of Block_dist.

2 Each template argument must be either Block_dist, Cyclic_dist, or Whole_dist.

3.2.7. Constructors [map.blockcyclic.constructors]

```
Map(Dim0 const& dist0, ..., Dimn const& distn) VSIP_NOTHROW;
```

Notation:

The parameter list contains VSIP_MAX_DIMENSION parameters, each a constant reference to the corresponding type Dim*i* in the template parameter list. Each parameter has a default argument Dim*i*().

Requires:

The number of subblocks specified by the data distribution parameters (this->num_subblocks()) must be less than or equal the total number or processors running the data-parallel program (vsip::num_processors() or equivalently vsip::processor_set().size()).

Effects:

Constructs a Map object whose processor set is the full processor set of the data-parallel program (vsip::processor_set()). The *i*th sub-block is mapped to the *i*th processor in the processor set.

Postconditions:

this->num_processors() == vsip::processor_set.size(). this- >processor_set() contains the same processors as vsip::processor_set(), in the same order.

```
template <typename BlockT>
Map(const_Vector<processor_type, BlockT> processor_set,
    Dim0 const &dist0, ..., Dimn const &distn)
  VSIP_NOTHROW;
```

Notation:

The parameter list contains VSIP_MAX_DIMENSION data distribution parameters, each a constant reference to the corresponding type Dim*i* in the template parameter list. Each parameter has a default argument Dim*i*().

Requires:

processor set.size() must be at least the number of subblocks specified by the data distribution parameters (this->num_subblocks()). This number is the product of each data distribution's number of subblocks. This function is present only if an implementation permits instantiation of const Vector<processor_type>.

Effects:

Constructs a Map object with the specified processor set processor_set. The *i*th sub-block is mapped to the *i*th processor in the processor set.

Postconditions:

`this->num_processors() == processor_set.size()`. `this->processor_set()` contains the same processors as processor_set, in the same order.

3.2.8. Accessor functions [map.blockcyclic.accessors]

1 For notational simplicity in this subclause, let dd[i] indicate the ith constructor data distribution object when i >= 0 && i < VSIP_MAX_DIMENSION .

2

```
distribution_type distribution(dimension_type d) const VSIP_NOTHROW;
```

Requires:

d < VSIP_MAX_DIMENSION.

Returns:

The distribution type of the dth data distribution dd[d].distribution().

3
```
length_type num_subblocks(dimension_type d) const VSIP_NOTHROW;
```

Requires:
d < VSIP_MAX_DIMENSION.

Returns:
The number of subblocks for the dth data distribution dd[d].num_subblocks().

4
```
length_type cyclic_contiguity(dimension_type d) const VSIP_NOTHROW;
```

Requires:
d < VSIP_MAX_DIMENSION.

Returns:
The cyclic contiguity of the d th data distribution dd[d].cyclic_contiguity().

5
```
length_type num_subblocks() const VSIP_NOTHROW;
```

Returns:
The total number of subblocks for *this map. This is the product of each data distribution's number of subblocks. this->num_subblocks() = $\prod_{d=0}^{D_{max}-1}$ this->num_subblocks(d). where D_{max} == VSIP_MAX_DIMENSION

6
```
length_type num_processors() const VSIP_NOTHROW;
```

Returns:
The total number of processors for *this' processor set. (`this->num_processors() == this->processor_set().size()`).

3.2.9. Subblock and processor iterator accessors [map.blockcyclic.gridfn]

1
```
index_type subblock(processor_type pr) const VSIP_NOTHROW;
```

Returns:
The subblock held by processor pr, or no_subblock if pr does not hold a subblock.

2
```
index_type subblock() const VSIP_NOTHROW;
```

Returns:
The subblock held by the local processor, or no_subblock if the local processor does not hold a subblock.

3
```
const_Vector<processor_type,
      implementation-defined> processor_set() const VSIP_THROW(std::bad_alloc);
```

Returns:
The processor set for *this.

4
```
processor_iterator processors_begin(index_type sb) const
      VSIP_NOTHROW;
```

Requires:
sb to be a valid subblock of *this this->num_subblocks()) or no_subblock.

Returns:
The beginning of a sequence containing only processor_set.get(sb).

5
```
processor_iterator processors_end(index_type sb) const VSIP_NOTHROW;
```

Requires:
sb to be a valid subblock of *this (0 <= sb < this->num_subblocks()) or no_subblock.

Returns:
The end of the sequence returned by processors_begin(sb).

3.3. Local_map map [map.localmap]

1 The class Local_map is a *map* that describes data that is stored locally on a single processor and is not distributed.

```
namespace vsip
{
  class Local_map
  {
  public:
    typedef unspecified processor_iterator;

    Local_map();

    distribution_type distribution(dimension_type) const VSIP_NOTHROW;
    length_type num_subblocks(dimension_type) const VSIP_NOTHROW;
    length_type cyclic_contiguity(dimension_type) const VSIP_NOTHROW;

    length_type num_subblocks() const VSIP_NOTHROW;
    length_type num_processors() const VSIP_NOTHROW;

    index_type subblock(processor_type) const VSIP_NOTHROW;
    index_type subblock() const VSIP_NOTHROW;

    const_Vector<processor_type, implementationdefined > processor_set() const
      VSIP_THROW(std::bad_alloc);

    processor_iterator processor_begin(index_type) const VSIP_NOTHROW;
    processor_iterator processor_end (index_type) const VSIP_NOTHROW;
  };
}
```

3.3.1. Constructors [map.localmap.constructors]

1
```
Local_map() VSIP_NOTHROW;
```

Effects:
Constructs a Local_map object with the local processor as the processor set.

3.3.2. Accessor functions [map.localmap.accessors]

1
```
distribution_type distribution(dimension_type d) const VSIP_NOTHROW;
```

Requires:
d < VSIP_MAX_DIMENSION.

Returns:
whole.

2
```
length_type num_subblocks(dimension_type d) const VSIP_NOTHROW;
```

Requires:
d < VSIP_MAX_DIMENSION.

Returns:
1.

3
```
length_type cyclic_contiguity(dimension_type d) const VSIP_NOTHROW;
```

Requires:
d < VSIP_MAX_DIMENSION.

Returns:
0.

4
```
length_type num_subblocks() const VSIP_NOTHROW;
```

Returns:
1.

5
```
length_type num_processors() const VSIP_NOTHROW;
```

Returns:
1.

3.3.3. Subblock and processor iterator accessors [map.localmap.gridfn]

1
```
index_type subblock(processor_type pr) const VSIP_NOTHROW;
```

Returns:
0 if pr is the local processor, no_subblock otherwise.

2
```
index_type subblock() const VSIP_NOTHROW;
```

Returns:
0 (the subblock held by the local processor).

3
```
const_Vector<processor_type, implementation-defined>
processor_set() const VSIP_THROW(std::bad_alloc);
```

Returns:
A vector containing the local processor.

4
```
processor_iterator processors_begin(index_type sb) const VSIP_NOTHROW;
```

Requires:
sb to be a valid subblock (sb == 0).

Returns:
The beginning of a sequence containing only the local processor.

5
```
processor_iterator processors_end(index_type sb) const VSIP_NOTHROW;
```

Requires:
sb to be a valid subblock (sb == 0).

Returns:
The end of the sequence returned by processors_begin .

3.4. Replicated_map map [map.replicatedmap]

1 The class Replicated_map is a *map* that describes data that is replicated. Each processor in the map's processor set owns an entire copy of the data.

```
namespace vsip
{
  template <dimension_type Dim>
  class Replicated_map
  {
  public:
    typedef unspecified processor_iterator;

    Replicated_map();
    template <typename Block>
    Replicated_map(const_Vector<processor_type, Block>);

    distribution_type distribution(dimension_type) const VSIP_NOTHROW;
    length_type num_subblocks(dimension_type) const VSIP_NOTHROW;

    length_type cyclic_contiguity(dimension_type) const VSIP_NOTHROW;

    length_type num_subblocks() const VSIP_NOTHROW;
    length_type num_processors() const VSIP_NOTHROW;

    index_type subblock(processor_type) const VSIP_NOTHROW;
    index_type subblock() const VSIP_NOTHROW;

    const_Vector<processor_type, implementationdefined> processor_set()

    const VSIP_THROW(std::bad_alloc);
    processor_iterator processor_begin(index_type) const VSIP_NOTHROW;
    processor_iterator processor_end (index_type) const VSIP_NOTHROW;
  };
}
```

3.4.1. Template Parameters [map.replicatedmap.template]

1 Dim specifies the dimensionality of the Replicated_map. It is at least one and at most VSIP_MAX_DIMENSION.

3.4.2. Constructors [map.replicatedmap.constructors]

1
```
Replicated_map() VSIP_NOTHROW;
```

Effects:
Constructs a Replicated_map object with the full processor set.

2
```
template <typename Block>
Replicated_map(const_Vector<processor_type, Block> processor_set) VSIP_NOTHROW;
```

Effects:
Constructs a Replicated_map object with the processor set specified by processor_set.

3.4.3. Accessor functions [map.replicatedmap.accessors]

1
```
distribution_type distribution(dimension_type d) const VSIP_NOTHROW;
```

Requires:
d < VSIP_MAX_DIMENSION.

Returns:
whole.

2
```
length_type num_subblocks(dimension_type d) const VSIP_NOTHROW;
```

Requires:
d < VSIP_MAX_DIMENSION.

Returns:
1, if the current processor is part of *this’ processor set, 0 otherwise.

3
```
length_type cyclic_contiguity(dimension_type d) const VSIP_NOTHROW;
```

Requires:
d < VSIP_MAX_DIMENSION.

Returns:
0.

4
```
length_type num_subblocks() const VSIP_NOTHROW;
```

Returns:
1, if the current processor is part of *this’ processor set, 0 otherwise.

5
```
length_type num_processors() const VSIP_NOTHROW;
```

Returns:
The total number of processors for *this’ processor set.

3.4.4. Subblock and processor iterator accessors [map.replicatedmap.gridfn]

1
```
index_type subblock(processor_type pr) const VSIP_NOTHROW;
```

Requires:
pr to be a valid processor in *this’ processor set.

0, if pr is part of *this’ processor set, no_subblock otherwise.

2
```
index_type subblock() const VSIP_NOTHROW;
```

Returns:
0, if pr is part of *this’ processor set, no_subblock otherwise.

3
```
const_Vector<processor_type, implementation-de#ned> processor_set() const
  VSIP_THROW(std::bad_alloc);
```

Returns:
The processor set for *this.

4
```
processor_iterator processors_begin(index_type sb) const VSIP_NOTHROW;
```

Requires:
sb to be a valid subblock (sb == 0).

Returns:
The beginning of a sequence containing *this processor set.

5
```
processor_iterator processors_end(index_type sb) const VSIP_NOTHROW;
```

Requires:
sb to be a valid subblock (sb == 0).

Returns:
The end of the sequence returned by processors_begin(sb).

4. Blocks [block]

4.1. Dense block [block.dense]

1 All Dense class template specifications apply, except the semantics of user-specified storage are refined to account for distributed blocks.

4.1.1. Constructors, copy, assignment, and destructor [block.dense.constructors]

1
```
Dense(Domain<D> const &dom, T *const pointer, map_type const &map = map_type())
  VSIP_THROW((std::bad_alloc));
```

Requires:
: For all i such that `0 <= i < subblock_domain.size()`, `pointer[i] = T()` must be a valid C++ expression. subblock_domain is the domain of the subblock stored on the local processor if the block is distributed. subblock_domain is dom if the block is not distributed.

Effects:
: Constructs a modifiable Dense object with user-specified storage and distributed by the map map.

Throws:
: std::bad_alloc indicating memory allocation for the returned Dense failed.

Postconditions:
: If D == 1, *this will have a one-dimensional Domain<1> denoted domain with Index<1>es containing 0, …, dom.size()-1 . If D != 1, *this will have two domains: Domain<1> domain1 with Index<1>s containing 0, …, dom.size()-1 and a Domain<D> domainD with, for each 0 <= d < D, domainD[d].size() == dom[d].size(), domainD[d].stride() == 1, and domainD[d].first() == 0. The object's use count will be one. this->user_storage() == array_format.

Note:
: The block's values are unspecified. This block's values can only be accessed after an admit call and before its corresponding release call. When the block is admitted, the pointer[i] values listed above may be modified by the block.

2
```
Dense(Domain<D> const &dom, uT *const pointer, map_type const &map = map_type())
  VSIP_THROW((std::bad_alloc));
```

Requires:
: T must be a complex type, with underlying type uT (T = complex<uT>). For all i such that 0 <= i < 2*subblock_domain.size(), pointer[i] = uT() must be a valid C++ expression. subblock_domain is the domain of the subblock stored on the local processor if the block is distributed. subblock_domain is dom if the block is not distributed.

Effects:
: Constructs a modifiable Dense object with user-specified storage and distributed by the map map.

Throws:
std::bad_alloc indicating memory allocation for the returned Dense failed.

Postconditions:
If D == 1, *this will have a one-dimensional Domain<1> denoted domain with Index<1>es containing 0, …, dom.size()-1 . If D != 1, *this will have two domains: Domain<1> domain1 with Index<1>es containing 0, …, dom.size()-1 and a Domain<D> domainD with, for each 0 <= d < D, domainD[d].size() == dom[d].size(), domainD[d].stride() == 1, and domainD[d].first() == 0. The object's use count will be one. this->user_storage() == interleaved_format.

Note:
The block's values are unspecified. This block's values can only be accessed after an admit call and before its corresponding release call. When the block is admitted, the pointer[i] values listed above may be modified by the block.

3

```
Dense(Domain<D> const &dom, uT *const real_pointer, uT *const imag_pointer,
      map_type const &map = map_type())
  VSIP_THROW((std::bad_alloc));
```

Requires:
T must be a complex type, with underlying type uT (T = complex<uT>). For all i such that 0 <= i < subblock_domain.size(), real_pointer[i] = uT() and imag_pointer[i] = uT() must be valid C++ expressions. subblock_domain is the domain of the subblock stored on the local processor if the block is distributed. subblock_domain is dom if the block is not distributed.

Effects:
Constructs a modifiable Dense object with user-specified storage and distributed by the map map.

Throws:
std::bad_alloc indicating memory allocation for the returned Dense failed.

Postconditions:
If D == 1, *this will have a one-dimensional Domain<1> denoted domain with Index<1>es containing 0, …, dom.size()-1 . If D != 1, *this will have two domains: Domain<1> domain1 with Index<1>es containing 0, …, dom.size()-1 and a Domain<D> domainD with, for each 0 <= d < D, domainD[d].size() == dom[d].size(), domainD[d].stride() == 1, and domainD[d].first() == 0. The object's use count will be one. this->user_storage() == split_format.

Note:
The block's values are unspecified. This block's values can only be accessed after an admit call and before its corresponding release call. When the block is admitted, the real_pointer[i] and imag_pointer[i] values listed above may be modified by the block.

4.1.2. User-specified storage [block.dense.userdata]

1 For the definitions in this section, let

- D be the dimensionality of Dense block *this.
- subblock_domain be the domain of the subblock of *this stored on the local processor. If no subblock is stored on the local processor, subblock_domain is empty.
- l_idx = $(l_0, \cdots, l_{D-1})$ refer to a local index of the subblock of *this stored on the local processor, where $0 \le l_d$ < subblock_domain[d].size() for each dimension d of block *this.

- g_idx = $(g_0, \cdots, g_{D-1})$ refer to the global index corresponding to local index l_idx.
- linear_index(l_idx) be a function that converts a local index l_idx of block *this into a linear index in accordance with the dimension-ordering $d_0 \cdots d_{D-1}$ of block *this:

  linear_index(l_idx) = $\sum_{i=0}^{D-1}\left(l_{d_i} * \prod_{j=i+1}^{D-1}\text{this->size(D,}d_j\text{)}\right)$

2
```
void admit(bool update = true) VSIP_NOTHROW;
```

Effects:

If *this is not a block with user-specified storage or *this is an admitted block with user-specified storage, there is no effect. Otherwise, the block is admitted so that its data can be used by VSIPL ++ functions and objects.

If update == true, the values of *this are updated as appropriate for the value of this->user_storage().

Assuming this->user_storage() == array_format, for all local indices l_idx with corresponding global indices g_idx = $(g_0, \cdots, g_{D-1})$: this->get$(g_0, \cdots, g_{D-1})$ == pointer[linear_index(l_idx)] where pointer is the value returned by this->find.

Assuming this->user_storage() == interleaved_format, for all local indices l_idx with corresponding global indices g_idx = $(g_0, \cdots, g_{D-1})$: this->get$(g_0, \cdots, g_{D-1})$ == complex<T>(pointer[2*linear_index(l_idx)+0], pointer[2*linear_index(l_idx)+1]) where pointer is the value returned by this->find.

Assuming this->user_storage() == split_format, for all local indices l_idx with corresponding global indices g_idx = $(g_0, \cdots, g_{D-1})$: this->get$(g_0, \cdots, g_{D-1})$ == complex<T>(real_pointer[linear_index(l_idx)], imag_pointer[linear_index(l_idx)]) where real_pointer and imag_pointer are the values returned by this->find.

If update == false, the result of this->get$(g_0, \cdots, g_{D-1})$ for all g_idx = $(g_0, \cdots, g_{D-1})$ such that 0 <= g_d < this->size(D, d), is undefined.

Note:

Invoking admit on an admitted block is permitted. The intent of using a false update flag is that, if the data in the user-specified storage is not needed, then there is no need to force consistency between the block's storage and the user-specified storage possibly through copies.

3
```
void release(bool update = true) VSIP_NOTHROW;
```

Effects:

If *this is not a block with user-specified storage or *this is a released block with user-specified storage, there is no effect. Otherwise, the block is released so that its data cannot be used by VSIPL++ functions and objects.

If update == true, the values in the user-specified storage are updated as appropriate for the value of this->user_storage().

Assuming this->user_storage() == array_format, for all local indices l_idx with corresponding global indices g_idx = $(g_0, \cdots, g_{D-1})$: this->get$(g_0, \cdots, g_{D-1})$ == pointer[linear_index(l_idx)] where pointer is the value returned by this->find.

Assuming this->user_storage() == interleaved_format, for all local indices l_idx with corresponding global indices g_idx = $(g_0, \cdots, g_{D-1})$: this->get$(g_0, \cdots, g_{D-1})$ == complex<T>(pointer[2*linear_index(l_idx)+0], pointer[2*linear_index(l_idx)+1]) where pointer is the value returned by this->find.

Assuming this->user_storage() == split_format, for all local indices l_idx with corresponding global indices g_idx = $(g_0, \cdots, g_{D-1})$: this->get$(g_0, \cdots, g_{D-1})$ == complex<T>(real_pointer[linear_index(l_idx)], imag_pointer[linear_index(l_idx)]) where real_pointer and imag_pointer are the values returned by this->find.

If update == false, the values in the user-specified storage are undefined.

Note:

Invoking release on a released block is permitted. The intent of using a false update flag is that, if the data in the block's storage is no longer needed, then there is no need to force consistency between the block's storage and the user-specified storage possibly through copies.

4

```
void release(bool update, T*& pointer) VSIP_NOTHROW;
```

Requires:

*this must either be a block with user-specified storage such that this->user_storage() equals array_format, or a block without user-specified storage.

Effects:

If *this is not a block with user-specified storage, pointer is assigned NULL, but there are no other effects. If *this is a released block with user-specified storage, pointer is assigned the value returned by this->find, but there are no other effects.

Otherwise, the block is released so that its data may not be used by VSIPL++ functions and objects. pointer is assigned the value returned by this->find. If update == true, the values in the user-specified storage are updated. For all local indices l_idx with corresponding global indices g_idx = $(g_0, \cdots, g_{D-1})$: this->get$(g_0, \cdots, g_{D-1})$ == pointer[linear_index(l_idx)]

If update == false, the values in the user-specified storage are unspecified.

Note:

Invoking release on a released block is permitted. The intent of using a false update flag is that, if the data in the block's storage is no longer needed, then there is no need to force consistency between the block's storage and the user-specified storage possibly through copies.

5

```
void release(bool update, uT*& pointer) VSIP_NOTHROW;
```

Requires:

*this must either be a block with user-specified storage such that this->user_storage() equals interleaved_format, or a block without user-specified storage. T must be a complex type with underlying type uT (T = complex<uT>).

Effects:

If *this is not a block with user-specified storage, pointer is assigned NULL, but there are no other effects. If *this is a released block with user-specified storage, pointer is assigned the value returned by this->find, but there are no other effects.

Otherwise, the block is released so that its data may not be used by VSIPL++ functions and objects. pointer is assigned the value returned by this->find.

If update == true, the values in the user-specified storage are updated. for all local indices l_idx with corresponding global indices g_idx = $(g_0, \cdots, g_{D-1})$: this->get$(g_0, \cdots, g_{D-1})$ == complex<T>(pointer[2*linear_index(l_idx)+0], pointer[2*linear_index(l_idx)+1])

If update == false, the values in the user-specified storage are unspecified.

Note:

Invoking release on a released block is permitted. The intent of using a false update flag is that, if the data in the block’s storage is no longer needed, then there is no need to force consistency between the block’s storage and the user-specified storage possibly through copies.

6

```
void release(bool update, uT*& real_pointer, uT*& imag_pointer) VSIP_NOTHROW;
```

Requires:

*this must either be a block with user-specified storage such that this->user_storage() equals interleaved_format or split_format, or a block without user-specified storage. T must be a complex type with underlying type uT (T = complex<uT>).

Effects:

If *this is not a block with user-specified storage, real_pointer and imag_pointer are assigned NULL, but there are no other effects. If *this is a released block with user-specified storage, real_pointer and imag_pointer are assigned the values returned by this->find, but there are no other effects. Otherwise, the block is released so that its data may not be used by VSIPL++ functions and objects. real_pointer and imag_pointer are assigned the values returned by this->find.

If update == true, the values in the user-specified storage are updated. Assuming this->user_storage() == interleaved_format, for all local indices l_idx with corresponding global indices g_idx = $(g_0, \cdots, g_{D-1})$: this->get$(g_0, \cdots, g_{D-1})$ == complex<T>(real_pointer[2*linear_index(l_idx)+0], real_pointer[2*linear_index(l_idx)+1]) Assuming this->user_storage() == split_format, for all local indices l_idx with corresponding global indices g_idx = $(g_0, \cdots, g_{D-1})$: this->get$(g_0, \cdots, g_{D-1})$ == complex<T>(real_pointer[linear_index(l_idx)], imag_pointer[linear_index(l_idx)])

If update == false, the values in the user-specified storage are unspecified.

Note:

Invoking release on a released block is permitted. The intent of using a false update flag is that, if the data in the block’s storage is no longer needed, then there is no need to force consistency between the block’s storage and the user-specified storage possibly through copies.

7 [*Note:* No change to find member function specifications.]

8

```
void rebind(T* const pointer) VSIP_NOTHROW;
```

Requires:

!this->admitted() && this->user_storage() == array_format. For all i such that 0 <= i < subblock_domain.size(), pointer[i] = T() must be a valid C++ expression. subblock_domain is the local domain of the block stored on the local processor if the block is distributed or dom if the block is not distributed.

Effects:

If *this is a block with user-specified storage, replaces the block's user-specified storage pointer with pointer. If *this is not a block with user-specified storage, this function has no effect.

9

```
void rebind(uT* const pointer) VSIP_NOTHROW;
```

Requires:

!this->admitted(). this->user_storage() equals interleaved_format or split_format. T must be a complex type with underlying type uT (T = complex<uT>). For all i such that 0 <= i < 2 * subblock_domain.size(), pointer[i] = uT() must be a valid C++ expression. subblock_domain is the domain of the subblock stored on the local processor if the block is distributed. subblock_domain is dom if the block is not distributed.

Effects:

If *this is a block with user-specified storage, replaces the block's user-specified storage pointer with pointer. If *this is not a block with user-specified storage, this function has no effect.

Postconditions:

this->user_storage() == interleaved_format.

10

```
void rebind(uT*const real_pointer, uT*const imag_pointer) VSIP_NOTHROW;
```

Requires:

!this->admitted(). this->user_storage() equals interleaved_format or split_format. T must be a complex type with underlying type uT (T = complex<uT>). For all i such that 0 <= i < subblock_domain.size(), real_pointer[i] = uT() and imag_pointer[i] = uT() must be valid C++ expressions. subblock_domain is the domain of the subblock stored on the local processor if the block is distributed. subblock_domain is dom if the block is not distributed.

Effects:

If *this is a block with user-specified storage, replaces the block's user-specified storage pointers with real_pointer and imag_pointer. If *this is not a block with user-specified storage, this function has no effect.

Postconditions:

this->user_storage() == split_format.

5. Views [view]

5.1. Definitions [view.view]

1 [*Note:* [view.view] occurs in a separate document excepting the additional definitions specified below.]

2 Every view meets the requirements in Table 5.1, “Parallel view requirements”. In Table 5.1, “Parallel view requirements”, V denotes a view class, v denotes a value of type V, and p denotes a value of type index_type indicating a valid patch of the subblock of v held by the local processor (0 <= p < num_patches(v)).

Table 5.1. Parallel view requirements

expression	return type
`V::local_type`	implementation-defined
`V::local_patch_type`	implementation-defined
`v.local()`	V::local_type
`v.local(p)`	V::local_patch_type

3 [*Note:* Every view has an associated block which is responsible for storing or computing the data in the view ([view.view]). Every block has an associated map ([view.block]). Every map describes how data stored in a block can be distributed over multiple processors by dividing the block into a set of disjoint subblocks, whose union contains all of the block’s indices ([map.map]). Each subblock is an ordered set of indices ([map.map]). A patch is a maximal subset of a subblock with contiguous indices ([map.map]).]

4 The *local subblock view* of a distributed view is a view to the subblock stored on the local processor. If the local processor does not hold a subblock, the local subblock view is empty.

5 local_type is the type of a view’s local subblock view. It shall have the same dimension and same value type as the view. Its’ block type is unspecified.

6 The local() member function returns the view’s local subblock view.

7 local_patch_type is the type of a patch subview of a view’s local subblock view. It shall have the same dimension and same value type as the view. Its’ block type is unspecified.

8 The local(p) member function of a view returns a view of patch p of view’s local subblock.

5.2. Vector [view.vector]

1 All *Vector* class template specifications apply to the *Vector*<T, Block> class template except the two-parameter constructor Vector(length_type, const T&) and the one-parameter constructor Vector(length_type) are replaced by the constructors below and an additional typedef and member functions are added to access the local view of a distributed view.

5.2.1. Local View Types [view.vector.types]

1 local_type specifies the type of the local subblock view of *Vector*. The type is a *Vector* with a value type T and an unspecified block type.

2 local_patch_type specifies the type of a patch subview of a local subblock of *Vector*. The type is a *Vector* with a value type T and an unspecified block type.

5.2.2. Constructors [view.vector.constructors]

1

```
Vector(length_type len, T const& value,
       typename block_type::map_type const& map = typename block_type::map_type())
  VSIP_THROW((std::bad_alloc));
```

Requires:
`len > 0.`

Effects:
Constructs a modifiable, zero-indexed Vector object containing exactly len values equal to value with a map map.

Throws:
std::bad_alloc indicating memory allocation for the underlying Dense block failed.

2

```
Vector(length_type len,
       typename block_type::map_type const& map = typename block_type::map_type())
  VSIP_THROW((std::bad_alloc));
```

Requires:
`len > 0.`

Effects:
Constructs a modifiable, zero-indexed Vector object containing exactly len unspecified values with a map map.

Throws:
std::bad_alloc indicating memory allocation for the underlying Dense block failed.

5.2.3. Accessors [view.vector.accessors]

1

```
local_type local() VSIP_NOTHROW;
```

Effects:
Returns the local subblock view. If `*this` is a distributed view, a view of the local processor's subblock is returned. If `*this` is a local view, `*this` is returned.

Notes:
The domain of the local view is equivalent to the `subblock_domain(view)`.

2

```
local_patch_type local(index_type patch) VSIP_NOTHROW;
```

Requires:
patch to be a valid patch of the local `subblock (0 <= patch < num_patches(view))`.

Effects:
Returns a subview of the local subblock view corresponding to patch patch.

Notes:
`view.local(patch)` is equivalent to `view.local()(local_domain(view, map.subblock(), patch)`

5.3. Matrix [view.matrix]

1 All *Matrix* class template specifications apply to the *Matrix*<T, Block> class template except the three-parameter constructor `Matrix(length_type, length_type, const T&)` and the two-parameter constructor Matrix(length_type, length_type) are replaced by the constructors below and an additional typedef and member function are added to access the local view of a distributed view.

5.3.1. Local View Types [view.matrix.types]

1 local_type specifies the type of the local subblock view of *Matrix*. The type is a *Matrix* with a value type T and an unspecified block type.

2 local_patch_type specifies the type of a patch subview of a local subblock of *Matrix*. The type is a *Matrix* with a value type T and an unspecified block type.

5.3.2. Constructors [view.matrix.constructors]

1

```
Matrix(length_type rows, length_type columns, T const& value,
       typename block_type::map_type const& map = typename block_type::map_type())
  VSIP_THROW((std::bad_alloc));
```

Requires:
rows > 0 . columns > 0 .

Effects:
Constructs a modifiable, zero-indexed Matrix object containing exactly rows * columns values equal to value with a map map.

Throws:
std::bad_alloc indicating memory allocation for the underlying Dense block failed.

2

```
Matrix(length_type rows, length_type columns,
       typename block_type::map_type const& map = typename block_type::map_type())
  VSIP_THROW((std::bad_alloc));
```

Requires:
rows > 0 . columns > 0 .

Effects:
Constructs a modifiable, zero-indexed Matrix object containing exactly `rows * columns` unspecified values with a map map.

Throws:
std::bad_alloc indicating memory allocation for the underlying Dense block failed.

5.3.3. Accessors [view.matrix.accessors]

1

```
local_type local() VSIP_NOTHROW;
```

Effects:
Returns the local subblock view. If `*this` is a distributed view, a view of the local processor's subblock is returned. If `*this` is a local view, `*this` is returned.

Notes:
The domain of the local view is equivalent to the subblock_domain(view).

2
```
local_patch_type local(index_type patch) VSIP_NOTHROW;
```

Requires:
patch to be a valid patch of the local `subblock (0 <= patch < num_patches(view))`.

Effects:
Returns a subview of the local subblock view corresponding to patch patch.

Notes:
`view.local(patch)` is equivalent to `view.local()(local_domain(view, map.subblock(), patch)`

5.4. Tensor [view.tensor]

1 All *Tensor* class template specifications apply to the *Tensor*<T, Block> class template except the four-parameter constructor `Tensor(length_type, length_type, length_type, const T&)` and the three-parameter constructor `Tensor(length_type, length_type, length_type)` are replaced by the constructors below and an additional typedef and member function are added to access the local view of a distributed view.

5.4.1. Local View Types [view.tensor.types]

1 local_type specifies the type of the local subblock view of *Tensor*. The type is a *Tensor* with a value type T and an unspecified block type.

2 local_patch_type specifies the type of a patch subview of a local subblock of *Tensor*. The type is a *Tensor* with a value type T and an unspecified block type.

5.4.2. Constructors [view.tensor.constructors]

1
```
Tensor(length_type z_length,
       length_type y_length,
       length_type x_length,
       T const& value,
       typename block_type::map_type const& map = typename block_type::map_type())
  VSIP_THROW((std::bad_alloc));
```

Requires:
z_length > 0 . y_length > 0 . x_length > 0 .

Effects:
Constructs a modifiable, zero-indexed Tensor object containing exactly `z_length * y_length * x_length` values equal to value with a map map.

Throws:
std::bad_alloc indicating memory allocation for the underlying Dense block failed.

2
```
Tensor(length_type z_length,
       length_type y_length,
       length_type x_length,
       typename block_type::map_type const& map = typename block_type::map_type())
  VSIP_THROW((std::bad_alloc));
```

Requires:
z_length > 0 . y_length > 0 . x_length > 0 .

Effects:
Constructs a modifiable, zero-indexed Tensor object containing exactly `z_length * y_length * x_length` unspecified values with a map map.

Throws:
std::bad_alloc indicating memory allocation for the underlying Dense block failed.

5.4.3. Accessors [view.tensor.accessors]

1
```
local_type local() VSIP_NOTHROW;
```

Effects:
Returns the local subblock view. If *this is a distributed view, a view of the local processor's subblock is returned. If `*this` is a local view, `*this` is returned.

Notes:
The domain of the local view is equivalent to the subblock_domain(view).

2
```
local_patch_type local(index_type patch) VSIP_NOTHROW;
```

Requires:
patch to be a valid patch of the local subblock (0 <= patch < num_patches(view)).

Effects:
Returns a subview of the local subblock view corresponding to patch patch.

Notes:
`view.local(patch)` is equivalent to `view.local()(local_domain(view, map.subblock(), patch)`

5.5. Support Functions [view.support]

1 VSIPL++ provides a set of parallel support functions to query distributed views.

5.5.1. Definitions [view.support.defn]

1 The domain representing the size of a subblock is called the *subblock domain*.

2 A subblock is comprised of one or more patches (num_patches()). Each patch consists of a maximal sequence of elements with contiguous global indices.

3 The set of global indices represented by a subblock-patch is called the *global domain*.

4 The set of local indices storing a subblock-patch is called the *local domain*.

5 The global domain and local domain of a subblock-patch are element conformant.

6 The union of the global domains of all subblocks-patches for a view exactly overlaps with view's domain.

7 The union of the local domains of patches for a view's subblock exactly overlaps with the subblock domain of the view's subblock.

8 If view is a D-dimensional view and g_idx = $\left(g_0, \cdots, g_{D-1}\right)$ is a global index in the domain of view (for each $0 \le d < D$, $0 \le i_d <$ view.size(d)), then there exists a subblock-patch pair sb-p whose global domain gdom contains index $\left(g_0, \cdots, g_{D-1}\right)$.

9 Moreover, if $(p_0, \cdots, p_{D-1})$ are defined such that g_d = gdom[d].first() + p_d * gdom[d].stride(), then l_idx = $(l_0, \cdots, l_{D-1})$, where l_d = ldom[d].first() + p_d * ldom[d].stride() and ldom is the local domain of subblock-patch pair sb-p, is the local index of the local subblock storing the value of global index $(g_0, \cdots, g_{D-1})$ in the view view.

10 The following relationships hold on all processors:sb == subblock_from_global_index(view, g_idx),p == patch_from_global_index(view, g_idx),gdom == global_domain(view, sb, p),ldom == local_domain(view, sb, p),l_idx == local_from_global_index(view, g_idx), g_idx == global_from_local_index(view, sb, l_idx), l_d == local_from_global_index(view, d, g_d), and g_d == global_from_local_index(view, d, sb, l_d).

11 The following relationships hold on processors owning subblock sb (i.e. processors where sb == view.block().map().subblock()):local_view == view.local(),local_view.get$(l_0, \cdots, l_{D-1})$ == view.get$(g_0, \cdots, g_{D-1})$,gdom == global_domain(view, p),ldom == local_domain(view, p), andg_d == global_from_local_index(view, d, l_d).

5.5.2. Functions [view.support.fcn]

1 *<vsip/parallel.hpp>* provides all declarations in this sub-sub-clause unless otherwise indicated.

2
```
template <typename ViewT> Domain<ViewT::dim>
subblock_domain(ViewT const &view, index_type sb) VSIP_NOTHROW;
```

Requires:
Subblock sb to be a valid subblock of view view (0 <= sb < num_subblocks(view)) or no_subblock.

Effects:
If sb is a valid subblock, returns the domain representing the extent of view's subblock sb. If sb == no_subblock, returns an empty domain.

3
```
template <typename ViewT> Domain<ViewT::dim>
subblock_domain(ViewT const &view) VSIP_NOTHROW;
```

Effects:
Returns the domain representing the extent of view's subblock stored on the local processor, or an empty domain if the local processor does not hold a subblock.

Notes:
Equivalent to subblock_domain(view, subblock(view)).

4
```
template <typename ViewT> Domain<ViewT::dim>
local_domain(ViewT const &view, index_type sb, index_type p) VSIP_NOTHROW;
```

Requires:
sb to be a valid subblock of view (0 <= sb < num_subblocks(view)) or no_subblock. p to be a valid patch of view's subblock sb (0 <= p < num_patches(view, sb)).

Returns the local domain for subblock sb patch p of view.

5
```
template <typename ViewT> Domain<ViewT::dim>
local_domain(ViewT const &view, index_type p=0) VSIP_NOTHROW;
```

Requires:
p to be a valid patch of view’s local subblock (0 <= p < num_patches(view, subblock(view))).

Effects:
Returns the local domain for the local subblock patch p of view.

Notes:
Equivalent to local_domain(view, subblock(view), p).

6

```
template <typename ViewT> Domain<ViewT::dim>
global_domain( ViewT const &view, index_type sb, index_type p) VSIP_NOTHROW;
```

Requires:
sb to be a valid subblock of view (0 <= sb < num_subblocks(view)) or no_subblock. p to be a valid patch of view’s subblock sb (0 <= p < num_patches(view, sb)).

Returns the local domain for subblock sb patch p of view.

7

```
template <typename ViewT> Domain<ViewT::dim>
global_domain(ViewT const &view, index_type p=0) VSIP_NOTHROW;
```

Requires:
p to be a valid patch of view’s local subblock (0 <= p < num_patches(view, subblock(view))).

Effects:
Returns the local domain for the local subblock patch p of view.

Notes:
Equivalent to global_domain(view, subblock(view), p).

8

```
template <typename ViewT> length_type
num_subblocks(ViewT const &view) VSIP_NOTHROW;
```

Effects:
Returns the number of subblocks of view.

Notes:
Equivalent to view.block().map().num_subblocks().

9

```
template <typename ViewT> length_type
num_patches(ViewT const &view, index_type sb) VSIP_NOTHROW;
```

Requires:
sb to be either a valid subblock of view (0 <= sb < num_subblocks(view)) or no_subblock.

Returns the number of patches in subblock sb of view, or 0 if sb == no_subblock.

10

```
template <typename ViewT> length_type
num_patches(ViewT const &view) VSIP_NOTHROW;
```

Effects:
Returns the number of patches for the local subblock of view. Returns 0 if there is no local subblock.

Notes:
Equivalent to num_patches(view, subblock(view)).

11
```
template <typename ViewT> index_type
subblock(ViewT const &view, processor_type pr) VSIP_NOTHROW;
```

Effects:
Returns the subblock of view view held by processor pr, or no_subblock if pr does not hold a subblock.

Notes:
Equivalent to view.block().map().subblock(pr).

12
```
template <typename ViewT> index_type
subblock(ViewT const &view) VSIP_NOTHROW;
```

Effects:
Returns the subblock of view view held by the local processor, or no_subblock if the local processor does not hold a subblock.

Notes:
Equivalent to view.block().map().subblock().

13
```
template <typename ViewT> index_type
subblock_from_global_index(ViewT const &view, Index<ViewT::dim> const &g_idx) VSIP_NOTHROW;
```

Requires:
g_idx to be a valid global index of view view (0 <= g_idx[d] < view.size(d) for each d such that 0 <= d < ViewT::dim).

Effects:
Returns the subblock sb containing global index g_idx. g_idx is in global_domain(view, sb, p) where p = patch_from_global_index(view, g_idx).

14
```
template <typename ViewT> index_type
patch_from_global_index(ViewT const &view, Index<ViewT::dim> const &g_idx) VSIP_NOTHROW;
```

Requires:
g_idx to be a valid global index of view view (0 <= g_idx[d] < view.size(d) for each d such that 0 <= d < ViewT::dim).

Effects:
Returns the patch p containing global index g_idx. g_idx is in global_domain(view, sb, p) where sb = subblock_from_global_index(view, g_idx).

15
```
template <typename ViewT> Index<ViewT::dim>
local_from_global_index(ViewT const &view, Index<ViewT::dim> const &g_idx) VSIP_NOTHROW;
```

Requires:
g_idx to be a valid global index of view view (0 <= g_idx[d] < view.size(d) for each d such that 0 <= d < ViewT::dim).

Effects:

Returns the index l_idx into the local view that corresponds to the value at global index g_idx in the global view.

The local index l_idx is valid in the local view of the subblock holding global index g_idx, as determined by subblock_from_global_index.

16
```
template <typename ViewT> index_type
local_from_global_index(ViewT const &view, dimension_type d, index_type g_i) VSIP_NOTHROW;
```

Requires:

d to be a valid dimension view view (0 <= d < ViewT::dim).

g_i to be a valid index of dimension d of view view (0 <= g_i < view.size(d)).

Effects:

Returns the local dimension-d index l_i in the local view corresponding to global dimension-d index i.

Notes:

If g_idx = $(g_0, \cdots, g_{D-1})$, l_idx = $(l_0, \cdots, l_{D-1})$, and l_idx = local_from_global_index(view, g_idx), then for all 0 <= d < D , l_d = local_from_global_index(view, d, g_d).

17
```
template <typename ViewT> Index<ViewT::dim>
global_from_local_index(ViewT const &view, index_type sb Index<ViewT::dim> const &l_idx)
  VSIP_NOTHROW;
```

Requires:

l_idx to be a valid local index of subblock sb of view view (0 <= l_idx[d] < subblock_domain(view, sb)[d].size() for each d such that 0 <= d < ViewT::dim).

Effects:

Returns the global index g_idx corresponding to the local index l_idx in the view of the processor holding subblock sb.

18
```
template <typename ViewT> Index<ViewT::dim>
global_from_local_index(ViewT const &view, Index<ViewT::dim> const &l_idx) VSIP_NOTHROW;
```

Requires:

l_idx to be a valid local index of the subblock of view view held on the local processor. (0 <= l_idx[d] < subblock_domain(view)[d].size() for each d such that 0 <= d < ViewT::dim).

Effects:

Returns the global index g_idx corresponding to the local index l_idx in the local view on the local processor.

Notes:

Equivalent to global_from_local_index(view, subblock(view), idx)

19
```
template <typename ViewT> index_type
global_from_local_index(ViewT const &view, dimension_type d, index_type l_i) VSIP_NOTHROW;
```

Requires:

d to be a valid dimension view view (0 <= d < ViewT::dim). l_i to be a valid local index of dimension d of view view subblock on local processor (0 <= l_i < subblock_domain(view)[d].size()).

Effects:

Returns the global dimension-d index g_i corresponding to the local dimension-d index i in the local view.

Notes:

If g_idx = $(g_0, \cdots, g_{D-1})$, l_idx = $(l_0, \cdots, l_{D-1})$, and g_idx = global_from_local_index(view, l_idx), then for all 0 <= d < D, g_d = global_from_local_view(view, d, l_d).

6. Data Parallel Programs [dpp]

6.1. Definitions [dpp.defn]

1 A *local view* is stored on the local processor and visible only to the local processor. Views of blocks with map type Local_map and other map types specified by the implementation are local views.

2 A *distributed view* is globally visible to all processors executing a data-parallel VSIPL++ program. It represents data that is distributed over one or more processors. Views of blocks with map type Map, Replicated_map, and other map types specified by the implementation are distributed views.

Implementations will specify their level of support (0 through 3) for distributed views as indicated in [dpp.distlevel].

3 Subviews of a view retain the same local/distributed status as the original view.

4 For distributed views, the local() member function returns the local subblock view. This is a local view to the distributed view's subblock stored on the local processor.

5 For local views, the local() member function returns the view itself.

6 An assignment operation is one that assigns a new value to either a scalar variable or a view variable (with multiple element values) that is the result of performing a computation on one or more scalar and view values.

7 A local assignment is one where all views in the assignment (both source and destination) are local.

8 A distributed assignment is one where all views in the assignment (both source and destination) are distributed.

Implementations will specify their level of support (0 through 2) for distributed operations as indicated in [dpp.oplevel].

9 Assignments must not mix both local and distributed views, with the following exception: a distributed view may be used in an expression with local views if its local subblock view is being accessed via the local() member function.

10 The processor set of a distributed assignment is the union of processor sets for the maps of the distributed views in the assignment.

11 A distributed assignment is a *valid data-parallel assignment* if all processors in the assignment's processor set execute the assignment with the same *context*, where context means all local variables used to index an element of a distributed view, used to specify a subview of a distributed view, or used as a value affecting the result of an expression with distributed views.

This includes, but is not limited to, indices used for get/put element-wise access of distributed views, indices used to select sub-dimensional subviews of distributed views, domains used to select same-dimensional subviews of distributed views, and scalar values used in scalar-view element-wise expressions.

It is the library user's responsibility to ensure that all processors have the same context. Otherwise, behavior is undefined.

12 A program is said to be a *valid data-parallel program* if it meets the following conditions:

- Each assignment performed by the program is either a valid data-parallel assignment or a local assignment.
- For each subset of processors S in the program's global processor set, all valid data-parallel assignments with processor sets that have a non-empty intersection with S are executed in the same order by all processors in the set S.

13 The result of a valid data-parallel program is independent of the mappings applied to its distributed views, modulo precision differences.

6.2. Distributed Data Support Levels [dpp.distlevel]

1 For distribution of data, implementations shall provide one of the following support levels:

Level 0:
: Distribution of data is not supported (not a parallel implementation).

Level 1:
: One dimension of data may be block distributed.

Level 2:
: Any and all dimensions of data may have block distributions.

Level 3:
: Any and all dimensions of data may have block distributions or cyclic distributions.

2 [*Note:* Each level is a superset of the previous level.]

6.3. Distributed Operation Support Levels [dpp.oplevel]

1 For distributed operation types, implementations shall provide one of the following support levels:

Level 0:
: No distributed operations supported (not a parallel implementation).

Level 1:
: Distributed element-wise operations supported.

Level 2:
: Distributed element-wise operations supported (same as level 1) and selected distributed non-elementwise operations supported as stated by the implementation.

2 Implementation providing level 1 or level 2 support for distributed operation types shall also define distributed data locality support level:

Level 1:
: All data required to perform the operation must be on the local processor.

Level 2:
: Data required to perform the operation can be located on any processor.

3 [*Note:* Each level is a superset of the previous level.]

<table>
<tr><th colspan="3">Revision History</th></tr>
<tr><td>Revision 1.0</td><td>Approved: 2006-04-07</td><td></td></tr>
<tr><td colspan="3">Initial VSIPL++ parallel specification.</td></tr>
<tr><td>Revision 1.1</td><td>Approved: 2011-10-08</td><td></td></tr>
<tr><td colspan="3">• Change the document title.
• [map.replicatedmap.accessors] [map.replicatedmap.gridfn] : Fix Replicated_map accessor semantics (objects may be replicated only on some processors).
• Split [view] into [block] and [view].
• [support.types] : Relax restrictions on processor_type and remove superfluous processor_difference_type definition.</td></tr>
<tr><td>Revision 1.2</td><td>Approved: 2012-12-10</td><td></td></tr>
<tr><td colspan="3">• Various formatting changes are applied as part of the adoption of the specification by the Object Management Group.</td></tr>
</table>