

VSIPL++ Parallel Specification¹

1.1

Mentor Graphics

Publication date: 2012-03-15

¹This specification developed under subcontract 601-02-S-0109 under U.S. Government contract F30602-00-D-0221.

© 2005 by Georgia Tech Research Corporation. All rights reserved

© 2011 by Mentor Graphics. All rights reserved

A non-exclusive, non-royalty bearing license is hereby granted to all persons to copy, distribute, and produce derivative works for any purpose, provided that this copyright notice and following disclaimer appear on all copies:

THIS LICENSE INCLUDES NO WARRANTIES, EXPRESSED OR IMPLIED, WHETHER ORAL OR WRITTEN, WITH RESPECT TO THE SOFTWARE OR OTHER MATERIAL INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OR MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF PERFORMANCE OR DEALING, OR FROM USAGE OR TRADE, OR OF NON-INFRINGEMENT OF ANY PATENTS OF THIRD PARTIES. THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY OF THE ABOVE PARTIES.

The U.S. Government has a license under these copyrights, and this material may be reproduced by or for the U.S. Government.

Published 2011.

Printed in the United States of America

11 10 09 08 07 06 05 04 5 4 3 2 1

Table of Contents

Foreword	vi
1. intro	1
1.1. intro.scope	1
1.2. intro.refs	1
2. support	2
2.1. support.types	2
2.1.1. support.types.par	2
2.2. support.functions	2
2.3. support.constants	3
3. map	4
3.1. map.map	4
3.2. map.blockcyclic	5
3.2.1. map.blockcyclic.distribute	6
3.2.2. map.blockcyclic.blockdistribution	6
3.2.3. map.blockcyclic.cyclicdistribution	7
3.2.4. map.blockcyclic.wholedistribution	8
3.2.5. map.blockcyclic.mapclass	8
3.2.6. map.blockcyclic.template	9
3.2.7. map.blockcyclic.constructors	9
3.2.8. map.blockcyclic.accessors	10
3.2.9. map.blockcyclic.gridfn	10
3.3. map.localmap	11
3.3.1. map.localmap.constructors	11
3.3.2. map.localmap.accessors	11
3.3.3. map.localmap.gridfn	12
3.4. map.replicatedmap	13
3.4.1. map.replicatedmap.template	13
3.4.2. map.replicatedmap.constructors	13
3.4.3. map.replicatedmap.accessors	13
3.4.4. map.replicatedmap.gridfn	14
4. block	15
4.1. block.dense	15
4.1.1. block.dense.constructors	15
4.1.2. block.dense.userdata	16
5. view	21
5.1. view.view	21
5.2. view.vector	21
5.2.1. view.vector.types	21
5.2.2. view.vector.constructors	22
5.2.3. view.vector.accessors	22
5.3. view.matrix	22
5.3.1. view.matrix.types	22
5.3.2. view.matrix.constructors	23
5.3.3. view.matrix.accessors	23
5.4. view.tensor	23
5.4.1. view.tensor.types	23
5.4.2. view.tensor.constructors	24
5.4.3. view.tensor.accessors	24
5.5. view.support	24
5.5.1. view.support.defn	24
5.5.2. view.support.fcn	25

6. dpp	29
6.1. dpp.defn	29
6.2. dpp.distlevel	30
6.3. dpp.oplevel	30
7. changes	31

List of Tables

3.1. Map requirements 4
5.1. Parallel view requirements 21

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++ Specification, 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 for more information on VSIPL++ and for reference implementations.

1.1. *Scope*

[intro.scope]

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

1.2. *Normative references*

[intro.refs]

1 This document “VSIPPL++ Parallel Specification (1.1)” is part of the VSIPPL++ Specification and is incorporated via reference into the “VSIPPL++ Specification (1.1)”.

2 [*Note:* This document and the other VSIPPL++ specification documents use the same section notation even though the chapter and section 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
<code>m.num_subblocks()</code>	<code>length_type</code>	
<code>m.num_processors()</code>	<code>length_type</code>	
<code>m.processor_set()</code>	<code>const_Vector< processor_type, implementation-defined></code>	
<code>m.distribution(d)</code>	<code>distribution_type</code>	<code>d < Dim</code>
<code>m.num_subblocks(d)</code>	<code>length_type</code>	pre: <code>d < Dim</code> post: greater than zero
<code>m.cyclic_contiguity(d)</code>	<code>length_type</code>	<code>d < Dim</code>
<code>m.subblock()</code>	<code>index_type</code>	
<code>m.subblock(pr)</code>	<code>index_type</code>	<code>pr</code> in <code>processor_set</code>
<code>M::processor_iterator</code>	<i>unspecified</i>	

<code>m.processors_begin(sb)</code>	<code>processor_iterator</code>	$0 \leq sb < \text{num_subblocks}()$
<code>m.processors_end(sb)</code>	<code>processor_iterator</code>	$0 \leq sb < \text{num_subblocks}()$

- 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 of 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 i th constructor data distribution object when $i \geq 0$ && $i < \text{VSIP_MAX_DIMENSION}$.

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

Requires: `d < VSIP_MAX_DIMENSION`.

Returns: The distribution type of the d th 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 d th 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 $D_{\max} == \text{VSIP_MAX_DIMENSION}$

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: $d < \text{VSIP_MAX_DIMENSION}$.

Returns: whole.

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

Requires: $d < \text{VSIP_MAX_DIMENSION}$.

Returns: 1.

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

Requires: $d < \text{VSIP_MAX_DIMENSION}$.

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 `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: $d < \text{VSIP_MAX_DIMENSION}$.

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: $d < \text{VSIP_MAX_DIMENSION}$.

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).

4.1. 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 $0 \leq i < \text{subblock_domain.size}()$, $\text{pointer}[i] = T()$ 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 $0, \dots, \text{dom.size}()-1$. If $D \neq 1$, `*this` will have two domains: `Domain<1>` `domain1` with `Index<1>s` containing $0, \dots, \text{dom.size}()-1$ and a `Domain<D>` `domainD` with, for each $0 \leq d < D$, `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` ($T = \text{complex}<uT>$). For all i such that $0 \leq i < 2 * \text{subblock_domain.size}()$, $\text{pointer}[i] = uT()$ 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 $0, \dots, \text{dom.size}()-1$. If $D \neq 1$, `*this` will have two domains: `Domain<1>` `domain1` with `Index<1>es` containing $0, \dots, \text{dom.size}()-1$ and a `Domain<D>` `domainD` with, for each $0 \leq d < D$, `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 $0 \leq i < \text{subblock_domain.size}()$, `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 $0, \dots, \text{dom.size}()-1$. If $D \neq 1$, `*this` will have two domains: `Domain<1>` domain1 with `Index<1>`es containing $0, \dots, \text{dom.size}()-1$ and a `Domain<D>` domainD with, for each $0 \leq d < D$, `domainD[d].size() == \text{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.
- $l_idx = (l_0, \dots, l_{D-1})$ refer to a local index of the subblock of `*this` stored on the local processor, where $0 \leq l_d < \text{subblock_domain}[d].\text{size}()$ for each dimension d of block `*this`.
- $g_idx = (g_0, \dots, g_{D-1})$ 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 $d_0 \dots d_{D-1}$ of block `*this`:

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

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 $g_idx = (g_{0'} \dots, g_{D-1})$: `this->get($g_{0'} \dots, g_{D-1}$) == 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 $g_idx = (g_{0'} \dots, g_{D-1})$: `this->get($g_{0'} \dots, g_{D-1}$) == 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 $g_idx = (g_{0'} \dots, g_{D-1})$: `this->get($g_{0'} \dots, g_{D-1}$) == 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($g_{0'} \dots, g_{D-1}$)` for all $g_idx = (g_{0'} \dots, g_{D-1})$ such that $0 \leq g_{d'} < \text{this->size}(D, d)$, 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 $g_idx = (g_{0'} \dots, g_{D-1})$: `this->get($g_{0'} \dots, g_{D-1}$) == 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 $g_idx = (g_{0'} \dots, g_{D-1})$: `this->get($g_{0'} \dots, g_{D-1}$) == 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 $g_idx = (g_{0'} \dots, g_{D-1})$: `this->get($g_{0'} \dots, g_{D-1}$) == 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 = $(g_0 \dots, g_{D-1})$: this->get($g_0 \dots, g_{D-1}$) == 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 = $(g_0 \dots, g_{D-1})$: this->get($g_0 \dots, g_{D-1}$) == 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, ..., g_{D-1})`: `this->get(g_0, ..., 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, ..., g_{D-1})`: `this->get(g_0, ..., 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`.

5.1. 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 ($0 \leq p < \text{num_patches}(v)$).

Table 5.1. Parallel view requirements

expression	return type
<code>V::local_type</code>	implementation-defined
<code>V::local_patch_type</code>	implementation-defined
<code>v.local()</code>	<code>V::local_type</code>
<code>v.local(p)</code>	<code>V::local_patch_type</code>

- 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 ($0 \leq \text{patch} < \text{num_patches}(\text{view})$).

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 VSIP++ 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 $g_idx = (g_{0'} \dots, g_{D-1'})$ is a global index in the domain of view (for each $0 \leq d < D$, $0 \leq i_d < \text{view.size}(d)$), then there exists a subblock-patch pair sb-p whose global domain gdom contains index $(g_{0'} \dots, g_{D-1'})$.
- 9 Moreover, if $(p_{0'} \dots, p_{D-1'})$ are defined such that $g_d = \text{gdom}[d].\text{first}() + p_d * \text{gdom}[d].\text{stride}()$, then $l_idx = (l_{0'} \dots, l_{D-1'})$, where $l_d = \text{ldom}[d].\text{first}() + p_d * \text{ldom}[d].\text{stride}()$ 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 $(g_{0'} \dots, g_{D-1'})$ in the view view.
- 10 The following relationships hold on all processors: $sb == \text{subblock_from_global_index}(\text{view}, g_idx), p == \text{patch_from_global_index}(\text{view}, g_idx), \text{gdom} == \text{global_domain}(\text{view}, sb, p), \text{ldom} == \text{local_domain}(\text{view}, sb, p), l_idx == \text{local_from_global_index}(\text{view}, g_idx), g_idx == \text{global_from_local_index}(\text{view}, sb, l_idx), l_d == \text{local_from_global_index}(\text{view}, d, g_d)$, and $g_d == \text{global_from_local_index}(\text{view}, d, sb, l_d)$.
- 11 The following relationships hold on processors owning subblock sb (i.e. processors where $sb == \text{view.block().map().subblock}()$): $\text{local_view} == \text{view.local}(), \text{local_view.get}(l_{0'} \dots, l_{D-1'}) == \text{view.get}(g_{0'} \dots, g_{D-1'})$, $\text{gdom} == \text{global_domain}(\text{view}, p), \text{ldom} == \text{local_domain}(\text{view}, p)$, and $g_d == \text{global_from_local_index}(\text{view}, d, l_d)$.

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 ($0 \leq sb < \text{num_subblocks}(\text{view})$) 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 ($0 \leq sb < \text{num_subblocks}(\text{view})$) or no_subblock. p to be a valid patch of view's subblock sb ($0 \leq p < \text{num_patches}(\text{view}, sb)$).

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 ($0 \leq p < \text{num_patches}(\text{view}, \text{subblock}(\text{view}))$).

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 ($0 \leq sb < \text{num_subblocks}(\text{view})$) or no_subblock. p to be a valid patch of view's subblock sb ($0 \leq p < \text{num_patches}(\text{view}, sb)$).

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 ($0 \leq p < \text{num_patches}(\text{view}, \text{subblock}(\text{view}))$).

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 ($0 \leq sb < \text{num_subblocks}(\text{view})$) 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` ($0 \leq g_idx[d] < view.size(d)$ for each d such that $0 \leq d < ViewT::dim$).

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` ($0 \leq g_idx[d] < view.size(d)$ for each d such that $0 \leq d < ViewT::dim$).

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` ($0 \leq g_idx[d] < view.size(d)$ for each d such that $0 \leq d < ViewT::dim$).

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` ($0 \leq d < ViewT::dim$).

`g_i` to be a valid index of dimension `d` of view `view` ($0 \leq g_i < view.size(d)$).

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

Notes: If $g_idx = (g_{0'} \dots, g_{D-1'})$, $l_idx = (l_{0'} \dots, l_{D-1'})$, and $l_idx = \text{local_from_global_index}(\text{view}, g_idx)$, then for all $0 \leq d < D$, $l_{d'} = \text{local_from_global_index}(\text{view}, d, g_{d'})$.

```
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 ($0 \leq l_idx[d] < \text{subblock_domain}(\text{view}, sb)[d].\text{size}()$ for each d such that $0 \leq d < \text{ViewT::dim}$).

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. ($0 \leq l_idx[d] < \text{subblock_domain}(\text{view})[d].\text{size}()$ for each d such that $0 \leq d < \text{ViewT::dim}$).

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 $\text{global_from_local_index}(\text{view}, \text{subblock}(\text{view}), \text{idx})$

```
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 ($0 \leq d < \text{ViewT::dim}$). l_i to be a valid local index of dimension d of view view subblock on local processor ($0 \leq l_i < \text{subblock_domain}(\text{view})[d].\text{size}()$).

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

Notes: If $g_idx = (g_{0'} \dots, g_{D-1'})$, $l_idx = (l_{0'} \dots, l_{D-1'})$, and $g_idx = \text{global_from_local_index}(\text{view}, l_idx)$, then for all $0 \leq d < D$, $g_{d'} = \text{global_from_local_view}(\text{view}, d, l_{d'})$.

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.]

Revision History		
Revision 1.0	Approved: 2006-04-07	
Initial VSIPL++ parallel specification.		
Revision 1.1	Approved: 2011-10-08	
<ul style="list-style-type: none">• 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.		