

## Developer's Guide

# Table of Contents

## [Quick Start](#)

- [Installation](#)
- [Usage](#)
- [Misc knobs](#)

## [Overview](#)

- [Delayed Execution](#)
- [Parallelism Extraction](#)
- [Multi-Process Scheduling \(MPI\)](#)

## [Containers](#)

- [Vector](#)
- [Partitioned Vector](#)
- [Iterators](#)
- [Future and Atomic](#)
- [Matrix](#)
- [Diagonal Matrix](#)
- [Tiled Matrix](#)
- [Custom Types](#)

## [Algorithms](#)

- [Partitioned Vector Algorithms](#)
- [Matrix Algorithms](#)
- [Tiled Matrix Algorithms](#)

# Quick Start

## Installation

Ambient is a header-only library. This means that installing the sources somewhere is enough:

```
cmake .  
make install
```

**Note:** commands above will install the sources to /opt/ambient (use -DCMAKE\_INSTALL\_PREFIX=<path> to change it).

### Numeric dependencies

Ambient itself doesn't have any requirements except for having C++14 compatible compiler (i.e. -std=c++14 flag) but Ambient's numeric::matrix class has a few dependencies and it's the right place to make sure they are satisfied if you are going to use it:

```
mkdir _build && cd _build  
cmake -DAMBIENT_REGRESSION_TESTS=ON ..  
make  
make test  
make install
```

Primarily, you need to make sure that [Intel® MKL](#) or compatible math package is present (note: in a few algorithms we directly depend upon [Plasma library](#) so it might be needed as well).

**Note:** Use -DMKL\_LIB\_DIR, -DMKL\_LIBRARIES to specify the math library and -DPLASMA\_LIB\_DIR, -DPLASMA\_LIBRARIES to specify Plasma library.

The configuration being used will be saved to <install path>/share/AmbientConfig.cmake for your convenience.

# Usage

## Compilation

Let's compile your application against Ambient sources. In the simplest form it's as easy as this:

```
mpicxx -std=c++14 -O3 -DNDEBUG -I/opt/ambient/include app.cpp
```

In example above we are using the default settings: MPI is assumed to be used in MPI\_THREAD\_FUNNELED mode and threading backend is determined from the compiler being used (Cilk in case of icc, OpenMP in case of gcc and none otherwise).

Otherwise, it's easy to choose an exact flavor:

- use -DAMBIENT\_MPI to select MPI mode (MPI\_DISABLE or MPI threading mode are accepted)
- and -DAMBIENT\_THREADING to select the threading backend (CILK, OPENMP or SERIAL)

For example below we are going to use Ambient without MPI (as shared-memory-only) and with OpenMP instead of Cilk:

```
icpc -std=c++14 -O3 -DNDEBUG -I/opt/ambient/include app.cpp -DAMBIENT_MPI=MPI_DISABLE  
-DAMBIENT_THREADING=OPENMP -openmp
```

## Compilation: numeric dependencies

In case of numeric::matrix class being used, one will have to link against the Math/Plasma libraries (which means listing the libraries to link against and their paths). The exact flags will depend upon the math library but in case of Intel® MKL the flags can be calculated using the [link line advisor](#) (shortcut for icpc: -mkl=sequential).

**Important:** choose sequential implementation to avoid oversubscribing unless you know what you are doing (see [mkl\\_parallel.hpp](#) utility).

If Plasma-based algorithms are needed - one will have to link against some of its libraries, typically these are "-lplasma -lcoreblas -llapacke -lquark".

The respective libraries includes are generally *not* needed as Ambient already contains declarations of the functions it actually needs.

It might be a good idea to finalise the configuration using CMake during the installation and reuse the saved configuration.

## Execution

When built, the application can be executed as usual: mpiexec/mpirun as any other MPI application or "as is" when shared-memory-only.

To customise the runtime configuration just use the threading backend environment variables (i.e. CILK\_NWORKERS or OMP\_NUM\_THREADS). Then the actual configuration (MPI, threading and other parameters) can be checked by setting the AMBIENT\_VERBOSE environment variable prior to running an executable.

We have included a few examples at share/examples for you to be able to start right away. Otherwise, be sure to read the [overview](#) section of this document to understand the model behind the framework.

# Misc knobs

To get you fully packed - here's a list of optional knobs that might be useful:

## Compilation flags:

- DAMBIENT\_DEFAULT\_IB : default blocking factor of tiled containers (2048)
- DAMBIENT\_INSTR\_BULK\_CHUNK : number of bytes in the chunk of bulk memory used for instructions logging (~16MB)
- DAMBIENT\_DATA\_BULK\_CHUNK : number of bytes in the chunk of bulk memory used for temporary objects (~64MB)
- DAMBIENT\_COMM\_BULK\_CHUNK : number of bytes in the chunk of bulk memory used for communications (~64MB)

**Note:** comm bulk chunk should fit any memory-continuous object being sent to other nodes

- DAMBIENT\_CHECK\_BOUNDARIES : enables boundary overflow checks for spatial memory copies (disabled)
- DAMBIENT\_SERIAL\_COLLECTION : strip thread-safety in operations collection (disabled)

## Environment flags:

export AMBIENT\_DATA\_BULK\_LIMIT=P : limit the data bulk's memory consumption by P percents of total memory (10)

export AMBIENT\_COMM\_BULK\_LIMIT=P : limit the communication bulk's memory consumption by P percents of total memory (20)

export AMBIENT\_MKL\_PARALLEL=1 : to enable mkl\_parallel utility (if [mkl\\_parallel.hpp](#) is included)

# Overview

This section focuses on explanation of the general concepts behind the Ambient framework. The core concept to keep in mind (that is connected to the rest of the framework's mechanics) is the "object versioning". In practice it means a slight change in how one reasons about classical object oriented programming. In contrast to the classical programming approach where the object's memory location is fixed (unless one reallocates its data) and frequently modified by means of some computational operations - the Ambient enabled code keeps objects memory conceptually read-only: every modification instead creates a new separate *version* of the object in memory. (Note: In practice that would mean a really large memory-footprint - so for the classical hardware unnecessary states are optimised away keeping memory footprint on a par with its classical analogue).

In the following sub-sections we will focus on how this concept benefits us from the point of view of application parallelisation, providing the reader with the key technical details about the implementation.

## Delayed Execution

In order to make implicit parallelisation possible, Ambient implements the concept of the delayed execution. When one calls a computational operation on an object - instead of executing it right away (like it would happen in classical programming) - it gets saved for the later execution when more operations are available to be executed in parallel.

Before discussing caveats of delayed execution implementation, let's see some syntax sugar. The developer writing his own functions operating on Ambient-enabled data-types has a choice between explicit and implicit call forms:

- The explicit form is fairly similar to `std::async` call:

```
template < class Function, class... Args >
void bind( Function&& f, Args&&... args );
```

This form supports both classical functions and lambdas. So typical invocation would look like this:

```
ambient::vector<int> vec;
ambient::bind([](ambient::vector<int>& v){ /* */ }, vec);
```

- The implicit form is designed to keep code appearance close to its natural form: one just has to export the target function under some alias name which can be called then the same way as the original function:

```
AMBIENT_STATIC_BIND_TEMPLATE(foo, foo_async);
```

or for not templated functions:

```
AMBIENT_STATIC_BIND(foo, foo_async);
```

The key difference comparing to `std` style here is that Ambient-enabled function calls don't return any handles or objects. Instead, an explicit synchronisation (i.e. `ambient::sync()` call) or the object's element access will trigger the actual computations.

When the function call is executed it does a number of specific actions in order to be able to execute this function later in parallel:

- During the compilation time, the function prototype is examined. If the function argument is not

marked as const, Ambient will create a new revision of the object that was passed expecting modification of the object. Otherwise (if const is present) the current revision will be used.

- Due to the limited life-time of C++ objects, Ambient will copy the non-Ambient data-types (i.e. int, size\_t) into its own heap-based memory (Ambient enabled data-types are protected against out-of-scope access since their memory is managed by Ambient).
- Finally the function will be placed into a graph of operations for the latter execution (keeping the sequential order of data modification intact due to dependency tree of the version control system).

## Parallelism Extraction

### Task parallelism

Now that functions are executed in a delayed manner - the whole program execution transforms into logging of which operations should be executed (alongside with corresponding objects versions). That is at any given moment we have a directed acyclic graph of the operations ready for parallel traversal.

The version control plays a critical role here:

- It makes sure that the execution order of the operations is correct by tracking whether the needed objects versions are already produced (hence making application parallel execution reproducible and eliminating possibility of race conditions).
- And, second, what's especially important from the parallelism point of view, version control allows a much higher degree of parallelism since we can deal with multiple versions of an object at the same time.

Consider an example where one uses a matrix N times for the read operations, then modifies it and uses it another N times for other read operations. Classically only N operations can be executed in parallel while Ambient enabled code will have no problem performing 2N operations in parallel using two revisions of the matrix at the same time. Similarly Ambient makes it easy to implicitly extract parallelism from any target application algorithm.

Once the operations set is sufficiently large, one either accesses the container's memory or explicitly calls `ambient::sync()` to trigger the actual execution.

At that moment Ambient will start parallel traversing of the collected graph using Cilk, OpenMP or other threading libraries to schedule the functions execution (tasks) across different threads.

### Data parallelism

The aforesaid concerns dominantly the task parallelism. That is the described model is effective when different operations are working on different data in parallel. But it doesn't help much with data parallelism when the same operation needs to be applied to multiple data in parallel. So specifically for the problems requiring data parallelism we have designed a set of containers that keep the data in a partitioned form. Hence allowing task parallelism to be used for data parallelism.

An example of such container would be the matrix [tiles class](#): it represents a set of square matrices that compose the original matrix (where each matrix-tile is an Ambient-enabled data-type with versioning semantics). So when one calls a matrix multiplication operation - it will be decomposed into series of matrix multiplications amongst tiles composing the original matrix. Hence providing efficient data parallelism even for a single algebraic operation.

Similarly a more general purpose container would be a [partitioned vector class](#). Analogously to tiles class, it deals with the original vector as a set of vector chunks. This container already supports a subset of STL

algorithms and is open for extension.

**Note:** Ambient doesn't prevent one from using the parallelisation capabilities of the underlying threading framework inside Ambient's bound calls.

An example of such use would be calls to parallel variants of [Intel® Math Kernel Library](#) functions inside bound calls (see the README.md for the list of the corresponding environment variables).

## Multi-Process Scheduling (MPI)

Classically, in order to run an application on a cluster, one normally has to resort to low-level MPI parallelisation where one has to explicitly write on which MPI rank the code should be executed and transfer back and forth the necessary data via send/receive calls. Thus polluting the source code with transfers code which is usually extremely complicated by itself.

In contrast, Ambient was designed from the start to tackle the problems that come with the distributed-memory execution model:

**First** of all, we abstracted away the need to write the transfers code: when the operation is submitted for the execution Ambient already knows what to do since it has a DAG of all operations. So at the time of evaluating function call's arguments Ambient will make sure to issue asynchronous MPI calls to transfer the required data to the target nodes.

Versioning of the objects plays an important role here as well. Since the revision (once its produced) cannot be modified, we can safely issue a transfer without caring about modifications of the memory that needs to be transfered. Plus, due to the fact that we already know which operations will require this version of data, we can construct a binary tree of the nodes to speedup the transfer automatically.

**Second**, when it comes to the question "which nodes should execute this operation" - we do the scheduling. Ambient evaluates the node-affinity of the current versions of the arguments (and their memory footprints) then selecting the optimal operation destination.

While in the first version of the framework the scheduling was completely automatic, we found out that automatic scheduling frequently produces sub-optimal operations distribution when looking at application's execution as a whole. Which, in contrast with shared-memory execution model, enforces a huge penalty on application's performance. Hence we decided to give developers the freedom to schedule parts of the application execution as they see fit: explicitly scheduled regions of the code will always abide the developer's intentions and the unmarked regions will then automatically deduce the node-affinity.

Here's how it works. In terms of language mechanics, Ambient blends in with the C++ scope concept: one has to create an instance of `ambient::actor` that will act on the current C++ scope (all operations between actor's construction and destruction will be executed on the respective node).

```
ambient::partitioned_vector<ambient::vector<int> > a(length);
{
    ambient::actor guard(ambient::scope::begin());
    ambient::sequence(a.begin(), a.end());
}
```

In the example above one can see that we have placed the operations inside brackets (`ambient::sequence`) to be executed at the first node inside of the `ambient::scope`. The latter is the way how one treats different application's processes in Ambient. Unlike MPI that uses directly node ranks - Ambient works with `ambient::scope` iterators that in turn would point to some specific ranks (depending upon the `ambient::scope` being used).

In turn, `ambient::scope` behaviour is similar to the stack model: at any given moment one can access only the inner-most scope that was created from the parent scope's iterators (the outer-most scope already contains all of the processes). Such design enforces code flexibility, while abstracting functions from the run-time



process configuration, which can be dynamic in case of fault tolerance.

Here's a brief example to give an idea of how to use the actors/scopes.

```
void generate(ambient::partitioned_vector<ambient::vector<int> >& v, int value){  
    // iterating through the given scope's processes  
    for(auto it = ambient::scope::begin(); it != ambient::scope::end(); it++){  
        ambient::actor guard(it);  
        v.partition(it - ambient::scope::begin()).init(value);  
    }  
}  
  
// using half of the processes available (NP/2)  
ambient::scope group(ambient::scope::begin(), ambient::scope::size()/2);  
// creating vector with NP/2 partitions  
ambient::partitioned_vector<ambient::vector<int>, N> a(N*ambient::scope::size());  
// calling function inside the scope of NP/2 processes  
generate(a, 13);
```

# Containers

## Vector

Ambient vector is a general purpose vector similar to `std::vector`. This class main purpose is to be used as a container data-type in [partitioned\\_vector class](#). This class has versioning semantics and can be directly used with custom asynchronous functions.

**Important:** in order to provide functionality similar to `std::vector` this class has polymorphic behaviour: methods called while in asynchronous mode will behave differently than during serial execution. The need to do that comes from the memory reallocation policy: while in asynchronous mode one can't change vector's capacity and vice versa - the actual vector size (if changed) is not known before execution is complete hence operating with "cached" size.

The good practice would be changing both vector size and capacity while in serial mode.

### Class Methods

(constructor)	constructs the vector (memory is allocated upon init)	<i>serial mode only</i>
operator =	assigns values to the container	<i>serial mode only</i>
init	fill the vector with some value (optional)	<i>serial mode only</i>
load	compute operations (enables element access when serial)	<i>serial mode only</i>

**Element access:** ("load" first if in serial mode)

at	access specified element with bounds checking
operator[]	access specified element
front	access the first element
back	access the last element
data	direct access to the underlying array
begin, cbegin	return an iterator to the beginning
end, cend	return an iterator to the end

### Capacity:

empty	checks whether the container is empty	
size	return the number of elements	
capacity	return the number of elements that can be held in currently allocated storage	
cached_size	return the <i>latest known</i> number of elements	
measure	measure and return the current number of elements	<i>serial mode only</i>
auto_reserve	reserves storage as needed	<i>serial mode only</i>

reserve	reserves storage	<i>serial mode only</i>
shrink_to_fit	reduces memory usage by freeing unused memory	<i>serial mode only</i>

#### Modifiers:

clear	clears the contents
insert	inserts one element
erase	erases one element
push_back	adds elements to the end
pop_back	removes the last element
resize	changes the number of elements stored
swap	swaps the contents

## Partitioned Vector

Unlike [ambient::vector](#) - `partitioned_vector` is a container that doesn't have versioning semantics but instead provides an easy way of using underlying container that does have versioning semantics (i.e. `ambient::vector`). Primarily, this is a class that one would use instead of `std::vector` to parallelise the target application. An object of this class would denote a vector that consists of multiple chunks that are stored independently in the distributed or shared memory.

### Class Methods

(constructor)	constructs the vector (memory is allocated upon first use)
(destructor)	deconstructs all the chunks
operator =	assigns values to the container

#### Element access:

operator[]	access specified element
begin, cbegin	return an iterator to the beginning
end, cend	return an iterator to the end

#### Partition access:

partition	access specified chunk
locate	return a chunk that contains specified element

#### Capacity:

empty	checks whether the container is empty
size	return the number of elements

#### Modifiers:

resize	changes the number of elements stored
normalize	resizes chunks to IB (overall content is unchanged)

# Iterators

There are two distinct ideas behind implementation of Ambient iterators:

- To abstract from the data-types partitioning in order to simplify algorithms development.
- Decouple iterators from the actual memory (since the latter might be remote or non-existent).

## **associated\_iterator**

As the name goes - this iterator class stores position information but routes dereferences through the associated object that owns this iterator.

Note: this iterator doesn't handle underlying partitioning.

### **Class Methods**

(constructor)	construct the iterator (using container and position)
operator ++	increase position by one
operator +=	increase iterator position
operator -=	decrease iterator position
operator *	access the actual element
get_container	return the container (owner)

## **block\_iterator**

This iterator deals with the partitioned-structure of the container. It's purpose is to iterate through the blocks in a range instead of elements (typically to spawn an asynchronous operation for each block).

### **Class Methods**

(constructor)	construct the iterator (using begin and end iterators)
operator ++	move to the next block
operator !=	check if position is different
operator *	return the current block
n_blocks	return the number of blocks in range
offset	return current position

## **block\_tuple\_iterator**

Similarly to `block_iterator`, this iterator iterates through the containers blocks. But in this case it cares about blocks mutual alignment in the objects of a tuple. That is it will iterate through maximally available memory-continuous regions (that are continuous in all objects of a tuple).

### **Class Methods**

(constructor)	construct the iterator (using begin iterators and region length)
operator ++	move to the next common region
operator !=	check if position is different
locate	return the current block (for the specified element in a tuple)

## block\_pair\_iterator

A trivial case of the block\_tuple\_iterator class (added for convenience): tuple contains only two objects with the same blocking factor.

### Class Methods

(constructor)	construct the iterator (using begin offsets and region length)
operator ++	move to the next common region
end	check if the end is reached

# Future and Atomic

## Future

Most of the time, after all of the heavy lifting is done using partitioned data-types, one has to get out with a single scalar (as an example one might consider vector reduction). Since all non-Ambient data-types are simply copied over for the bind calls - one cannot use a reference and expect it to store the output later after the computation. Instead a valid output scalar might have been a pointer but it won't be accessible by other processes.

Therefore the ambient::future data-type was introduced - Ambient will make sure to transfer its memory to all of the processes as soon as the value was calculated. But unlike other Ambient data-types - future doesn't support versioning - so it will trigger computations as soon as its value is accessed or used.

### Class Methods

(constructor)	construct the future (i.e. from the initial value)
(copy constructor)	copy construct the future
(destructor)	free the descriptor
operator =	swap or calculate and assign

### Access:

get	get the value (async only)
set	set the value (async only)
load	calculate and return the value (serial only)
operator T	same as load

## Atomic

While the `ambient::future` is appropriate to use most of the time - it might limit the degree of parallelism depending upon the application. For such applications we have added `ambient::atomic` that behaves exactly as other versioned data-types (except being specialised for a single scalar payload). This data-type eliminates the need to synchronize and broadcast its contents after an operation has finished but it imposes a bigger processing overhead comparing to `ambient::future`. *Use it with caution only if necessary.*

### Class Methods

(constructor)	construct the atomic (i.e. from the initial value)
(copy constructor)	copy construct the future
operator =	copy construct and swap
init	init atomic with the value
swap	swap the contents
<b>Access:</b>	
	("load" first if in serial mode)
get	get the value
set	set the value
load	calculate the value (serial only)

## Matrix

The Ambient matrix class main purpose is to be used as a container data-type in [tiled\\_matrix class](#). This class has versioning semantics and can be directly used with custom asynchronous functions. Unlike in multiple numeric libraries - this realisation doesn't have an LDA parameter - that is any resize of the matrix will create a new matrix instead with subsequent copy of the data. This has proven to be beneficial in terms of performance - the algorithms deal with non-strided memory faster (while resizing of the tiled matrix needs to alter only the border tiles).

### Class Methods

(constructor)	constructs the matrix (memory is allocated upon use)
(copy constructor)	copy constructs the matrix (the memory points to source matrix current version)
operator =	swap with copy-constructed matrix (if the type is the same)
num_rows	return the number of rows
num_cols	return the number of columns
addr	calculate the element's offset inside of the memory region
trace	return "future" of the matrix trace
<b>Modifiers:</b>	
swap	swap the matrices
resize	resize the matrix (swap with a new matrix + copy)
transpose	transpose the matrix (inplace)
conj	conjugate the matrix (inplace)

operator +=	add another matrix (inplace)
operator -=	subtract another matrix (inplace)
operator *=	multiply by scalar (inplace)
operator /=	divide by scalar (inplace)
<b>Element access:</b>	
	("load" first if in serial mode)
operator ()	access specified element
data()	direct access to the underlying array

## Diagonal Matrix

This class is a special case of the [matrix class](#) for the diagonal matrices. It uses matrix class with the number of columns set to one as a container.

### Class Methods

(constructor)	constructs the matrix (memory is allocated upon use)
num_rows	return the number of rows
num_cols	return the number of columns (= num_rows)
addr	calculate the element's offset inside of the memory region
get_data	return the underlying container
size	return the size of the matrix (= num_rows)
resize	resize the matrix (swap with a new matrix + copy)

### Element access:

operator ()	access specified element
operator []	access specified diagonal element

## Tiled Matrix

Unlike [ambient::matrix](#) - *tiles* class is a container that doesn't have versioning semantics but instead provides an easy way of using underlying container that does have versioning semantics (i.e. `ambient::matrix`). Primarily, this is a class that one would use to parallelise the target application. An object of this class would denote a matrix that consists of multiple tiles that are stored independently in the distributed or shared memory.

### Class Methods

subset	return matrix that is a subset of this matrix
identity_matrix	return identity matrix (only diagonal tiles are filled)
(constructor)	constructs the matrix (tiles memory is allocated upon use)
(copy constructor)	constructs the matrix (tiles are copy-constructed)
(destructor)	delete all the tiles
operator =	swap with copy-constructed matrix
num_rows	return the number of rows
num_cols	return the number of columns (= num_rows)
trace	return "future" of the matrix trace
empty	return if the size is zero
save	write matrix to hdf5 archive
load	read matrix from hdf5 archive

#### Modifiers:

swap	swap the contents
transpose	transpose the matrix (inplace)
conj	conjugate the matrix (inplace)
resize	resize the matrix
operator +=	add another matrix (inplace)
operator -=	subtract another matrix (inplace)
operator *=	multiply by scalar (inplace)
operator /=	divide by scalar (inplace)

#### Content access:

operator ()	access specified element
operator []	access specified tile (1D)
tile	access specified tile (2D)
locate	access specified element's tile
addr	get the element's offset inside its tile

## Custom types

When using Ambient, depending upon the application, it might be necessary to implement a custom problem-specific type. The easiest way to do that (keeping Ambient's versioning semantics) is to derive from `ambient::block` class. This class acts as a memory-block with a variable size (determined by constructor's parameters). Otherwise, nothing prevents one from implementing his own data-type from scratch by analogy with `ambient::block`.

### Block Class Methods

(constructor)	construct the column-major 2D-block (memory is allocated upon use)
init	fill the block with the value



lda	return the y-dimension of the block
data	direct access to the underlying array
operator ()	access specified element

# Algorithms

## Partitioned Vector Algorithms

copy	copies a range of elements to a new location
fill	assigns a range of elements a certain value
find	finds the first element satisfying specific criteria
for_each	applies a function to a range of elements
generate	saves the result of a function in a range
reduce	reduce a range of elements (i.e. sum up)
remove	removes elements satisfying specific criteria
replace	replaces all values satisfying specific criteria with another value
sequence	fills the range with a sequence of numbers
sort	sorts a range into ascending order (bitonic sort)
transform	applies a function to a range of elements
unique	removes consecutive duplicate elements in a range

## Matrix Algorithms

conj	return conjugate of the matrix
copy_rt, copy_lt	copy upper/lower triangular matrix
copy_block (_s, _sa)	copy/copy-scale/copy-scale-add a 2D-block
exp	compute matrix exponential
fill_identity, fill_random, fill_random_hermitian, fill_value	fill matrix with the corresponding values
geev, heev	compute eigenvalues and eigenvectors of the matrix
gemm, gemm_fma	perform matrix-matrix multiplication
geqrt (gelqt)	perform QR-factorization (or LQ)
inverse	return inverse of the matrix
is_hermitian	return if the matrix is hermitian
norm_square	calculate squared norm of the matrix

overlap	calculate dot product between elements of two matrices
scale	scale matrix elements
sqrt	square root of a diagonal matrix (inplace)
svd	perform singular value decomposition
trace	return trace of the matrix
transpose	return transpose_view of the matrix

## Tiled Matrix Algorithms

adjoint	return adjoint of the matrix
conj	return conjugate of the matrix
copy_block (_s, _sa)	copy/copy-scale/copy-scale-add a 2D-block
exp (_hermitian)	compute matrix exponential
geev, heev	compute eigenvalues and eigenvectors of the matrix
gemm	perform matrix-matrix multiplication
gemm_strassen	perform matrix-matrix multiplication (strassen algorithm)
generate (_hermitian)	generate matrix values
inverse	return inverse of the matrix
is_hermitian	return if the matrix is hermitian
merge	merge all tiles into one
norm_square	calculate squared norm of the matrix
overlap	calculate dot product between elements of two matrices
qr (lq)	perform QR-factorization (or LQ)
remove_cols (_rows)	remove columns (rows) of the matrix
split	split one matrix into IB-tiles
svd	perform singular value decomposition
sqrt	return square root of a diagonal matrix
trace	return trace of the matrix
transpose	return transpose_view of the matrix