DASHMM 2.0.0-alpha.0 User Guide

Jackson DeBuhr[1]
Bo Zhang [2]

[1]jdebuhr@indiana.edu
[2]zhang416@indiana.edu

Contents

Chapter 1

Introduction to DASHMM

The *Dynamic Adaptive System for Hierarchical Multipole Methods* (DASHMM) is a C++ library providing a general framework for computations using multipole methods. In addition to the flexibility to handle user-specified methods and expansions, DASHMM includes built-in methods and expansions, including the Barnes-Hut (BH) and Fast Multipole Method (FMM), and expansions implementing the Laplace, Yukawa and low-frequency Helmholtz kernels.

DASHMM is designed to make its adoption and use as easy as possible, and so the interface to DASHMM provides both an easy-to-use basic interface (see Chapter 3) and a more advanced interface (see Chapter 4). The basic interface allows a user to get up and running with multipole methods as quickly as possible. However, more advanced use-cases, especially for users that are implementing their own methods and expansions, will need to explore the advanced interface.

DASHMM is built using the advanced runtime system, HPX-5, but the basic, and much of the advanced, interface does not require any knowledge of how to use HPX-5 directly. Instead, DASHMM insulates users from the specific details of HPX-5, allowing expression of the multipole method application in higher level concepts than the specifics of threads and execution control structures. Nevertheless, it can be helpful to have a sense of the conceptual underpinnings of HPX-5, and how those relate to DASHMM. Information about this can be found in this guide in both the basic and advanced interface. Even more information can be found at the HPX-5 website (https://jacksondebuhr.github.io/dashmm/).

For a description of the techniques that go into the parallel execution of DASHMM, and for a discussion of the conceptual framework, please see the code paper: *"DASHMM: Dynamic Adaptive System for Hierarchical Multipole Methods"* in Communications in Computational Physics, Vol. 40 (2016), No. 4, pp. 1106-1126.

This document covers version 2.0.0-alpha.0 of DASHMM. For the latest news and updates of DASHMM, please visit the DASHMM website: https://www.crest.iu.edu/projects/dashmm/.

The DASHMM project has adopted semantic versioning (http://semver.org). Chapters 3 and 4 should be considered to be the specification of the interface to the library. However, please note that as this document is being fine-tuned, there is the possibility of the odd omission or typo. Every attempt will be made to make this as complete and correct as possible, but during the early life of the document, some errors are to be expected.

In the following, snippets of code or the names of code constructs will be set in a fixed width font. For example, `main()`. Unless otherwise indicated, every construct presented in this guide is a member of the `dashmm` namespace.

Chapter 2

Installing DASHMM

This chapter outlines what is needed to install and use of DASHMM, and gives a brief sketch of the demo programs included with DASHMM.

2.1 Prerequisites

DASHMM v. 2.0.0-alpha.0 depends on one external library: HPX-5. The current version of DASHMM depends on version 4.1.0 of HPX-5 or later, which may be found at https://hpx.crest.iu.edu/. Please see the official HPX-5 documentation for instructions on how to build, install and run HPX-5 on your system.

The DASHMM build system relies on the `pkg-config` utility to specify the needed HPX-5 compilation and linking options, so it is important to add the correct path for HPX-5 to your `PKG_CONFIG_PATH` environment variable. For example, assuming HPX-5 is installed in `/path/to/hpx`, this can be accomplished using `bash` with:

```
export PKG_CONFIG_PATH=/path/to/hpx/lib/pkgconfig:$PKG_CONFIG_PATH
```

2.2 Building DASHMM

The DASHMM library is straightforward to build. DASHMM uses CMake for its build system, and so CMake of version at least 3.4 is required to build DASHMM. To build DASHMM, perform the following steps.

1. Unpack the source code into some convenient directory. For the sake of discussion, this guide assumes that the code has been unpacked in `/path/to/dashmm/`.

2. Create some directory in which to build DASHMM. For the sake of argument, assume that this is `/path/to/dashmm/build/`.

3. From that directory, run `cmake .. -DCMAKE_INSTALL_PREFIX=/path/to/dashmm/install/`. It is *not* recommended to use the default value for `CMAKE_INSTALL_PREFIX`. There are various additional options that CMake will use during this process, see the CMake documentation for a description of these. For instance, to change the compiler used, specify `-DCMAKE_C_COMPILER` and `-DCMAKE_CXX_COMPILER` when running `cmake`.

4. Run `make` from `/path/to/dashmm/build`.

5. Run `make install` from `/path/to/dashmm/build/`. This will install the library and the header files in the specified place.

6. Further build targets are available, including `basic`, `stepping` and `user`, which build the various demo programs included with DASHMM. These will not be installed by `make install`, and will instead be built in `/path/to/dashmm/build/demo/`.

Because DASHMM is heavily templated, a good deal of the code is contained in the header files. This means that much of the compilation of the DASHMM code will occur when the user code is compiled. This can increase the compilation time of user code, but the added flexibility of DASHMM is worth the minor increase in compilation time.

2.3 Linking against DASHMM

To build a program using the DASHMM library, only a few things need to be done. One must specify where to look for the header files, and where to look for the built library. Further, because DASHMM relies on HPX-5, one must also specify how to find HPX-5. For HPX-5 this is the easiest with the `pkg-config` utility.

Assuming that DASHMM was installed in `/path/to/dashmm/install/`, to compile code (with, for example `g++`) one must specify the following arguments for compilation:

```
-I/path/to/dashmm/install/include $(shell pkg-config --cflags hpx)
```

Similarly, one must specify the following arguments for linking:

```
-L/path/to/dashmm/install/lib -ldashmm $(shell pkg-config --libs hpx)
```

For an example of automatically pulling and building DASHMM inside another project (with CMake), see the AFMPB project: https://github.com/zhang416/afmpb.

2.4 DASHMM demo programs

Included with DASHMM are several test codes that demonstrate the use of the library. These are found in the `/path/to/dashmm/demo` subfolder. Detailed information about each example can be found in the provided `README` file. Each can be built by running `make <target>`, where `<target>` is the demo program name.

2.4.1 Basic demo

The `basic` demo code creates a random distribution of source and target points and computes the potential at the targets due to the sources using any of the built-in kernels provided with DASHMM. A user can request a summary of the options to the test code by running the code with `--help` as a command line argument, or by reading `/path/to/dashmm/demo/basic/README`.

2.4.2 Time-stepping demo

The `stepping` demo code creates a distribution of particles and computes their acceleration and integrates their motion forward in time. Note that the point of the demo is not to provide a great time integrator, so many fine points of creating a good integrator are skipped. Instead,

`stepping` demonstrates those features of DASHMM that enable time-stepping codes as a use-case for DASHMM. A user can request a summary of the options to the code by running the code with `--help` as a command line argument, or by reading `/path/to/dashmm/demo/stepping/README`.

2.4.3 User-defined expansion demo

The `user` demo provides a skeleton code that implements a new expansion type. The code is documented, and the requirements of the Expansion concept are outlined in the in-source comments. This example should be considered to be more advanced, and would require investigation of the advanced interface to understand completely.

Chapter 3

Basic Guide to DASHMM

In this chapter, the basic interface to DASHMM will be covered. Generally speaking, the basic user interface to DASHMM is anything needed to employ the provided methods and kernels in applications. For instructions on how to define and use new methods and kernels, please see chapter 4.

One of the goals of DASHMM is to make it easy to use multipole methods, and so the basic interface targets ease-of-use. Additionally, the library aims to make it easy to perform *parallel* multiple method computations. However, the advanced dynamic techniques that DASHMM employs are not simple to use directly, so the library is constructed in a way that allows the user to get the benefits of parallel execution without having to write the parallel code themselves. That being said, it can be useful to have a sense of what underlies the conceptual framework of DAHSMM. And so, the following section will cover these concepts at a level that might be relevant for basic use of the library. More details can be found in Chapter 4.

3.1 DASHMM concepts

This section covers both the conceptual framework of the multipole methods supported by DASHMM and the parallelization of those methods. More explanation of DASHMM's conceptual framework can be found in the following chapter, or in the code paper.

3.1.1 Multipole method abstractions

DASHMM is a templated library allowing for the description of a general set of multipole methods with a small set of template parameters. To use DASHMM, one must specify four types: the *Source* data type, the *Target* data type, the *Expansion* type and the *Method* type. Within certain limits, each of these can be varied independently, for example, a given expansion might be used with a number of methods, allowing users to easily experiment and select the method that most meets their needs.

DASHMM includes a number of built-in expansion and method types, which are described in sections 3.8 and 3.9.

Source

The Source type gives the structure of the source point data. There are few requirements on the Source type. Typically the minimum requirements are the position and charge of the source. The

`position` is of type `Point` and the `charge` is of type `double`. For example, the following is a minimal Source type that works with every expansion provided with DASHMM.

```
struct SourceData {
  Point position;
  double charge;
};
```

3

Beyond these required members, anything might be added to a Source type. This allows the user to associate application specific data to the sources in the evaluation.

The Source records will be copied across the network, and so if the type is non-trivial, the user of DASHMM must provide a subclass of `Serializer` to handle the serialization and deserialization of Source objects. See section 4.5 for details.

Target

The Target type gives the structure of the target point data. There are similarly few requirements on the Target type. Each Target type will need a `position` of type `Point` and a member to store the result. The details on the exact requirements can be found in the documentation of the individual expansions below. Typically this is a member `phi` of type `std::complex<double>`, which has been aliased as `dcomplex_t` in DASHMM. So the following would work for many of the included DASHMM expansions:

```
struct TargetData {
  Point position;
  std::complex<double> phi;
};
```

Beyond the required members, anything might be added to the Target type. A typical choice is an identifier of some kind because DASHMM evaluations will sort the input data to suit the parallel computation, and points can then be identified after the computation.

The Target records will be copied across the network, and so if the type is non-trivial, the user of DASHMM must provide a subclass of `Serializer` to handle the serialization and deserialization of Target objects. See section 4.5 for details.

Expansion

The particular potential or interaction that is being computed with the multipole moment is called the kernel. For example, the Laplace kernel is the traditional potential from electrostatics or Newtonian gravitation. In DASHMM, kernels are not represented directly. Instead, Expansions are created that implement the needed operations for the particular kernel. The distinction is that there are often multiple ways to expand a given kernel to be used in a multipole method computation. Each Expansion represents a way (or a closely related set of ways) that the potential is expanded into an approximation.

The Expansion type is a template type over two parameters, the Source and Target types being employed. It is the Expansion type that places restrictions on the Source and Target types; the Expansion requires certain data to compute from (taken from the Sources), and it will produce certain data (into the Targets). This allows the Expansion to operate on any data that meets its requirements, meaning that a very general set of source and target data types can be supported.

For details on creating user-defined Expansions, please see chapter 4.

Method

The final major abstraction in DASHMM is the Method. This type specifies how the Expansion is used on the provided Source data to compute the interaction at the locations specified in the Target data. The Method is responsible for connecting the various Expansions representing the hierarchically subdivided set of Source and Target locations with the appropriate operations provided by the Expansion. It is the Method that allows one to perform both a Barnes-Hut computation as well as the Fast Multipole Method.

The Method type is a template over three types: the Source, the Target and the Expansion. For the Method, it is not important exactly how the various operations are implemented, but only that they are implemented. The details of the Expansion, and thus the central details of the particular interaction being studied, are hidden and unimportant to the method.

Despite this generality, it is possible to create a Method that only works for certain expansions. Indeed, included in DASHMM is the `LaplaceCOM` expansion, which does not provide implementations for all of the operations needed by the `FMM` or `FMM97` methods. This is intentional, as the style of expansion in `LaplaceCOM` is not terribly well suited to the Fast Multipole Method. Nevertheless, great flexibility and generality is possible with DASHMM.

For details on creating user-defined Methods, please see chapter 4.

3.1.2 Parallelization abstractions

DASHMM uses the advanced runtime system HPX-5 for its parallelization. HPX-5 provides a number of features that allow DASHMM to naturally express the parallelism and data dependence of the computation directly in programming constructs. However, the flexibility of HPX-5 comes with a significant amount of effort to learn the system. One major goal of DAHSMM is to get the benefits of the dynamic adaptive techniques enabled by HPX-5 without the end-user having to write directly to HPX-5 constructs, and DASHMM is successful in this regard. It is, nevertheless, useful to have some notion of a few concepts from HPX-5 for the basic use of DASHMM. For more details on HPX-5, please visit https://hpx.crest.iu.edu/. For more details on the use of HPX-5 in DASHMM, please see the advanced use guide in the next chapter, or the code paper.

PGAS

HPX-5 provides a partitioned global address space (PGAS) that DASHMM uses for the data during the computation.

That the address space is partitioned means that though the addresses are unified into a global space, each byte of the global address space is served by the physical memory on one particular locality of the system. Further, the mapping from global address to physical address is fixed. A locality is similar to the concept of a rank in an MPI program. That the address space is global means that there is a single virtual address space allowing any locality to refer to data, even if that data is not stored in the same physical memory of the referring locality.

Practically speaking for users of DASHMM, the important part of the global address space provided by HPX-5, as used by DASHMM, is that it is partitioned. Each locality will have direct access to a portion of the data, and indirect access to all of it.

Execution model

The execution model for a DASHMM program is one very similar to the SPMD model. The program written to use DASHMM will be run on every locality in the allocated resources. And

at certain points, the execution will be handed off to DASHMM and ultimately HPX-5 by making DASHMM library calls. Inside DASHMM, the execution is very dynamic and is formed of a large number of small, interdependent tasks. This complication, however, is hidden from the user. Instead, DASHMM presents an interface where each locality participates in collective calls, with each presenting possibly different data to the DASHMM library. So in many ways, using basic DASHMM will be very similar to using MPI.

3.2 Basic types

DASHMM defines a number of basic types that are used throughout the system, and which might be needed by users of the library.

3.2.1 `ReturnCode`

DASHMM calls will return values of the type `ReturnCode` where it is reasonable to do so. The possible values are: `kSuccess`, `kRuntimeError`, `kIncompatible`, `kAllocationError`, `kInitError`, `kFiniError` and `kDomainError`. See specific library calls for cases where each might be returned.

3.2.2 `dcomplex_t`

Many kernels return potential values that are complex numbers. DASHMM provides `dcomplex_t` as an alias to `std::complex<double>`.

3.2.3 `Point`

The `Point` class is used to represent locations in three dimensional space. `Point` is expected for giving the locations of sources and targets. It has the following members and relevant non-member operations:

```
Point::Point(double x = 0, double y = 0, double z = 0)
```

Construct a point from a given set of coordinates. This can also be used to default construct a point.

```
Point::Point(double *arr)
```

Construct a point from a C-style array. It is important that `arr` should contain at least three members.

```
Point::Point(const Point &pt)
```

Copy construct a point.

```
Point Point::scale(double c) const
```

Return a point whose coordinates have all been scaled by the factor `c`.

```
double Point::operator[](size_t i) const
```

Indexing access to the coordinates of the point. `i` must be in the range $[0, 2]$.

```
double Point::x() const
```

Return the x coordinate of the point.

```
double Point::y() const
```

Return the y coordinate of the point.

```
double Point::z() const
```

Return the z coordinate of the point.

```
double Point::norm() const
```

Return the 2-norm of the point.

```
void Point::lower_bound(const Point &other)
```

This computes the lowest coordinate in each direction of this point and `other` and sets this point's coordinate to that value.

```
void Point::upper_bound(const Point &other)
```

This computes the highest coordinate in each direction of this point and `other` and sets this point's coordinates to that value.

```
double point_dot(const Point &left, const Point &right)
```

Treat the points as if they are vectors and take their dot product.

```
Point point_add(const Point &left, const Point &right)
```

Perform a component-wise addition of `left` and `right` and return a point with the result.

```
Point point_sub(const Point &left, const Point &right)
```

Perform a component-wise subtraction of `right` from `left` and return a point with the result.

3.3 Initializing DASHMM

DASHMM must be initialized and finalized to be used. There are some DASHMM operations that must only occur before initialization, and some that can only occur after initialization. All DASHMM operations must occur before the library is finalized.

```
ReturnCode init(int *argc, char ***argv)
```

Initialize the runtime system supporting DASHMM and allocate any resources needed by DASHMM. The addresses of the command line arguments must be provided as the behavior of HPX-5 can be controlled by these arguments. Any arguments dealing with HPX-5 directly will be removed and `argc` and `argv` will be updated accordingly.

`init()` returns `kSuccess` if the system is successfully started, and `kRuntimeError` otherwise. If `init()` returns `kRuntimeError` all subsequent calls to DASHMM will have undefined behavior.

All other DASHMM library calls must occur after `init()`. However, some DASHMM related objects must be constructed before the call to `init()`. See below for details.

This is a collective call; all localities must participate.

```
ReturnCode finalize()
```

This will free any DASHMM specific resources and shut down the runtime system. No other calls to DASHMM must occur after the call to `finalize()`. This is a collective call; all localities must participate.

3.4 SPMD utilities

DASHMM provides a small number of traditional SPMD utilities to make certain things simpler.

```
int get_my_rank()
```

This returns the rank of the calling locality.

```
int get_num_ranks()
```

This returns the number of ranks available.

```
void broadcast(T *value)
```

This performs a broadcast of the given value at rank 0 to all other ranks. This is a template over the type `T`. This is a collective operation. Each rank must provide the address of a type `T` object. For rank 0, this will provide the address of the value to share; for all other ranks this provides the address into which the value broadcast from rank 0 will be stored. The type `T` must be trivially copyable.

3.5 Evaluation

The central object in any DASHMM evaluation is the `Evaluator` object. This object not only manages the registration of certain actions with the runtime system, but also provides the interface to performing the multipole method evaluation.

The `Evaluator` object is a template over four types: the source type, the target type, the expansion type and the method type. The `Evaluator` for a given set of types must be declared before the call to `init()`. For example:

```
dashmm::Evaluator<Source, Target, dashmm::Laplace, dashmm::FMM97> eval{};
```

would be an `Evaluator` for the Laplace kernel using the advanced FMM method for two user-defined types `Source` and `Target` implementing the data that the user requires of the source and target points.

Specifying the full type of the evaluator will cause the template to expand out all of the needed actions to actually implement the evaluation using HPX-5, and will also register those actions with HPX-5.

```
ReturnCode Evaluator::evaluate(
    const Array<Source> &sources,
    const Array<Target> &targets,
    int refinement_limit,
    const Method<Source, Target, Expansion<Source, Target>> *method,
    int n_digits,
    const std::vector<double> *kernelparams)
```

Perform a multipole method evaluation. The arguments to this method are as follows:

- `const Array<Source> &sources`: the `Array` containing the source data.
- `const Array<Target> &targets`: the `Array` containing the target data.
- `int refinement_limit`: the refinement limit of the tree. The sources and targets will be placed into a hierarchical partitioning of space. This partitioning will end when there are fewer sources or targets than the supplied refinement limit in the region under consideration.
- `const Method<Source, Target, Expansion<Source, Target>> *method`: an instance of the method to use for the evaluation. A few methods require parameters at construction, so this is passed in to provide those parameters to the evaluation.
- `int n_digits`: the accuracy parameter for the `Expansion` in use. This is often the number of digits of accuracy required.
- `const std::vector<double> *kernelparams`: the parameters for the kernel evaluation. These are those quantities that are constant for each use of the particular expansion. See the individual expansions for details about what needs to be provided.

Note that during evaluation, the records in the source and target arrays may be sorted. So a separate identifier should be added to the source and target types if the identity of the sources or targets needs to be tracked. Other than the sorting, the only change to the data in the target array will be the output potential or other field value (as specified by the chosen expansion). The only change to the source array beyond the sorting will only occur in the case that the source and target arrays are the same, in which case the previous comment about the targets also applies to the source.

This is a collective call; all localities must participate.

The possible return values are `kSuccess` when there is no problem, and `kRuntimeError` when there is a problem with the execution.

3.6 DASHMM Array

DASHMM provides an array construct that represents a distributed collection of records. The `Array` object is a template type over the record. For basic use of `Array` the record type should be

trivially copyable. For more advanced record types, see section 4.5. When an `Array` is created, as in

```
Array<T> source_data{};
```

no memory is yet allocated for the array. To allocate the memory that will serve the array, one must use `allocate()`. Array objects can be thought of as a traditional array, but which is broken into one part for each locality in the system. The parts, or *segments* of the array, could have different lengths. Some segments can even have no records, meaning that a given locality does not have a share of the data.

`Array` objects are intended to be employed in user code, and so most members of the interface are SPMD in nature, and are collective operations.

```
Array<T>::Array()
```

`Array` objects have a single constructor, which does not allocate any memory. When default constructed, an `Array` will be invalid (see `valid()` below).

This is not a collective operation. To use an array, each locality must create an `Array` object. The individual `Array` objects will be bound into a unified object using `allocate()`.

```
bool Array<T>::valid() const
```

This returns if the array is valid, that is, it refers to some global memory. When initially constructed, an `Array` will be invalid, and can only be made valid by allocating memory for the records (see `allocate()`).

This method is not collective.

```
size_t Array<T>::count() const
```

This returns the number of records in the segment of the array on the calling locality. This is a collective call, and it is an error to call this method on an invalid array. Each calling rank will receive a different result from this method.

```
size_t Array<T>::length() const
```

This returns the total length of the entire array. The result of `length()` is equal to the sum of the results of `count()` from each rank. This is a collective call, and it is an error to call this method on an invalid array.

```
ReturnCode Array<T>::allocate(size_t record_count, T *segment = nullptr)
```

This allocates the memory to serve an array with the given counts on each locality. After this call the array will be valid unless there is an error, which will be indicated by the return code. Possible return values are: `kSuccess` if the array is successfully allocated; `kDomainError` is the object already has an allocation; `kAllocationError` is the global memory cannot be allocated; or `kRuntimeError` if there is some error in the runtime.

This is a collective call. Each locality can provided a different number of records to allocate via the `record_count` parameter. The resulting allocation will match the input of `allocate()`. A locality can provide `0` as an argument, so long as at least one rank asks for a non-zero number of

records. For instance, one locality might allocate all of the records for an array, or each locality might own a portion of the overall records.

The optional second argument allows users to provide the data that will make up the segments of the array. If this argument is `nullptr` (the default), then DASHMM will allocate the memory for each segment. If this argument is not null, then DASHMM will assume ownership of the provided memory, and will build the resulting array with the provided data. Providing an incorrect `record_count` for this use case will cause undefined behavior.

This will return `kDomainError` if the object already has an allocation, `kAllocationError` if memory is unable to be allocated, `kRuntimeError` if there is an error with the runtime system, or `kSuccess` if the method completes without a problem.

```
ReturnCode Array<T>::destroy()
```

This will destroy the memory allocated for this array object. It is an error to call this on an invalid array. This is a collective call. The result of the method will be either `kRuntimeError` if there is an error in the runtime, or `kSuccess` otherwise.

```
ReturnCode Array<T>::get(size_t first, size_t last, T *out)
```

This method gets data from an array object, and places the requested records into the buffer provided by `out`. The range of records that is retrieved is specified by `first` (inclusive) and `last` (exclusive).

This is a collective call. Each locality will provide a different range of records, and a different local buffer into which the retrieved data will be placed. `first` and `last` are given in terms of the segment of the array on this locality. To discover the number of records on the calling locality, use `count()`.

Note that this is a copy of the data; changes to the retrieved values will not be reflected in the array object.

This method will return one of the following: `kRuntimeError` if there is an error with the runtime; `kDomainError` if the provided index range is inconsistent with the array object; or `kSuccess` otherwise.

NOTE: Are we going to deprecate this?

```
ReturnCode Array<T>::put(size_t first, size_t last, T *in)
```

This method puts data into an array object, copying the specified records from the buffer provided by `in`. The range of records that is copied is specified by `first` (inclusive) and `last` (exclusive).

This is a collective call. Each locality will provide a different range of records, and a different local buffer from which the retrieved data will be placed. `first` and `last` are given in terms of the segment of the array on this locality. To discover the capacity of the array on the calling locality, use `count()`.

Note that this places a copy of the records into the address space; subsequent changes in the local data will not be reflected in the array object.

This method will return one of the following: `kRuntimeError` if there is an error with the runtime; `kDomainError` if the provided index range is inconsistent with the array object; or `kSuccess` otherwise.

NOTE: Are we going to deprecate this?

```
std::unique_ptr<T[]> Array<T>::collect()
```

This method collects all of the records in the array and returns a new local allocation containing the records at locality 0. This is, largely speaking, a convenience feature. Note that this will allocate a copy of the entirety of the array on one locality.

This is a collective call. The returned smart pointer will only be valid on locality zero. All other ranks will receive a null pointer.

Note that this provides a copy of the data; changes to the returned data will not be reflected in the array object.

```
T *Array<T>::segment(size_t &count)
```

This method provides direct access to the local segment of an array. Because some DASHMM routines will modify the segments of arrays, the length of the returned array data cannot be known ahead of time. So, in addition to the segment's address, the number of records is returned via the `count` parameter. It is possible that this will return `nullptr`, but this will occur only when `count` is zero.

This is a collective call. Each rank will receive the address and count of its own segment.

Note that this provides direct access to the data; changes to the data will persist in the object.

Note that calls to some other DASHMM routines will invalidate the address returned by this method.

3.7 Array For Each Actions

To avoid the round trip from and to the global address space via `Array`'s `get()` and `put()` methods, one can make use of the `ArrayForEachAction` type. This type specifies an action to be performed on the records of an array. Then, together with a method of the `Array` object, this allows for some computation to occur on the records of an array.

This class is a template requiring two parameters: the type of records for the array to which the action will be applied (hereafter `T`), and a type specifying an environment to provide to the action (hereafter `E`).

The action is specified during construction of an object of type `ArrayForEachAction`. This is provided as a function pointer, for a function with a particular signature. So that DASHMM can use the action on every locality, like the `Evaluator` object, any `ArrayForEachAction` objects must be defined before `init()` is called.

The function implementing the action must take three arguments, the first is a `T *` giving the data on which to act, the second is a `const size_t` giving the number of records on which to act, and the third is `const E *`, where `E` is the environment type of the `ArrayForEachAction`. For example:

```
void update_position(T *data, const size_t count, const E *env) {
  for (size_t i = 0; i < count; ++i) {
    data[i].position += data[i].velocity * env->delta_t;
  }
}
```

is an action that might perform a position update for a time-stepping code. It is important to note that the action implementation can place requirements on the array record type `T`. In the previous

example, type `T` needs to have a member called `velocity`. The first argument to the action will be a correctly offset pointer into a contiguous chunk of records. The action should only assume that `data[0]` through `data[count - 1]` are available for use.

```
ArrayForEachAction<T, E>::map_function_t
```

This class aliases the type of function that can implement the action. It is:

```
void (*)(T *, const size_t, const E *)
```

```
ArrayForEachAction<T, E>::ArrayForEachAction(map_function_t f)
```

This constructs an array map action object. The provided function pointer will give the action that is performed on the array. To use an action, the associated `ArrayForEachAction` must be defined before `init()` is called. This object will register the needed actions with the runtime system.

For any given action `f` only a single `ArrayForEachAction` should be defined.

```
ReturnCode Array<T>::forEach(ArrayForEachAction<T, E> &act, const E *env)
```

Once an `ArrayForEachAction` is defined, it can be used on a specific array by calling that array's `map()` method. This will cause the action represented by `act` to be applied on all entries of this array. The action ultimately works on segments of the array. The environment, `env` is provided unmodified to each segment.

This is a collective call. It is an error to call `forEach()` on an invalid array.

3.8 Built-in methods

DASHMM includes a number of built-in methods that are ready to use for problems: the Barnes-Hut method, two forms of the Fast Multipole Method and a Direct summation method. These will be covered in detail below. To successfully compile, all operations in the Expansion concept need to be implemented for a given expansion. However, some methods only require a subset of the full complement of operations. The built-in methods will be covered below, in order of increasing complexity.

3.8.1 `Direct`

The `Direct` method is primarily intended for use as a comparison for computing the exact result for a given set of sources. There is some parallelism implemented for this method, so the execution time is reduced somewhat. However, for realistic problem sizes, this method should be avoided.

The only operation that is used by the `Direct` method is `StoT`, so any Expansion implementing that operator can be used with the `Direct` method.

Please see the demo program `demo/basic` included with DASHMM for an example use case of `Direct` and note that it is only applied to a very small subset of the target locations.

Construction of `Direct` is simple, as it can only be default constructed.

3.8.2 BH

The `BH` method implements the Barnes-Hut algorithm in the framework of DASHMM. The implemented method uses the simple multipole acceptance criterion parameterized by a single angle. If a multipole expansion is centered a distance R away from a given point and the multipole expansion is associated with a region of size D then the multipole expansion is used only if $D/R < \theta_C$ for some critical angle θ_C.

In practice, since the hierarchical partitioning of the source and target locations in DASHMM produces leaves that can contain multiple particles, the multipole acceptance criterion is evaluated pessimistically: the radius is computed from the nearest possible location in a given target tree node to the multipole in question. This will tend to produce results with a slightly smaller error, with a slightly larger execution time.

When creating a `BH` object, the opening angle θ_C is specified:

```
BH bh_method(0.6);
```

Generally the opening angle should be less than one. As the angle approaches zero, the method approaches the direct summation method. The critical angle for a particular `BH` object can be accessed with `double BH::theta()`, which returns the angle specified at creation time. A default constructed `BH` uses a critical angle of `0`.

In addition to the direct contribution, `StoT`, to use `BH` an expansion must also implement fully the following operations: `StoM`, `MtoM` and `MtoT`.

3.8.3 FMM

The `FMM` method implements the Fast Multipole Method in its original form. To create an `FMM` method, no arguments are needed as the decisions about which expansions to use in which situations are all made based on fixed geometric considerations.

The following operations must have a full implementation in an expansion to be used with `FMM`: `StoT`, `StoM`, `StoL`, `MtoM`, `MtoL`, `MtoT`, `LtoL`, and `LtoT`.

3.8.4 FMM97

The `FMM97` method implements the Fast Multipole Method in the form that uses exponential expansions and the merge-and-shift technique. No parameters are needed to construct an `FMM97` method.

The following operations must have a full implementation in an expansion to be used with `FMM97`: `StoT`, `StoM`, `StoL`, `MtoM`, `MtoL`, `MtoT`, `LtoL`, `LtoT`, `MtoI`, `ItoI` and `ItoL`.

3.9 Built-in expansions

DASHMM includes a number of built-in expansion that are ready to use for applications. Each expansion will have a different set of implemented operations which will restrict their use for certain methods. Further, each expansion will place requirements on the source and target types used during a DASHMM evaluation. These details will be covered for each expansion below.

3.9.1 Laplace

The Laplace expansion is a spherical harmonic expansion of the Laplace potential that is designed to handle sources with both signs of charge without losing accuracy, unlike LaplaceCOM and LaplaceCOMAcc below. This potential is scale-invariant, so there are no kernel parameters that are needed in the call to evaluate(). However, calls to evaluate must supply an accuracy parameter giving the number of digits of accuracy that are requested.

Though this expansion is in principle compatible with every method included with DASHMM, it is designed for the FMM and FMM97 methods.

This expansion imposes the following restrictions on the source type: a member of type Point with the name position must be provided; a member of type double with the name charge must be provided.

This expansion imposes the following restrictions on the target type: a member of type Point with the name position must be provided; a member of type dcomplex_t with the name potential must be provided.

3.9.2 Yukawa

The Yukawa expansion is a spherical harmonic expansion of the Yukawa potential. This potential is scaling variant, so a single kernel parameter must be provided to evaluate(). Calls to evaluate must also supply an accuracy parameter that gives the number of digits of accuracy required.

Though this expansion is in principle compatible with every method included with DASHMM, it is designed for the FMM97 method.

This expansion imposes the following restrictions on the source type: a member of type Point with the name position must be provided; a member of type double with the name charge must be provided.

This expansion imposes the following restrictions on the target type: a member of type Point with the name position must be provided; a member of type dcomplex_t with the name potential must be provided.

3.9.3 Helmholtz

The Helmholtz expansion expands the Helmholtz potential in the low-frequency regime using spherical harmonics, spherical Bessel and Hankel functions. This potential is scaling variant, so a single kernel parameter must be provided to evaluate(). Calls to evaluate must also supply an accuracy parameter that gives the number of digits of accuracy required.

Though this expansion is designed for use with the FMM97 method.

This expansion imposes the following restrictions on the source type: a member of type Point with the name position must be provided; a member of type double with the name charge must be provided.

This expansion imposes the following restrictions on the target type: a member of type Point with the name position must be provided; a member of type dcomplex_t with the name potential must be provided.

3.9.4 LaplaceCOM

The LaplaceCOM expansion is a center of mass expansion of the Laplace potential. This form of the expansion extends to the quadrupole term, and because of the choice of center has an identically zero dipole term. This expansion can be used to compute the potential. See LaplaceCOMAcc for an

equivalent expansion that computes the acceleration. This potential is scale-invariant so no kernel parameters are needed in the call to `evaluate()`. Further, the number of terms in the expansion is fixed, so the accuracy parameter to `evaluate()` is ignored.

This expansion is only compatible with the `BH` or `Direct` methods; it does not implement all the needed operations for use with `FMM` or `FMM97`.

This expansion imposes the following restrictions on the source type: a member of type `Point` with the name `position` must be provided; a member of type `double` with the name `charge` must be provided.

This expansion imposes the following restrictions on the target type: a member of type `Point` with the name 'position' must be provided; a member of type `dcomplex_t` with the name `potential` must be provided.

3.9.5 `LaplaceCOMAcc`

The `LaplaceCOM` expansion is a center of mass expansion of the Laplace potential. This form of the expansion extends to the quadrupole term, and because of the choice of center has an identically zero dipole term. This expansion can be used to compute the acceleration. See `LaplaceCOM` for an equivalent expansion that computes the potential. This potential is scale-invariant so no kernel parameters are needed in the call to `evaluate()`. Further, the number of terms in the expansion is fixed, so the accuracy parameter to `evaluate()` is ignored.

This expansion is only compatible with the `BH` or `Direct` methods; it does not implement all the needed operations for use with `FMM` or `FMM97`.

This expansion imposes the following restrictions on the source type: a member of type `Point` with the name `position` must be provided; a member of type `double` with the name `charge` must be provided.

This expansion imposes the following restrictions on the target type: a member of type `Point` with the name `position` must be provided; a member `double acceleration[3]` must be provided.

Chapter 4

Advanced Guide to DASHMM

In this chapter, the rest of the interface to DASHMM will be presented. Some of the material presented here provides further information on constructs presented in the previous chapter. In other cases, the coverage of a library construct is complete in the previous chapter, and so those topics will not be repeated here. Finally, some constructs not needed for basic DASHMM usage are presented in their entirety

The fundamental difference between the basic and advanced interface to DASHMM is that the advanced interface is needed when a user is implementing a new Expansion or Method. This will require a user to be familiar with more of the details of how the execution is performed, and will thus require some more details about how HPX-5 provides parallelism to DASHMM. However, the extent to which a user will have to learn HPX-5 directly is still extremely limited.

The arrangement of material in this chapter will parallel the arrangement in the previous.

4.1 DASHMM Concepts

This sections covers the conceptual framework of DASHMM's implementation of general multipole methods. The following abstractions form the basis for the implementation of multipole methods in DASHMM.

4.1.1 The Dual Tree

The multipole method framework implemented in DASHMM is general enough to allow for situations where the sources of an interaction, and the locations where the interaction is to be computed are different. As a result, DASHMM uses the Dual Tree construction. This involves a hierarchical partitioning of the problem domain for both the source and target points. This leads naturally to two trees, one each for sources and targets. These tree are constructed to be compatible with one another: the root of the source and target trees represent the same volume. Each node of both trees can have up to eight children, each having half the side length of the parent.

When referring to particular volumes in the tree, DASHMM often uses an integer index giving which level of the tree and where on that level the node is. This makes it possible to refer to the volumes of the tree nodes without needing to wrangle with floating point arithmetic.

When these trees are constructed, the source and target points are ultimately assigned to a node of the associated tree. The refinement of the tree is adaptive in DASHMM; a node is refined, adding up to eight children, until the number of points in a node is below some given threshold. In this way, the tree adapts to the distribution of source and target points.

4.1.2 The DAG

Ultimately, a multipole method computation is the creation and evaluation of a directed acyclic graph (DAG). This DAG encodes in its nodes the various approximations, or expansions, that are used to the effects of the sources represented by a given node. Encoded in the edges of the DAG are the operations that transform the various approximations into other approximations.

DASHMM constructs two versions of the DAG representing a given multipole computation: the first is an explicit representation that is used during the discovery of the overall structure of the DAG and which is used to distribute the work among the available localities; the second is an implicit representation that lives in the data, synchronization objects, and the interconnections among the tasks that are performed in the actual computation.

The explicit DAG is represented in DASHMM with objects that an advanced user of DASHMM might need to interact with. The implicit DAG comprises a number of HPX-5-aware objects that manage the parallel execution. The exact use and form of these objects is not necessary for even an advanced user; DASHMM insulates the user from these details.

4.1.3 Expansion Roles

Each node of the DAG is associated with a node of either the source or target tree. This leads to the four expansion roles defined in DASHMM. Nodes associated with the typical expansions (multipole and local) are termed 'normal'. Nodes associated with expansions that are used in advanced techniques such as merge-and-shift are called 'intermediate' expansions. This leads to the four roles for expansions: two kinds for each of two trees.

4.1.4 Operations

The abstraction of the multipole method in DASHMM allows for the following operations. Of course, for a particular use-case, these operations may take on meanings different from the typical meaning for FMM or BH. The notation for the following operations all take the form `AtoB`, where `A` is one of {`S, M, L, I`} and `B` is one of the following {`M, L, I, T`}. `S` is a node of the DAG that represents the leaves of a source tree; it represents a set of sources. `M` is a node of the DAG representing a normal expansion in the source tree. `L` is a node of the DAG representing a normal expansion in the target tree. `T` is a node of the DAG representing a leaf of the target tree; it represents a set of target locations. `I` represents the intermediate expansions on either the source or target trees (no distinction is made in this notation as the source or target side of the `I` can be inferred from the operation).

The `StoM` operation is any operation that produces an approximation or summary version of the source data in a given node. These operations generate multipole moments from sources in typical use cases.

The `MtoM` operation is used to combine summaries of a set of sources into summaries that apply to larger volumes, or to larger number of sources. Typical uses of this operation produce a hierarchy of multipole expansions for every node of the source tree.

The `MtoL` operation is used to translate summaries of a set of sources into a summary that applies to a volume in the target tree. This is the operation that distinguishes FMM from BH; the creation of the local expansions allows for efficient evaluation of large parts of the source distribution.

The `LtoL` operation is used to translate a local expansion that applies for a parent into a local expansion that applies in the child's volume.

The `MtoT` operation computes the effect of a multipole expansion on a set of targets.

The `LtoT` operation computes the effect of a local expansion on a set of targets.

The `StoT` operation computes the direct effect of a set of sources on a set of targets. If a method only schedules `StoT` operations, it will be the same as the direct summation method.

The `StoL` operation computes the effect of a set of sources on a volume of the target tree.

The `MtoI` operation transforms the summary information on the source tree into another form that certain advanced methods might find useful. The use of these intermediate expansions is what distinguishes FMM and FMM97. The intermediate expansion is associated with a source tree node.

The `ItoI` operation translates the intermediate representation on the source tree into an intermediate representation on the target tree. The forms of these representations may well be different.

The `ItoL` operation translates the intermediate representation on the target tree into a local expansion on the target tree.

4.1.5 Expansions and Views

There is a distinction between an Expansion in the mathematical sense, and the Expansions in DASHMM. An Expansion in DASHMM might contain multiple mathematical expansions. It can be the case that a given method might need to keep several versions of summary data for a given set of sources. In this case, a DASHMM expansion presents multiple Views. Each View is a single mathematical expansion. The various views that are implemented in an Expansion can be related to one another, but it can also support the ability to have unrelated mathematical expansions to allow for a single evaluation to compute for multiple kernels at the same time.

4.1.6 Kernel details

The mathematical concept of a Kernel is included in DASHMM's concept of an Expansion. Ultimately the Method does not care about how the operations are performed, just that they exist. So it is the Expansion that implements the kernel. In the most direct sense, the kernel can be seen in the implemented `StoT` operation.

Some kernels will have parameters that control the exact shape of the potential. For example, the built-in Yukawa kernel is scaling variant, and so the scale must be provided to DASHMM. These kernel parameters also impact the expansions that are employed in Expansion. Further, often there is some one-time precomputation that can occur that produces values that are used in the various operations. To allow for this precomputation, DASHMM employs the abstraction of a Kernel table. This table is created once before the operations are employed and is made available to all ranks. The table represents any values needed by the expansions and operations that does not change as the inputs to the operations change.

4.1.7 Methods and DAG creation

There are four main operations that a Method will employ during DAG discovery.

The first is *generate*. This operation is responsible for creating an approximation at the leaves of a source tree from the source records. This is often the generation of the multipole expansions from the source data.

The second is *aggregate*. In aggregate, the summary expansions computed at the child of a given node are combined into a summary that corresponds to all of the sources that are descendants of the given node.

The third is *inherit*. The inherit operation will take summary information for target nodes and will create summary information for the children of that node. Note that operations that end at a

target leaf are not discovered in this operation.[1]

The fourth is *process*. It is in process that the bulk of the work of creating the DAG is done. This is responsible for forging any connection between the two trees. Notice that generate and aggregate only connect nodes on the source tree, and that inherit only connects nodes in the target tree. All other connections are made in process.

4.2 Basic types

The following basic types are needed when implementing user-defined methods and expansions.

4.2.1 `Operation`

DASHMM organizes the transformations between various forms of the potential expansion into a set of operations. The `Operation` scoped enumeration identifies the operations that DASHMM recognizes. They are: `Nop`, `StoM`, `StoL`, `MtoM`, `MtoL`, `LtoL`, `MtoT`, `LtoT`, `StoT`, `MtoI`, `ItoI` and `ItoL`.

4.2.2 `ExpansionRole`

DASHMM creates Expansion objects in a number of roles. The role is essentially a description of which tree in the dual tree an expansion is most closely associated with, and whether the expansion is a primary or intermediate expansion. The `ExpansionRole` is an enumeration with the following members: `kSourcePrimary`, `kSourceIntermediate`, `kTargetPrimary`, `kTargetIntermediate` and `kNoRoleNeeded`. The latter is for situations that that require an operation from the Expansion type, but which do not require expansion data. The prototypical example of this is `StoT`.

4.2.3 `Index`

Each node of both the source and target trees can be identified with four integers: the level of the tree (starting with 0 for the root), and the position of the low corner of the node in units of the node size at the given level. The index allows DASHMM to not only perform accurate positional comparisons, but also to provide an ordering of the nodes of the tree at a given level.

```
Index::Index(int ix = 0, int iy = 0, int iz = 0, int lvl = 0)
```

Construct an Index with the given on-level position and level.

```
int Index::x() const
```

Return the on-level position in the x direction.

```
int Index::y() const
```

Return the on-level position in the y direction.

```
int Index::z() const
```

Return the on-level position in the z direction.

[1] This is an asymmetry that will be rectified in future versions of DASHMM.

```
int Index::level() const
```

Return the level of the index.

```
Index Index::parent(int num = 1) const
```

Return the `num`-th parent of this `Index`. By default, this gives the index of the immediate parent.

```
Index Index::child(int which) const
```

Return the given child of this Index. `which` is a composite value that indicates with each bit if the child is on the left or right in that direction. For example for `which == 6`, the child is the left child in the z-direction, and the right child in both the y and x directions.

```
int Index::which_child() const
```

Return which child of its parent this node is.

```
bool Index::operator==(const Index &other) const
```

Equality operator.

4.2.4 DomainGeometry

To represent the computational domain of the sources and targets, DASHMM uses `DomainGeometry` object. This object can be used to convert an `Index` into `Point`s for various locations in the volume represented by that index. The domains represented by DASHMM are cubical regions.

```
DomainGeometry::DomainGeometry()
```

Default construct a domain to begin at the origin, with a zero side length.

```
DomainGeometry::DomainGeometry(Point low, double size)
```

Construct a domain with the given low corner, and the given side length.

```
DomainGeometry::DomainGeometry(Point low, Point high, double f = 1.0)
```

Construct a domain from the given low and high corners. The given region need not be cubical. The resulting object will represent the smallest cube that contains the specified volume. Additionally, the final parameter can be used to enlarge the resulting cube by a fixed fraction.

```
double DomainGeometry::size() const
```

Return the side length of the represented cubical volume.

```
Point DomainGeometry::low() const
```

Return the low corner of the represented volume.

```
Point DomainGeometry::high() const
```

Return the high corner of the represented volume.

```
Point DomainGeometry::center() const
```

Return the center of the represented volume.

```
Point DomainGeometry::low_from_index(Index idx) const
```

Return the low corner of the region represented by the given index.

```
Point DomainGeometry::high_from_index(Index idx) const
```

Return the high corner of the region represented by the given index.

```
Point DomainGeometry::center_from_index(Index idx) const
```

Return the center of the region represented by the given index.

```
double DomainGeometry::size_from_index(Index idx) const
```

Return the size of the region represented by the given index.

4.3 Initializing DASHMM

Initialization of DASHMM via `init()` requires providing the command line arguments to the program. This is to provide the opportunity to HPX-5 to detect any command line arguments that modify its behavior. A full description of the available options can be found in the HPX-5 documentation. Here, we shall cover those that are most relevant for programs using DASHMM.

NOTE: The material in this section should not be considered to be part of the library's interface, and are subject to modification out of the control of the DASHMM development team.

The following HPX-5 command line arguments are the most relevant to DASHMM:

```
--hpx-help
```

Display a help message giving a brief description of all available options.

```
--hpx-threads
```

Specify the number of scheduler threads that HPX-5 will use per rank. Typically, one thread per core gives best results, but fewer is sometimes useful in scalability studies.

```
--hpx-heapsize
```

Specify the size in bytes of the amount of global address space available to each rank. The default is frequently too low for large problem sizes. This, however, should not be set to take all of the system memory.

4.4 Evaluation

4.4.1 Distribution Policy

The `Evaluator` object has one additional feature that was not covered in the previous chapter. In addition to the parameters outlined in Chapter 3, one final optional parameter is available.

Each Method has a distribution policy that specifies how the DAG nodes are to be distributed around the available resources. To allow for these policies to have some parameters selectable at runtime, the `Evaluator::evaluate` method accepts a distribution policy object. This allows for users to fine-tune the behavior of the distribution if needed. When the final argument is not supplied, DASHMM employs a default constructed object of the type `Method<Source, Target, Expansion>::distropolicy_t`, where `Method, Source, Target` and `Expansion` are the template arguments supplied to the particular instance of `Evaluator`.

4.4.2 Expanded API

In addition to the one-size-fits-all approach implemented in the `evaluate()`, `Evaluator` objects expose sub-steps of the evaluation process. This is to enable uses in iterative methods, where the same DAG is used multiple times. Rather than creating the Dual Tree and discovering the DAG each iteration, they can be created only once, and reused.

```
DualTreeHandle Evaluator::create_tree(
    const Array<Source> &sources,
    const Array<Target> &targets,
    int refinement_limit)
```

Create a Dual Tree object in the global address space from the given `sources` and `targets`, and with the given `refinement_limit`. The return value of this method is an opaque handle to the created Dual Tree. This value, being a handle, can be freely copied.

This is a collective call and all ranks must participate.

```
std::unique_ptr<DAG> Evaluator::create_DAG(
    DualTreeHandle tree,
    int n_digits,
    const std::vector<double> *kernel_params,
    const Method<Source, Target, Expansion<Source, Target>> *method,
    distropolicy_t distro = distropolicy_t{})
```

Given a `tree`, an accuracy parameter `n_digits`, the kernel parameters, `kernel_params` to use for this DAG, an instance of a Method object, and an optional distribution policy instance, this method will create the DAG corresponding to the given method for the given Dual Tree.

This is a collective call, and all ranks must participate.

```
ReturnCode Evaluator::execute_DAG(DualTreeHandle tree, DAG *dag)
```

Once the DAG has been constructed, this method will execute the work represented by the given DAG. This will return either `kSuccess` on successful execution, or `kRuntimeError` if there is a problem during the execution.

After this call, the DAG is left in a used state. If the DAG is to be used again, it must be reset (see `reset_DAG` below).

This is a collective call, and all ranks must participate.

```
ReturnCode Evaluator::reset_DAG(DAG *dag)
```

This will take a DAG that has been executed with the `execute_DAG` method, and will reset the internal data structures so that it might be used again. Note that this does not actually perform the execution of that DAG, merely prepares it to be executed.

This will return either `kSuccess` if there is no error, or `kRuntimeError` in the case of an error.

This is a collective call, and all ranks must participate.

```
ReturnCode Evaluator::destroy_DAG(
    DualTreeHandle tree,
    std::unique_ptr<DAG> dag)
```

This will destroy a DAG. This will return either `kSuccess` if the destruction proceeds without error, or will return `kRuntimeError`.

This is a collective call, and all ranks must participate.

```
ReturnCode Evaluator::destroy_tree(DualTreeHandle tree)
```

This method destroys the Dual Tree specified by `tree`. This will return either `kSuccess` after successfully destroying the tree, or `kRuntimeError` if there is some error.

This is a collective call, and all ranks must participate.

4.5 Serializer

Source and Target data may at times need to be copied around the system, either locally or across the network. To allow for non-trivial Source and Target types, DASHMM allows the user to specify a serialization manager. This is a subclass of the `Serializer` abstract class, which has the job of correctly serializing and deserializing the data in a Source or Target record. Each `Array` has a serialization manager associated with it that takes responsibility for this serialization.

4.5.1 `TrivialSerializer`

DASHMM comes with one built in subclass of `Serializer`, called `TrivialSerializer`, which is actually a template over a single template parameter. It can work for any trivially-copyable type, and is the default serialization manager installed when a new `Array` is allocated.

4.5.2 `Serializer` Interface

The interface that all subclasses of `Serializer` must implement is given below.

```
size_t Serializer::size(void *object) const
```

This method will return the serialized size in bytes of the given object. The interface does not contain any information about the type of the object, and so it is up to the user of DASHMM to assure that the subclass is used only for compatible objects. Typically, inside the implementation, the provided address will be typecast.

```
void *Serializer::serialize(void *object, void *buffer) const
```

This method will serialize the given object into the provided `buffer`. This method then returns the address `buffer + bytes`, where `bytes` is the number of bytes of the serialized form of `object`.

```
void *Serializer::deserialize(void *buffer, void *object) const
```

This method will deserialize an object from `buffer` into the provided `object`. This object already exists; this method fills in the correct values for the object, rather than creating a new object. The return value from this method is `buffer + bytes` where `bytes` is the serialized size of the object.

4.6 Array

When an `Array` contains records of a non-trivial type, the user of the array must provide an object that derives from `Serializer`, which has the task of serialization of those records. Without explicitly setting the serialization manager, an `Array` will initially have a `TrivialSerializer<T>` as its serialization manager, where `T` is the record type of the array.

At any point, an `Array`'s serialization manager can be changed with the following method of `Array`.

```
ReturnCode Array<T>::set_manager(std::unique_ptr<Serializer> manager)
```

This method will set the serialization manager for the array to the provided `Serializer` instance. This is a collective call, and all ranks must participate. Further, though it would be technically possible to do otherwise, each rank should provide an object of the same subclass of `Serializer`.

4.7 Array For Each Actions

The `ArrayForEachAction` utility class supports a third template parameter that controls the level of parallelism employed in the mapping of the work to the records in the array. This parameter is an integer that has the following meaning: if the argument is zero, the array will be handled in a single chunk; if the argument is positive, the array will be handled in a number of chunks equal to the number of HPX-5 scheduler threads times the provided argument.

The default value of this argument is 1, so the default operation of the map will be to split each rank's portion of the array into a number of equally sized pieces equal to the number of scheduler threads. Unless there is the possibility for variation in the amount of computation that the mapped action performs per record, it is likely that the default value will be sufficient. If, however, the work for each record is variable, better performance may be achieved with smaller chunks, and thus larger third arguments to the template.

The full declaration of `ArrayForEachAction` is as follows:

```
template <typename T, typename E, int factor = 1>
class ArrayForEachAction;
```

4.8 DAG objects

After the dual tree is constructed, DASHMM creates an explicit representation of the DAG for the given method applied to the just constructed tree. This DAG is used to perform a work distribution, and to act as a scaffold from which the actual expansion data is instantiated. For users wishing to

implement their own methods, the DAG objects are the objects that will be needed. In DASHMM, the Method builds a DAG from the dual tree. There are 4 classes that make up the DAG system for DASHMM: `DAG`, `DAGInfo`, `DAGNode` and `DAGEdge`, which will be covered in turn.

4.8.1 DAGEdge

The edges connecting nodes in the DAG is described with the `DAGEdge` type. This simple type holds pointers to the source and target `DAGNode`s connected by the edge, the `Operation` that the edge represents, and an integer `weight` that gives an estimate of the communication cost of the edge. The `weight` is optionally used by the distribution policy to aid in the decision about data placement around the system. The full definition of `DAGEdge` is as follows:

```
DAGNode *DAGEdge::target
```

Target node of the edge.

```
Operation DAGEdge::op
```

Operation to perform along edge.

```
int DAGEdge::weight
```

Estimate of communication cost required if the edge were to span localities.

4.8.2 DAGNode

The nodes of the DAG are represented by the simple type `DAGNode`. It contains the following public members:

```
std::vector<DAGEdge> DAGNode::out_edges
```

The edges of the DAG that start at this node.

```
int DAGNode::locality
```

The locality to which this node will be assigned by the distribution. Note that this will most often be set by the distribution policy. To indicate that the locality is not set, a value of `-1` should be used.

```
int DAGNode::color
```

A color that might be used by the distribution policy. This has meaning only in the context of the distribution policy.

```
DAGNode::DAGNode(Index i)
```

Construct a DAG node. This will set the index to the given value, give the locality a value of `-1`, and default construct the remaining members of the node.

```
size_t DAGNode::in_count() const
```

Returns the number of incoming edges to this node.

```
size_t DAGNode::out_count() const
```

Returns the number of outgoing edges from this node.

```
Index DAGNode::index()
```

Returns the index of the tree node to which this DAG node is associated.

```
bool DAGNode::is_parts() const
```

Indicates if this node represents sources or targets.

```
bool DAGNode::is_normal() const
```

Indicates if this node represents a multipole or local expansion.

```
bool DAGNode::is_interm() const
```

Indicates if this node represents an intermediate expansion.

4.8.3 DAG

The `DAG` object is given to a distribution policy when computing the localities of the nodes in the DAG. This is another simple object that contains four containers of the nodes of the DAG. These containers separate the nodes by their position in the DAG.

The `DAG` object has the following members:

```
std::vector<DAGNode *> DAG::source_leaves
```

Nodes of the DAG that are associated with leaves of the source tree. These nodes will have no incoming edges.

```
std::vector<DAGNode *> DAG::source_nodes
```

All other DAG nodes that are associated with nodes of the source tree.

```
std::vector<DAGNode *> DAG::target_leaves
```

Nodes of the DAG that are associated with leaves of the target tree. These nodes will have no outgoing edges. In some methods, these are not associated with the leaves of the target tree, but rather those nodes of the target tree after which more refinement for the method would only induce unnecessary overhead.

```
std::vector<DAGNode *> DAG::target_nodes
```

All other DAG nodes that are associated with node of the target tree.

4.8.4 DAGInfo

The DAGInfo object contains all the information related to the DAG for each node of the source and target trees. These objects will be the primary means by which a Method interacts with the DAG.

Each DAGInfo object represents up to three DAG nodes: a particle node, the normal DAG node, and an intermediate DAG node. The particle node represents the terminal nodes of the DAG: either the information about the sources, or the information about the targets. These DAG nodes do not represent expansions. The normal DAG node represents expansions with a role of kSourcePrimary or kTargetPrimary. The intermediate node represents expansions with a role of kSourceIntermediate or kTargetIntermediate. A particular method might not use certain expansion roles, so a DAGInfo object might not contain the normal or intermediate DAG node. Further, not all tree nodes are leaves, and so only some DAGInfo objects will have a particles node.

This object manages the concurrent modification of the DAG. However, that management is hidden from users of this object.

The DAGInfo methods that a user might need to use in a Method are covered below.

```
Index DAGInfo::index() const
```

Return the index of the volume represented by the DAG nodes owned by this object.

```
bool DAGInfo::add_normal()
```

Add a normal DAG node to this object. This will return true if this call added the node. Otherwise, if the DAGNode already owns a normal node, this call will do nothing, and return false.

```
bool DAGInfo::add_interm()
```

Add an intermediate DAG node to this object. This will return true if this call added the node. Otherwise, if the DAGNode already owns an intermediate node, this call will do nothing, and return false.

```
bool DAGInfo::add_parts()
```

Add a particles DAG node to this object. This will return true if this call added the node. Otherwise, if the DAGNode already owns a particles node, this call will do nothing, and return false.

```
bool DAGInfo::has_normal() const
```

Predicate returning if this object owns a normal DAG node.

```
bool DAGInfo::has_interm() const
```

Predicate returning if this object owns an intermediate DAG node.

```
bool DAGInfo::has_parts() const
```

Predicate returning of this object owns a particles DAG node.

```
void DAGInfo::set_normal_locality(int loc)
```

Set the locality for the normal DAG node, if it exists, to the given locality, `loc`.

```
void DAGInfo::set_interm_locality(int loc)
```

Set the locality for the intermediate DAG node, if it exists, to the given locality, `loc`.

```
void DAGInfo::StoM(DAGInfo *source, int weight)
```

Connect the particles DAG node of `source` to the normal DAG node of this object with operation `StoM`, and assign the resulting edge the given `weight`. It is an error to call this method if this object does not own a normal DAG node, and if the `source` does not own a particles DAG node.

```
void DAGInfo::StoL(DAGInfo *source, int weight)
```

Connect the particles DAG node of `source` to the normal DAG node of this object with operation `StoL`, and assign the resulting edge the given `weight`. It is an error to call this method if this object does not own a normal DAG node, and if the `source` does not own a particles DAG node.

```
void DAGInfo::MtoM(DAGInfo *source, int weight)
```

Connect the normal DAG node of `source` to the normal DAG node of this object with operation `MtoM`, and assign the resulting edge the given `weight`. It is an error to call this method if this object does not own a normal DAG node, and if the `source` does not own a normal DAG node.

```
void DAGInfo::MtoL(DAGInfo *source, int weight)
```

Connect the normal DAG node of `source` to the normal DAG node of this object with operation `MtoL`, and assign the resulting edge the given `weight`. It is an error to call this method if this object does not own a normal DAG node, and if the `source` does not own a normal DAG node.

```
void DAGInfo::LtoL(DAGInfo *source, int weight)
```

Connect the normal DAG node of `source` to the normal DAG node of this object with operation `LtoL`, and assign the resulting edge the given `weight`. It is an error to call this method if this object does not own a normal DAG node, and if the `source` does not own a normal DAG node.

```
void DAGInfo::MtoT(DAGInfo *target, int weight)
```

Connect the normal DAG node of this object to the particles DAG node of `target` with operation `MtoT`, and assign the resulting edge the given `weight`. It is an error to call this method if this object does not own a normal DAG node, and if the `target` does not own a particles DAG node.

```
void DAGInfo::LtoT(DAGInfo *target, int weight)
```

Connect the normal DAG node of this object to the particles DAG node of `target` with operation `LtoT`, and assign the resulting edge the given `weight`. It is an error to call this method if this object does not own a normal DAG node, and if the `target` does not own a particles DAG node.

```
void DAGInfo::StoT(DAGInfo *source, int weight)
```

Connect the particles DAG node of `source` to the particles DAG node of this object with operation `StoT`, and assign the resulting edge the given `weight`. It is an error to call this method if this object does not own a particles DAG node, and if the `source` does not own a particles DAG node.

```
void DAGInfo::MtoI(DAGInfo *source, int weight)
```

Connect the normal DAG node of `source` to the intermediate DAG node of this object with operation `MtoI`, and assign the resulting edge the given `weight`. It is an error to call this method if this object does not own an intermediate DAG node, and if the `source` does not own a normal DAG node.

```
void DAGInfo::ItoI(DAGInfo *source, int weight)
```

Connect the intermediate DAG node of `source` to the intermediate DAG node of this object with operation `ItoI`, and assign the resulting edge the given `weight`. It is an error to call this method if this object does not own an intermediate DAG node, and if the `source` does not own an intermediate DAG node.

```
void DAGInfo::ItoL(DAGInfo *source, int weight)
```

Connect the intermediate DAG node of `source` to the normal DAG node of this object with operation `ItoL` operation, and assign the resulting edge the given `weight`. It is an error to call this method if this object does not own a normal DAG node, and if the `source` does not own an intermediate DAG node.

4.9 ViewSet

The `ViewSet` object is a means by which a subset of the possible views of an Expansion can be selected. The DASHMM concept of Expansion allows for multiple mathematical expansions to be contained in the Expansion object. This is a feature used by some advanced methods. A `ViewSet` essentially indexes over the possible views of an Expansion. For example, if an Expansion has six versions (or six views) of the data, then a `ViewSet` might select a subset of these (e.g., views 2, 4 and 5). Like anything in C++, the indices are zero based. The particular meaning of each index is defined only with respect to the Expansion's implementation.

```
ViewSet::ViewSet()
```

Default construct the object.

```
ViewSet::ViewSet(
    ExpansionRole role,
    const Point &center = Point{},
    double scale = 1.0)
```

Create an empty `ViewSet`, while setting a few vital features: the `role` of the represented expansion, the `center` around which it is defined, and the scaling factor `scale` associated with the represented expansion.

```
void ViewSet::clear()
```

Clear out the object. After `clear()`, it will be as if the object were default constructed.

```
void ViewSet::add_view(int index)
```

Add a view with the given `index`. This is an incomplete view.

```
void ViewSet::add_view(int index, size_t bytes, char *data)
```

Add a view with the given `index`, size in `bytes` and `data`. This is a complete view, indicating not only which index, but also the data backing the view.

NOTE: The `ViewSet` does not assume ownership of `data`.

```
void ViewSet::set_bytes(int view, size_t bytes)
```

Set the size of a given view. `view` gives which view of this object to set the size of. That is, `view` is not an index into the original Expansion, but an index into the set of views represented in this object.

```
void ViewSet::set_data(int view, char *data)
```

Set the data for a given view. `view` gives which view of this object to set the size of. That is, `view` is not an index into the original Expansion, but an index into the set of views represented in this object.

```
void ViewSet::set_role(ExpansionRole role)
```

Set the `role` of this object.

```
void ViewSet::set_center(const Point &center)
```

Set the `center` of this object. This represents the point around which the expansion is defined.

```
void ViewSet::set_scale(double s)
```

Set the `scale` of this object.

```
int ViewSet::view_index(int view) const
```

Return the index in the original Expansion that is represented by the given `view`.

```
size_t ViewSet::view_bytes(int view) const
```

Return the size in bytes of the given `view`.

```
char *ViewSet::view_data(int view) const
```

Return the data of the given `view`.

```
ExpansionRole ViewSet::role() const
```

Return the role of the represented expansion.

```
Point ViewSet::center() const
```

Return the center of the represented expansion.

```
double ViewSet::scale() const
```

Return the scale of the represented expansion.

```
int ViewSet::count() const
```

Return the number of views in this object.

```
size_t ViewSet::bytes() const
```

Return the total size of the data represented by this object. This includes not only the data of the views themselves, but some metadata used during serialization of this object.

4.10 Tree nodes

The nodes of the dual tree are implemented as the class `Node<T>`, which is a template over the record type, either Source or Target. Typically these types are aliased to `sourcenode_t = Node<Source>`, and `targetnode_t = Node<Target>`. The members of this class that might be needed by implementers of new Methods are detailed below.

```
Index Node<T>::idx
```

The index of the tree node.

```
Node<T> *Node<T>::parent
```

The parent of this node in the tree. This is only `nullptr` for the root.

```
Node<T> *Node<T>::child[8]
```

The children of this node in the tree. These might be `nullptr`.

```
DAGInfo Node<T>::dag
```

A `DAGInfo` object providing a handle to the relevant portions of the DAG for this node.

```
bool Node<T>::is_leaf() const
```

A utility predicate answering the question: Is this node a leaf?

4.11 Built-in distribution policies

The distribution policy controls on which localities the nodes of the DAG are ultimately placed. Each Method is required to define a distribution policy to be used with that method (see below). DASHMM has a few built-in distribution policies that can be used in user-defined methods. Additionally, there is a default policy defined that represents the current best all-around policy available in DASHMM. To make use of this default in a user-defined Method, one merely has to include the following in the class definition:

```
using distropolicy_t = dashmm::DefaultDistributionPolicy;
```

`DefaultDistributionPolicy` is defined in `dashmm/defaultpolicy.h`.

Each distribution policy only sets the locality of nodes that are not automatically set by DASHMM. DAG nodes that have an automatically determined locality include: the source DAG nodes, the target DAG nodes, and those normal expansions on the source or target side representing the same node of the source or target tree as a source or target DAG node.

The following distribution policies are included with DASHMM:

`SingleLocality`

This distribution policy places all of the DAG nodes on a single locality in the system. The locality can be selected in the constructor, and has a default value of 0: `SingleLocality(int loc = 0)`.

`RandomDistro`

This distribution places DAG nodes around the available localities at random. Note that this is generally speaking a bad idea, but the scaling of this distribution is not bad, even if the raw performance is bad. The seed for the RNG can be set using the constructor, which has a default value: `RandomDistro(int seed = 137)`.

`BHDistro`

This distribution policy, which is the policy used for the `BH` method, is compatible with any method; there is no dependence on its operation on the sorts of DAG nodes or the operations performed on the DAG edges. This policy works by starting at the target DAG nodes and working backwards through the rest of the DAG. The locality of a node being examined is set to the locality which minimizes the communication with other localities. In deciding what is the minimal communication, the weight of the DAG edges is used to approximate the cost of the message.

`FMM97Distro`

This distribution policy only applies to the `FMM97` method. It explores the source and target tree structure within the DAG. The normal and intermediate DAG nodes associated with a node in the source tree, and the normal DAG node associated with a node in the target tree, are placed on the locality that contains the descendant source or target DAG nodes. To determine the placement of the intermediate DAG node associated with a node in the target tree, the policy considers the weight of the DAG edges to minimize communication cost, and the color of the DAG edges to increase slack time to hide communication latency.

4.12 User-defined Expansions

To define an expansion, a user need only create a class that conforms to the following interface. For more details on the relation of Expansions to the mathematics, please see section 4.1 above. To implement a user-defined Expansion, one must be familiar with the following DASHMM constructs: `Point`, `ExpansionRole`, `ViewSet`, `Index`, `Operation` and `dcomplex_t`.

In the following, we shall take the name of the user-defined Expansion to be `Expansion`, but one can use any other name that one wishes.

Expansions are template types with two parameters, the Source and Target types. The following would be the declaration of the user-defined Expansion:

```
template <typename Source, typename Target>
class Expansion;
```

It is not required, but the following aliases will be assumed to be have been defined in the following description of the Expansion interface:

```
using source_t = Source;
using target_t = Target;
using expansion_t = Expansion<Source, Target>;
```

The latter introduces some brevity, while the former are mostly for completeness.

```
Expansion::Expansion(
    ExpansionRole role,
    double scale = 1.0,
    Point center = Point{})
```

This constructor creates the expansion object with the given `center`, `scale` and `role`. There is no obligation on the part of the expansion to use these inputs, but they will be provided for those expansions that will need these data. The scale provided to this constructor will be produced by `Expansion::compute_scale`.

```
Expansion::Expansion(const ViewSet &views)
```

This constructor interprets the provided `views` as the data serving this expansion. Unlike the previous constructor, this constructor produces an object that does not own any data. Instead, it interprets existing data.

Further, this constructor must be able to operate in a mode where `views` is empty. This 'shallow' mode of construction is for cases where the expansion data is not needed, such as `StoT`, but where some of the kernel parameters might be needed.

```
Expansion::~Expansion()
```

The destructor should free the data of the expansion. In some instances, the expansion will not own any data, and so this should do nothing. Only if the expansion is `valid()` should this ever delete allocated memory.

```
void Expansion::release()
```

This will release the internal data for an expansion. These objects need to support the ability to export the data making the expansion, losing ownership of the data in the process. Further, expansion objects need to be able to be constructed in a shallow way from existing data. `release()` breaks the association. After `release()`, calls to `valid()` must return false.

The simplest implementation of this is to have the object store a pointer to memory allocated on the heap (as in `new char [size]`), and `release()` can just set that pointer to `nullptr`.

```
bool Expansion::valid(const ViewSet &views) const
```

Returns if the indicated views are valid. An expansion is valid if it has data associated with it. If `views` is empty, this will check all views.

```
int Expansion::view_count() const
```

Return the current number of views for this object. This will either be the full number for an object created with the first constructor, or a smaller number for an expansion created by interpretation in the second constructor.

```
ViewSet Expansion::get_all_views() const
```

Get all current views of this object.

```
ExpansionRole Expansion::role() const
```

Return the expansion role of this object.

```
Point Expansion::center() const
```

Return the point around which the expansion is defined.

```
size_t Expansion::view_size(int view) const
```

Return the view size for the specified `view`. This returns the number of terms in the expansion for the given view. Do not confuse this with the size in bytes of the data in a given view, which can be obtained with `get_all_views()`.

```
dcomplex_t Expansion::view_term(int view, size_t i) const
```

Get term `i` of the given `view`. The term is returned as a complex number, so real-valued expansions must return a complex number.

```
std::unique_ptr<expansion_t>
Expansion::S_to_M(Point center, source_t *first, source_t *last) const
```

Create a multipole expansion for a given set of sources. The expansion will have the given `center`, and the sources are gives as pointers to the `first` and one past the `last` record. The returned expansion will have a role of `kSourcePrimary`.

```
std::unique_ptr<expansion_t>
```

```
Expansion::S_to_L(Point center, source_t *first, source_t *last) const
```

Create a local expansion for a given set of sources. The expansion will have the given `center`, and the sources are gives as pointers to the `first` and one past the `last` record. The returned expansion will have a role of `kTargetPrimary`.

```
std::unique_ptr<expansion_t>
Expansion::M_to_M(int from_child) const
```

Change the center of a multipole expansion. The shift in the center is specified through `from_child`, which indicates the child from which the expansion is being converted. This expansion will have a role of `kSourcePrimary`. The returned expansion will have a role of `kSourcePrimary`.

```
std::unique_ptr<expansion_t>
Expansion::M_to_L(Index s_index, Index t_index) const
```

Convert a multipole expansion to a local expansion. To specify the change the source and target indices (`s_index` and `t_index`), are provided. This expansion will have a role of `kSourcePrimary`. The returned expansion will have a role of `kTargetPrimary`.

```
std::unique_ptr<expansion_t>
Expansion::L_to_L(int to_child) const
```

Convert a local expansion to a local expansion for a child. The child is specified by `to_child`, which gives the child of the tree node associated with this expansion that the resulting expansion should be associated with. This expansion will have a role of `kTargetPrimary`. The returned expansion will have a role of `kTargetPrimary`.

```
void Expansion::M_to_T(target_t *first, target_t *last) const
```

Apply the effect of a multipole expansion to a set of targets, specified by a pointer to the `first` and one past the `last` target. This expansion will have a role of `kSourcePrimary`.

```
void Expansion::L_to_T(target_t *first, target_t *last) const
```

Apply the effect of a local expansion to a set of targets, specified by a pointer to the `first` and one past the `last` target. This expansion will have a role of `kTargetPrimary`.

```
void Expansion::S_to_T(source_t *s_first, source_t *s_last,
                       target_t *t_first, target_t *t_last) const
```

Apply the direct interaction of a set of sources to a set of targets. The sources and targets are specified by pointers to the first and one past the last record.

```
std::unique_ptr<expansion_t> Expansion::M_to_I() const
```

Create an intermediate expansion from a multipole expansion. This expansion will have a role of `kSourcePrimary`. The returned expansion will have a role of `kSourceIntermediate`.

```
std::unique_ptr<expansion_t>
Expansion::I_to_I(Index s_index, Index t_index) const
```

Translate a source-side intermediate expansion into a target-side intermediate expansion. The `s_index` and `t_index` are the index of the tree nodes that are represented by this object, and the resulting expansion, respectively. This expansion will have a role of `kSourceIntermediate`. The returned expansion will have a role of `kTargetIntermediate`.

```
std::unique_ptr<expansion_t>
Expansion::I_to_L(Index t_index) const
```

Translate a target-side intermediate expansion into a local expansion. The target tree node's index, `t_index`, is provided. This expansion will have a role of `kTargetIntermediate`. The returned expansion will have a role of `kTargetPrimary`.

```
void Expansion::add_expansion(const expansion_t *temp)
```

Add the given expansion to this expansion. Typically this involves summing the coefficients, but can be more involved in some cases.

```
static void Expansion::update_table(
    int n_digits,
    double domain_size,
    const std::vector<double> &kernel_params)
```

Update a kernel table. This should generate or update a kernel table associated with this expansion type. The kernel table is a mechanism for precomputing values that are required by expansion operations. As the same table should serve every instance of this class, the table should be a static member of the class. This routine will either allocate and fill these tables (if this is the first call to this function during the program lifetime, or if the table has been explicitly deleted) or it will replace the values in the table without necessarily reallocating the memory for the table (if this is a subsequent call to this function).

The value of `n_digits` and `kernel_params` are ultimately provided by the call to `evaluate()`. The `domain_size` is computed by DASHMM after the tree is constructed.

```
static void Expansion::delete_table()
```

Destroy the kernel table. This should destroy any tables that exist that are associated with this type of expansion.

```
static double Expansion::compute_scale(Index index)
```

Compute the scale to pass into expansion constructors. The will only be called after the table exists, so the implementation can rely on the existence of the table. In particular, kernel parameters needed by this routine should be stored in the table.

```
static int Expansion::weight_estimate(
    Operation op,
    Index s = Index{},
    Index t = Index{})
```

Compute an estimate of the cost to send the result of the given operation across the network. In some expansions, the operation is enough to determine the cost. In others, the source and target indices, `s` and `t`.

4.13 User-defined Methods

To create a user-defined Method, one must create a class that conforms to the following interface. For a description of the four primary routines of a Method, please see 4.1. To implement a user-defined Method, one must be familiar with the following DASHMM constructs: the `Expansion` concept, nodes of the tree (both source and target), `DomainGeometry` and `DAGInfo`.

In the following we shall take the name of the user-defined Method to be `Method`, but one can use any name that one wishes.

Methods are templates over three parameters, the Source, Target and Expansion types. The following is the full declaration of a user-defined Method:

```
template <typename Source, typename Target,
          template <typename, typename> class Expansion>
class Method;
```

It is helpful to also define the following aliases as members of any Method:

```
using source_t = Source;
using target_t = Target;
using expansion_t = Expansion<Source, Target>;
using method_t = Method<Source, Target, Expansion>;
using sourcenode_t = Node<Source>;
using targetnode_t = Node<Target>;
```

The above aliases will be used in the following description.

```
Method::distropolicy_t
```

Any method must define this type to specify the distribution policy that is to be used with the method. This allows for the possibility that special details of the implemented `Method` might allow for better distribution of the DAG. If there is no such better distribution, or the implementer is not concerned, or if the default provided by DASHMM is found to be sufficient, then one can use `dashmm::DefaultDistributionPolicy` defined in `dashmm/defaultpolicy.h`.

```
Method::Method()
```

Methods must have a default constructor. Note that other constructors are allowed. Methods instances used for particular evaluations are copied so methods can support runtime parameters.

```
void Method::generate(sourcenode_t *curr, DomainGeometry *domain) const
```

This routine is called at the leaves of the source tree to generate the initial expansions from the source data. The leaf node in question is provided via `curr`. For use in cases where the size of the node is relevant, the problem `domain` is also provided. The typical interaction with `curr` is to use its `dag` member.

`generate` is responsible for creating the DAG nodes that the method requires for the given source tree leaf node. Further, any operations between these nodes should be scheduled during `generate` (via the many `DAGInfo` methods). The typical operation scheduled during `generate` is `StoM`.

```
void Method::aggregate(sourcenode_t *curr, DomainGeometry *domain) const
```

This routine is called for the internal nodes of the source tree to generate any expansions needed by the method that are associated with non-leaf source tree nodes. The current node, `curr`, and the overall `domain` are provided.

`aggregate` is responsible for creating the DAG nodes that the method requires. Further, operations between nodes should be scheduled. The typical operation for `aggregate` is `MtoM`.

```
void Method::inherit(
    targetnode_t *curr,
    DomainGeometry *domain,
    bool curr_is_leaf) const
```

In `inherit` information from parents in the target tree are propagated to children. In addition to the node, `curr` and the `domain`, whether is the current node is a leaf if indicated with `curr_is_leaf`. This is necessary as some methods might allow for internal target tree nodes to nonetheless act as if they were leaves.

`inherit` is responsible for creating the DAG nodes that the method requires, and for scheduling the operations between DAG nodes. The typical operation for `inherit` is `LtoL`.

```
void Method::process(
    targetnode_t *curr,
    std::vector<sourcenode_t *> &consider,
    bool curr_is_leaf,
    DomainGeometry *domain) const
```

The most complex work of the method often takes place in `process`. In addition to the current node, the domain geometry, and if this leaf is a node, a vector of source tree nodes are provided. This vector, `consider`, gives the list of source nodes that might have an impact on the calculation of the potential for the target locations represented by this node.

`consider` should be examined, and where possible elements should be removed from the vector when their effect on the targets in `curr` can be computed. That is, if some operation is scheduled between a node in `consider` and `curr`, then that source node has been handled, and can be removed. In some instances, a node might be removed from `consider` and the children of that node will be added to consider.

At the end of `process`, the set of nodes in `consider` will be different than were passed into this function. The resulting `consider` will be passed to the children of `curr`, unless `curr_is_leaf` was true. In that case, it should be that `consider` will have been completely used by the call to `process`.

```
void Method::refine_test(
    bool same_sources_and_targets,
    const targetnode_t *curr,
    const std::vector<sourcenode_t *> &consider) const
```

This method determines if the given target tree node should be refined, or if it represents a leaf of the DAG. This decision is based on `curr`, `consider` and if the sources and targets are identical in this evaluation. The latter is provided as `same_sources_and_targets`.

This method should return `true` if the target tree should be refined further, and `false` otherwise.

4.14 User-defined distribution policies

For users defining their own Methods, it can potentially be helpful to also define a distribution policy for that method if the built-in methods cannot take advantage of details of the generated DAG. This section outlines the concept of `DistributionPolicy`, giving details on the required interface. Distribution policies have a relatively small interface. For the sake of discussion, we shall call the user-defined policy `Policy`, but one can give any name to the policy.

```
Policy::Policy()
```

Distribution policies require either a default constructor, or a constructor with all arguments given default values.

```
void Policy::compute_distribution(DAG &dag)
```

This is the main distribution method. After the DAG is fully discovered, it will be passed into this function where the localities of every DAG node need to be given a value. Because DASHMM sets a few localities in a fixed way, only those DAG nodes with a locality set to `-1` (the default value) should have a locality assigned. Changing the locality of a DAG node that has already had a locality assigned will lead to program failure and termination.

Other than the previous two conditions, there are no other restriction on the implementation. However, it is generally best to avoid very deep recursion. HPX-5 threads each have a stack associated with them, and these stacks are generally small, given that there will be a large number of threads. Thus, it is much easier to overrun a thread's stack.

```
void Policy::assign_for_source(DAGInfo &dag, int locality)
```

The routine is called for every source tree node during DAG discovery. It is an opportunity to set localities when the DAG nodes are created, and not all at once in `compute_distribution`.

WARNING: There is inadequate contention management available to safely use this routine without a thorough understanding of the internal workings of DASHMM. It is strongly advised to not give this method a non-trivial implementation.

NOTE: This is an experimental feature, and should not be considered to be part of the DASHMM API; it may be removed in the future.

```
void Policy::assign_for_target(DAGInfo &dag, int locality)
```

The routine is called for every target tree node during DAG discovery. It is an opportunity to set localities when the DAG nodes are created, and not all at once in `compute_distribution`.

WARNING: There is inadequate contention management available to safely use this routine without a thorough understanding of the internal workings of DASHMM. It is strongly advised to not give this method a non-trivial implementation.

NOTE: This is an experimental feature, and should not be considered to be part of the DASHMM API; it may be remove in the future.