DASHMM Basic User Guide

The Dynamic Adaptive System for Hierarchical Multipole Moments (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 expansion, DASHMM includes built-in methods and expansions, including the Barnes-Hut (BH) and Fast Multipole Method (FMM), and two expansions implementing the Laplace Kernel used in electrostatics and Newtonian gravitation. Although only the Laplace kernel is currently implemented in DASHMM, in future versions more kernels will be added.

This document covers the basic use of the DASHMM library. More functionality is exposed through the advanced interface to DASHMM, including the ability to define and register methods and expansions with the library. For instructions on the advanced interface, please see the DASHMM Advanced User Guide. Instructions for installing and building applications using the library can be found in the DASHMM Installation Guide. Finally, the latest information, resources and tutorials can be found at the DASHMM webpage: https://www.crest.iu.edu/projects/dashmm/

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.

Introduction to DASHMM

DASHMM is a library built using C++ templates parameterized by four types: a source data type, a target data type, an expansion type and a method type. The source and target data types are used to specify the information that the user presents to DASHMM. These give positions for both the sources and targets, as well as providing the charge for the sources, and a location in which to store the results of the evaluation in the targets. The expansion conceptually encompasses the kernel being used (e.g. Laplace) and the details about how that kernel is expanded into multipole moments (e.g. using spherical harmonics). The method encompasses both how the various expansions are generated and how those expansions are used.

DASHMM is built using the advanced runtime system, HPX-5, but the basic interface to DASHMM does not require any knowledge of how to use HPX-5. Instead, the basic interface handles the distribution of the parallel work. The model of use for the basic interface to DASHMM is to take a serial code and make some calls to the DASHMM library. In so doing, the program will make use of parallel resources. In this mode of operation, when execution is outside a DASHMM library call, the runtime can be considered to be not operating. Data saved in the global address space provided by HPX-5 persists between calls, but the runtime is not executing any operations.

To present data to DASHMM, the user will make use of the `Array` object. `Array` objects live in the global address space provided by HPX-5. The user will need to allocate an `Array`, and put or get memory from that array.

The evaluation of a particular multipole method is accomplished through an `Evaluator` object. These objects have two responsibilities: they register a number of required actions with the HPX-5 runtime, and they perform multipole method evaluations. `Evaluator`s take input from `Array` objects, and place the results into an `Array` object.

Types Defined by DASHMM

DASHMM provides a few types that used throughout the system. The following are those needed in the basic interface.

Return codes from DASHMM library calls are all of the `ReturnCode` (non-scoped) enumeration type. The possible values are `kSuccess`, `kRuntimeError`, `kIncompatible`, `kAllocationError`, `kInitError`, `kFiniError`, and `kDomainError`. See below for cases in which these values might be returned.

DASHMM aliases `std::complex<double>` as `dcomplex_t`, which is used in a number of places in the basic interface.

Finally, there is a `Point` class that encapsulates the notion of a three-dimensional location. `Point`s are used to specify the locations of sources and targets. A full description of `Point` can be found in `/path/to/dashmm/include/dashmm/point.h`, but the most important methods will be covered here. `Point`s are constructed from three `double`s giving the position of the point. The coordinates of the point can be accessed by name (the `x()`, `y()` and `z()` methods) or by index (e.g. `somepoint[1]` for the y coordinate).

Initializing DASHMM

To use DASHMM, one must initialize and finalize the library. These tasks are accomplished with two library functions.

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

To start the runtime system, a user must call `init` and provide the address of the command line arguments used to launch the program. The runtime has some settings that can be controlled from the command prompt, and so these must be handed to the runtime. For a list of command line arguments that the runtime accepts, please see the DASHMM Advanced User Guide, or the HPX-5 documentation. After `init`, any runtime related command line argument will have been removed from `argv`, and `argc` will be updated accordingly.

Only a single call to `init` is allowed in any program that makes use of DASHMM. The majority of the use of DASHMM must occur *after* the call to `init`. For important exceptions, see Evaluators and Mapping Actions onto Arrays below.

On successful initialization of the runtime and of DASHMM, `kSuccess` is returned. Otherwise `kInitError` is returned indicating some problem.

```
ReturnCode finalize()
```

Before finishing a program, the user must call `finalize` to shut down the runtime. Once `finalize` returns, the user's program is free to continue performing any other work required, but all DASHMM and runtime resources will have been destroyed. So any data that is to be read from DASHMM's internal state must occur before this call.

All calls to DASHMM library routines must occur before the call to `finalize`. Only a single call to `finalize` is allowed in any program that makes use of DASHMM.

If `finalize` is successful, `kSuccess` will be returned. Otherwise, `kFiniError` will be returned.

Array Objects

To provide data to DASHMM, one must make use of `Array` objects. An `Array` is a class that handles a chunk of associated global memory. Further, `Array` is a template class, with a parameter that can be any trivially copyable type. The relevant members of `Array<T>` are (see Mapping Actions onto Arrays below for another useful member of `Array<T>`):

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

Array objects possess a single constructor that takes one argument that has a default value. For the basic use of DASHMM, this default value is all that is needed. Interested users can consult the DASHMM Advanced User Guide for more information.

```
ReturnCode Array<T>::allocate(size_t record_count)
```

After an array object is created, `allocate` will allocate space in the global address space for `record_count` records whose size is `sizeof(T)`. The distribution of these records is chosen by DASHMM. Further, for `Array`s used during an evaluation, the distribution and the ordering of the records are subject to change. This allows DASHMM to make changes that will benefit the scaling and performance of the evaluation. It is an error to call `allocate` more than once before a call to `destroy`.

This routine returns `kSuccess` on success, `kRuntimeError` if there is an error from the runtime, `kAllocationError` if the request fails because of a lack of available resources, and `kDomainError` if the `Array` already has some global memory allocated to it.

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

This function instructs DASHMM to reclaim the resources used by an array object. It is an error to use `put` or `get` after the object has been deallocated. On success, this routine return `kSuccess`. If there is an error from the runtime, this routine returns `kRuntimeError`.

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

Once an array is allocated, the records can be filled with a call to `put`. The provided `in_data` is copied into the array's records in the range [`first`, `last`). It is the user's responsibility to assure that the buffer pointed to by `in_data` contains sufficient data to fill the given number of records.

This routine returns `kSuccess` on successful put, `kRuntimeError` if there is an error with the runtime, or `kDomainError` if there are problems with the given range (e.g. `last` is beyond the end of the array).

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

Data may be retrieved from an array using a call to `get`. Similar to `put`, the data retrieved is from records in the range [`first`, `last`). The data is placed into `out_data`. It is the user's responsibility to assure that `out_data` has sufficient capacity for the retrieved data.

The routine returns `kSuccess` on success, `kRuntimeError` if there is a runtime error, or `kDomainError` if the range specified is incompatible with the underlying array.

Source and Target Data

Evaluation of multipole methods require source and target data provided by an Array. DASHMM relies on C++ templates to allow for some flexibility for the user. Two important type parameters that appear in DASHMM evaluations are the Source and Target types. These types specific what data each source and target carries. Each expansion will place different requirements on the Source and Target types, but one requirement is universal: the types used for Sources and Targets must be trivially copyable.

For the built-in expansions provided with the current version of DASHMM, the requirements on Source types are identical. The source must provide a member of type `Point` called `position`, and a member of type `double` called `charge`. Additional data may be provided beyond that that the user might fine useful for their particular case. For example, the following would be a valid Source type:

```
struct SourceData {
  Point  position;    // where is the source?
  double charge;      // what is its charge?
  int    something;   // user data ignored by DASHMM
  double number;      // user data ignored by DASHMM
};
```

The built-in expansions provided with the current version of DASHMM all place identical requirements on the Target type. The target must provide a member of type `Point` called `position`, and a member of type `dcomplex_t` called `phi` for most cases. See the specific expansions below for details.

It is permissible to have both the Source and Target types be the same (for instance, if you are computing the self-interaction of a number of bodies) so long as all required fields are present. Note in particular, the `position` will serve as position for both the sources and targets in this case.

Evaluators

The central object in DASHMM is the `Evaluator` object. This object not only sets up the specific case for the chosen method and expansion, but also performs the evaluation. `Evaluator` objects are template classes that require four template arguments: the Source type, the Target type, the Expansion type and the Method type. Source and Target types have been discussed above. The built-in expansion and method types will be covered below.

An `Evaluator` object should only be created once for each combination of template arguments. When `Evaluator`s are constructed, they handle registration of various actions (for more on actions, see the HPX-5 documentation) specific for the chosen types. This registration must happen only once, so it is an error to create multiple instances of a particular `Evaluator`. Fortunately, there is no hardship imposed by this restriction because `Evaluator`s are stateless, and so having multiple instances does not benefit the user.

The single instance of a given `Evaluator` must be created before `dashmm::init` is called. Creation of `Evaluator`s after `init` is an error.

```
Evaluator<Source, Target, Expansion, Method>::Evaluator()
```

Construction of an `Evaluator` instance is simple; there are no arguments to `Evaluator`'s constructor. For example, if we wish to perform a multipole evaluation using `FMM` on a `LaplaceSPH` expansion for `UserSource` and `UserTarget`, then one creates an object as follows:

```
Evaluator<UserSource, UserTarget, LaplaceSPH, FMM> eval{};
```

Any Evaluator object will define a number of type aliases from their template arguments. In particular `source_t`, `target_t`, `expansion_t` and `method_t` are defined. The first two are of little utility, but the latter two are a savings given that both expansions and methods are themselves template classes (see the sections on built-in methods and expansion for more). `expansion_t` and `method_t` both contain the fully specified type. In the previous example, we would have `expansion_t` being equivalent to `LaplaceSPH<UserSource, UserTarget>`, and `method_t` being equivalent to `FMM<UserSource, UserTarget, LaplaceSPH>`.

```
ReturnCode Evaluator<Source, Target, Expansion, Method>::evaluate(
        Array<Source> &sources, Array<Target> &targets,
        int refinement_limit,
        const method_t &method,
        const expansion_t &expansion)
```

The central call in the basic interface to DASHMM is `evaluate`. This routine performs the multipole method evaluation, computing the potential at the specified `targets` as a result of the specified `sources`. This will use the provided `method`, and the provided `expansion` of the potential in question. The built-in methods can be obtained from some methods covered below.

The source and target points are provided via DASHMM `Array` objects. DASHMM will likely sort both arrays, and so the user should not count on the ordering of the records after a call to evaluate. If the original ordering is important, the user will need to add an index to the Source and Target types. It is

permissible to have both `sources` and `targets` be the same array so long as the Source and Target types are identical.

During the evaluation, two hierarchical space-partitioning trees are constructed, one each for the sources and targets. The `refinement_limit` specifies the refinement termination criterion. If a given tree node contains fewer sources or targets than the `refinement_limit`, the partitioning halts.

The user must provide actual objects of `expansion_t` and `method_t` to `evaluate`. The particular Expansion or Method types might have parameters that specify fully their behavior (for instance, the number of digits of accuracy for the `LaplaceSPH` expansion, or the critical angle for the `BH` method; see below).

`evaluate` can be called multiple times from the same instance of an `Evaluator`. These evaluations do not have to share any arguments with other evaluations, so the user can use completely different data, the same data but different parameters for the passed in expansion, or every argument can be identical.

This routine returns `kSuccess` on successful evaluation, or `kRuntimeError` if there is some error in the runtime.

Built-in Expansions

Expansions in DASHMM represent not only a particular kernel, but a particular way of expanding the given potential. Many expansion operations use either source or target data and so Expansions in DASHMM are also template classes, parameterized by the Source and Target types.

Currently, DASHMM provides one built-in kernel, the Laplace kernel. This potential represents the potential of a point charge in electrostatics, or a point mass in Newtonian gravitation. This kernel is exposed through three different expansions.

`LaplaceCOM<Source, Target>::LaplaceCOM()`

This expansion is an expansion about the center of mass of the represented sources. This expansion includes contributions up to the quadrupole term, but because it is an expansion about the center of mass, the dipole term is identically zero. This expansion is well suited to gravitation and the `BH` method, and should not be used for problems with both signs of charge. This expansion is not compatible with `FMM`; this expansion does not implement all of the required operations for use with `FMM`.

This expansion requires `Source` to have a member of type `Point` called `position` and a member of type `double` called `charge`. This expansion requires `Target` to have a member of type `Point` called `position` and a member of type `dcomplex_t` called `phi`.

The `LaplaceCOM` type defines a number of other methods, but the only relevant method for basic use of DASHMM is the constructor. An object of this type must be passed into `Evaluator::evaluate`.

`LaplaceCOMAcc<Source, Target>::LaplaceCOMAcc()`

This is similar to the LaplaceCOM, except that this will compute the acceleration rather than the potential.

This expansion requires `Source` to have a member of type `Point` called `position` and a member of type `double` called `charge`. This expansion requires `Target` to have a member of type `Point` called `position` and a member of type `double [3]` called `acceleration`.

`LaplaceSPH<Source, Target>::LaplaceSPH(int n_digits)`

This expansion is intended for use with the `FMM` This expansion implements all relevant operations, and provides a variable length expansion so that the user provided accuracy limit, given by `n_digits`, can be reached.

Before an object of this type can be constructed for a given n_digits, the user must call `laplace_sph_precompute(int n_digits)`. This routine performs some one-time initialization of coefficients used by `LaplaceSPH`, and needs to be called only once for each value of `n_digits`.

This expansion requires `Source` to have a member of type `Point` called `position` and a member of type `double` called `charge`. This expansion requires `Target` to have a member of type `Point` called `position` and a member of type `dcomplex_t` called `phi`.

Built-in Methods

Methods in DASHMM need to know on which Expansion type they are being applied. And because Expansions need to know the Source and Target types, Methods in DASHMM are templates over the Source, Target and Expansion types.

DASHMM currently provides three built-in methods for immediate use, the Barnes-Hut Method, the Fast Multipole Method, and a direct summation method. As with the built-in expansions, the built-in methods have other routines. But also, just like the built-in expansion, it is only the constructor that is needed in basic use of DASHMM.

`BH<Source, Target, Expansion>::BH(double theta)`

This instance of the BH method will use the provided `theta` as the critical angle for deciding if a given expansion is usable. This criterion matches exactly the criterion introduced by Barnes & Hut. When `theta` is 0, this method is effectively the same as `Direct`.

`FMM<Source, Target, Expansion>::FMM()`

Users of `FMM` will not need to specify the error tolerance with this function. Instead, the error tolerance is specified by the expansion; the more terms in the expansion, the lower the error. See `LaplaceSPH` above for details.

`Direct<Source, Target, Expansion>::Direct()`

This method performs the potential computation using the direct summation technique. This is intended as a means for comparing approximate potentials computed with another method with the 'exact' answer. As it is direct summation, this method will take a long time to finish for even modest source and target counts.

Mapping Actions onto Arrays

DASHMM also allows the user to specify functions that are applied to each element of an `Array`. This is useful in cases where multiple evaluations occur with simple work occurring between each. The prototypical example of this is a time-stepping code that uses multipole methods to compute the acceleration of the particles which then have their positions updated. This mechanism is provided through the `ArrayMapAction` object.

There are two reasons to let DASHMM handle such a task: the memory stored in the Array object is in HPX-5's global address space, and DASHMM can parallelize the work. If DASHMM is not used for this, the user would need to copy the Array data out of the global address space with Array::get before operating on it. Then, the user would need to call Array::put to return the data to the global address space so the might perform another multipole moment evaluation.

The `ArrayMapAction` object is a template class with two template parameters. The first specifies the record type of the array to which this action will be applied. The second specified the type of the environment that will be passed to the action. The environment is any data that is needed by the action that is not stored in the array records. For the example of time-stepping, the environment would contain at least the time step.

Like the `Evaluator` object, the `ArrayMapAction<T, E>` should be declared before `dashmm::init()`; the runtime needs to have the action represented by the `ArrayMapAction` registered. The constructor for the ArrayMapAction takes a single argument, a pointer to the function implementing the action to be performed on the records of the array. Once the object is constructed, a member of `Array<T>` will apply the action. For an example that uses this facility, please see `/path/to/dashmm/demo/stepping`.

`ArrayMapAction<T, E>::ArrayMapAction(`

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

This will create an object that represents an action that can be applied to each record of an `Array<T>` object. The type `E` specifies the type of an environment that will be passed to the function `fcn` on each application to a record in the array.

The first argument to the provided function will be the base address of a subset of the records to which the action will be applied. The second argument to the provided function will give the number of records that this invocation of the function will treat. The third argument gives the total offset in the array to the first record treated by this function. Note that this does not mean that the indexing inside the function should begin with the third argument; instead, the indexing should begin with 0 and run up to but not including the first argument. The third argument is provided for convenience if it should be

important to know in a specific case what the absolute offset is in the array. The last argument to the function will be an environment provided by the user.

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

This will apply the given action to all the records in the array on which the `map` method is called. The provided environment will be used for each invocation of the function. This will return `kSuccess`.

DASHMM Example

This section presents a short example of the use of DASHMM. Details of the functions called can be found above. In the following we assume that there are two functions that generate the source and target data using information from the command line, and one function to do something with the results. Also, to keep the example brief, return codes are not checked. Real applications should check the result of each DASHMM library call.

```
#include <dashmm/dashmm.h>

struct source {
  Point position;
  double charge;
};

struct target {
  Point position;
  Dcomplex_t phi;
};

dashmm::Evaluator<source, target, LaplaceSPH, FMM> fmmeval{};

int main(int argc, char **argv) {
  dashmm::init(&argc, &argv);

  int n_sources{0};
  source *S = generate_sources(argc, argv, &n_sources);
  dashmm::Array<source> S_handle{};
  s_handle.allocate(n_sources);
  s_handle.put(0, n_sources, S);

  int n_targets{0};
  target *T = generate_targets(argc, argv, &n_targets);
  dashmm::Array<target> T_handle{};
  T_handle.allocate(n_targets);
  T_handle.put(0, n_targets, T);

  laplace_sph_precompute(6);
  LaplaceSPH<source, target> expansion{6};

  FMM<source, target, LaplaceSPH> method{};

  int refinement_limit{10};
  fmmeval.evaluate(S_handle, T_handle, refinement_limit,
                   method, expansion);

  T_handle.get(0, n_targets, T);
  do_something_with_results(T, n_targets);

  S_handle.destroy();
  T_handle.destroy();
  delete [] S;
  delete [] T;
  dashmm::finalize();

  return 0;
}
```