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.

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, make some calls to the DASHMM library, and 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. More options are available in the DASHMM Advanced User Guide, including interoperability with legacy MPI codes, or use in HPX-5 aware applications.

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.

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/

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.

Objects living in the global address space provided by DASHMM have handles of the type `ObjectHandle`. These can be thought of like an opaque pointer. Routines that create objects and return references to those objects will return `ObjectHandle`s. Routines that require references to objects will take `ObjectHandle`s.

Specification of a particular multipole method is via a class derived from an abstract base class, `Method`. Typically, unless the user is implementing their own methods, routines in DASHMM will use pointers to the base class. For details on the required interface for a derived `Method`, and how to register new methods with DASHMM, please see the DASHMM Advanced User Guide.

Specification of a particular multipole expansion is via a class derived from an abstract base class, `Expansion`. Typically, unless the user is implementing their own expansions, routines in DASHMM will use pointers to the base class. For details on the required interface for a derived `Expansion`, and how to register new expansions with DASHMM, please see the DASHMM Advanced User Guide.

Basic Interface

The basic interface to DASHMM comprises seven functions. Two for starting and stopping the runtime system, four for providing data to, and getting data from, DASHMM, and `evaluate`, which performs the multipole method calculation.

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

To start the runtime system, a user must call `init` and provide references to 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. After `init`, any runtime related command line argument will have been removed from `argv`, and `argc` will be updated accordingly. In addition to setting up the runtime, `init` will setup some internal bookkeeping for the DASHMM library.

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

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, and to free the DASHMM specific resources acquired. 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.

```
ReturnCode allocate_array(size_t count, size_t size,
                          ObjectHandle *obj)
```

To provide data to DASHMM, one must make use of array objects. This function will request an array object that can hold `count` objects, each with a size in bytes of `size`. If DASHMM is successful in creating an array of the requested size, a handle to that array will be returned via the parameter `obj`.

This routine provides no user control of the distribution of the records in the array, leaving it instead to DASHMM to provide a good guess. Further, the distribution of the data may change during execution.

This routine returns `kSuccess` on success, `kRuntimeError` if there is an error from the runtime, and `kAllocationError` if the request fails because of a lack of available resources.

```
ReturnCode deallocate_array(ObjectHandle obj)
```

This function instructs DASHMM to reclaim the resources used by an array object. The object handle provided must be a handle to an object allocated with `allocate_array`. It is an error to use `array_put` or `array_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_put(ObjectHandle obj, size_t first, size_t last,
                     void *in_data)
```

Once an array object is created, to copy values into that array so that DASHMM might use them (in `evaluate`, for example) one must use `array_put`. The provided `in_data` is copied into the array object with handle `obj`, in the 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 the provided object handle is not an array, or if there are problems with the given range (e.g. `last` is beyond the end of the array).

```
ReturnCode array_get(ObjectHandle obj, size_t first, size_t last,
                     void *out_data)
```

Once DASHMM has produced results into an array, the data may be obtained from DASHMM using `array_get`. Similar to `array_put`, the array object is specified by its handle, `obj`. 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 provided object handle is not an array, or if the range specified is incompatible with the underlying array.

```
ReturnCode evaluate(ObjectHandle sources, int spos_offset,
                    int q_offset, ObjectHandle targets,
                    int tpos_offset, int phi_offset,
                    int refinement_limit,
                    std::unique_ptr<Method> method,
                    std::unique_ptr<Expansion> 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 points are provided via a DASHMM array object. The user provides a handle to the array, the offset in each record to the position (three `double` values stored contiguously), `spos_offset`, and the offset in the record to the charge (a double), `q_offset`. Similarly, the user must specify an object handle to the DASHMM array containing the target records, an offset in each target record to the target position (three double values stored contiguously), `tpos_offset`, and the offset in each target record to the location to save the computed potential value (two doubles to allow for complex potentials), `phi_offset`.

DASHMM will only modify `targets` by putting the potential results in the specified offset for each record. Similarly, DASHMM will not modify `sources`, unless `sources` and `targets` are the same array.

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 than the `refinement_limit`, the partitioning halts.

This routine return `kSuccess` on successful evaluation, `kIncompatible` if the specified method and expansion cannot be used together, or `kRuntimeError` if there is some error in the runtime.

Built-in Methods

DASHMM currently provides three built-in methods for immediate use, the Barnes-Hut Method, the Fast Multipole Method, and a direct summation method. Instances of these methods can be obtained through three members of the `dashmm` namespace.

```
Method *bh_method(double theta)
```

The returned `Method` object will be usable anywhere in DASHMM requiring a method to be specified. 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[1].

[1] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. Nature, 324, 446-449 (1986)

`Method *fmm_method()`

The returned `Method` object will be usable anywhere in DASHMM requiring a method to be specified. The user will not need to specify the error tolerance with this function. For FMM, the error tolerance is instead specified by the expansion; the more terms in the expansion, the lower the error. See Built-In Expansion, below for details.

`Method *direct_method()`

The returned `Method` object will be usable anywhere in DASHMM requiring a method to be specified. 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.

Built-in Expansions

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 with two different variations of the multipole expansion.

`Expansion *laplace_COM_expansion()`

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 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.

`Expansion *laplace_sph_expansion(int n_digits)`

This expansion is intended for use with the FMM (obtained from `fmm_method`). 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.

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.h>

struct source {
  double pos[3];
  double q;
};

struct target {
  double pos[3];
  double phi;
};

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

  int n_sources{0};
  source *S = generate_sources(argc, argv, &n_sources);
  int n_targets{0};
  target *T = generate_targets(argc, argv, &n_targets);

  ObjectHandle S_handle;
  dashmm::allocate_array(n_sources, sizeof(source), &S_handle);
  dashmm::array_put(S_handle, 0, n_sources, S);

  ObjectHandle T_handle;
  dashmm::allocate_array(n_targets, sizeof(target), &T_handle);
  dashmm::array_put(T_handle, 0, n_targets, T);

  int refinement_limit{10};
  dashmm::evaluate(S_handle, offsetof(source, pos),
                   offsetof(source, q),
                   T_handle, offsetof(target, pos),
                   offsetof(target, phi), refinement_limit,
                   dashmm::fmm_method(),
                   dashmm::laplace_sph_expansion());

  dashmm::array_get(T_handle, 0, n_targets, T);
  do_something_with_results(T, n_targets);

  dashmm::deallocate_array(S_handle);
  dashmm::deallocate_array(T_handle);
  delete [] S;
  delete [] T;

  dashmm::finalize();
  return 0;
}
```