Commun. Comput. Phys.
doi: 10.4208/cicp.030316.310716sw

Vol. **20**, No. 4, pp. 1106-1126
October 2016

COMPUTATIONAL SOFTWARE

DASHMM: Dynamic Adaptive System for Hierarchical Multipole Methods

J. DeBuhr[1,*], B. Zhang[1], A. Tsueda[2], V. Tilstra-Smith[3] and T. Sterling[1]

[1] *Center for Research in Extreme Scale Technologies, School of Informatics and Computing, Indiana University, Bloomington, IN, 47404, USA.*
[2] *College of Arts and Sciences, Loyola University Chicago, Chicago, IL, 60660, USA.*
[3] *Department of Physics and Mathematics, Central College, Pella, IA, 50219, USA.*

Received 3 March 2016; Accepted (in revised version) 31 July 2016

Abstract. We present DASHMM, a general library implementing multipole methods (including both Barnes-Hut and the Fast Multipole Method). DASHMM relies on dynamic adaptive runtime techniques provided by the HPX-5 system to parallelize the resulting multipole moment computation. The result is a library that is easy-to-use, extensible, scalable, efficient, and portable. We present both the abstractions defined by DASHMM as well as the specific features of HPX-5 that allow the library to execute scalably and efficiently.

AMS subject classifications: 15A06, 31C20, 68N19

Key words: Barnes-Hut method, fast multipole method, Laplace potential, ParalleX, runtime software.

Program summary

Program title: DASHMM

Nature of problem: Evaluates the Laplace potentials at N target locations induced by M source points.

Software license: BSD 3-Clause

CiCP scientific software URL: `http://www.global-sci.com/code/dashmm-0.5.tar.gz`

Distribution format: .gz

Programming language(s): C++

*Corresponding author. *Email address:* `jdebuhr@indiana.edu` (J. DeBuhr)

Computer platform: x86_64

Operating system: Linux

Compilers: GCC 4.8.4 or newer; icc (tested with 15.0.1)

RAM:

External routines/libraries: HPX-5 2.1.0 or later

Running time:

Restrictions: Currently only supports shared memory, but library will be extended to multiple nodes

Supplementary material and references: `https://www.crest.iu.edu/projects/dashmm/` `http://hpx.crest.iu.edu/`

Additional Comments:

1 Introduction

Multipole methods are a key computational kernel in a wide variety of scientific applications spanning multiple disciplines. However, these applications are emerging as scaling-constrained when using conventional parallelization practices. Emerging, dynamic task management execution models can go beyond conventional programming practices to significantly improve both efficiency and scalability for algorithms exhibiting irregular and time-varying executions. Multipole method calculations are an example of an irregular application that would benefit from the use of a dynamic adaptive runtime system approach. In this paper, we present DASHMM, a library leveraging the power and expressibility of an experimental runtime system to improve the efficiency and scalability of multipole method calculations, to solve ever more challenging end-user problem domain applications.

The Barnes-Hut (BH) [5] method and the Fast Multipole Method (FMM) [11], both exemplars of multipole methods, are used extensively in applications [7, 10, 19, 23]. The fundamental building blocks of these methods are quite similar, and so it is worth constructing a generalization of multipole methods that encompasses both BH and FMM. In practice, many variations and improvements on the original forms of these methods are used; a library that provides only a few rigidly defined methods would be of little use. Instead, DASHMM outlines a set of abstractions that are general enough to allow both BH and the FMM to be implemented as well as many variations on the theme of multipole methods. With the abstractions defined, the task is to effectively parallelize the resulting general multipole method.

Considerable prior effort has been exerted on multipole methods. The first parallel FMM algorithm is introduced by Greengard and Gropp in its 2D uniform version on shared memory architecture [9]. Zhao and Johnsson studied the parallelization of 3D uniform FMM on the connection machine [34]. The parallelization strategy follows

closely to the levelwise implementation of the sequential FMM. This means, operations at any scale do not start until those at the level below (above) in the upward (downward) pass finish. This approach has been kept mostly intact in many following works. The first work on adaptive FMM came from Board et al. [13, 14] in the context of molecular dynamics simulation. Subsequent work on parallel multipole methods focused on load balancing among available computing resources. On distributed memory architectures methods like orthogonal recursive bisection, locally essential tree, and hashed oct-tree [26, 27] are used in different application contexts [15–17, 20, 25, 29, 31]. Special attention was taken to hide communication latency within the computation. Other representative work on distributed-memory machines include the distribution independent adaptive tree [3], the parallel graph partitioning algorithm from Teng [24], and the one used in PetFMM [8]. These works consider not only the tree structure but also the interaction edges in the FMM when making their load balancing decision. On shared memory architecture, the most representative early work is the costzone method [21, 22]. With the deployment of multicore, GPUs, and coprocessors, there have been many efforts on porting the multipole methods on these architectures [12, 30] and a new focus is on the data-driven implementation that tries to address the strong scaling challenges in traditional implementations [2, 4, 18, 32, 33].

Multipole method computations are naturally expressible as a directed acyclic graph (DAG). Effective parallelization of multipole methods then amounts to an effective parallelization of the traversal of this DAG [6, 28]. The structure of the DAG depends critically on the details of the source and target locations. This dependence requires that the DAG, and the resulting execution, be discovered at runtime. DASHMM uses an advanced runtime system to map the nodes and edges of the DAG into literal execution constructs. This allows for sophisticated control of the execution that can make better use of the computational resources leading to improved efficiency and scalability.

The runtime system used for DASHMM, HPX-5 [1], provides a rich set of tools for parallelizing applications. Along with this richness comes a certain level of complication in using the runtime system. To effectively use HPX-5, one must rethink how to perform parallel computations. If DASHMM cannot hide this complication from the user – unless they want to discover it – then it will fail to provide a general framework that is also easy to use.

These considerations lead us to the following desiderata for DASHMM:

- Ease of use: The time from selection of DASHMM to a working application should be minimized where possible. This means providing common use cases as well as hiding the details of the parallelization until the user is ready to explore the available options.

- Extensible: The library must provide a means to apply the dynamic adaptive techniques to user-specified problems. Without this ability, the ultimate utility of the library is severely restricted to only those kernels provided by the library.

- Efficient: If the computational resources are not used effectively, there will be times during which some computational elements are not being used. Many problems use less than 10% of the system's peak performance. This is a waste of both time and energy.
- Scalable: Many problems have grown so large that parallel computing is not optional, and so the library should provide benefits beyond the increase in total memory as more computational resources are used for a given problem.
- Portable: Time spent porting code to take advantage of emerging hardware is time not spent doing end-science.

All of these goals taken together speak to productivity and performance, not only of the DASHMM user, but of the computational resources employed by DASHMM.

The organization of this paper is as follows. Section 2 presents the abstractions employed by DASHMM to produce a general and extensible library. The conceptual framework for the parallel computation, and the resulting runtime system, are presented in Section 3. The installation, use of the demonstration code, and some performance results are outlined in Section 4. In Section 5 we conclude with a discussion of future extensions of DASHMM.

2 Generalizing multipole methods

This section starts with a brief review of the algorithmic structures of BH and FMM, two exemplars of the multipole method. Then, it presents the abstractions employed by DASHMM to produce a general and extensible library. Here, each *source* point s_i has strength m_i, and each *target* point t_j is a location where force field or potential needs to be computed. Depending on applications, the source ensemble $\{s_i\}$ and the target ensemble $\{t_j\}$ can be the same, partially overlapping, or completely disjoint. The computational domain is defined to be the smallest box that contains $\{s_i\}$ and $\{t_j\}$.

2.1 Barnes-Hut

The first phase of the BH method hierarchically subdivides the computational domain, creating a tree structure of the source ensemble. The root node of the source tree represents the entire computational volume, with each child node representing a fraction of the volume of its parent. The hierarchy continues until the leaves of the source tree contain fewer than a given number of sources.

Next, BH contains an upward pass of the source tree that generates the multipole expansion for each node. This uses two fundamental operations, the source-to-multipole (S2M) operator, which generates the multipole expansion for a single leaf of the tree, and the multipole-to-multipole (M2M) operator, which uses the multipole expansion of a child of a given node to compute the multipole expansion for that node itself. In this way,

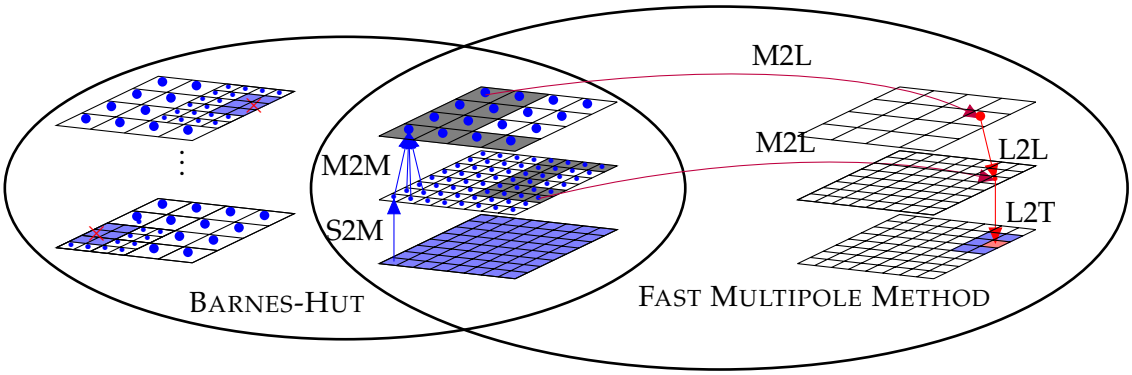

Figure 1: Illustration of the two dimensional BH and FMM algorithms using four levels of subdivisions. A blue (red) shaded box (node) represents a cluster of source (target) locations while a blue (red) dot represents the multipole (local) expansion. The red cross marked on the left represents one particular target location. The gray shaded region is the interaction list region for the FMM.

information about the particle distribution propagates upward toward the root of the source tree, with each source node having a multipole expansion for the set of particles inside its volume.

These multipole moments are used in the final phase of the BH algorithm. For each location of interest, the source tree is traversed from the root downward, and a set of nodes is considered. A source node under consideration is used if the approximation represented by the multipole moments in that node would not contribute too much error compared to the exact solution. Typically, this amounts to a decision about how far away the set of particles represented by the multipole moment is from the point of interest. If the source node can be used, its effect at the target point is computed from the approximation. If it cannot be used, the children of the source node are then considered in turn. If the source node cannot be used and it has no children, the direct interaction for each source in the leaf node is computed for the point of interest. This is demonstrated schematically in part of Fig. 1.

2.2 The fast multipole method

FMM also starts with the hierarchical subdivision of the computational domain. To accommodate source and target ensembles that are different, a separate tree is constructed for both the sources and the targets. The subdivision of each tree halts either by reaching a prescribed level of subdivision (*uniform* version) or when the points inside the node of the tree are fewer than the prescribed threshold (*adaptive* version). The right part of Fig. 1 illustrates a uniform version with four levels of subdivision. In the figure, the gray shaded region at each level of the source tree is the so-called *interaction list region* of the target node at the same subdivision level containing the target point of interest—those source nodes far away enough enough that their expansions can be used, but not so far away that their parent's expansion can be used. In the adaptive version, the sur-

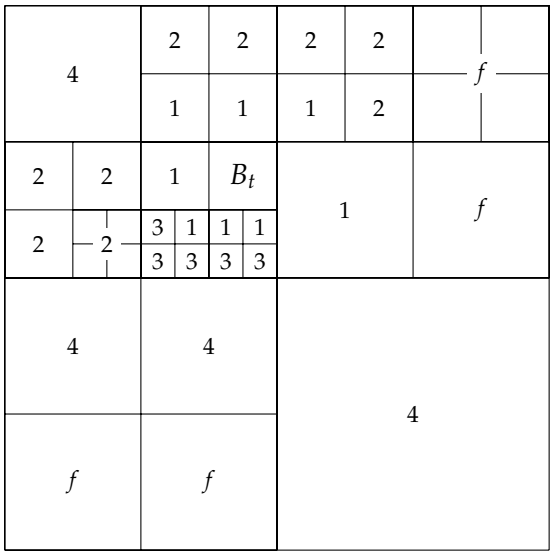

Figure 2: Illustration of the four types of lists associated with a node of the target tree B_t in 2D.

roundings of a node of the target tree is more complicated, and is organized into four categories, usually referred as *lists* (see Fig. 2 for a 2D illustration). For a target node B_t, the definitions of the lists are as follows.

1. $L_1(B_t)$, the list-1 of B_t, is empty if B_t is a non-leaf node. Otherwise, it contains all the leaf source nodes that are adjacent to B_t.

2. $L_2(B_t)$, the list-2 of B_t, is the interaction list of B_t.

3. $L_3(B_t)$, the list-3 of B_t, is empty if B_t is a non-leaf node. Otherwise, it contains all the nodes that are not adjacent to B_t, but whose parents are adjacent to B_t.

4. $L_4(B_t)$, the list-4 of B_t, consists of all the leaf source nodes that are adjacent to B_t's parent, but not to B_t itself.

FMM shares the same upward pass that generates the multipole expansions as in BH, but performs more operations on the target nodes afterwards. First, the multipole expansion of the source nodes in the interaction list region are translated into local expansions using the multipole-to-local (M2L) operator. This translation allows the effect of a distant source node to be applied only once to a given target node at the same level of refinement, rather than multiple times at the descendant leaves of the given node. Second, the target node inherits the local expansion from its parent using the local-to-local (L2L) operator. This allows the local expansion of higher level nodes to be applied to nodes at a finer level of refinement. At the finest subdivision level, the FMM resumes operation at the particle level; the local expansion of the target node containing the target point is evaluated using the local-to-target (L2T) operator and interactions with neighboring

source particles are computed via the direct pairwise interactions. In its adaptive version, FMM has two additional operators to process lists 3 and 4. Specifically, each list 3 source node's expansion is evaluated using the multipole-to-target (M2T) operator, and each list 4 source node contributes a local expansion about the target node's center using the source-to-local (S2L) operator.

2.3 DASHMM abstraction

Despite their differences, these two methods can be unified into a more general framework. The fundamental feature of these methods that allows some form of unification is that each method essentially describes a DAG. The nodes of this DAG are the expansions of the potential due to clusters of source particles, and translations of those expansions. The edges of the DAG are the transformations that produce the expansions. The difference between these multipole methods is the precise topology of the DAG.

The general framework for multipole methods provided by DASHMM comprises trees for both sources and targets, a method, and expansions. The trees are typical structures that describe the partitioning of the source and target particles. They also act as a scaffold on which the multipole method is evaluated. The precise topologies of the two trees will affect the exact details of the DAG, but conceptually, the trees and the DAG are distinct. The method is those details about how the scaffold provided by the trees is used to generate a DAG. Finally, the expansions are the nodes of the DAG and represent the bulk of the computations in any multipole method evaluation. The trees and the method being employed will dictate the exact set of expansions that are needed, and the interrelations among those expansions.

In the above descriptions of BH and FMM there were no specific details of the exact potential or interaction that is being computed. All that is needed are the general notions of computing expansions from sources, translating expansions in various ways, and applying expansions to targets. The details of these operations will depend critically on the potential under consideration, but the fact of needing these operations does not.

This leads to the first major unifying abstraction in DASHMM: the Expansion. Expansions contain the specific details that are needed to perform the fundamental operations on the multipole expansions. Expansions are implemented in DASHMM as subclasses of an abstract base class called `Expansion`. Each subclass implements a different kernel, and means of representing the multipole moments. In principle, a new kernel can be used by providing the Expansion that implements the operators needed for the method of choice. DASHMM currently provides two expansions that implement the Laplace kernel with two different series, one that works well with FMM, and one specialized to BH. However, the user is able to define and register their own expansions with DASHMM, allowing the library to apply to any other potential for which a subclass of `Expansion` can be defined.

The second major unifying abstraction in DASHMM is the Method. In the descriptions of the specific cases of BH and FMM, there was similar work that had to be per-

formed, and similar use patterns of the expansion operators. These similarities can be lifted from the specific cases of BH and FMM to produce the Method abstraction. In DASHMM, methods are implemented as an abstract base class, the subclasses of which provide the specific cases of multipole methods. Currently, DASHMM provides a BH and an FMM `Method` subclass. Like with `Expansion`, the user is able to define and register other `Method` subclasses with the library.

The abstraction employed by DASHMM for methods reduces down to four operations during the traversal of the DAG: *generate*, *aggregate*, *inherit* and *process*. By varying the work performed in these four operations, the resulting `Methods` are able to reproduce both BH and FMM with ease, while allowing for variations on both methods, as well as generalizations to other overall methods. Indeed, BH and FMM differ only in their implementations of *inherit* and *process*.

The *generate* operation is responsible for producing the initial multipole expansions from the source points at the leaves of the source tree. Then, *aggregate* will translate and combine multipole expansions in the source tree from the children of a given node of the source tree. *Inherit* translates an expansion from the parent of a given target node into its children. *Process*, given a list of source nodes to consider, will make decisions about the usability of the expansions in the given list relative to a given target node, and take action on this usability, be that by translating and combining expansions from the source tree (like the M2L operations in FMM), computing the direct interaction between a leaf of the source tree and a leaf of the target tree, or by passing a node on to the children of this target node for further processing.

By putting together the `Expansion` and `Method` abstractions with tree structures that use the `Method` to generate and connect `Expansions`, DASHMM is able to support a wide variety of multipole methods in a single framework. With this framework in place, the task of DASHMM becomes one of efficiently and scalably executing the traversal of the DAG.

3 Dynamic adaptive computation

Problem domains, such as that addressed by the DASHMM library, stress the capabilities and methods of conventional programming and computing practices, requiring alternative dynamic means to adapt to changing conditions of the applications and their underlying computer platforms. This is due to intrinsic computational properties of irregular time-varying data structures and data-dependent flow control that cannot be predicted and therefore cannot be optimized by static scheduling from user- or compiler-driven approaches. These challenges are further aggravated by increasing uncertainty due to variable latency-driven asynchrony of remote access and service requests. To address these and other emerging obstacles to next generation FMM and BH algorithms on ultra-scale computers through the end of this decade and into the next, a new strategy is required that exposes, exploits, and employs the opportunities of dynamic adaptive computation

through advanced runtime system software and the innovative execution models upon which these systems are based. Such dynamic adaptive techniques are likely to deliver substantial gains in efficiency and scalability, both of which are essential to effective exploitation of future multi-core heterogeneous Peta/Exascale architectures. For this reason, the strategy employed by DASHMM is to use a dynamic adaptive runtime system, HPX-5, which is derived from, and is an approximation of the ParalleX execution model.

3.1 ParalleX overview

ParalleX is an experimental execution model that serves as a computational paradigm to guide the design of interoperable system layers and govern their operation of resource management and task scheduling. An execution model is an overall system-wide strategy to define and coordinate parallel actions and the distributed data upon which they operate. Historically, different execution models have been created to exploit the opportunities exposed by new technologies and to address the challenges they impose. Examples of such previous generation execution models include vectors, SIMD-Array, and Communicating Sequential Processes. ParalleX is derived to minimize sources of performance degradation including starvation, latency, overhead, and waiting due to shared resource contention. Its strategy is to support *guided* computing through runtime system hardware and software in lieu of conventional *ballistic* computing that statically determines scheduling. The anticipated achievement through this innovation in adaptive control is significant improvement in efficiency and scalability of future systems and the applications they support. ParalleX addresses starvation by exposing and exploiting a broad range of form, granularity, synchronization, and scheduling of parallelism. It addresses the challenge of latency of data movement by moving work to the data through its *parcel* active messages and by hiding latency through automatic overlap of computation and communication by means of rapid task context switching. Overheads are eliminated or substantially reduced by optimized and specialized mechanisms (potentially supported in hardware) for task context switching and creation as well as synchronization and parcel to task instantiation. Shrinking of overheads also reduces starvation by enabling finer granularity of parallel tasks and thereby exposing greater concurrency for fixed size workloads. Contention is precluded in many cases through introspection and adaptive resource management, task scheduling, load balancing and work stealing. Overheads are further reduced through an intrinsic global address space, allowing computing to perform remote accesses directly. Global memory usage is assisted by means of advanced local control objects such as dataflow and futures based synchronization. Each of these semantic constructs and their implied implementations are considered in greater detail in the following subsections.

3.1.1 Active Global Address Space (AGAS)

A system-wide name space is enabled by means of a dynamic hierarchy of ParalleX processes, each a first-class object, that serves as a context for some subset of the entire com-

putation while still supporting access to global shared data. This is realized through an active global address space that permits direct efficient memory usage across an entire application data set, minimizing copies or user software data movement management. It further enables load balancing of changing data set sizes and shapes that evolve throughout the computation. The global address space is active in the sense that a virtual object may be moved across a physical computer system (e.g., between processing nodes) without changing its virtual address differentiating it from more common PGAS (i.e., partitioned global address space) methods. AGAS also provides a naming convention that supports performance portability across varying types, scales, and generations of machines while facilitating user productivity, especially for those problems with changing data structures and workloads.

3.1.2 Compute complexes

Almost all computation is directed and performed by a localized construct referred to as a *compute complex*. A compute complex is local in the sense that all of its operations are performed within the resident *locality*, the physical compute element (sometimes referred to as a "node") that guarantees bounded response time and integrity of atomic sequences of operations. A simple example of a compute complex is a thread or task as used by more conventional models. However, a compute complex can differ in a number of ways that may prove important in the management and execution of dynamic adaptive applications like DASHMM. Compute complexes provide a wide range of parallelism granularity from near fine grain to heavy weight. Unlike some alternatives (e.g., codelets), a complex is preemptive such that it can be started and stopped at will when necessary either due to computational restrictions (waiting for an intermediate result) or to optimize ordering of execution to best adjust to prevailing system hardware conditions. Perhaps of greater importance is that a complex can access global data and perform global mutable side-effects. Some other task-oriented computation is purely functional with value exchanges limited to input argument and output results. Finally, an essential feature that distinguishes a compute complex from more conventional models is that it too is a first-class object. This means that a computation can refer to and manipulate itself; this is a very elegant and powerful capability critical to efficiency of dynamic applications.

Within a compute complex is fine-grained parallelism most closely associated with the venerable static data flow model, replacing the diversity of alternative conventional fine-grain operations like speculative execution, branch prediction, prefetching, instruction level parallelism, among others. However, for conventional processor cores, this generalized dataflow technique can be transformed into these more usual forms through compilation techniques and therefore prove effective in the short term within the limits of the processor. When instantiated, a compute complex does not require strict initialization where all the operands need to be available in order to start. Rather, a critical minimum set of arguments is sufficient with internal synchronization managing late binding. This avoids the inadvertent consequences of an implicit barrier, permits greater overlapping of operations, and supports eager evaluation.

A runtime system consumes precious resources and, as suspended complexes accrue, can be starved of sufficient capacity to continue. To ensure scalability ParalleX supports a unique modality for suspended complexes when appropriate: as *depleted threads*. This is an important unifying principle that treats complexes waiting for progress of computation as a not-so-simple form of synchronization object referred to as a local control object (see below). For the purposes of this discussion the important aspect of depleted threads is that they exploit their status as first class objects and do not require any runtime system resources in this state. Also, compute complexes can be moved from one locality to another in the migration state. Although not anticipated to be a frequently used mechanism, migration permits load balancing and supports resilience as well as dynamic resource usage (for example when the operating system can provide more resources than originally available or requires some resources back for other purposes).

3.1.3 Parcels

ParalleX parcels support event-driven computation and are transaction-oriented means of invoking actions at the physical site of data elements upon which to be operated. A parcel is a specification of an action, an object upon which the action is performed, some data or references to be employed by the operation, and an indication of what is to happen upon completion of the specified action. This serves as a general template for describing any generalized action anywhere in the system.

Parcels are critical for a number of reasons. From an abstract point of view of the computation, they guarantee symmetry of semantics independent of whether the spawned task is relatively local to the calling complex (on the same locality) or remote (on a separate locality). The semantics are the same and the distributed positioning of data and actions can change between execution instances without changes to the work characterization (i.e., program). In order to facilitate this, a parcel is not a first-class object, thereby keeping the flexibility needed for different operational modalities. Parcels also address the challenge of operational latency. A parcel reduces latency by moving work to the data in a physically distributed structure assuming that most of the data to be processed is separated from the operational requirement. It can also hide latencies when combined with pre-emptible compute complexes by overlapping computation with communication. A special case of the parcel is "percolation" which supports heterogeneous computing as in the instance of GPUs integrated within a distributed system. Here the destination is not an abstract name/address of an argument object but a physical function system component. Values and task specifications are passed to the function unit while it is performing a different task and possibly returning computed results from a third task previously performed. The data movement is overlapped with the work being performed thus hiding the latencies and overhead of the relatively precious resource.

3.1.4 Local Control Objects

In addition to hardware program counters associated with each processor core, ParalleX supports a second layer of control state: *local control objects* or LCOs. Like ParalleX pro-

cesses and compute complexes, LCOs are first class objects. LCOs manage and exploit asynchrony of operation and communication, and in many ways they are similar to conventional synchronization objects such as barriers, semaphores, and mutexes. In most cases they are variants of higher order constructs from prior art including the two most important of these, *dataflow* and *futures*. Dataflow LCOs launch a new complex instantiation when all or a necessary subset of operand values have been computed and conveyed. Future LCOs allow metadata operations to be performed on a variable even while the actual value is being computed. In this case the computation may be either eager or lazy. ParalleX supports global graphs of LCOs to be formed, distributed, and altered as the computation evolves. For instance, a typical continuation response to a parcel invocation is the creation or accessing of an LCO. A form of LCO is also employed within compute complexes serving as single-assignment registers (write-once, read-many) incorporating control state indicating arrival of values.

3.2 HPX-5

As a proof-of-concept and one of the early reductions to practice, HPX-5 is a runtime software system that approximates the parallel semantics of the ParalleX execution model. It is used both for experimentation and validation of the model, and for user applications such as DASHMM. The work presented here uses version 2.1 of HPX-5, though any subsequent version of HPX-5 will work equally well. An added benefit of building DASHMM using HPX-5 and standard-conforming C++11 is that DASHMM automatically can be run on any system on which HPX-5 can run. In this section, we shall cover those features of HPX-5 that are crucial to the operation of DASHMM, and how those features are used by DASHMM to traverse the DAG.

3.2.1 Lightweight threads

HPX-5 is designed around lightweight multi-threading. The work performed by any HPX-5 computation is handled by an HPX-5 thread. These threads are multiplexed onto a set of heavyweight operating system-level threads (called scheduler threads). Lightweight threads can easily be swapped into and out of an executing state, allowing the HPX-5 scheduler to rapidly swap in work that can proceed when one thread of execution becomes blocked waiting for some data or some event.

In DASHMM, all execution is performed by a lightweight HPX-5 thread. The primary task of any HPX-5 program is to describe which threads are to be instantiated, and what is the action of those threads.

3.2.2 Parcels

Parcels, a form of active message, are the primary means by which lightweight threads are instantiated in HPX-5. A parcel contains a description of the action to be performed as well as the address of the data on which the action is to be performed: a parcel is execution transformed into data. Parcels allow the system to not only send data to work,

as is more traditional, but also allows the system to send work to data. In this way, the arrival of a parcel, through the arrival of a message, causes the execution to occur; HPX-5 provides *message-driven* computation.

For each edge of the DAG in DASHMM, a parcel is prepared that describes the operation to apply (M2L, M2M, and so on) and the data on which to operate (the expansion that is being transformed). The result of the transformation is then accumulated into another expansion. For example, an edge connecting two expansions with an M2M operation will be mapped into a parcel that will read the expansion at the source of the edge, will perform the translation, and then will send the results to the expansion at the destination of the edge.

The large number of edges in the DAG for a typical evaluation becomes a large number of tasks to be performed in the parallel execution. However, there are some dependencies on how those tasks are ordered. All input edges to a node in the DAG must be executed before any output edges may be performed. Also, the destination expansion must exist before the transformation can begin. In DASHMM, the creation time of the expansions and the execution time of the transformations are not known, and so there is a need for synchronization among the lightweight tasks.

3.2.3 Local Control Objects

Local Control Objects (LCOs) are event-driven, lightweight, globally-allocated synchronization objects that encode data and control flow. Each LCO has a set of events that must occur before the LCO is triggered, allowing threads waiting on the LCO to proceed, or enabling new threads to be instantiated. Threads may wait on an LCO or attempt to get the values stored in an LCO, thereby suspending the thread's execution if the LCO is not yet triggered. HPX-5 offers a rich set of LCOs that allows for fine control over the execution.

One example is an *and* LCO. This LCO behaves like an and gate, only triggering once a certain number of set operations have been performed. This allows for situations where a set of events must occur before a computation can proceed. For example, an and LCO is ideal if a computation for a parent node cannot be performed until the children of that node have finished their computation.

The primary data in a DASHMM evaluation are the expansions, be they generated at the source points, or by translation from other expansions; the expansions are the nodes of the DAG. Given that the nodes also encode the dependencies of the computation, it is natural to promote the expansions to being LCOs. The expansion LCO acts in some ways like an and LCO, requiring a certain number of inputs before triggering. However, unlike the and LCO, expansion LCOs take input data: namely, an expansion. The multiple input expansions are combined into one expansion by the expansion LCO. A further complication is that when some expansion LCOs are created the number of inputs required is not available until the DAG is fully discovered. However, the user-defined LCO provided by HPX-5 is flexible enough to handle this use-case. The expansion LCO is an example of the more general concept of the dataflow LCO.

After all the input dependencies are met, the expansion LCO triggers, and the output edges can then be processed. All that is required to meet this synchronization criterion is for the action taken by the lightweight thread implementing the edges of the DAG to wait on the expansion LCO at the source of the edge. However, this simple approach can be improved by using another feature of HPX-5.

3.2.4 `call_when`

It is frequently the case that some actions require input data before they can be executed. One way to use HPX-5 in this case is to send a parcel that takes the first step of waiting on an LCO representing the data needed. However, that will cause a lightweight thread to be instantiated only to be immediately suspended, consuming some physical resources. It would be better to instead only send the parcel once the LCO has been triggered.

HPX-5 provides a mechanism for precisely this sort of dependent parcel. Any LCO can have parcels attached to it that are only sent once the LCO has been triggered; this is the 'when' in `call_when`. This takes the notion of an LCO as a control structure one step further by giving them the ability to explicitly cause execution to occur, and not only implicitly allow execution to occur.

Indeed, the edges of the DAG in DASHMM are exactly this sort of action. The input expansion data must be ready before the translation can occur, and so any action that is scheduled would immediately block on the input LCO. Instead, when DASHMM schedules work, it does so in the dependent fashion provided by `call_when`. Only once the input expansion LCO is triggered are the parcels implementing the outgoing edges from the node of the DAG sent. This is demonstrated schematically in Fig. 3.

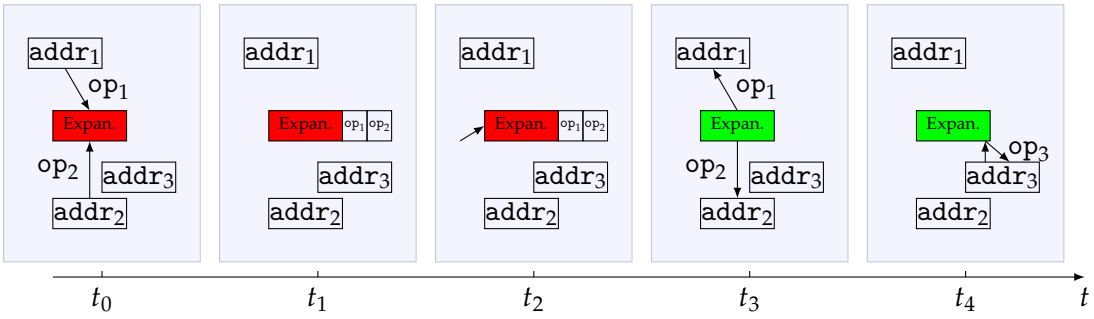

Figure 3: A schematic of the function of the Expansion LCO in DASHMM. Parcels arrive from various objects (e.g., $\texttt{addr}_1$) which will perform some operation ($\texttt{op}_1$) on the expansion data stored in the LCO (t_0). The Expansion LCO is initially untriggered, and so the operations are scheduled, but are not executed (t_1). Eventually, the Expansion LCO is triggered by the arrival of its final input (t_2). At this point, the scheduled operations can be executed (t_3), typically by sending a parcel to the requester with the data stored in the expansion. Requests that arrive after the LCO has been triggered can be executed immediately (t_4).

4 Software installation and numerical examples

4.1 Installation

DASHMM depends on one external library: HPX-5. The current version of DASHMM (v. 0.5) depends on version 2.1.0 of HPX-5, which can be found at `https://hpx.crest.iu.edu/download`

The current version of DASHMM is specialized to shared memory operation, so there is little to be specified in the configuration of HPX-5 for use with this version of DASHMM. Assuming that you have unpacked the HPX-5 source into the folder `/path/to/hpx` and wish to install the library into `/path/to/install` the following steps should build HPX-5 and install it.

1. `cd /path/to/hpx`
2. `./configure --prefix=/path/to/install`
3. `make`
4. `make install`

The previous commands will build HPX-5 and install it into the given location. This configuration uses many of the default HPX-5 options, and may not be ideal for specific systems. Please see the official HPX-5 documentation for more details.

The DASHMM build system relies on the `pkg-config` (version 0.26 or newer) 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`. While one is at it, it is useful to modify the following environment variables to point to the newly installed HPX-5. For example, with `bash`:

```
export PATH=/path/to/install/bin:$PATH
export LD_LIBRARY_PATH=/path/to/install/lib:$LD_LIBRARY_PATH
export PKG_CONFIG_PATH=/path/to/install/lib/pkgconfig:$PKG_CONFIG_PATH
```

Once the prerequisites are met, one needs to perform the following steps to build the DASHMM library.

1. Unpack the source code into some convenient directory. For the sake of discussion, we assume that the code has been unpacked in `/path/to/dashmm`.
   Change to the DASHMM directory into which you have unpacked the code. There are a number of subfolders. Initially, the most relevant of these are `doc/` and `demo/`.
2. In `/path/to/dashmm` can be found `Makefile`. There are few (if any) changes that need to be made to this file for successful compilation. The most likely change to make would be to modify the compiler used. Any compiler supported by HPX-5 will be able to compile DASHMM provided it also supports the C++11 standard.

For example, to change the compiler from the default (g++) to the Intel C++ compiler, one needs to replace `CXX = g++` with `CXX = icpc`.

3. Run `make` from `/path/to/dashmm`. This should build the library statically, and it will be ready to link with your code.

To build a program using the DASHMM library, only a few things need to be done. DASHMM builds in place, so when compiling code that uses the library, 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 easiest with the `pkg-config` utility.

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

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

Similarly, one must specify the following arguments for linking:

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

Examples of this can be found in the demo programs included with DASHMM.

4.2 Built-in methods and expansions

Version 0.5 of DASHMM includes two built-in methods: FMM and BH, which are outlined briefly above. The included FMM does not use a tunable multipole acceptance criterion (MAC). Instead, it uses the geometric cut that the source node be well-separated from the target node. Thus, the built-in FMM method does not require any parameters to use. The BH method uses the classic MAC specified by a critical angle, θ_c. This critical angle is a free parameter of the method specifiable by the user.

DASHMM includes two build-in expansions, both representing the Laplace potential. The built-in expansion that is designed to be used with FMM expands the potential using spherical harmonics, and can be used for a specifiable number of digits of accuracy. This expansion is performed around the center of the node containing the particle. The expansion that is intended to be used with the BH method instead performs a Taylor expansion of the potential and keeps only up to the quadrupole term. The expansion is performed around the center of charge of the represented particles. Further, it is only intended for use with charges of the same sign. The particular choice of expansion for BH is selected to match what is commonly done in astrophysical simulations, a field in which BH has been extensively applied.

4.3 Basic usage

Included with DASHMM is a test code that demonstrates a simple use of the library. This code is given in `/path/to/dashmm/demo/basic/`. It can be built by running `make` in the

Table 1: DASHMM speedup. Almost linear strong (weak) scalability result is shown in each row (column) of the matrix.

d	N	BH		FMM	
		$p=6$	$p=12$	$p=6$	$p=12$
cube	1	5.92	11.57	5.85	11.37
	2	5.90	11.45	5.88	11.54
	4	5.89	11.65	5.91	11.69
Sphere	1	5.91	11.55	5.85	11.21
	2	5.86	11.39	5.77	11.23
	4	5.86	11.57	5.80	11.34
Plummer	1	5.93	11.46	5.85	11.29
	2	5.88	11.48	5.83	11.36
	4	5.88	11.65	5.82	11.33

`demo/basic/` folder once the library has been built. The simple `Makefile` in `demo/basic/` is an example of how to link your code with the DASHMM library.

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

The performance of DASHMM is demonstrated here using this test code, particularly focusing on the resulting scalability. The tests were performed on a workstation with dual Xeon E5-2609 v3 processors, running at 1.9 GHz clock rate, and 64 GB of memory. The operating system is Ubuntu 14.04 with the 3.13.0-45-generic Linux kernel. The code was compiled using the GNU compiler. Results of the performance tests are shown in Fig. 4 and Table 1. The configurations of the tests can be summarized as follows:

- Source and target ensemble are different, but have the same size and distribution. The problem sizes used are 1, 2, and 4 million.
- Three data distributions are tested: (a) Uniform distribution inside a cube; (b) Uniform distribution on a spherical surface; (c) Plummer distribution.
- Masses carried by the source points have a magnitude uniformly distributed on the interval [1,2]. For the BH method the masses are all positive, and for FMM the masses have a random sign.
- Three digits of accuracy were required of the FMM cases, which required 55 terms in the expansion.
- The critical angle used in the multipole acceptance criterion for the BH cases was 0.6, and the expansion of the potential included up to the quadrupole moment.

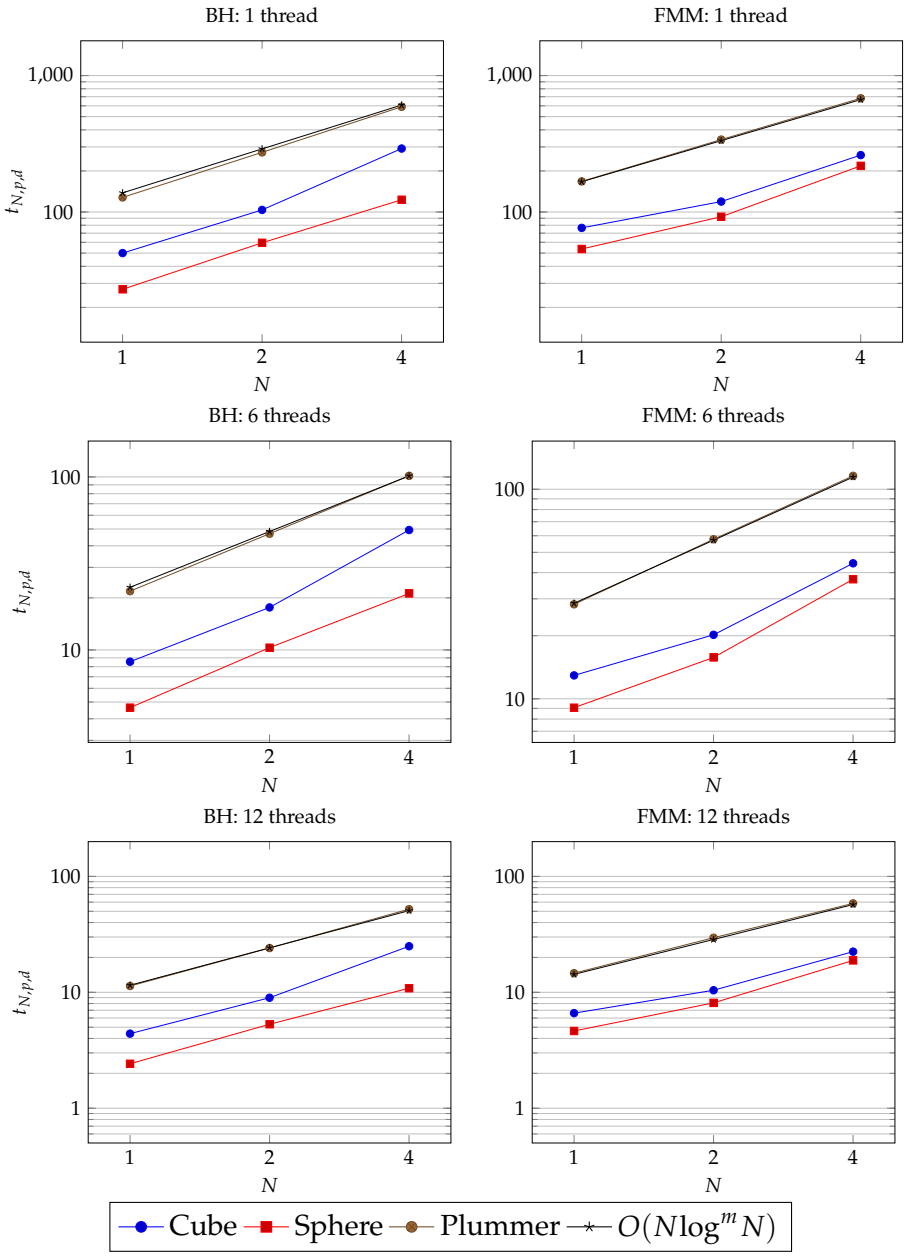

Figure 4: DASHMM performance. $t_{N,p,d}$ is measured in seconds and N is given in millions. Each panel shows the execution time for the cube, sphere, and plummer distribution at thread count p, and a curve depicting the arithmetic complexity of the underlying method, normalized to the results of the Plummer distribution. Parameter m in the legend entry $\mathcal{O}(N\log^m N)$ is 1 for the BH method on the left column and 0 for the FMM method on the right column.

- For each problem size N and thread count p, we define the execution time for a particular data distribution d as

$$t_{N,p,d} = \min_s \left\{ \sum_{i=1}^{10} t^i_{N,p,d,s} / 10 \right\},$$

where $\{s\}$ is the set of thresholds examined. Speedup $S_{N,p,d}$ is defined as

$$S_{N,p,d} = t_{N,1,d} / t_{N,p,d}.$$

4.4 User-defined expansion

One key feature of DASHMM is that it is extensible: Users can define their own `Expansions` and `Methods` to treat their problem while still achieving the efficiency and scalability of dynamic adaptive methods. A second demonstration program released with DASHMM (`demo/user_expansion/`) provides a skeleton code outlining the means by which a user might define their own `Expansion` subclass, and use it with DASHMM. The needed documentation appears as comments in the source code.

5 Conclusion

DASHMM is a library that implements a general framework for parallel multipole method computations. The library is built on a dynamic adaptive runtime system, HPX-5, and leverages the flexibility of that system to improve the efficiency and scalability of the resulting multipole method computations. The library provides built-in methods (FMM, BH) and expansions (Laplace kernel), as well as the ability to extend the library with user-defined expansions and methods. The resulting framework is easy-to-use and portable.

Future versions of DASHMM will continue to extend the functionality of the system as well as the usability in applications. Foremost on the list of future features is a fully distributed implementation that has some capability to automatically balance the work across the available resources. This will rely in some measure on the global address space provided by HPX-5 to allow each object in the system to have a single virtual address that can serve both as its identifier as well as a target for parcels.

In terms of usability, DASHMM will be extended with more built-in kernels and methods. Further, the output of the method will be generalized to allow for the acceleration to be returned from an Expansion as well as the potential. This will make time-stepping codes easier to construct using DASHMM. Further, the interface to DASHMM will be expanded to allow for more advanced control of the underlying computation, allowing for repeat uses of the same DAG, or to allow for multiple kernels for the same sources, or even some direct control of the underlying runtime system.

Acknowledgments

This work was supported by National Science Foundation grant number ACI-1440396. Authors AT and VT-S gratefully acknowledge the support of the National Science Foundation's REU program.

References

[1] HPX-5. https://hpx.crest.iu.edu/about.
[2] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi. Task-based FMM for multicore architectures. *SIAM J. Sci. Comput.*, 36:C66–C93, 2014.
[3] S. Aluru, J. Gustafson, G. M. Prabhu, and F. E. Sevilgen. Distribution-independent hierarchical algorithms for the N-body problem. *J. SuperComput.*, 12:303–323, 1998.
[4] A. Amer, N. Maruyama, M. Pericás, K. Taura, R. Yokota, and S. Matsuoka. Fork-join and data-driven exeuction models on multi-core architectures: Case study of the FMM. *Lect. Notes. Comput. Sc.*, 7905:255–266, 2013.
[5] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, December 1986.
[6] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Comput.*, 38:37–51, 2012.
[7] W. C. Chew, J. M. Jin, E. Michielssen, and J. M. Song. *Fast and Efficient Algorithm in Computational Electromagnetics*. Artech House, 2001.
[8] F. A. Cruz, M. G. Knepley, and L. A. Barba. PetFMM–A dynamically load-balancing parallel fast multipole library. *Int. J. Numer. Meth. Eng.*, 85:403–428, 2011.
[9] L. Greengard and W. Gropp. A parallel version of the fast multipole method. *Comput. Math. Appl.*, 20:63–71, 1990.
[10] L. Greengard, MC Kropinski, and A. Mayo. Integral Equation Methods for Stokes Flow and Isotropic Elasticity in the Plane. *J. Comput. Phys.*, 125:403–414, 1996.
[11] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73:325–348, 1987.
[12] N. A. Gumerov and R. Duraiswami. Fast multipole methods on graphics processors. *J. Comput. Phys.*, 227:8290–8313, 2008.
[13] J. A. Board Jr., J. W. Causey, and J. F. Leathrum Jr. Accelerated molecular dynamics simulation with the parallel fast multipole algorithm. *Chem. Phys. Lett.*, 198:89–94, 1992.
[14] J. F. Leathrum Jr. and J. A. Board Jr. Mapping the adaptive fast multipole algorithm onto MIMD systems. In *Unstructured Scientific Computation on Scalable Multiprocessors*, pages 161–177, Nags Head, NC, USA, 1992.
[15] J. Kurzak and B. M. Pettitt. Communications overlapping in fast multipole particle dynamics methods. *J. Comput. Phys.*, 203:731–743, 2005.
[16] J. Kurzak and B. M. Pettitt. Massively parallel implementation of a fast multipole method for distributed memory machines. *J. Parallel Distr. Com.*, 65:870–881, 2005.
[17] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A massively parallel adaptive fast-multipole method on heterogeneous architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[18] H. Ltaief and R. Yokota. Data-driven execution of fast multipole methods. *CoRR*, abs/1203.0889, 2012.
[19] B. Lu, X. Cheng, J. Huang, and J. McCammon. Order N algorithm for computation of electrostatic Interactions in biomolecular systems. *Proceedings of the National Academy of Sciences*, 103:19314–19319, 2006.
[20] A. Rahimian, I. Lashuk, S. K. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200K cores and heterogeneous architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.
[21] J. Singh, C. Holt, J. Hennessy, and A. Gupta. A parallel adaptive fast multipole method. In *SC 93': Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, 1993.
[22] J. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-Hut, fast multipole, and radiosity. *J. Parallel Distr. Com.*, 27:118–141, 1995.
[23] V. Springel, J. Wang, M. Vogelsberger, A. Ludlow, A. Jenkins, A. Helmi, J. F. Navarro, C. S. Frenk, and S. D. M. White. The Aquarius Project: the subhaloes of galactic haloes. *MNRAS*, 391:1685–1711, December 2008.
[24] S. Teng. Provably good partitioning and load balancing algorithms for parallel adaptive n-body simulation. *SIAM J. Sci. Comput.*, 19:635–656, 1998.
[25] H. Wang, T. Lei, J. Li, J. Huang, and Z. Yao. A parallel fast multipole accelerated integral equation scheme for 3D Stokes equations. *Int. J. Numer. Meth. Eng.*, 70:812–839, 2007.
[26] M. Warren and J. Salmon. Astrophysical n-body simulation using hierarchical tree data structures. In *SC 92': Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, 1992.
[27] M. Warren and J. Salmon. A parallel hashed oct-tree n-body algorithm. In *SC 93': Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, 1993.
[28] W. Wu, G. Bosilca, A. Bouteiller, M. Faverge, and J. Dongarra. Hierarchical DAG scheduling for hybrid distributed systems. In *IPDPS*, Hyderabad, India, 2015.
[29] L. Ying, G. Biros, D. Zorin, and H. Langston. A new parallel kernel-independent fast multipole method. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, 2003.
[30] R. Yokota, J. P. Bardhan, M. G. Knepley, L. A. Barba, and T. Hamada. Biomolecular electrostatics using a fast multipole BEM on up to 512 GPUs and a billion unknowns. *Comput. Phys. Commun.*, 182:1272–1283, 2011.
[31] Y. Yuan and P. Banerjee. A parallel implementation of a fast multipole based 3D capacitance extraction program on distributed memory multicomputers. *J. Parallel Distr. Com.*, 61:1751–1774, 2001.
[32] B. Zhang. Asynchronous task scheduling of the fast multipole method using various runtime systems. In *Proceedings of the Forth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, Edmonton, Canada, 2014.
[33] B. Zhang, J. Huang, N. P. Pitsianis, and X. Sun. Dynamic prioritization for parallel traversal of irregularly structured spatio-temporal graphs. In *Proceedings of 3rd USENIX Workshop on Hot Topics in Parallelism*, 2011.
[34] F. Zhao and S. L. Johnsson. The parallel multipole method on the connection machine. *SIAM J. Sci. Stat. Comp.*, 12:1420–1437, 1991.