2017 IEEE International Parallel and Distributed Processing Symposium Workshops

Scalable Hierarchical Multipole Methods using an Asynchronous Many-Tasking Runtime System

Jackson DeBuhr, Bo Zhang, Luke D'Alessandro
Center for Research in Extreme Scale Technologies, School of Informatics and Computing
Indiana University, Bloomington, IN 47404
Email: {jdebuhr, zhang416, ldalessa}@indiana.edu

***Abstract*—Hierarchical Multipole Methods (HMMs) are an important class of methods in scientific and engineering applications. They are challenging to parallelize for contemporary and emerging platforms using existing programming models. Asynchronous many-tasking (AMT) execution models provide abstractions suitable for HMMs and promise scalability in the context of future exascale systems. In our work we (1) present a novel AMT-based HMM framework on HPX-5—an experimental exascale interface and runtime system, (2) evaluate its performance and scalability for a range of interaction types and data sets, and (3) reflect on the costs and benefits of using HPX-5 for this class of applications.**

***Index Terms*—Hierarchical Multipole Methods, Asynchronous Many-tasking, HPX**

I. Introduction

Hierarchical Multipole Methods (HMMs), such as Barnes-Hut (BH) and the Fast Multipole Method (FMM), compute approximate solutions to N-body or N-body like problems using $O(N \log N)$ or $O(N)$ arithmetic operations. They have been applied to huge range of applications [1]–[3]. These methods can all ultimately be viewed as building and executing a directed acyclic graph (DAG) that represents the flow of information from the sources of the interaction, through a series of approximations, to the target locations of interest. Seen in this form, HMMs offer a huge amount of inherent parallelism.

Parallel distributed memory HMM implementations often target single program, multiple data (SPMD) execution models and use MPI directly for communication. The DAG is partitioned and distributed across a cluster of compute nodes where each node maintains its own vertices and a replicated shadow region [4]–[6]. Within each compute node, the DAG is executed in a strict levelwise fashion, hewing closely to the description in the original paper [7]. Special attention is taken during execution to overlap communication within computation. With hardware advances in multicore processors and many-core accelerators, work within each level is processed in parallel. However, inputs to each vertex in the DAG come from multiple levels and some inputs can be processed earlier than in a levelwise schedule. Thus strict levelwise implementations cannot exploit all of the available parallelism, limiting their strong scaling behavior.

An asynchronous many-tasking (AMT) runtime system that provides lightweight, dependence-aware tasks offers a natural fit for the parallelism inherent in HMMs, and has the potential to address some of the scaling challenges facing HMMs. Instead of explicitly managing and scheduling the interdependent DAG operations using SPMD threads or processes, users may simply instantiate them as tasks managed by the runtime system. Such an approach has been explored in many recent studies with promising results [8]–[13]. However, this work has been limited to *shared* memory architectures.

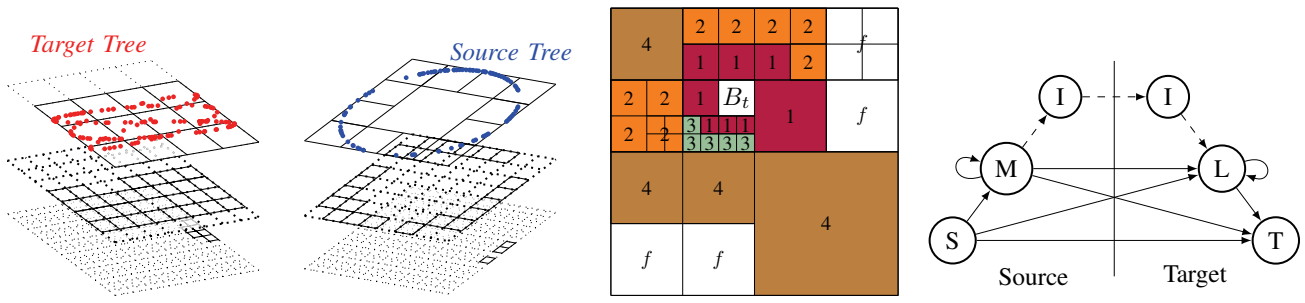
The Dynamic Adaptive System for Hierarchical Multipole Methods (DASHMM) is a scientific framework providing generic HMM implementations using the HPX-5 exascale runtime system [14], [15]. The initial implementation [16], though limited to shared memory platforms, accomplished two design objectives. First, DASHMM is generic in that the exact method and interaction used are parameters, and the parallelization employed by DASHMM is agnostic to many of these specific details. Second, the DASHMM programmer interface is independent of HPX-5 and no knowledge of the runtime system is required from its end-users. Recently, the operation of DASHMM was seamlessly extended to distributed memory platforms.

From the runtime developer's perspective, DASHMM offers a comprehensive test of many aspects of the runtime system. Varying the concrete method varies the DAG topology and available parallelism. Altering the distribution of the data alters the length of the critical path, the number of tasks on the critical path, and the number of network messages that must be sent. Adjusting the required accuracy adjusts the grain size (FLOPS and bytes transferred per task) of the computation. Switching the interaction type can create non-uniform message sizes.

This paper presents a parallel implementation of HMM evaluation using the experimental exascale runtime system HPX-5, and measures how well the runtime system handles the parallelism that unfolds dynamically during the execution, how well the runtime can prioritize tasks along the critical path, and how well any communication is hidden behind computation.

The rest of this paper is organized as follows. Section II reviews the relevant mathematical features of FMM. Section III discusses HPX-5 and in section IV the manner in which it is used by DASHMM. Section V presents DASHMM's strong scaling performance and quantitatively measures the resources utilization. Finally, section VI presents our conclusions from this work.

978-0-7695-6149-3/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPSW.2017.88



(a) Dual Tree (b) Lists (c) FMM operators

Fig. 1. (a) The FMM hierarchically partitions the source and target ensembles into two trees of nested boxes. (b) Each box on the target tree is connected with up to four types of boxes of the source tree based on the concept of well-separatedness. (c) Computation starts from the leaf source boxes, move upwards along the source tree, migrate towards the target tree, and move downwards to the leaf target boxes. The basic FMM involves two types of expansions, multipole (M) and local (L) expansions, and eight operators (shown in solid lines) that connect them to the sources (S) and targets (T). The advanced version introduces a third expansion, the intermediate expansion (I), and three additional operators(shown in dashed lines). In particular, the M→L operation is decomposed into a chain of M→I, I→I, and I→L operations in the advanced FMM.

II. Fast Multipole Method

N-body and N-body like problems are characterized by the concrete function being evaluated, an ensemble of source points that contribute to the function, and an ensemble of target points at which the user would like to evaluate the function. In traditional N-body problems the source and target ensembles are identical but in other contexts they are often only partially overlapping or disjoint. Naive N-body solutions are composed of loops over each source for each target and run in $O(N^2)$ time.

HMMs leverage the fact that the interaction between targets and sources in "far-away" regions can be approximated and performed in aggregate. HMMs adaptively partition the source and target ensembles into trees and organize computations in terms of a set of expansions, computed for nodes in these trees. These expansions are series approximations to the exact interaction providing a user-defined accuracy. The various expansions form the nodes of a DAG, and they are connected with edges representing certain transformations of the series representations. In this way, the DAG represents influence from the source ensemble, moving through a set of approximations, on the target ensemble. The specific method employed determines what sort of edges connect which nodes in the DAG. Of the two HMMs already built into DASHMM, the FMM involves more operation diversity and DAG complexity. Below we review the FMM, which is the method we evaluate in Section V.

As in any HMM, the FMM first hierarchically partitions the computational domain—the smallest cubic box that contains the source and target ensembles—into a tree of nested boxes. Child boxes are created by dividing the parent box equally along each dimension, taking care to prune empty children. The partition is *adaptive*: one stops partitioning a box when it contains fewer than a prescribed number of points, called the *threshold*. Boxes without children are called *leaf* boxes. This process creates two trees, one for each ensemble. As noted above the source and target ensembles can be identical, partially overlapping, or far apart, thus the dual trees can be identical, partially overlapping, or completely disjoint. Figure 1a demonstrates an adaptive partition for a partially overlapping source and target ensemble.

After finding the dual tree, FMM then proceeds with two *basic* types of expansion: each source box has a *multipole* expansion, and each target box has a *local* expansion. Central to the FMM is the concept of well-separatedness which gives the conditions under which these two types of expansion are valid. A box A is said to be *well-separated* from a box B if the distance between the centers of A and B is larger than a β-dilation of A's radius, where β depends on the dimension. Multipole expansions represent the enclosed sources' influence in well-separated regions; local expansions represent the influence of well-separated sources.

Each box B_t in the target tree is connected with up to four sets of boxes in the source tree, called *lists*, $L_i(B_t)$, $i = 1, 2, 3, 4$ (see Figure 1b). $L_1(B_t)$ is nonempty if B_t is a leaf and contains leaf source boxes that are not well-separated from B_t. $L_2(B_t)$ is the set of source boxes that are well-separated from B_t but whose parents are not well-separated from B_t's parent. $L_3(B_t)$ exists if B_t is leaf. For each box $B_s \in L_3(B_t)$, B_t is well-separated from B_s, but not from its parent. $L_4(B_t)$ includes leaf source boxes that are well-separated from B_t but not from B_t's parent. Note that when the source and target ensembles are not identical, it is possible that a non-leaf target box is well-separated from boxes in the source tree. In this situation, the sub-tree below the target box can be pruned to reduce arithmetic complexity [11].

In its basic version, the FMM uses eight operators (see solid lines in Figure 1c). The computation starts with computing the multipole expansions for source boxes in a bottom-up approach. This is done by using the source-to-multipole (S→M) operator at a leaf source box and multipole-to-multipole (M→M) operator at a non-leaf source box. Next, the multipole expansions are passed to the target boxes along lists 2 and 3. Multipole expansions received along list 3 are

evaluated at the target position using the multipole-to-target (M→T) operator. Multipole expansions received along list 2 are converted into a local expansion using the multipole-to-local (M→L) operator. Each target box also processes its list 4 using the source-to-local (S→L) operator. Finally, local expansions are passed downwards from parent to child target boxes using the local-to-local (L→L) operator. At a leaf target box, the local expansion is evaluated using the local-to-target (L→T) operator, and list 1 boxes are processed using the source-to-target (S→T) operator. Note that most of the edges in the FMM DAG are list 2 as each target box can have up to 189 boxes in this list. A typical FMM execution is dominated by time performing the M→L translations.

Advanced FMMs [2], [17] further employ the so-called *merge-and-shift* technique to improve performance. The idea is to explore the overlap of list 2 edges among boxes of the same parent. If there are M_1 boxes sharing M_2 boxes in their list 2, instead of doing $M_1 \times M_2$ translations, one can first merge the M_2 expansions into one and then shift it to the M_1 expansions using $M_1 + M_2$ translations. The mathematical foundation for these operations is a third type of expansion, called *intermediate* expansions. This expansion type introduces three new operations that move information from the source tree to the target tree: multipole-to-intermediate (M→I), intermediate-to-intermediate (I→I), and intermediate-to-local (I→L) (see dashed line in Figure 1c). With this technique, the average number of M→L translations per target box can be reduced from 189 to 40. The FMM evaluated in this paper implements this technique. Unlike the basic FMM, where there is an one-to-one mapping from boxes to expansions, each source (target) box in the advanced version is associated with both a multipole (local) and an intermediate expansion.

III. HPX-5

HPX-5 (High Performance ParalleX) is an experimental AMT programming model and runtime developed at Indiana University [14], [15]. Its design is governed by the ParalleX exascale execution model [18] and it aims to enable programs to run unmodified on systems from a single SMP to large clusters and supercomputers with thousands of nodes.

HPX-5 defines a broad API that covers most aspects of the system. Programs are organized as diffusive, message driven computation, consisting of a large number of lightweight threads and active messages, executing within the context of a global address space, and synchronizing through the use of lightweight synchronization objects. The HPX-5 runtime is responsible for managing global allocation, address resolution, data and control dependence, and scheduling threads and the network.

The HPX-5 *global address space* provides a global shared memory space abstraction and serves as the basis for computation. Global allocation is performed through a set of dynamic allocators that provide individual, block cyclic, and user-defined allocation for blocks of memory. Access to data in the global address space is provided through an asynchronous `memput/memget` API. Explicit global address translation can be performed in order to operate on local machine virtual aliases. Finally, raw global addresses serve as the targets for HPX-5's active message *parcels*, described below. Localities (roughly equivalent to MPI processes) are mapped into the global address space and can be accessed through indices allowing messages to target localities as in other active message runtimes.

Parcels form the basis of parallel computation in HPX-5. They contain a description of the action to be performed, argument data, and continuation information (not explicitly used in DASHMM) and are sent to the global address on which the action is to be performed. The HPX-5 scheduler invokes parcels as lightweight threads once they reach their destination. This parcel–thread equivalence is key to abstracting the difference between shared and distributed execution in HPX-5. Sending a parcel is equivalent to, and the only means of, spawning a lightweight thread. In shared memory execution it just happens that all target addresses are on a single locality. Unlike many other AMT runtimes, HPX-5 is designed around cooperative threading and not simply run to completion tasks, however this additional semantic power is not currently necessary for DASHMM.

Program data and control dependencies are represented in memory by *local control objects* (LCOs). An LCO is an event-driven, lightweight, globally addressable synchronization object that co-locates data and control information. All LCOs have input slots, predicates that evaluate functions of the inputs and may determine that an LCO has been triggered, and continuations (i.e., dependent threads and parcels) that will be executed once the LCO is triggered. This allows the user to build fully dynamic dataflow networks managed by the runtime. A simple example of an LCO is a reduction that performs a sum across its inputs. HPX-5 is delivered with a number of classes of built-in LCOs, e.g., futures and reduction types, and permits user-defined LCO classes as well. A user-defined LCO encodes the data that it represents, the task performed when an input becomes available, and the predicate under which the LCO is considered to be triggered. Section IV describes how DASHMM implements its execution DAG in terms of a network of user-defined LCOs.

IV. Parallel DAG Evaluation

The FMM is widely used in an iterative procedure where the same DAG is evaluated multiple times for different inputs. In this use case, the cost of any initial setup can be amortized over the many evaluations. Further, though there is opportunity for parallelism when building the hierarchical partitioning of the data and in other setup tasks, the amount of parallelism is greatest in the evaluation phase, meaning the first place to look for benefits from adopting an AMT approach is during the evaluation. While all phases of DASHMM (and thus the FMM) are parallel, DAG evaluation is the phase of the overall execution that has seen the most development effort.

DASHMM builds two representations of the DAG for a given evaluation, an explicit DAG that is used by the framework during partitioning and distribution, and a second,

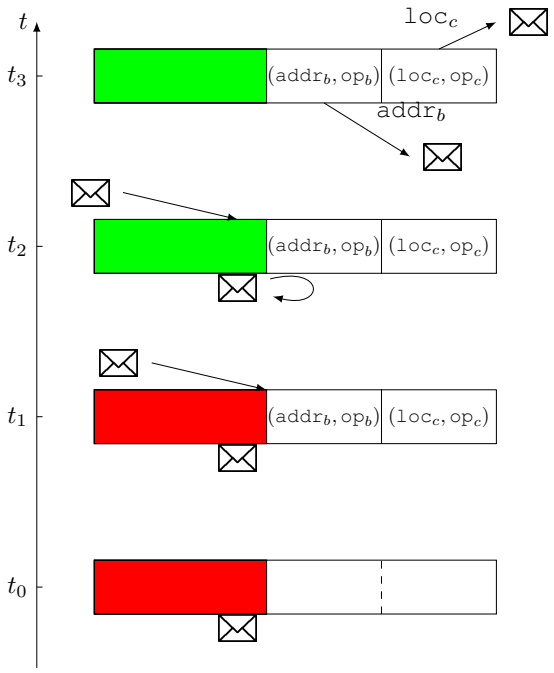


Fig. 2. Demonstration of parallel DAG evaluation via a dataflow network of HPX-5 Local Control Objects (LCOs). At t_0, an LCO for an expansion that needs one input data and has two out-going edges is allocated. At t_1, an HPX-5 parcel arrives and registers two continuations (out-going edges) for the LCO. At t_2, the input data arrives, triggers the LCO, and invokes the registered continuations. At t_3, the continuations are processed resulting in two additional dataflow messages.

executable DAG implicitly represented using HPX-5's LCO infrastructure. DASHMM creates the explicit DAG in three steps. First, the source and target ensembles are sorted at a coarse level based on the number of localities and then distributed equally across localities. Next, each locality performs adaptive hierarchical partitioning on its assigned points. Finally the results of this parallel partitioning are compactly shared across localities. The explicit DAG representation is used in a *distribution policy* to assign the nodes of the implicit DAG to the available localities. The creation of the explicit DAG and the allocation and placement of the implicit LCO dataflow graph do not need to be carried out in sequential order, and the DASHMM API permits the distribution policy to make placement decisions for the implicit DAG during explicit DAG generation.

The only constraint on the distribution of the implicit DAG is that the nodes representing the multipole expansion of a source leaf box, or the local expansion of a target left box, are fixed to match the a-priori distribution of the source and target data. The distribution policy used in Section V is designed for FMMs that implement the merge-and-shift technique [2], [17]. In the implicit DAG, the node representing the multipole or intermediate expansion of a source box, or the local expansion of a target box, is fixed to the locality owning that box. The node representing the intermediate expansion of a target box is placed by trying to minimize communication cost while increasing slack time to hide communication latency.

From the explicit DAG, DASHMM knows the LCO operation type and out edge degree, and thus its size. Once the location of a node of the implicit DAG is computed using the distribution policy, an appropriate LCO is allocated. The custom LCO type used in DASHMM stores both the node expansion data and the DAG out edge data (see Figure 2 time t_0). The expansion data varies from kernel to kernel but is essentially a set of complex double values. The out edge data consists of the set of edge operation types and the target DAG node global addresses, and is sorted into local and remote edges. As the individual DAG nodes are allocated and their global addresses fixed, DASHMM will concurrently backfill the predecessor out edge global addresses necessary for execution (see Figure 2 time t_1).

During execution, the custom LCO will continuously reduce input data into the stored expansion data. Once the last input arrives, the LCO will be triggered by HPX-5 and all of its dynamically registered continuations will be executed as lightweight threads. DASHMM registers a single continuation for its implicit DAG LCO that is responsible for processing the out edge list (Figure 2 time t_2 and t_3). Conceptually, processing an out edge requires transforming the expansion data along each edge and setting the edge's target LCO. Local edge operations—that is edges for which the target DAG node is on the same locality—are performed sequentially and input into their target LCOs which may trigger future asynchronous evaluation.

As the LCOs reside in the global address space provided by HPX-5, remote edges could be processed in the same manner. The subsequent tasks will always occur at the correct locality, even if that requires network traffic (handled by HPX-5). However, this means that if a given LCO has multiple out edges to a given locality then DASHMM would send transformed data for each edge across the network. Instead, DASHMM manually examines the out edges and sends only a single coalesced active-message parcel containing the expansion data and the relevant out edges to any given locality, where they are then evaluated as normal.

In this way, DASHMM sets up an implicit DAG of LCOs and parcels representing tasks to perform that ultimately will result in the computation of the potentials at each target location due to each source.

V. Results

A. Strong Scaling

The strong scaling performance of the DAG evaluation portion of DASHMM for the FMM was evaluated on Indiana University's Cray XE6/XK7 supercomputer, Big Red II. Each compute node has two AMD Opteron 16-core Abu Dhabi x86_64 CPUs and 64 GB of RAM. All compute nodes are connected through the Cray Gemini interconnect. The tests used the 4.0.0 version of HPX-5 [19] with the Photon network [15] and a mild modification of version 1.0.0 of DASHMM [20] that added the ability to trace DASHMM execution

events (see below). HPX-5 was configured with a statically partitioned global address space implementation and local randomized workstealing for node-local thread scheduling. Both codes were compiled using GNU compiler version 5.3.0 with optimization flag `'-O3'`.

The source and target ensembles used in the tests were of the same size, were distinct, and were drawn from the same type of distribution. Two types of distributions were used: in the first, points were distributed uniformly in a cube; in the second, points were distributed uniformly on the surface of a sphere. The cube data produces dual trees that are fairly uniform, with every leaf in the tree having the same depth, and so the critical path is shorter. The sphere data produces much more non-uniform trees with a longer critical path.

The tests used two interaction types (kernels) for each source and target distribution: the scale-invariant Laplace kernel ($1/r$), which gives the typical potential for electrostatics or Newtonian gravitation, and the scale-variant Yukawa kernel ($e^{-\lambda r}/r$, with $\lambda > 0$), which, for example, gives the potential for the screened Coulomb interaction. The merge-and-shift technique was implemented for each kernel. For the scale-variant Yukawa kernel, the length of the intermediate expansion depends on the depth in the hierarchy.

Every test used 60 as the refinement threshold and required 3-digits of accuracy [17, Eq.(57)]. For each test, the number of source and target points was selected to be the largest that fits in a single node's memory. Practically, this is set by the maximum size that can be run with the Yukawa kernel. For cube data, the largest problem that could fit on one node was 60 million sources and targets. For sphere data, the largest problem that could fit on one node was 42 million sources and targets.

Using the same source and target ensembles, and the same refinement threshold, an evaluation of the Laplace or Yukawa kernel will produce the same DAG with the same kinds of operations to perform. The details of those operations will change between the Laplace and Yukawa case, but the number and kind of operations will be identical. This allows a comparison of the effect of grain size on the resulting scaling. Generally speaking, the specific operations for the Yukawa kernel are heavier than the equivalent for the Laplace kernel.

The left side of Figure 3 shows the average evaluation time t_n of five runs for each combination of core count n, kernel and distribution; the right side shows the speedup relative to one node, t_{32}, for each set of runs. During execution, each core had one HPX-5 scheduler thread. At 4096 cores, the final scaling efficiency of each run is as follows: 60% for cube Laplace, 74% for cube Yukawa, 62% for sphere Laplace, and 69% for sphere Yukawa. Generally speaking, as the tasks become heavier moving from Laplace kernel to Yukawa kernel, the scaling is improved. The scaling efficiency starts to visibly deviate from the ideal line from 512 cores on.

B. Resource Utilization

To understand the scaling behavior of DASHMM, we collected event traces marking the beginning and ending of the various operations performed by DASHMM during the execution. These include the translations of approximations, the evaluation of approximations at target locations, the accumulation of contributions to the data represented by the expansion LCOs, and the direct interaction between nearby source and target locations. These events are used as a way to measure the percentage of time during the execution that is spent doing the work of DASHMM, as opposed to the work that HPX-5 must perform to manage the computation.

TABLE I
COUNT, SIZE AND MIN/MAX IN-/OUT-DEGREE OF DAG NODES.

Type	Count	Size [B]	d^{in}_{min}	d^{in}_{max}	d^{out}_{min}	d^{out}_{max}
S	2097148	32-1920	0	0	9	28
M	2396732	880	1	8	1	2
I_s	2396732	5472	1	1	7	26
I_t	2396672	25536	56	208	1	8
L	2396672	880	1	2	1	8
T	2097152	40-2400	9	28	0	0

TABLE II
COUNT, MESSAGE SIZE AND AVERAGE EXECUTION TIME OF DAG EDGES.

Type	Count	Size [B]	t_{avg} [μs]
S→T	55742860	32-1920	1.89
S→M	2097148	880	10.9
M→M	2396668	880	4.60
M→I	2396732	5280	29.6
I→I	59992216	912-2736	1.75
I→L	2396736	880	38.4
L→L	2396672	880	4.45
L→T	2097152	880	13.5

The DAG resulting from an FMM evaluation is quite large, and so the resulting execution traces are also quite large. To produce a workable amount of data in the traces only 30 million source and target locations in a cube were used. We collected traces for $n = 64, 128$ and 512 cores. In each case the DAG is identical, and it is only the distribution of that DAG onto the available computational resources that changes. For this case, there are 13.8 million nodes, and 129 million edges.

Table I gives details about the six classes of node in the DAG, including the number of those nodes, the size in bytes of the data represented by the node, and the min/max in-/out-degrees of the node. The subscripts on the two classes of intermediate nodes indicates to which tree the node is most closely associated. The sizes for the source and target nodes are given as a range because the number of source or targets is between 1 and 60 for each leaf of either tree. There are very similar numbers of the six classes of node. The intermediate nodes stand out both in message size and connectivity; the bulk of the work in FMM is in the generation of the intermediate expansions.

Table II shows a summary of the DAG edges, including the type of nodes connected by the edge, the number of such edges, the size in bytes of the data transferred through the edge, and the average execution time (in the 128 core run) of the various operation types. The sizes of the S→T operation is a range as the number of sources in a given leaf

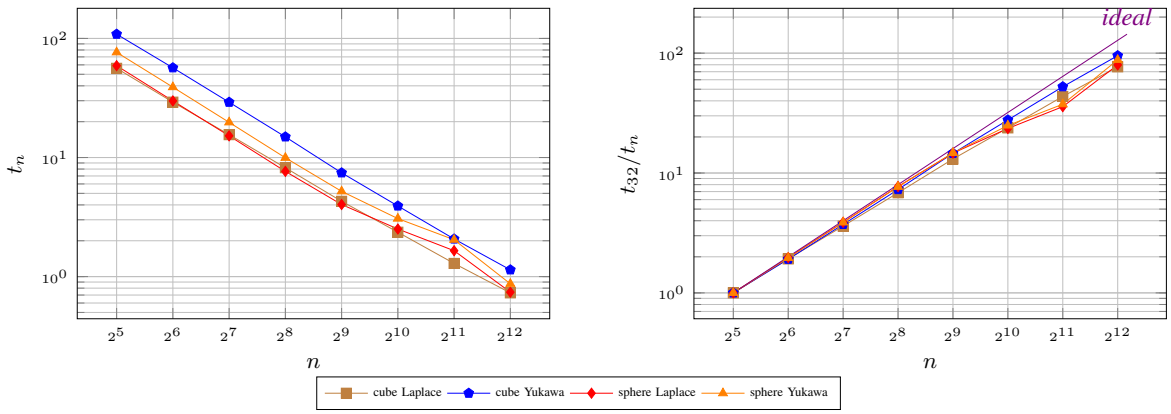


Fig. 3. The time to completion t_n (left) and speedup relative to 32 cores t_{32}/t_n (right) as a function of core count, n, for the four strong scaling runs: cube Laplace (brown square), cube Yukawa (blue pentagon), sphere Laplace (red diamond) and sphere Yukawa (orange triangle). There were 60 million sources and targets for cube data, and 42 million sources and targets for sphere data.

is between 1 and 60. The size of the I→I operation data is a range because the amount of information that must be conveyed from a source intermediate expansion to a target intermediate expansion depends on the relative positions of the two nodes. The single largest contribution to the edges is the I→I operation; much of the effort of FMM is to move information from the source tree to the target tree. Though, this work is relatively simple, and so the average time to execute one of these edges is the smallest of any class.

To measure how well the work represented by the DAG edges is mapped onto the available computational resources, we define a utilization fraction to be the fraction of time spent performing these DASHMM events out of the entire available time. This fraction is computed in a series of time intervals of uniform size, $\Delta t_k = t_{k+1} - t_k = \Delta t_{total}/M$, where M is the number of intervals, and Δt_{total} is the total elapsed evaluation time. If $\Delta t_k^{(i)}$ is the amount of time spent doing operations of class i (e.g. S→M or L→T), during time interval k, and n is the number of scheduler threads in all localities, then

$$f_k^{(i)} = \frac{\Delta t_k^{(i)}}{n \Delta t_k} \tag{1}$$

is the utilization fraction for that event class. From these, one easily forms the total utilization fraction:

$$f_k = \sum_i f_k^{(i)} = \frac{\sum_i \Delta t_k^{(i)}}{n \Delta t_k}. \tag{2}$$

Shown in Figure 4 is the total utilization fraction for a few runs with cube data, using the Laplace kernel on $n = 64, 128$ and 512 cores as computed from the trace files collected for the smaller run of 30 million source and 30 million target points. Figure 4 is plotted as a function of interval k because each core count takes a different absolute time to execute.

For most of the execution, the system is performing work that directly addresses DASHMM's needs. We suspect that the deficit from full utilization to the average of about 90% that is achieved is largely due to dynamic memory allocation and

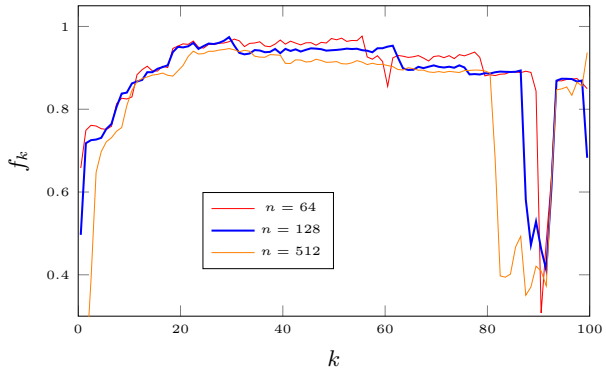


Fig. 4. Total utilization fraction, f_k, as a function of time interval, k, for 2-, 4-, and 16-locality runs of 30 million source and target points with cube data using the Laplace kernel. The utilization was computed for 100 uniform intervals of the total evaluation time for each case: 34.6 s for 64 cores, 17.6 s for 128 cores and 4.55 s for 512 cores. Each run shows a high average utilization with a period of underutilization. The relative width of this period increases with locality count, showing a primary reason for scaling inefficiencies at high core counts.

memory copies related to the implementation of dynamic non-local DAG out edge handling. Asynchronous HPX-5 managed network progress accounts for a small fraction of this deficit as well. For a similar figure, made for a run on a single node where networking and copying is not needed, the total utilization hovers near 98% for the bulk of the execution. The deficit in utilization at the beginning of the execution is due to the fact that the first tasks are only just being performed, and the amount of ready parallel work has yet to saturate the system. Around 20% of the way through the execution, each case has reached a plateau. This saturation level decreases somewhat with increasing numbers of localities, likely due to the increased number of non-local out edges and increased overhead in HPX-5 management of a larger system.

Clearly visible in this figure is a dip in utilization near the end of the evaluation. This dip reflects a period of reduced

supply of tasks in the system. Further, as we go to larger core counts, the width of this region is a larger relative fraction of the total execution time. This is the predominant reason for the scaling inefficiencies seen in Figure 3. As the core count increases, the relative cost on the execution of this constrained concurrency is increased. This also explains the improved scaling for the Yukawa cases: the larger overall execution time for the larger grain size means that the number of available tasks has a smaller impact on the scaling.

C. Scheduling & Critical Path

This period of underutilization is a clear impediment to improved scaling. One suspected cause would be an imbalance in the workload across the localities used. However, an examination of the event traces in detail shows that the workload is well-balanced, with each locality reaching the region at the same time. Instead, it is a result of the HPX-5 exploring the DAG using an inefficient schedule that is oblivious to the critical path.

The DAG arising from FMM can be divided into three sets of operations: those that move data up the source tree, those that move data from the source to the target tree, and those that move data down the target tree. In this general flow, the critical path moves up the source tree and back down the target tree, so it is vital to perform all of the upward work on the source tree as early as possible, this includes both the S→M and M→M operations. The operations bridging the trees include M→I, I→I, and I→L. Finally, the operations yielding the final results are S→T, L→L, and L→T.

Figure 5 shows the utilization fraction by class for the 128 core run: the top panel shows the operations that move data up the source tree; the middle panel shows those operations that move data from the source to the target tree; and the bottom panel shows those operations that contribute to the final values at the target locations. In each panel, two dashed lines indicate the period of underutilization that is visible in Figure 4.

The importance of performing the computations up the source tree cannot be understated. Without these data, many of the subsequent operations are blocked. The top panel of Figure 5 shows that the work of S→M and M→M is being scheduled throughout the execution up to the underutilized region. Ideally, these tasks should be performed as soon as they are ready, and the root level of the source tree would be reached as soon as possible. It is also worth noting the vertical scale of the top panel as compared to the other two panels; the absolute amount of work in the upward pass is fairly small, meaning it could easily be completed very early using a better schedule.

At around the time that the final M→M operations finish, the final M→I operations occur. These then allow for a final burst of I→I work that translates information from the source tree to the target tree. Then, this also allows for the last of I→L.

To compute the final result at each target location a given DAG node needs the local expansion from its parent. Following this dependence up to (near the) root, this means

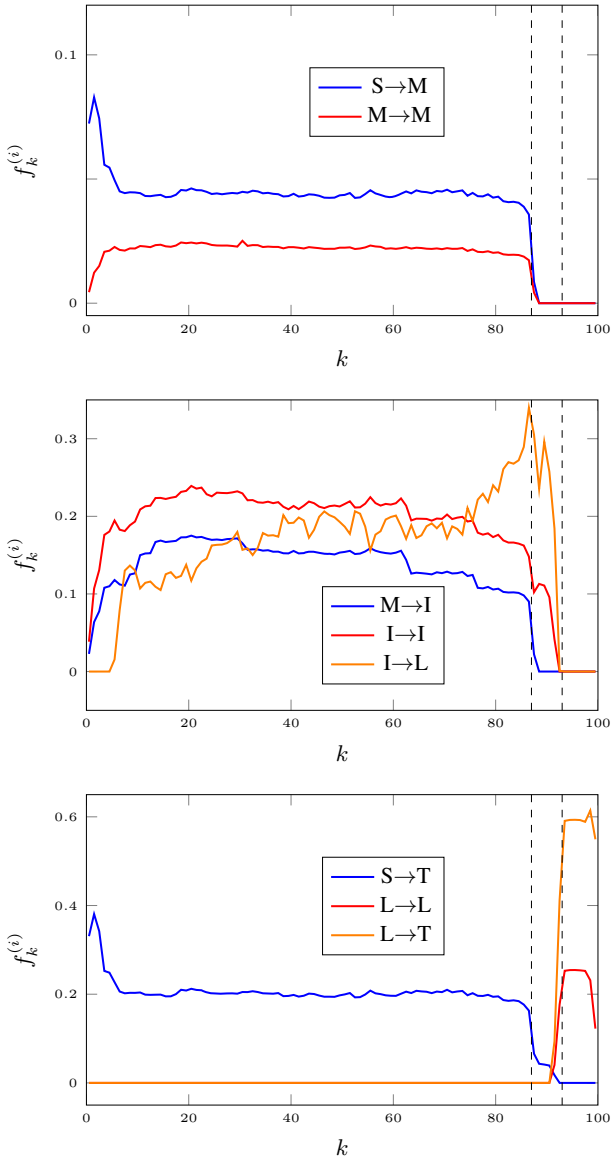


Fig. 5. Utilization fractions by class, $f_k^{(i)}$ as a function of time interval, k, for the 128 core run. Each of the 100 intervals is of uniform size, dividing equally the 17.6 s of the entire evaluation. The dashed lines in each panel indicate the start and end of the underutilized region seen in Figure 4. (top) The operations up the source tree. (middle) The operations that link the source tree to the target tree. (bottom) The operations that contribute to the final values computed at the target locations.

that there is a severe bottleneck at the top of the tree, after which the amount of available parallelism rises sharply. This is seen in the bottom panel of Figure 5. At the end of the underutilization phase, the amount of available work explodes, the utilization fraction rises sharply, and the pathology ends. The final operations during the execution are the translation and evaluation of the local expansions.

Figure 5 illustrates clearly the limitation to scaling that DASHMM is experiencing. The lack of any mechanism to

specify critical work to HPX-5 means that all work is treated equally, leading to the most important work to occur even up to 83% through the execution. If this could be addressed by the introduction of priorities for the tasks, even so simple a system as a binary choice between low and high priority, this underutilization could largely be eliminated. Early execution of the most critical work up the source tree would allow the work at the bottleneck at the top of the trees to occur earlier, overlapped with other less critical work. This phase would then be eliminated, and the resulting scaling would be improved.

It is also worth mentioning that there are some features of Figure 5 that are encouraging. During much of the early execution, there is a wide variety of work occurring, indicating that the evaluation is indeed driven by the data. As various expansions become available, the translations of those expansions are then computed, demonstrating HPX-5's capability to expose and exploit the parallelism inherent in the FMM. More importantly, the I→I operations that dominate the FMM's execution and involves heavy network traffic are executed at a constant utilization fraction up to the underutilized region, suggesting communication latency is well hidden behind computation.

VI. Conclusions

We have presented a novel implementation of FMM using the advanced, experimental runtime system, HPX-5. The complexity of FMM and the resulting DAG provides a good test of the ability of HPX-5 to manage and execute an extremely large number of tasks across a large number of cores. The fine-grained data-driven implementation was tested out to 4096 cores for a variety of problem sizes and type.

Generally speaking, the implementation of HMM in DASHMM scales well out to 4096 cores. At higher core counts, a period of constrained concurrency limits the ultimate scalability. This constrained concurrency is due to an inability of HPX-5 to address the priority of tasks in its current implementation. This application has a set of tasks that should be performed as soon as possible, but were delayed and computed throughout most of the execution.

The sequential execution of out edges maximizes cache locality as the large input for each edge computation is reused, but sacrifices parallelism and makes it difficult to prefer the critical paths through the graph. Unfortunately the current HPX-5 infrastructure is not able to help here. There is no way to indicate to HPX-5 that two parallel operations will benefit from running together, nor is there a way to prioritize critical operations.

However, now that the issue has been measured and understood, it will be straightforward to treat, either by extending the interface of HPX-5 to accept priority hints, or by using detailed knowledge of the current HPX-5 scheduler to present work to HPX-5 in an order that will emphasize the critical tasks. The benefit of this improvement is easy to estimate. Given the known widths of the starved region, and under the simple assumption that the utilization during those times would return to its saturated value, one can estimate how long the work occurring during that phase would take. The effect is to increase the scaling efficiency by 10% or more.

In creating DASHMM, the parallelization strategy was completely redesigned to take advantage of the opportunities afforded by HPX-5. This redesign started not with existing parallel implementations, but instead from the fundamental parallelism in HMMs. This change of perspective was not obvious or easy. Further, once the design of how we were going to fit HMMs into the conceptual framework provided by ParalleX was complete, that strategy must be implemented in actual HPX-5 constructs, which provide only a subset of the functionality promised by ParalleX.

The implementation process presents its own set of challenges. For example, the initial implementation of DASHMM kept the bulk of the expansion data inside the LCOs in the global address space. However, HPX-5 has some alignment requirements on data in the global address space that led to padding, which significantly lowered the maximum problem size that was treatable. As a result, much of the expansion data was removed from the global address space. Additionally, debugging nontrivial code that uses HPX-5 is very involved. The asynchronous nature of the resulting execution can lead to some very subtle bugs, and the lack of custom tools makes tracking bugs down more difficult. Further, if bad data is passed into a parcel, then that bad data makes its way through HPX-5 for a time before appearing in user code, and so the source of the trouble is obfuscated, making these errors time-consuming to fix.

The effort involved with creating an implementation to take advantage of the parallelism offered by HPX-5 is large. But the results of that effort, we believe, justify the expenditure. The resulting implementation hews more closely to the inherent parallelism in the method, and to the description of the work as a set of tasks to perform. In working on the implementation, we have routinely suggested new features for HPX-5 that benefit the system in general. And with the realization of the direct impact of a lack of priority hints on the scaling of this application, which is a good proxy for a wide class of dynamic adaptive algorithms, means that the effort will continue to pay off both for DASHMM and for HPX-5.

Acknowledgments

The authors were supported in part by National Science Foundation grant number ACI-1440396. This research was supported in part by Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute, and in part by the Indiana METACyt Initiative. The Indiana METACyt Initiative at IU is also supported in part by Lilly Endowment, Inc.

References

[1] J. Carrier, L. Greengard, and V. Rokhlin, "A fast adaptive multipole algorithm for particle simulations," *SIAM J. Sci. Stat. Comp.*, vol. 9, pp. 669–686, 1988.

[2] L. Greengard and J. Huang, "A new version of the fast multipole method for screened Coulomb interactions in three dimensions," *J. Comput. Phys.*, vol. 180, pp. 642–658, 2002.

[3] W. C. Chew, M. S. Tong, and B. Hu, *Integral Equations Methods for Electromagnetic and Elastic Waves*. Morgan & Claypool, 2008.
[4] 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.
[5] ——, "A parallel hashed oct-tree n-body algorithm," in *SC 93': Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, 1993.
[6] F. A. Cruz, M. G. Knepley, and L. A. Barba, "PetFMM–A dynamically load-balancing parallel fast multipole library," *Int. J. Numer. Meth. Eng.*, vol. 85, pp. 403–428, 2011.
[7] L. Greengard and V. Rokhlin, "A new version of the fast multipole method for the Laplace equation in three dimensions," *Acta Numer.*, vol. 6, pp. 229–269, 1997.
[8] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based FMM for multicore architectures," *SIAM J. Sci. Comput.*, vol. 36, pp. C66–C93, 2014.
[9] 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.*, vol. 7905, pp. 255–266, 2013.
[10] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, 2011, pp. 12–25.
[11] B. Zhang, J. Huang, N. P. Pitsianis, and X. Sun, "recFMM: Recursive Parallelization of the Adaptive Fast Multipole Method for Coulomb and Screened Coulomb Interactions," *Communications in Computational Physics*, vol. 20, pp. 534–550, 2016.
[12] H. Ltaief and R. Yokota, "Data-driven execution of fast multipole methods," *CoRR*, vol. abs/1203.0889, 2012.
[13] 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.
[14] A. Kulkarni, L. Dalessandro, E. Kissel, A. Lumsdaine, T. Sterling, and M. Swany, "Network-managed virtual global address space for message-driven runtimes," in *Proceedings of the 25th International Symposium on High Performance Parallel and Distributed Computing (HPDC 2016)*, 2016.
[15] E. Kissel and M. Swany, "Photon: Remote memory access middleware for high-performance runtime systems," in *In Proceedings of the 1st Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM) Workshop*, 2016.
[16] J. DeBuhr, B. Zhang, A. Tsueda, V. Tilstra-Smith, and T. Sterling, "DASHMM: Dynamic Adaptive System for Hierarchical Multipole Methods," *Communications in Computational Physics*, vol. 20, pp. 1106–1126, Oct. 2016.
[17] H. Cheng, L. Greengard, and V. Rokhlin, "A fast adaptive multipole algorithm in three dimensions," *J. Comput. Phys.*, vol. 155, pp. 468–498, 1999.
[18] M. Cimini, J. G. Siek, and T. Sterling, "The Semantics of ParalleX, v1.0," School of Informatics and Computing, Indiana University Bloomington, Tech. Rep. TR726, May 2016.
[19] "HPX-5," https://hpx.crest.iu.edu/about.
[20] "DASHMM," https://crest.iu.edu/projects/dashmm.