Asynchronous Task Scheduling of the Fast Multipole Method using various Runtime Systems

Bo Zhang
Center for Research in Extreme Scale Technologies, Indiana University, Bloomington, IN, 47408
zhang416@indiana.edu

***Abstract*—In this paper, we explore data-driven execution of the adaptive fast multipole method by asynchronously scheduling available computational tasks using Cilk, C++11 standard thread and future libraries, the High Performance ParalleX (HPX-5) library, and OpenMP tasks. By comparing these implementations using various input data sets, this paper examines the runtime system's capability to spawn new task, the capacity of the tasks that can be managed, the performance impact between eager and lazy thread creation for new task, and the effectiveness of the task scheduler and its ability to recognize the critical path of the underlying algorithm.**

***Keywords*—Fast Multipole Method, Data-driven, Cilk, C++11, HPX-5, OpenMP**

I. Introduction

The fast multipole method (FMM) [1], [2] is recognized as one of the top ten algorithms of the last century. The algorithm is capable of computing the pairwise interaction of N particles in $O(N)$ arithmetic operations to any prescribed accuracy requirement. As a result, it has been widely applied to accelerate many N-body problems in molecular dynamics simulation [3]. It has also been applied to accelerate the iterative solver in discretized systems resulting from boundary element or boundary integral formulations of problems in the field of computational electromagnetics [4], computational fluid dynamics and solid mechanics [5].

The execution of the FMM can be analyzed as traversing a directed acyclic graph (DAG), where a node represents the computation associated with a particular spatial location and a directed edge indicates the dependency from a predecessor node to a successor node. Scheduling policies for DAGs have been well studied [6], [7]. The effectiveness of a variety of DAG schedulers adopted in different runtime systems has been examined both theoretically [8], [9] and empirically using simple benchmark tests [10], [11]. What has not been studied, however, is the capability and effectiveness of task schedulers when they handle a more sophisticated algorithm like the FMM. This paper attempts to address this issue and considers four different runtime systems, including Cilk [6], [12], C++ thread and future libraries [13], the High Performance ParalleX (HPX-5) library [14], and OpenMP tasks. Here, we use runtime for ease of reference because strictly speaking, C++ thread and future libraries are not a runtime system. In the paper, Cilk refers to the implementation of CilkPlus provided by Intel. HPX-5 is a research level runtime system implementing the ParalleX execution model [15], which is an experimental execution model designed to mitigate the key sources of performance decay identified by the SLOWER performance model [16].

There are several features of the DAG corresponding to the FMM execution that challenge the capability and effectiveness of the runtime systems. In the DAG, each node has different in-degree and out-degree that cannot be known quantitatively until execution time. The input data for each node becomes ready at different times and each requires different amounts of processing time. Partial input data can be processed at each node, but to avoid increasing the algorithm's arithmetic complexity, the output of the node cannot be released to its successor nodes until all inputs have been processed. As a result, at any time in the course of execution, there are potentially a large number of tasks for the scheduler to consider. Not all of these tasks are on the critical path of the execution. Moreover, a task created at an earlier time does not necessarily represent more work than the one created at a later time.

Overall, this paper makes the following contributions. 1) It shows how to asynchronously schedule tasks to achieve a data-driven FMM execution using Cilk, C++11 standard thread and future libraries, HPX-5, and OpenMP task. 2) It compares the runtime system's ability to create new tasks and the capacity of the tasks that can be managed. 3) It compares the performance difference between a compiler-based approach and a library-based approach, and measures the performance impact caused by eager and lazy thread creation. 4) It tests the effectiveness of the task scheduler and examines its ability to recognize the critical path of the algorithm.

The remainder of this work is structured as follows. We describe the general structure of FMM and special feature of the version used in this paper in Section II. We summarize the related work in Section III. We show how to schedule tasks asynchronously to achieve the data-driven execution of the FMM and some implementation details in Section IV. We examine the implementations using various input sets, and compare the performance, capability, and effectiveness of the runtime systems in Section V. We give our conclusion and directions for future work in Section VI.

II. Overview of the Fast Multipole Method

This section provides an abstract description of the algorithmic structure of the FMM. Mathematical details are beyond the scope of this paper and can be found in Refs [1], [2], [17].

The FMM first determines the smallest bounding box that contains all the interacting particles as its computational domain and then performs a hierarchical partition by dividing the box equally along each dimension repeatedly. Boxes with no particles are pruned in this process. The partition is called *uniform* if one specifies the maximum refinement level and *adaptive* if one stops partitioning a box when it contains fewer than a prescribed number of particles. The partition process naturally results in a tree structure and boxes without children are called *leaf* boxes. Strictly speaking, this tree is a fusion of two spatial partitions in one spatial hierarchy: one partitions the source ensemble and the other partitions the target ensemble. Target points are locations where the force field or potential needs to be computed. Depending on the applications, the source and target ensembles may be identical, partially overlapping, or completely disjoint in the spatial domain. As a result, the source and target trees can be identical, partially overlapping, or completely different. The FMM algorithm introduces extra edges to the trees that offer direct path between boxes on the source and target trees. Specifically, a box B_s on the source tree is connected with a box B_t on the target tree if: (1) B_s and B_t are of the same refinement level and the spatial volume enclosed by the two boxes are disjoint, referred to as *non-adjacent*; and (2) the parent boxes of B_s and B_t are adjacent. When such an edge exists between B_s and B_t, box B_s is also said to be in the *interaction list region* of box B_t.

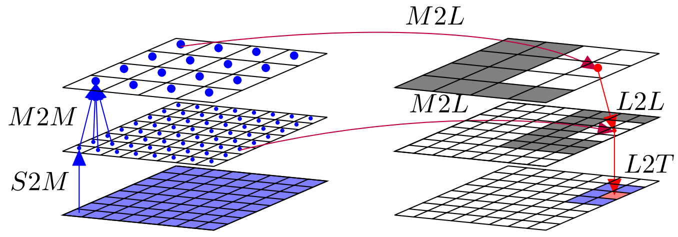

Fig. 1. An example of two-dimensional G_{FMM} results from a four-level uniform partition. Nodes corresponding to the root level and the first partition level are not drawn. In the figure, a blue (red) shaded node represents a cluster of source (target) points while a blue (red) dot represents the multipole (local) expansion of the corresponding source (target) node. There are two inter-scale traversals in G_{FMM}. One traversal occurs on the source tree, using the source-to-multipole ($S2M$) operator at the leaf node and multipole-to-multipole ($M2M$) operator at the non-leaf node to generate all the multipole expansions. The other traversal occurs on the target tree, using the local-to-local ($L2L$) operator to pass down local expansions from ancestor nodes to their descendants and the local-to-target ($L2T$) operator to evaluate the local expansions at the leaf target nodes. G_{FMM} also contains level-wise intra-scale traversals. For each target node, the multipole expansions of the source nodes in its interaction list region are translated into local expansions. The gray shaded region on the target tree depicts the interaction list region for two different target nodes.

The DAG for the FMM execution, denoted by G_{FMM}, have three major components: the source tree, the target tree, and level-wise bipartite graphs connecting the trees. Figure 1 shows an example of G_{FMM} in two dimensions resulting from a four-level uniform partition. To avoid confusion, we will use *box(es)* to refer to the spatial cluster of particles and *node(s)* to refer to the vertices in G_{FMM}. When the source and target ensembles are identical, there are two nodes in G_{FMM}—one on the source tree and the other on the target tree—corresponding to the same box. Data movement in G_{FMM} is from the source tree (blue, shown on the left) to the target tree (red, shown on the right), involving both inter-scale tree traversal and intra-scale traversal. On the source tree, the inter-scale traversal generates the multipole expansion for all the nodes. The multipole expansion for each node is a truncated series that approximates the far-field influence of the particles contained in the node up to a prescribed accuracy requirement. At a leaf node, the multipole expansion is generated from particle information using the source-to-multipole ($S2M$) operator. At a non-leaf node, the multipole expansion is generated by shifting the multipole expansions of the child nodes, using the multipole-to-multipole ($M2M$) operator. On the target tree, the operations are associated with local expansion, resulting from shifting the center of the multipole expansions. On the target tree, the inter-scale traversal passes the local expansions from the ancestors to their descendants using the local-to-local ($L2L$) operator and evaluates the local expansions at leaf nodes using the local-to-target ($L2T$) operator. During intra-scale traversal, the multipole expansions of the nodes on the source tree for a given target node's interaction list region are converted into local expansions using the multipole-to-local ($M2L$) operator. Conventionally, the traversals are executed level by level in two passes, starting with an upward pass on the source tree, followed by a downward pass on the target tree. Intra-scale operations are usually carried out during the downward pass.

The FMM implemented in this paper has one special feature called *merge-and-shift* technique [17]. The technique is developed to reduce the number of $M2L$ operations performed on nodes of the same parent since there exists a significant overlap of their individual interaction list regions. Particularly, if there are M_1 nodes sharing M_2 nodes in their interaction list regions, instead of doing $M_1 \times M_2$ translations, one can first merge the M_2 multipole expansions into one and then shift it to M_1 nodes using only $M_1 + M_2$ translations. Figure 2 shows a simple demonstration of this technique. By applying this technique, the average number of $M2L$ translations performed on each node can be reduced from 189 to 40. The mathematical foundation for this technique is the use of exponential expansion, multipole-to-exponential ($M2E$) operator, exponential-to-exponential ($E2E$) operator, and exponential-to-local ($E2L$) operator. This means, the $M2L$ translation is done in three steps. The multipole expansions of the source nodes are first translated into exponential expansions. Afterwards, merging and shifting the expansions involves component-wise manipulation only. Finally, the exponential expansion formed at each target node is translated back to the local expansion.

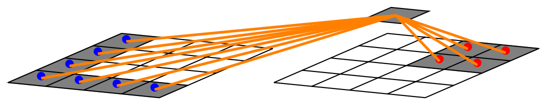

Fig. 2. Illustration of the merge-and-shift technique employed in the FMM tested in this paper. Each of the four shared nodes on the right side needs to convert the multipole expansions of the seven shaded nodes on the left side into local expansions. Instead of doing 7×4 translations, the 7 multipole expansions can be merged into 1 and then shifted to the four nodes on the right, using $7+4$ translations. In three dimensions, this technique could reduce the average number of $M2L$ operations performed for each target node from 189 to 40.

III. Related Work

Parallel FMM implementations often apply the so-called "MPI + X" approach. That is, the entire problem is partitioned and distributed over a cluster of compute nodes and MPI is used for communication. Within each compute node, the work is executed in a multi-threaded way to utilize the parallelism provided by the multicore processors or many-core accelerators.

The most common approach used to partition the work on distributed memory architectures is based on the *locally essential tree* (LET) method originally developed by Warren and Salmon [18]. The method uses a space-filling curve to index all the leaf nodes in G_{FMM}, partitions the curve into segments and distribute them evenly among available processes. On each process, the LET is defined as the union of all the nodes needed in the computation for all the owned leaf nodes and their ancestors. The LET provides the communication pattern for each process. It is usually constructed before the traversal on G_{FMM} happens. With the LET ready, care is often taken to hide communication within computation. Historically, LET was constructed in a top-down fashion and became the bottleneck in parallel execution when the number of available processes increases. A bottom-up approach [19] was developed to address this issue.

Another interesting approach on distributed memory architectures is the one used in PetFMM [20]. It partitions G_{FMM} into smaller components such that the number of components is more than the number of available processes but much smaller than the number of nodes in G_{FMM}. Next, it builds a new weighted graph to represent the computation within each component and communications between different components. The derived weighted graph is divided into as many equally weighted parts as the number of available processes, and the result is used to assign components to available processes. We point out that this approach has the advantage of considering intra-scale traversal cost in distributing the work. However, the size of the derived weighted graph grows when the number of available processes increases, which makes the partitioning decision more difficult to compute and the resulting algorithm possibly less scalable.

On shared memory architecture, parallel FMM often adopts relatively simple strategies. When implemented in a level-wise fashion, one can simply process all the nodes of the same level in parallel loops [21]. A slightly complicated approach is the *costzone* method [22]. The method estimates a cost for each node and partitions G_{FMM} into different branches of approximately equal costs. Each branch is then assigned to one available core or thread.

There are certain limitations of the above approaches. First, they are less *dynamic* than one would ideally expect. This means, once the data partition is completed, each process has a somewhat static assignment until the data is repartitioned. Second, they all rely on weight estimation. Without prior information, initial weights for each node are often identical or proportional to the number of particles contained in the node. In an iterative setting, these weights need to be adjusted in the subsequent steps. This is often done by inserting extra instructions to track execution time or number of certain operations occurred at each node at the current step so that they can be used to estimate a new cost for the next step. This not only introduces additional overhead, but also may not reflect the actual execution accurately. By contrast, data-driven approaches seem promising to address these issues, such as the ones in Refs. [23]–[25], and the several versions to be discussed in Section IV.

IV. Data-driven Parallel FMM

Unlike coarse-grained data partition approaches used in conventional parallel FMM implementations, data-driven approach can be interpreted as creating sufficient fine-grained tasks that reveal the structure of G_{FMM} and using the runtime system to schedule these tasks to available processing resources while obeying all dependency constraints. In this section, we show how to implement data-driven FMM using Cilk, C++11 standard thread and future libraries, HPX-5 library, and OpenMP tasks.

There are several issues to consider in a data-driven parallel FMM implementation. One issue is how to express these tasks. The simplest way would be to start from every leaf node on the target tree where the final result is rendered and trace backwards the entire history of data flow to create tasks. This, however, is undesirable because it could create many redundant tasks that ultimately increase the overall arithmetic complexity.

Notice that a node in G_{FMM} can be processed even if only a subset of its input data is available. This generates a series of questions in a data-driven implementation. Do we create new tasks for them? Can the tasks be created easily using the runtime system? When created eagerly, many of these tasks are not on the critical path of the execution until a later time. Can the task scheduler of the runtime system recognize this fact? If the scheduler fails to do so, what impact will it have on the performance?

As an extension to the C/C++ language, Cilk introduces three new keywords to implement task parallelism. Tasks can be spawned using `cilk_spawn` and `cilk_for` and synchronization is supported using `cilk_sync`. To avoid creating redundant tasks, one could use locks. Each thread that attempts to create a new task needs to grab the lock first to check whether the task has already been created. If the task has not been created, the thread spawns the new task and uses `cilk_sync` to wait for its completion. Otherwise, the thread has to grab the lock from time to time to check the status of this dependent task itself. This introduces significant overhead to the program and reduces scheduling opportunities from the task scheduler of the Cilk runtime. Alternatively, one could use hyperobjects since operations on each node are essentially reductions. However, there are two difficulties with this approach. First, both the input and output of the reduction are associated with vectors of double precision complex numbers. Second, the in-degree of the nodes in the target tree is usually large. These could potentially limit the problem size that can be handled.

A better strategy to use Cilk is to divide G_{FMM} into disjoint components such that at the cost of introducing synchronization barriers between components, one can eliminate the need of locks (or hyperobjects) and let Cilk's runtime handle all the synchronization points. Fortunately, this goal can be accomplished by dividing G_{FMM} into only two components: one component is the source tree and the remainder of G_{FMM} is the other component. It is clear from Figure 1 that completing work within the first component provides all the required input to the second component. Within each component, tasks can be created during a pre-order traversal of the source and target tree using `cilk_for`. The pre-order traversal guarantees that no redundant task will be created. The code snippet of the corresponding implementation is shown in Figure 3. We point out the OpenMP implementation considered in this paper basically replaces `cilk_for` inside the `Aggregate` and `DisAggregate` functions with `for, task, taskwait` directives.

```
void FMMCompute(void) {
  Aggregate(&snodes[1]);
  DisAggregate(&tnodes[1]);
}

void Aggregate(const Node *snode) {
  if (snode == NULL)
    return;

  if (snode->nchild) {
    // spawn tasks at a nonleaf source node
    cilk_for (int i = 0; i < 8; i++)
      Aggregate(snode->child[i]);

    // generate multipole expansion using M2M operator
    MultipoleToMultipole(snode);
  } else {
    // generate multipole expansion using S2M operator
    SourceToMultipole(snode);
  }
  // convert multipole into exponential expansion using
       M2E operator
  MultipoleToExponential(snode);
}

void DisAggregate(const Node *tnode) {
  if (tnode == NULL)
    return;

  // translate parent's local expansion using L2L operator
  LocalToLocal(tnode);
  ProcessList4(tnode);  // for adaptive fmm only
  if (tnode->nchild) {
    // complete M2L operation using merge-and-shift
    ExponentialToLocal(tnode);
    // spawn new tasks at a nonleaf target node
    cilk_for (int i = 0; i < 8; i++)
      DisAggregate(tnode->child[i]);
  } else {
    // evaluate local expansion using L2T operator;
    LocalToTarget(tnode);
    ProcessList13(tnode); // for adaptive fmm only
  }
}
```

Fig. 3. A data-driven parallel FMM implementation. G_{FMM} is partitioned into two separate components, equivalent to not spawning new tasks for the target nodes when they have partial input to process. The runtime system used is Cilk.

The above partition decision on G_{FMM} has a few consequences for the second concern mentioned at the beginning of this section. It is shown in Figure 1 that the completion of work along the source tree provides partial input to the nodes on the target tree. The partition adopted here prevents such tasks to be created and subsequently considered by the task scheduler. As a result, at the end of processing tasks within the first component, some resources could become idle due to insufficient work at the top portion of the source tree. However, we argue that the pros outweigh the cons. First, the work near the top of the source tree usually requires minimal processing time. Second, with fewer tasks to consider, the scheduler has an easier decision to make. Third, the partition allows the scheduler to focus on tasks on the critical path of the execution first.

```
void fmm::FMMCompute(void) {
  for (int i = 1; i <= nsboxes; i++)
    future_expo[i] = async(launch::async, &fmm::Aggregate,
         this, graph.getsbox(i));

  for (int i = 1; i <= ntboxes; i++)
    future_local[i] = async(launch::async, &fmm::
         DisAggregate, this, graph.gettbox(i));

  for (int i = 1; i <= ntboxes; i++)
    future_local[i].wait();
}

void fmm::Aggregate(const Node & snode) {
  if (snode.getnchild()) {
    // wait for multipole expansions of the child nodes.
    for (int i = 0; i < 8; i++) {
      int child = snode.getchild(i);
      if (child) future_mpole[child].wait();
    }
    MultipoleToMultipole(snode);
  } else {
    SourceToMultipole(snode);
  }
  promise_mpole[snode.getnodeid()].set_value();

  MultipoleToExponential(snode);
  promise_expo[snode.getnodeid()].set_value();
}

void fmm::DisAggregateconst Node & tnode) {
  if (tnode.getnchild()) {
    // wait for exponential expansions needed in the merge-
         and-shift operation
    const vector<int> & mslist = tnode.getmergeshiftlist();
    for (auto it = mslist.begin(); it != mslist.end(); it++)
      future_expo[*it].wait();
    ExponentialToLocal(tnode);
  }
  future_local[tnode.getparent()].wait();
  LocalToLocal(tnode);
  ProcessList4(tnode);
  if (tnode.getnchild() == 0) {
    LocalToTarget(tnode);
    ProcessList13(tnode);
  }
  promise_local[tnode.getnodeid()].set_value();
}
```

Fig. 4. Data-driven parallel FMM implementation using future and promise objects in C++11. The implementation does not partition G_{FMM}, equivalent to creating a new task for every node in G_{FMM} if it has partial input to process.

C++11 provides `future` and `promise` objects to manage asynchrony. A promise object is the asynchronous provider and is expected to set a value for a shared state at some point. A future object is an asynchronous return object that can retrieve the value of the shared state, waiting for it to be ready if necessary. The library further provides the `shared_future` object if the shared state needs to be retrieved multiple times once ready. These features offer more freedom to examine a variety of implementation choices. In addition to the approach adopted in the Cilk version, we can express every directed edge in G_{FMM} using a promise-shared future pair. This is equivalent to creating a new task for every node if the node has some partial input to process. Figure 4 shows the code snippet of this implementation. Here, at each source (target) node, the `Aggregate` (`DisAggregate`) function describes the input data the node depends on and the output data the node will produce. At each target node, the corresponding promise is set when all the related computation completes. When all the promises on the target tree are set, the entire computation terminates. We point out that inside the `async` call, in addition to `launch::async` which creates a new thread to execute the task asynchronously, there is another option called `launch::deferred` which executes the task on the calling thread the first time the result is requested.

HPX-5 is a research level runtime, implementing the ParalleX execution model. The ParalleX execution model makes use of lightweight threads and an active global address space that allows both code and data to move freely across the system in order to adapt to rapidly changing computational needs as well as environmental factors such as system load, power utilization, real-time events, and network performance. Unlike Cilk, C++11 and OpenMP, threads in HPX-5 are intended to work on distributed memory architectures as well. The HPX-5 library is being actively developed at Indiana University and the complete set of thread functionalities is being realized.

HPX-5 unifies the promise, future, and shared future objects in C++ into a single future object. It creates a lightweight thread by calling the `hpx_thread_create` function. This function takes four input parameters and two output parameters. The input parameters include a thread container as a context structure, thread option, thread entry function, and arguments to the entry function. The output parameters are the future representing the result and a handle of the thread. Unlike C++, HPX-5 does not provide a deferred option and all threads are created eagerly. However, the stack for the thread is only allocated when the thread is scheduled. The HPX thread calls `hpx_thread_exit` function to terminate its execution, at which time the future object specified in the `hpx_thread_create` function is automatically set. By using the future object and HPX-5 keywords, it is easy to implement one parallel FMM similar to the one shown in Figure 3 and another version similar to the one shown in Figure 4.

V. Performance Tests and Analysis

The FMM implemented in this paper computes the three dimensional electrostatic potential ϕ_i and force field $\mathbf{f}_i$ of N charged particles

$$\phi_i = \sum_{j=1, j\neq i}^{N} \frac{q_j}{\|\mathbf{x}_j - \mathbf{x}_i\|_2}, \quad \mathbf{f}_i = \nabla \phi_i.$$

All the codes were written in C/C++, restructured from the Fortran code available at [26].

The machine used for all the tests has two eight-core Xeon E5-2670 processors with hyper-threading disabled running at 2.6 GHz and 32 GB of RAM. Intel compiler version 13.0.1 was used to compile the HPX-5 library. The same compiler was also used to compile the Cilk, OpenMP, and HPX-5 FMM implementations with `-fast` flag. However, this version of the intel compiler does not support the `launch::deferred` option. Therefore, for the C++11 FMM implementation, the code was compiled using GNU compiler 4.8.1 with `-O3` option.

We used two types of data sets to test the performance of various implementations. For type I data, particles are uniformly distributed inside a cubic box of side length 1 and centered at the origin. For type II data, particles are distributed over the surface of a sphere of radius 1 and centered at the origin. For both data types, charges $\{q_i\}$ are distributed uniformly from the interval $[-0.5, 0.5]$. For each data distribution, we chose five different problem sizes, ranging from 10^4 to 10^6.

TABLE I. Duration of each basic FMM translation operator measured in microseconds. The results for $S2M$ and $L2T$ operators are the time caused by one particle. The other results are the time spent on processing one expansion of the respective input type.

Accuracy	$S2M$	$M2M$	$M2E$	$E2L$	$L2L$	$L2T$
3	0.56	3.24	9.01	2.80	3.79	0.88
6	1.81	15.2	62.7	15.3	18.3	3.32

TABLE II. Various information of the G_{FMM} used in the test. In all cases, the maximum number of particles allowed in a leaf node is set to be 40. In the table, N is the number of particles in the computation; L is the levels of partitions performed, or the levels of source (target) tree in G_{FMM}; $|G|$ is the number of nodes in G_{FMM}; C_{avg} is the average number of child nodes a non-leaf node has; P_{avg} is the average number of particles a leaf node has; the last three columns give the minimum, maximum, and average number of $M2L$ operations performed for nodes of the same parent using merge-and-shift technique.

Type	N	L	$\|G\|$	C_{avg}	P_{avg}	$M2L : (min, max, avg)$		
I	1e4	3	585	8.0	19.5	68	324	199.5
	5e4	4	4681	8.0	12.2	68	324	253.8
	1e5	5	4729	8.0	24.2	68	324	252.0
	5e5	5	37449	8.0	15.3	68	324	286.7
	1e6	6	47779	8.0	23.9	56	324	241.2
II	1e4	7	837	5.4	14.7	24	111	79.7
	5e4	9	4771	5.0	13.1	24	143	80.1
	1e5	10	7936	4.9	15.9	27	143	80.8
	5e5	12	40987	4.7	15.5	23	147	80.2
	1e6	13	90130	4.8	14.0	23	147	81.3

We imposed two different accuracy requirements in the tests, requiring three and six digits of accuracy of the computed potentials [17, Eq.(57)]. Higher accuracy requires longer expansions and will effectively increase the granularity of each node in G_{FMM}. Table I shows the duration of each basic FMM operators using the three and six digit accuracy requirements, measured in microseconds. In the table, the results for $S2M$ and $L2T$ operators are the time caused by one particle. The results for $M2M$, $M2E$, $E2L$, and $L2L$ operators are the time spent on processing one expansion of the input type.

The execution time of the FMM relies on an integer parameter denoted by s that specifies the maximum number of particles allowed in a leaf node. For each implementation, one could tweak the value of s for each input data (in terms of distribution and size), and each accuracy requirement to achieve the fastest execution time. Here, we choose to use the same s and set it to be 40 to create the same DAG in all the test cases in order to compare all the runtime systems. Table II summarizes various properties of G_{FMM} used in all the tests. For type I data, each non-leaf node has an average of eight child nodes, which makes the tree almost complete, except possibly at the finest partition level. The critical path, reflected by the levels of partition, for this distribution is short. For type II data, each non-leaf node has an average of five child nodes and the critical path is longer. The last three columns of the table report the minimum, maximum, and average number of $M2L$ operations performed among nodes of the same parent using the merge-and-shift technique. The wide range of the number of such operations is a major source of asynchrony

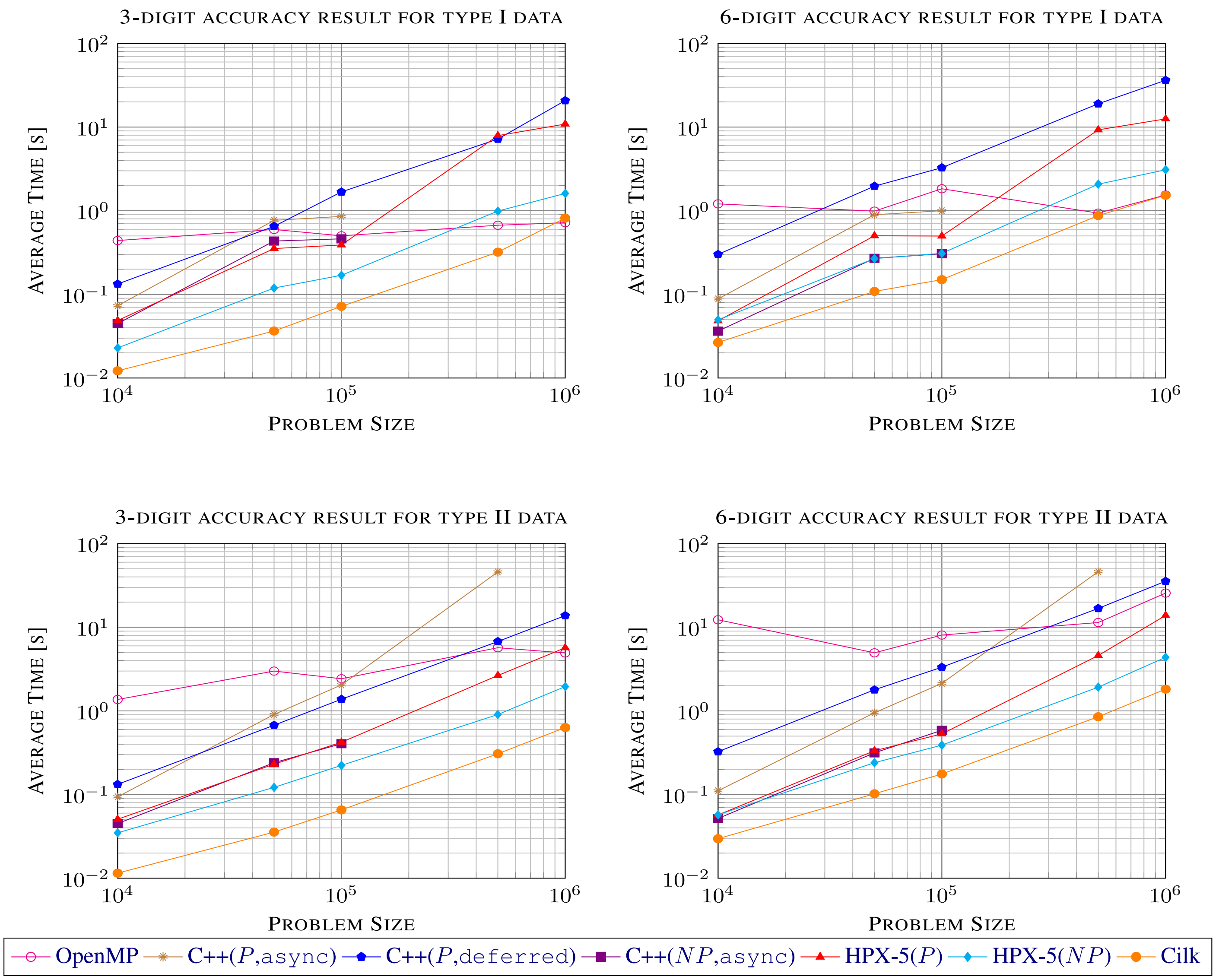

Fig. 5. The average execution time of various parallel FMM implementations for different problem size. Type I data is uniformly distributed inside a cubic box with side length 1 and centered at the origin. Type II data is distributed over the surface of a sphere of radius 1 and centered at the origin.

during the execution of the algorithm.

Following the discussion in Section IV, we consider eight different parallel FMM implementations in our tests, including one version using Cilk, one version using OpenMP task, two version using HPX-5, and four versions using C++11. The HPX-5 and C++11 versions are first differentiated by whether or not to create tasks for nodes having partial input to process. As this decision is equivalent to whether or not to partition G_{FMM}, we label the cases using P and NP, respectively. The C++ implementations are further categorized by the options used in the `async` function. In this way, the eight versions are referred to as Cilk, OpenMP, HPX-5(P), HPX-5(NP), C++(P, `async`), C++(P,`deferred`), C++(NP, `async`), and C++(NP, `deferred`). Figure 5 shows the performance of the eight implementations. All the cases were run ten times using all the sixteen cores and the average execution time is reported here.

We make a few comments on the missing data points for C++11-based implementations. The curve for the C++(NP, `deferred`) version is unavailable because the execution hangs even at the smallest problem size 10^4, implying that the entire structure of G_{FMM} is too complicated for the runtime to figure out. For the C++(NP, `async`) version, the missing data is because the number of threads to be created exceeds C++'s capability. For the C++(P, `async`) version, the missing data is because the program stalled for the given problem size.

Among all the C++11-based implementations, the versions using `async` option often run faster than the versions using `deferred` option. This shows the additional overhead associated with determining the first time the result of a deferred task is needed. However, such advantage can only be achieved at very small problem size before the `async` versions exceed C++'s thread creation limitation. The C++(P, `deferred`) version is the only one able to complete all the tests but its

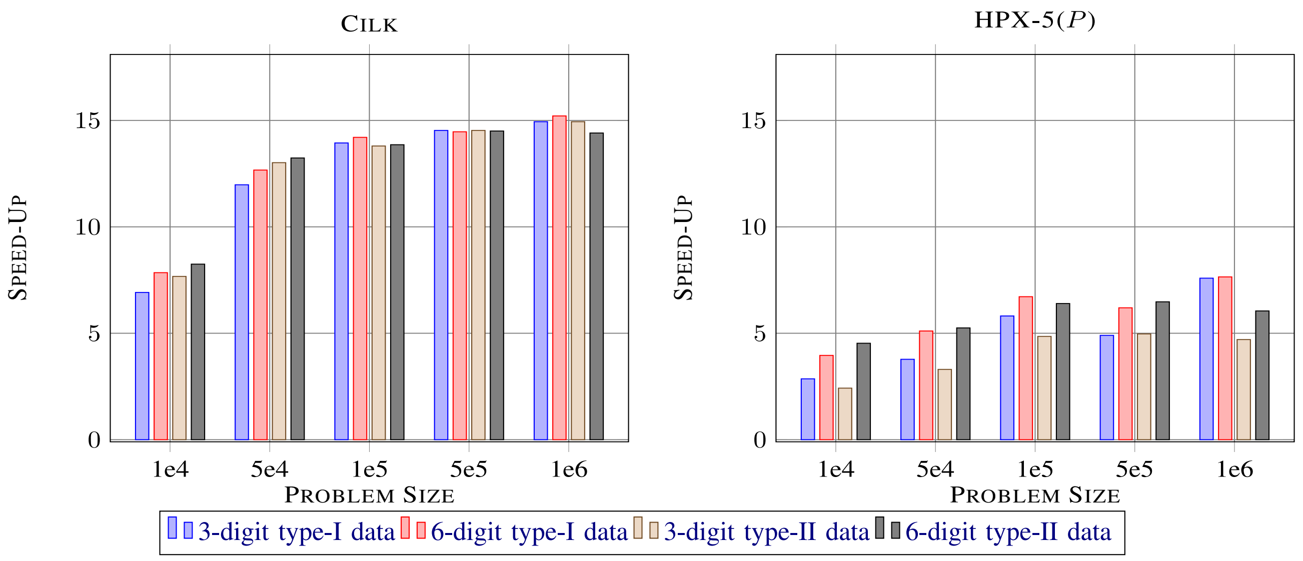

Fig. 6. Speed-up result of the Cilk and HPX-5(P) versions.

performance is the worst among all versions in most of the test cases. The curve for the C++(NP, `async`) version has a very similar trend as the one for the HPX-5(NP) version. For type I data, the HPX-5(NP) version runs faster than C++(NP, `async`) version for cases requiring three digits accuracy whereas the C++(NP, `async`) version performs better for cases requiring six digits accuracy. For type II data, the performance difference between the two versions is quite small.

For the OpenMP version, we tested multiple values of `OMP_MAX_ACTIVE_LEVELS` and the best execution times reported here were obtained by setting the value to two. The curves clearly reveal the task creation overhead. For type I data, the curves show no observable scaling trend, which suggests that there is insufficient computation to compensate the overhead. For type II data, we observe some scaling behavior when the problem size is over 10^5.

Excluding the C++(P, `deferred`) and OpenMP versions, the Cilk and HPX-5(P) versions that partitioned G_{FMM} have clear advantage over the versions not making this choice. To some extent, this reflects the limitation of the task scheduler's ability to recognize tasks along the critical path. The Cilk version is the clear winner among the eight versions in all the test cases, and is approximately 2X$\sim$3X faster than the second fastest version, the HPX-5(P) version. This performance difference clearly manifests the different overhead levels of the Cilk and HPX-5 runtime systems. There exist a few reasons contributing to the different overhead level between Cilk and HPX-5 runtime system. To begin, as a compiler-based runtime, Cilk is capable of performing deep static analysis for parallel execution, such as the fast-clone feature in Cilk [27]. Moreover, the runtime scheduler of Cilk creates a worker thread for each core present on a system and each worker maintains a deque for storing tasks yet to execute. When a worker thread executes a `cilk_spawn` or `cilk_for` statement, it simply pushes new tasks onto the tail of its deque and this does not involve any call into the OS nor involve the OS thread scheduler. In contrast, threads in HPX-5 have more states and need to be globally addressable and this inevitably increases the creation overhead. It is also important to point that that Cilk is an industry-level product and HPX-5 is still a research-level system at present.

Figure 6 shows the speed-up results for the Cilk and HPX-5(P) versions. Versions based on C++11 are not considered here because most of the versions could not complete in all the test cases and the C++(P,`deferred`) version is almost two orders of magnitude slower than the Cilk version. The OpenMP version is excluded because most test cases seem to be dominated by the overhead. The HPX-5(P) version is chosen because its performance is better than the HPX-5(NP) version. For smaller problem size, the speed-up result is close to 8X for the Cilk version and about 3X for the HPX-5(P) version. As the problem size grows larger, the speed-up gets close to 15X for the Cilk version and about 6X$\sim$7X for the HPX-5(P) version. Recall that we chose to use the same value of s in all the test cases and the speed-up results obtained here should be interpreted as a lower bound of the optimal speed-up for each individual version.

We point out that the problem sizes chosen to compare different runtimes were quite small compared to real FMM applications. The reason is because that C++'s implementation of standard thread library on Linux is done through the use of kernel thread. In comparison, Cilk and HPX-5 provide their own M:N libraries, and OpenMP provides API to control `MAX_ACTIVE_LEVELS`. It is worth mentioning that Cilk, HPX-5, and OpenMP versions can handle problem sizes up to 60 millions, which is the largest size one can fit in the testing machine. For larger problem size, the parallel efficiency of Cilk version remains above 90%; the HPX-5(P), HPX-5(NP), and OpenMP versions are roughly 3X, 10X, and 6X slower, respectively. We point out the OpenMP implementation considered here focus on testing the use of OpenMP tasks. One could develop more efficient OpenMP version. Nonetheless, this shows the promise of data-driven parallel FMM implementation provided the implementations uses the strengths of the runtime and the runtime itself has an appropriate level of overhead.

VI. Conclusion

In this paper, we discuss how to achieve a data-driven execution of the adaptive FMM by asynchronously scheduling available computational tasks. An important design decision to make is whether or not to create a new task for each node in G_{FMM} if the node has partial input to process. We choose four runtime systems—Cilk, C++11 standard thread and future libraries, HPX-5 library, and OpenMP tasks—as candidates to implement the parallel algorithm. By examining the features supported by each runtime, we developed eight different implementations of parallel FMM.

The performance evaluation of the eight versions provides answers to many questions quantitatively. As a compiler-based runtime, the Cilk version demonstrates the best performance in all the test cases, and is approximately 2X$\sim$3X faster than the next best version. The OpenMP version is dominated by overhead for small problem size. Versions using deferred task creation run slower most of the time. However, for C++11, eager task creation could quickly exceed the capacity of the runtime. The comparison between versions that partition G_{FMM} and the versions without partitioning G_{FMM} reveals the scheduler's difficulty with recognizing tasks along the critical path. Among all the versions, the Cilk version achieved 15X speed-up on the sixteen-core machine when the problem size is 10^6, without tuning the parameter s for different accuracy requirements and data distributions, showing the promise of the data-driven approach.

Although all the tests were performed on shared memory architectures, the results offer some measurements for the future development of the HPX-5 runtime that targets the distributed memory architecture. A few strategies that could improve its performance include preallocating a pool of threads and futures on startup instead of on demand, reducing the amount of work done during a context switch, using more efficient data structure for queuing and helping scheduler to recognize the critical path.

Acknowledgment

This work is supported by subcontract 172417 of the DOE grant DE-AC05-76RL01830 and contract PO 1208502 with Sandia National Laboratories. The author thanks Matthew Anderson and Jackson DeBuhr for their useful comments.

References

[1] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, pp. 325–348, 1987.

[2] J. Carrier, L. Greengard, and V. Rokhlin, "A fast adaptive multipole algorithm for particle simulations," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, pp. 669–686, 1988.

[3] B. Lu, X. Cheng, J. Huang, and J. McCammon, "Order N algorithm for computation of electrostatic Interactions in biomolecular systems," in *Proceedings of the National Academy of Sciences*, vol. 103, 2006, pp. 19 314–19 319.

[4] W. C. Chew, M. S. Tong, and B. Hu, *Integral Equations Methods for Electromagnetic and Elastic Waves*. Morgan & Claypool, 2008.

[5] L. Ying, G. Biros, and D. Zorin, "A Kernel-Independent Adaptive Fast Multipole Algorithm in Two and Three Dimensions," *Journal of Computational Physics*, vol. 196, pp. 591–626, 2004.

[6] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, pp. 720–748, 1999.

[7] S. Olivier, A. Porterfield, K. Wheeler, M. Spiegel, and J. Prins, "OpenMP task scheduling strategies for multicore NUMA systems," *International Journal of High Performance Computing Applications*, vol. 26, pp. 110–124, 2012.

[8] D. R. Lutz and D. N. Jayasimha, "What is an effective schedule?" in *SPDP*, 1991, pp. 158–161.

[9] G. E. Belloch, P. B. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *Journal of the ACM*, vol. 46, pp. 281–321, 1999.

[10] K. Wheeler, D. Stark, and R. Murphy, "A comparative critical analysis of modern task-parallel runtimes," Sandia National Laboratories, Tech. Rep., 2012.

[11] T. Gilmanov, M. Anderson, M. Brodowicz, and T. Sterling, "Application characteristics of many-tasking execution models," in *The 19th International Conference on Parallel and Distributed Processing Techniques and Applications*, 2013.

[12] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *Journal of Parallel and Distributed Computing*, vol. 37, pp. 55–69, 1996.

[13] "Information technology–Programming languages–C++," International Organization for Standardization, Standard, 2011.

[14] A. Tabbal, M. Anderson, M. Brodowicz, H. Kaiser, and T. L. Sterling, "Preliminary design examination of the ParalleX system from a software and hardware perspective," *SIGMETRICS Peformance Evaluation Review*, vol. 38, pp. 81–87, 2011.

[15] C. Dekate, M. .Anderson, M. Brodowicz, H. Kaiser, B. AdelsteinLelbach, and T. Sterling, "Improving the scalability of parallel N-body applications with an event-driven constraint-based execution model," *International Journal of High Performance Computing Applications*, vol. 26, pp. 319–332, 2012.

[16] T. Sterling, "Making slower faster," August 2013, presentation at the meeting of the ASCAC Subcommittee on Exascale Research Challenges.

[17] H. Cheng, L. Greengard, and V. Rokhlin, "A fast adaptive multipole algorithm in three dimensions," *Journal of Computational Physics*, vol. 155, pp. 468–498, 1999.

[18] 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.

[19] H. Sundar, R. S. Sampath, and G. Biros, "Bottm-up construction and 2:1 balance refinement of linear octrees in parallel," *SIAM Journal on Scientific Computing*, vol. 30, pp. 2675–2708, 2008.

[20] F. A. Cruz, M. G. Knepley, and L. A. Barba, "PetFMM–A dynamically load-balancing parallel fast multipole library," *International Journal for Numerical Methods in Engineering*, vol. 85, pp. 403–428, 2011.

[21] L. Greengard and W. Gropp, "A parallel version of the fast multipole method," *Computers & Mathematics with Applications*, vol. 20, pp. 63–71, 1990.

[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," *Journal of Parallel and Distributed Computing*, vol. 27, pp. 118–141, 1995.

[23] H. Ltaief and R. Yokota, "Data-driven execution of fast multipole methods," *CoRR*, vol. abs/1203.0889, 2012.

[24] A. Amer, N. Maruyama, M. Pericás, K. Taura, R. Yokota, and S. Matsuoka, "For-join and data-driven exeuction models on multi-core architectures: Case study of the FMM," *Lecture Notes in Computer Science*, vol. 7905, pp. 255–266, 2013.

[25] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based FMM for multicore architectures," *SIAM Journal on Scientific Computing*, vol. 36, pp. C66–C93, 2014.

[26] "Fast Multipole Methods," http://fastmultipole.org.

[27] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," *SIGPLAN Notices*, vol. 33, pp. 212–223, 1998.