論文 / 著書情報
Article / Book Information

Title	Tapas: An Implicitly Parallel ProgrammingFramework For Hierarchical N-body Algorithms
Author	Keisuke Fukuda, Motohiko Matsuda, Naoya Maruyama, Rio Yokota, Kenjiro Taura, Satoshi Matsuoka
Journal/Book name	The 22nd IEEE International Conference on Parallel And Distributed Systems, , , Page 1100-1109
Issue date	2016, 12
DOI	https://doi.org/10.1109/ICPADS.2016.0145
URL	http://www.ieee.org/index.html
Copyright	(c)2016 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other users, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works for resale or redistribution to servers or lists, or reuse of any copyrighted components of this work in other works.
Note	このファイルは著者（最終）版です。 This file is author (final) version.

Tapas: An Implicitly Parallel Programming Framework For Hierarchical N-body Algorithms

Keisuke Fukuda*†§, Motohiko Matsuda*, Naoya Maruyama*, Rio Yokota†, Kenjiro Taura‡ and Satoshi Matsuoka†
*RIKEN Advanced Institute for Computational Science, Hyogo, Japan
†Tokyo Institute of Technology, Tokyo, Japan
‡University of Tokyo, Tokyo, Japan
§Email:fukuda@matsulab.is.titech.ac.jp

***Abstract*—Tapas is our new C++ programming framework for hierarchical algorithms such as n-body, on large scale heterogeneous supercomputers. Although n-body and their variants are widely used in scientific applications, their correct implementations are often difficult on such modern machines, as the algorithms are irregular, complex, and involve explicit task parallel programming over distributed nodes. Encapsulating the complexities in a library or a framework has been challenging due to irregular data access over massively distributed memory. Tapas solves this by converting the users clean implicit-style parallel program into an inspector-executor style code on heterogeneous multi-core, multi-node environment solely by the use of C++ template metaprogramming. Prototype implementation of a Fast Multipole Method (FMM) on Tapas demonstrates 85% to 115% of the performance of ExaFMM, the fastest hand-tuned implementation of FMM to date, and scales to over 1,500 of CPU cores while efficiently utilizing hundreds of GPUs on heterogeneous machines such as TSUBAME2.5.**

I. Introduction

Exascale computing places significant research challenges in developing algorithms that can effectively exploit the capabilities of future computing systems [14]. One such algorithm that has the potential to extract the full potential of exascale systems is the Fast Multipole Method (FMM) [10, 11, 14].

Developing efficient implementations of such hierarchical algorithms for today's and future supercomputing systems, however, is a non-trivial and labor-intensive task. Although the algorithmic complexity lends itself to large-scale systems, efficiently exploiting the tree-based hierarchical structure of problem spaces as well as compute-intensive direct computations for near-field interactions on modern increasingly-complex systems requires thorough analyses and adaptations of implementations for each specific architecture. For example, the state-of-the-art technique to parallelize tree-based particle interactions on distributed memory systems is to construct Locally Essential Trees (LETs) that consist of sub-trees with overlapping nodes for resolving data dependencies among sub-trees. While the technique is well known and understood, its scalable implementation remains to be a challenging job for application scientists due to the inherent dynamic data dependencies that also vary among specific formulations of algorithms. The trend of diverging architectures towards exascale computing further exacerbates the problem since it is unlikely that a single implementation in conventional programming languages such as C and Fortran with prescriptive communications can perform equally well on different architectures.

There have been substantial efforts to ease the development of efficient scientific applications with high-level programming abstractions [6, 8, 12, 17, 18, 27, 34]. In particular, domain specific languages have been shown to be effective by greatly simplifying the task of performance critical programming such as efficient parallelization and architecture-specific optimization, allowing the programmer to focus on computational algorithms rather than implementation details [18, 27, 34]. However, since there has been little work that addresses the problem in the context of irregular algorithms such as FMM, most of the existing FMM implementations are developed from scratch with very limited use of software engineering discipline [2, 5, 26, 39], leading to poor productivity that is expected to be further problematic in the exascale era. There have been some efforts to couple more software engineering with FMM such as Charm++[22], X10 [28], StarPU [1], ParalleX [13, 41], OmpSs [30], MassiveThreads [35], QUARK [25], but the lack of performance comparisons of these codes against the hand-tuned state-of-the-art [2, 5, 26, 39] makes it difficult to assess the overhead of introducing such middleware into FMM codes. One of the very few exceptions is the work by Zandifar et al. [40], which we mention in Section VI.

We address the problem of application development productivity for a class of hierarchical N-body algorithms represented by, but not limited to, FMM. Specifically, we aim to realize a programming interface that allows separation of concerns between algorithmic and architectural issues such that a single implementation of a hierarchical N-body algorithm, once written, can efficiently run on different parallel architectures. To that end, we propose *Tapas*, a high-level implicitly-parallel programming framework for hierarchical N-body algorithms. Tapas provides programming constructs to define force interactions in a hierarchical and architecture-transparent manner, which are then automatically parallelized by the framework. Unlike regular computations such as stencils, data dependencies in our target problem domain are not statically determined, requiring dependence analyses at runtime. The key novelty in our framework is that the dependence problem is automatically resolved by a transparent inspector-executor method [33],

allowing automatic parallelization even for distributed memory environments.

This paper presents an implementation and evaluations of Tapas using C++ template meta programming. Despite implemented solely using standard C++ language features, the Tapas framework allows for automatic parallelization over distributed memory machines and shared-memory multi-core CPUs. Furthermore, compute-intensive kernels such as direct computations are automatically offloaded to CUDA-based GPU accelerators. The fact that it does not use any custom compilers or source-to-source translators except for standard C++ features has a practical advantage in the sense that it can be safely assumed to run on almost all major current and future platforms. To evaluate Tapas, we implement the same FMM algorithm as that of the ExaFMM library [39], one of the fastest known implementations of FMM to date, and compare their performances using a GPU-based heterogeneous supercomputer. We show that Tapas can achieve comparable performance as the hand-tuned FMM code and scale thousands of CPU cores while efficiently using GPUs. More specifically, our FMM implementation achieves 1.15x faster over ExaFMM in serial execution. It also shows good strong scalability up to 1536 cores. A GPU-offloading version shows good weak scalability with 150 and 300 GPUs and 2.04x to 5.77x speedup over the corresponding CPU execution of Tapas framework.

In summary, the contributions of this work include:

- We design a high-level framework for a class of hierarchical n-body algorithms such as FMM. Applications written with the framework are automatically parallelized by its inspector-executor-based automatic runtime dependence analysis.
- We develop a prototype implementation of the framework for heterogeneous GPU-based supercomputers. Experimental evaluations shows that our framework can achieve comparable performances as a hand-tuned FMM implementation using over 1,500 CPU cores as well as GPUs.

II. Hierarchical N-body Algorithms

A. Algorithm Overview

The N-body problem computes pairwise interactions for all pairs of bodies, resulting in $\mathcal{O}(N^2)$ calculations. As the computational complexity of the direct approach is often prohibitively high for realistic simulation problems, various approximation algorithms have been proposed such as static-cutoff-based method and the Particle-Mesh method. Hierarchical N-body algorithms, such as Barnes-Hut and the FMM, are another class of approximation algorithms, whose highly scalable performances have been successfully demonstrated on large-scale machines [4, 5, 21, 37].

In hierarchical algorithms, a multi-dimensional space is represented as a tree, T, where the root node, r, corresponds to the entire space, which is then disjointly decomposed into subspaces by its child nodes, $C(r)$. The decomposition is recursively applied until the leaf subspaces become small enough to contain up to k bodies, where k is an algorithmic parameter that balances the load of the near-field calculation with that of the far-field calculation. Contributions of bodies in the subspace of node i to *neighbor* bodies are calculated with pairwise direct methods, while contributions to *well-separated* bodies are approximated by using q_i, which is recursively computed from its child nodes. In the case of Barnes Hut and FMM, q_i corresponds to the multipole expansion. The exact condition when bodies are considered separated sufficiently far apart to approximate force interactions depends on the type of hierarchical algorithm. However, a common characteristic in these algorithms is that the farther the distance becomes, the approximation of the larger subspace is used, allowing significant reduction of the computational complexity. Furthermore, in algorithms such as FMM, the contribution of a subspace can also be computed against a far subspace rather than bodies, which is then propagated downward to its bodies.

The main focus of the paper is the FMM, which is one of the most complex and irregular one of such n-body algorithms. The FMM was originally developed to accelerate the $\mathcal{O}(N^2)$ direct N-body calculation to $\mathcal{O}(N)$ with algorithmically bounded errors[9, 16]. In its original form, the applicability of FMM is limited to problems that have a Green's function solution. This restriction was somewhat relaxed by the advent of kernel-independent [24] and black-box [15] versions of FMM, which do not require a Green's function but only an analytical expression of the kernel. Many of the original FMM researchers have now moved on to develop algebraic variants of FMM such as $\mathcal{H} - matrix$ [19], $\mathcal{H}^2 - matrix$ [20], hierarchically semi-separable (HSS) matrix [7], and hierarchically off-diagonal low-rank ($HODLR$) matrix [3]. These algebraic variants are becoming increasingly popular in fields that were previously dominated by dense linear algebra, since the complexity can be reduced from $\mathcal{O}(N^3)$ to $\mathcal{O}(Nlog^2N)$. Various scalable FMM codes have been developed and tested on large-scale platforms in Juelich [37], Oak Ridge [5], and Kobe [4, 21].

B. Implementation

Naturally, implementations of hierarchical N-body algorithms employ a tree data structure, where nodes represent subspaces and the parent-child relationship corresponds to the hierarchy of subspaces. Computations of q_i can be implemented with a post-order traversal of the tree, whereas the integration of forces with the hierarchical approximation can be expressed as a pre-order traversal. More specifically, as illustrated in Fig. 1, FMM can be implemented with a dual tree traversal that performs for a given pair of non-leaf nodes either approximating their interaction force or recursively visiting their children, depending on the distances between the bodies in the nodes [38]. Note that, while the force computation based on the tree traversal is not the only approach used in existing FMM implementations [24], the dual tree traversal allows for more flexible definition of well-separatedness, which can have a significant performance impact [38].

```
1: function INTERACT(i, j)
2:     if i and j are both leaf nodes then
3:         computes pairwise forces
4:     else if i and j are well separated then
5:         computes approximated forces
6:     else if j is leaf or i is larger than j then
7:         for all c ∈ C(i) do
8:             INTERACT(c, j)
9:         end for
10:    else
11:        for all c ∈ C(j) do
12:            INTERACT(i, c)
13:        end for
14:    end if
15: end function
```

Fig. 1. Tree traversal function in FMM. i and j denote tree nodes. The funcion is first called with the root node as the i and j arguments.

C. Distributed-Memory Parallelization

A common approach to parallelizing the above hierarchical N-body algorithms for multiple processes on distributed memory machines, divides the tree into subtrees with top-level nodes including the root being shared among them. Parallel processes exchange tree nodes that have interactions with remote processes, and augment the local subtrees with the remote data, which is called Locally Essential Trees (LETs) [36]. Once the LETs of the processes are constructed, they can independently compute the forces for the bodies in their subtrees.

Constructing LETs is not trivial as it is not a mere neighbor exchange. The data required for an LET depends on the distribution of the bodies that can vary over the course of simulation as well as the definition of the well-separatedness. Due to the former, a run-time analysis is required to resolve the data dependency. Furthermore, the run-time analysis is often implemented for a particular definition of the well-sepratedness with some parametric variations, making it difficult to reuse one implementation to different hierarchical algorithms.

D. Acceleration with GPUs

The N-body problem is one of the most suitable types of algorithms for accelerators such as GPUs thanks to its ample regular parallelism [29]. In hierarchical algorithms, however, the node-to-node approximation may not have sufficient parallelism to fully exploit throughput-optimized accelerators. Therefore, it is often the case that only certain computations with relatively high computation cost, such as the neighbor pairwise direct computation, are offloaded to accelerators [24] and the rest of computations as well as the data exchange for LET constructions are still done on the host CPU.

III. High-Level Framework for Hierarchical N-body Algorithms

While there have been extensive studies on developing efficient implementations of hierarchical n-body algorithms at scale, to the best of our knowledge, little attention has been paid to the cost of development of such implementations. Each study essentially develops its own implementation with almost no reuse of software components among existing implementations except for certain pieces of code that can be trivially encapsulated as library functions.

We aim to greatly simplify the development of high performance n-body applications by designing a versatile framework that allows the programmer to only focus on describing hierarchical n-body algorithms. Our framework, *Tapas*, provides the programmer with architecture-neutral programming interfaces such that a majority of implementation concerns for parallel systems are transparently managed by its generative programming capability. Furthermore, the framework runs on virtually any of existing and future systems as it is entirely implemented in the standard C++ programming language without relying on external tools for program introspection and transformation.

The rest of this section first discusses the guiding principles for the design of the framework, followed by the details of the framework API and illustrative examples.

A. Framework Design

Tapas is designed to allow the programmer to express a hierarchical n-body algorithm with a customizable definition of well-separatedness in a straightforward fashion. The basic programming constructs in Tapas consist of a standard tree data structure and its associated operations such as traversing parent-child edges. An array of bodies of a user-defined data type can be imported to a Tapas tree, which then builds a hierarchical representation of the bodies based on their positions in a multi-dimensional Euclid space. We define a small number of operations on trees so that tree traversals used in the hierarchical n-body algorithms including dual tree traversals can be implemented.

An important challenge in designing the framework is to achieve scalable parallel performance. Specifically, as discussed in Section IV-B, the state-of-the-practice to parallelize these algorithms on distributed memory systems is to construct LETs using MPI. While it would be interesting to employ advanced runtimes that support global address spaces such as Charm++˜citecharm, this paper focuses on automatically realizing the proven parallelization method. It is, however, not trivial in our framework as we do not assume any specific traversal patterns, precluding LET constructions as predefined library routines.

We note that the LET-based parallelization can be viewed as an inspector-executor method in the sense that constructing an LET inspects its local tree, followed by execution of force computations on the LET. Therefore, the problem of automatic LET construction can be viewed as realizing automatic inspector-executor from a given user traversal code.

To solve the problem, we derive a variant of a given code that interprets its traversal at run time using the C++ template metaprogramming capability. More specifically, we abstract the tree operations as template functors so that they can be changed arbitrarily without performance loss. Function pointers or virtual functions often disturb compiler's optimization such as static inline expansion. User traversal functors are also defined as template functions, which are instantiated twice at compile time: once for the inspection and another for the execution. In the inspection case, the user code performs the traversal as it is defined in the original code except for two modifications. First, any assignment to tree nodes is inactivated. Second, as remote trees are not accessible before an LET is constructed, we use a dummy tree that conservatively approximates each remote tree. The inspector then records the dependency to remote trees, which are resolved by an all-to-all data exchange. In the execution phase, since the LET for each local tree is already constructed, the original code can be executed as it is. We will describe more details on the automatic LET construction in Section IV. More technical details are described in Section IV-B

Similarly, to enable automatic parallelization over multi-core CPUs, we assume that no data dependency exists among traversals visiting different nodes. This implies that in a pre-order traversal of a single tree, for example, each child of a node can be visited in parallel without data races. The assumption restricts the expressiveness of the framework, however, to the best of our knowledge, it is commonly valid in our target problem domains.

B. API Overview

We realize Tapas as a C++ template framework providing basic programming constructs for tree-based hierarchical n-body implementations. We describe its main primitive data types, their associated operations for tree traversals, and the attribute mechanism.

The main data types of Tapas includes `Body` and `Cell`, which correspond to bodies and tree nodes, respectively. To import arrays of bodies defined in user code into a Tapas tree, `Body` is a user-defined type given to Tapas as a template parameter. Arbitrary types with n floating point or arbitrary plain old data type of fields can be used for target problems. Here the term "plain old data type" means C++ data types except pointers, references and classes with virtual functions. The byte offset of the coordinate fields also need to be given to Tapas as a template integer parameter. The byte offset is necessary because Tapas receives user's `Body` data types as a generic template parameter and does not inspect the internal structure of it. Tapas uses the byte offset to read body coordinates from user's data types.

`Cell` is a template type whose object represents a tree node and its corresponding sub region in the problem space. Once a `Body` array is imported into Tapas, a function to create a tree can be called, which returns its root node as a `Cell` object.

Type `Cell` defines several primitive operations commonly used in tree data structures such as `Center` and `IsLeaf`, which returns the center coordinate and a boolean value designating whether the cell is leaf, respectively.

```
1  // Pre-order tree traversal functor
2  struct TraversePreOrder {
3    template<typename Cell>
4    void operator()(Cell& p, Cell &c) {
5      // p is the parent, c is a child
6      ...; // do something using C
7      Map(TraversePreOrder(), C.SubCells());
8    }
9  };
10 // Post-order tree traversal functor
11 struct TraversePostOrder {
12   template<typename Cell>
13   void operator()(Cell &p, Cell &c) {
14     // p is the parent, c is a child
15     // To update parent, use Reduce() API
16     // Reduce values from all the children
17     // using reducing function 'sum'.
18     // The field name 'foo' is defined by the programmer.
19     Reduce(p. p.attr().foo, val, sum);
20
21     Map(TraversePostOrder(), c.SubCells());
22     ...; // do something using C;
23   }
24 };
25 void main() {
26   Cell root; // Tree root cell
27   // Start a pre-order traversal
28   Map(TraversePreOrder(), root);
29   // Start a post-order traversal
30   Map(TraversePostOrder(), root);
31 }
```

Fig. 2. Simplified example code snippets of pre-order and post-order traversals.

1) Tree Traversals: A tree traversal in Tapas is expressed as a user-defined *tree traversal function*, which is a C++ template functor accepting two `Cell` parameters, parent and child, to designate the tree nodes to visit. A call to `Map` with the function and the tree root `Cell` initiates a traversal of the tree. Recursive calls to `Map` in the user function with `Subcells` of the `Cell` parameter traverses down the tree. A tree traversal operation is eigher post-order (which is often refereed to as bottom-up or upward) or pre-order (top-down or downward) traversal.Fig. 2 illustrates examples of traversals.

Another key API function is `Reduce`. In post-order traversal, parallel store accesses from child cells to the parent cell must be coordinated to avoid write conflict. `Reduce` function controls the parallel interactions and keep parents data correct by reducing the written data from the children. Example usage of `Reduce` is also presented in Fig. 2.

Dual-tree traversals can be similarly implemented with functions `Map` and `Subcells`. User traversal functions for dual-tree traversals require two `Cell` parameters to specify the two trees to traverse. Function `Product` is provided to specify the product set of child nodes of both trees. An example of a dual tree traversal is illustrated in Fig. 3.

When a traversal reaches a leaf node, its bodies can be accessed with the `Bodies` function of the leaf node, which returns a `BodyIterator` object. Similar to tree traversal functions, a user-defined *body iterator function* can be defined

```
// Dual tree traversal functor
struct DualTreeTraversal {}
  template<typename Cell>
  inline void operator()(Cell C1, Cell C2) {
    ...; // do something using C1 and C2
    Map(DualTreeTraversal(),
        Product(C1.SubCells(), C2.SubCells()));
  }
};
void main() {
  Cell root; // Tree root cell
  // Start a dual tree traversal
  Map(DualTreeTraversal(), root, root);
}
```

Fig. 3. Simplified example code snippets of dual-tree traversal.

to visit each body. The `Map` function can be used to iterate through the bodies with the body traversal function.

2) Attributes: In order to represent computed force values as well as hierarchically computed approximations, Tapas allows both bodies and cells to be associated with *attributes* whose types are specified as template parameters. To explicitly prevent modifications of tree meta data during traversals such as corresponding coordinates, store accesses are only restricted to cell and body attributes during traversals. An attribute of each body can be accessed through the `Attr` function of the `Body` type. A cell attribute can be directly accessed with the `Attr` function of the `Cell` type.

Fig. 4 illustrates how the dual tree traversal in ExaFMM can be written in Tapas. Note that some detailes are omitted due to space limitations; a complete implementation can be found at https://git.io/vrsAz.

```
struct FMM_DTT {
  tmeplate<class Cell>
  inline void operator()(Cell C1, Cell C2,
                         float theta) {
    float dist2 = norm(C1.Center()-C2.Center());
    float R1 = 0; float R2 = 0;
    for (int d = 0; d < 3; ++d) {
      R1 = std::max(C1.Width(d), R1);
      R2 = std::max(C2.Width(d), R2);
    }
    R1 = (R1 / 2 * 1.00001f) / theta;
    R2 = (R2 / 2 * 1.00001f) / theta;
    if (dist2 > (R1 + R2) * (R1 + R2)) {
      M2L();
    } else if (C1.IsLeaf() && C2.IsLeaf()) {
      Map(P2P(), Product(C1.Bodies(),
                         C2.Bodies()));
    } else {
      SplitCell(C1, C2, R1, R2, theta);
    }
  }

  template<class Cell>
  inline void SplitCell(Cell C1, Cell C2,
                        float R1, float R2,
                        float theta) {
    if (C2.IsLeaf()) {
      Map(*this, Product(C1.Subcells(), C2), theta);
    } else if (C1.IsLeaf()) {
      Map(*this, Product(C1, Cj.Subcells()), theta);
    } else {
      Map(*this, Product(C1.Subcells(),
                         C2.Subcells()), theta);
    }
  }
};
```

Fig. 4. Simplified dual tree traversal in FMM.

C. Restrictions and limitations

There are several restrictions and limitations in programming on Tapas framework. Restrictions are from the programming models of Tapas. Certain kinds operations are inhibited or hidden from programmers to implement transparent paralleism and optimization under the hood by the framework. Restrictions on user's data types are already mentioned in Section III-B.

Limitations are mainly due to the current development status of the framework and it is our future work to remove them.

1) Restriction: side effect and thread safety: The user's code is expected to have no side effect except writing to the cells passed as function arguments. "Side effect" in this context includes I/O, writing/reading global variables, using random variables, and calling functions with such side effects. User's template functions or functor classes are transformed to inspector and executor using C++'s metaprogramming techniques and thus the code runs at least twice during a single execution. The output of the appliation will be thus inconsisted or unexpected if user's code has side effects.

2) Restriction: code for GPU: If the user wants to compiler the code for GPUs, a few more restrictions are introduced. Since the written code is directly passed to NVCC, operations that are not allowed in NVCC is not allowed in user's functions. More specificaly, calling external library functions is not allowed even if they don't have side effects.

3) Limitation: mutual interaction and parallelism: As shown in the evaluation section Section V, parallel efficiency on multithreaded execution is lower than manually written applications (ExaFMM in this specific case) due to Tapas' limitation on mutual interactions on multithreaded executions.

Mutual interaction is an optimization technique to reduce amount of computations. Particle simulations have two sets of bodies "target" and "source". Target bodies receive effect from source bodies and change their state in each timestep and source bodies don't. In many simulations target and source are identical set of bodies. In such cases interaction between particle A to B and B to A can be computed together and save computations. From our experience, mutual interaction can save computation time by up to 40% in ExaFMM.

Consider an interaction between cells C and D in three dimensional space. When both cells are split, 64 interactions happen between their children C_{1-8} and D_{1-8}. Mutual exclusion is necessary here to avoid store conflict and thus there is thoretically 8-way parallelism.

Users can write such application code in two ways: (A) one side split style, and (B) two-side split style. Fig. 5 shows the concept. Two-side style splits `C` and `D` together for a single

```
1  // (A) two-side split
2  // Split both of C and D at once
3  Map(Funct(), C.Subcells(), D.Subcells());
4
5  // (B-1) one-side split
6  Map(Funct(), C.Subcells(), D); // First split C
7
8  // (B-2) In the next level of recursion
9  // Split D (Cn is one of the C's children)
10 Map(Funct(), Cn, D.split());
```

Fig. 5. Simplified example of one-side split and two-side split

`Map` function call. One-side style splits C first, and splits D in the next recursion level.

Tapas cannot extract parallelism from code which is written in one-side split matter if mutual interaction optimization is activated. It is because both target cells and source cells are to be updated and all interaction must be serilized. In the specific case of Fig. 5, interactions in (B-1) `Map` must be serilized in terms of `D`, and (B-2) `Map` must be serialized on `Cn`. It is possible for the user to write his/her code always in two-side manner, but unnecessary splits increase the number of cell-to-cell interaction and possibly leads leads to longer execution time.

In ExaFMM, they solve the issue using a heuristic to split a pair of cells in the two-side way if there are more than X bodies (where X is 5000 by deafult) under their subtree. This heuristic increases parallelism while avoid unnecessary M2L interactions. This heuristics is, however, not availalbe in Tapas framework because Tapas does not expose number of bodies of non-leaf cells to programmers.

Also, the GPU version does not support mutual execution as of writing in the current version of Tapas.

These are our future work to remove this limitation.

D. Limitation: CPU SIMD operations

Acceleration by SIMD operations on CPUs is not supported as of writing.

E. Limitation: Advanced optimization

Some advanced optimizations are not implemented in Tapas. Those inlcude computation-communication overlapping, load balancing with weighted tree reconstruction, and advanced task scheduling.

IV. Implementation

This section describes implementation details on the tree construction, automatic parallelization, and GPU offloading.

A. Tree Construction

Although there are different types of trees to represent bodies in multi-dimensional spaces, one of the most common scheme is the octree in 3-D problems, where a region is evenly divided into eight sub regions or octants. Thus, while Tapas is intended to support all of the known major schemes, our current prototype only provides an octree-based tree construction. Specifically, function `Partition` can be used to import bodies and build its octree, which returns a `Cell` object corresponding to the tree root.

B. Distributed Memory Parallelization

As discussed above, one of the technical highlights of this paper is that we realize an automatic inspector-executor method to construct LETs for user-defined traversals by employing the C++ template metaprogramming capability. Specifically, in addition to the cell type representing a real tree node, Tapas internally has a *mock cell*, with which a traversal functor is also expanded to generate its inspector version. As presented in Fig. 2, Fig. 3, and Fig. 4, user functors take a template parameter `Cell`. The functors are instantiated with template arguments of mock cell and real cell to be an inspector and executor respectively. An inspector traversal does not modify any tree data including its attributes as its calls to `Attr` is statically overloaded to nullify assignments to the attributes. Instead, it records the cells visited during the traversal, which are then gathered by an MPI collective routine.

C. Multicore CPU Parallelization

Similar to the original ExaFMM, to parallelize tree traversals on shared-memory multi-core CPUs, we use MassiveThreads, a light-weight user-level threading library [35]. As is done in the original ExaFMM, we spawn new threads when traversing down trees with `Map` with `Subcells`. However, unlike the original version, the thread spawning is hidden in the `Map`template function. Note that in Tapas a single functor call can only modify the attributes of its parameter cells. Thus, it is legal to spawn multiple threads when traversing children of a tree node in a single tree traversal. In a dual tree traversal, however, as each child node is paired with all children of the other node, parallelizing the product set potentially results in data races. Similar to the original ExaFMM implementation, to avoid data races, we divide the product set into disjoint sub sets so that no single child appears in multiple sub sets, and parallelize each of the sub sets by spawning a new thread for each child pair.

D. GPU Offloading

As the Tapas framework is designed to expose data dependencies and parallelism, it is also possible to automatically exploit accelerators such as GPUs. Our current prototype implementation optionally supports offloading of body iterator functors such as pairwise direct force computations to GPUs. More specifically, when instructed, the framework wraps a user-given functor within a CUDA kernel function that is spawned with the total number of threads the same as the number of body pairs. As an optimization, we aggregate all calls to a body iterator within a traversal and dispatch all of them in once to reduce the overhead of kernel calls.

V. Evaluation

To evaluate the performance and scalability of Tapas framework, we have implemented FMM algorithm on top

of Tapas, which we refer to as TapasFMM. TapasFMM is based on ExaFMM. ExaFMM is one of the fastest FMM implementation[38]. We ported ExaFMM's computation kernels (P2M, M2M, M2L, P2P, L2L, L2P) and dual tree traversal code from the development branch onto Tapas. The code is modified to use Tapas' APIs, but the core algorithm and computations are identical to the originals. Other components of the application, such as space decomposition and tree construction, data management, distributed memory parallelism using MPI, shared memory parallelism using threads, are provided by Tapas framework.

A. Environment and configuration

We conduct all the experiments on TSUBAME2.5, which is a GPU-based supercomputer installed at Tokyo Institute of Technology. Each node has 54GB of memory, two 6-core Intel Xeon X5670 (12 cores in total) and three Tesla K20Xm GPUs. The compiler is 16.0.2 20160204 and MPICH2 version 3.1. Note that selection of C++ compiler is critical because Tapas heavily exploits static template metaprogramming techniques and compiler's inlining optimizations.

Since the current Tapas implementation does not support CPU SIMD acceleration, we use non-SIMD version of ExaFMM. ExaFMM supports SIMD operations in P2P interactions and the performance benefit is roughly 10% to 40% depending on `ncrit` parameter.

B. Serial performance

Fig. 6 shows serial performance evaluation of ExaFMM and TapasFMM. It is not trivial to fairly compare implementations of approximating algorithms because a degree of accuracy affects a number of computations. ExaFMM's main accuracy parameter is θ. It is a MAC (Multipole Acceptance Criteria) and typically between 0.3 and 0.5. Larger θ means lower accuracy and less computation and vice versa. We carefully select 0.34 and 0.345 for ExaFMM and TapasFMM, so both achieve the same level of accuracy. Errors compared to direct computations are $6.82e-06$ and $6.09e-6$ respectively. Note that such θ values differ between every single dataset.

Note that Since Tapas' inspector does not run and not included in the runtime of the evaluation since it is implemented for the necessity of data exchange between distributed processes.

C. Multithreaded Performance

Fig. 7 shows multicore scalability of the two implementations. We use the same number of bodies, θ and other configurations except number of threads and mutual interaction mode.

We evaluate the implementations with mutual interaction mode on and off because the current Tapas prototype has a limitation on mutual interaction and multicore parallel efficiency described in Section III-C3. On 12 threads, the performance of mutual and non-mutual Tapas are 53% and 58% of ExaFMM. As mentioned in Section III-C3, the performance degradation comes from a restriction of Tapas' programming model, and it's our future work.

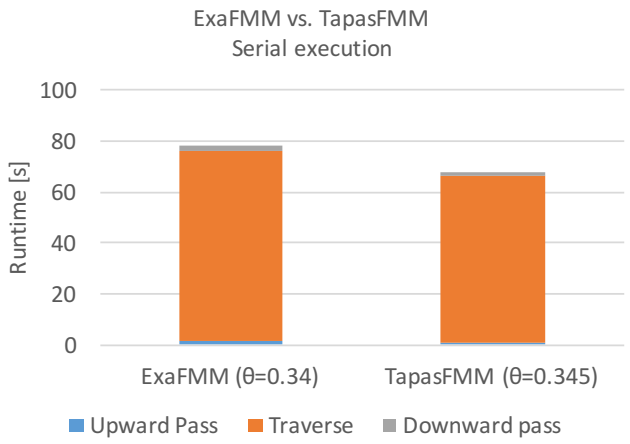

Fig. 6. Single process, single thread performance. 1,000,000 bodies of cube distribution, Ncrit=64, mutual interaction is activated.

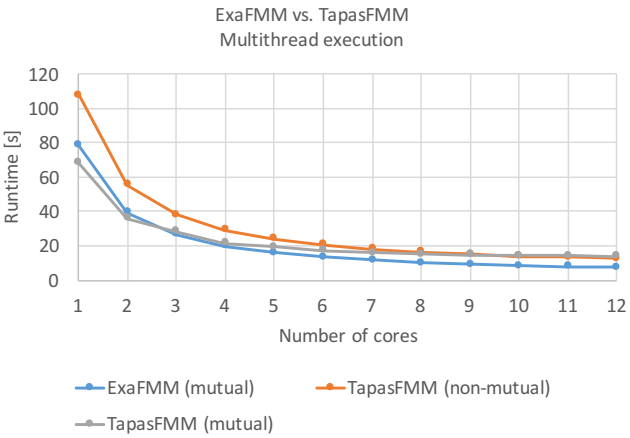

Fig. 7. Single process, single thread performance. 1,000,000 bodies of cube distribution, Ncrit=64, mutual interaction is activated for ExaFMM

D. Multinode Performance

Next, we show multiprocess scalability results in Fig. 8. The evaluation is strong scaling from 1 to 256 nodes of TSUBAME2.5, with one process per node and 12 threads per process.

We use summations of upward, traverse, downward times as total runtime, and exclude initial tree construction time. This is because cost of the first tree construction in a simulation depends on the physical data layout of bodies of input data. If bodies are generated with uniformly random positions and distributed over NP processes, $NP-1/NP$ portion of the bodies are exchanged on average between the processes in an all-to-all communication pattern, which is refereed to as particle shuffle. The cost would be much smaller if the bodies are loaded from a pre-sorted dataset. In addition, particle shuffling happends only in the first timestep of an execution and it is virtually negligible in multi-timestep production simulations.

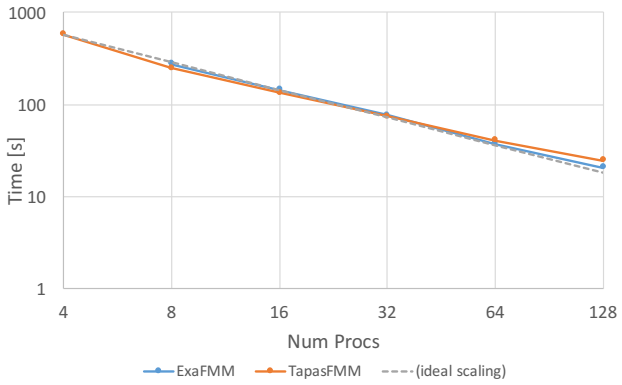

Fig. 8. Multi-process, multi-thread performance.
strong scaling of 310 million bodies of cube distribution,
12 threads/process Ncrit=64, mutual interaction is activated

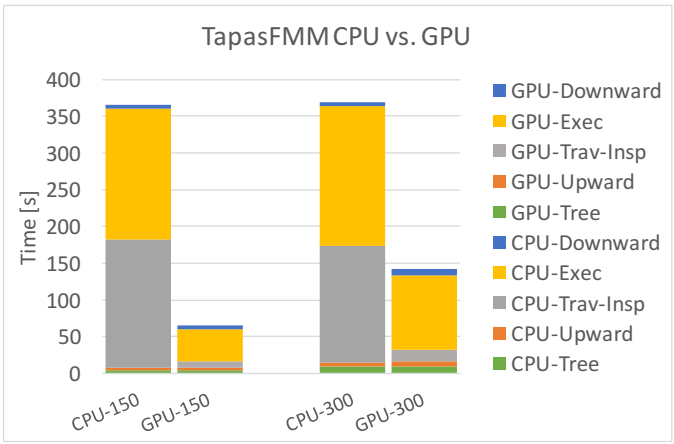

Fig. 9. Multi-process, multi-thread GPU performance of TapasFMM. Weak scaling of 150 and 300 processes with 1,000,000 bodies/process cube distribution,
Ncrit=1024/64, mutual interaction is NOT activated

Considering the fact implied by the previous multithreaded evaluation and Tapas' inspector overhead, Tapas' net traversal runtime is faster than ExaFMM in this configuration. However, we note that we use the same θ value 0.4 for both execution and the careful calibration of θ done in Fig. 6 is not done for this experiment. ExaFMM shows slightly better accuracy in this particular case. This is because it requires parameter sweep of θ and is not realistic for such a large scale dataset.

E. GPU Performance

Finally, GPU performance evaluation is shown in Fig. 9. The GPU version is compiled from the identical source code using NVIDIA's NVCC compiler. Unlike the previous experiments, We evaluate weak scalability of the GPU version because problem size of each process is limited by GPUs' device memory capacity. The current Tapas prototype does not implement techniques to overcome device memory capacity limitation.

Since TSUBAME2.5's computation nodes have three GPUs each, we run three processes per node, and each process uses four cores and one GPU. We use 50 and 100 nodes for the experiment, which run 150 and 300 processes respectively, with 1 million bodies per process.

The parameter `ncrit` of FMM is crucial for the performance. It controls the balance of direct and approximate computations. Since GPUs are highly optimized for massively parallel and regular computations, larger `ncrit`, which means larger amount of P2P direct computation and less M2L approximation, is suitable for GPUs. Tuning `ncrit` for performance is highly challenging and beyond the scope of this paper, we thus empirically choose `ncrit=1024`. The baseline CPU version's `ncrit` is 64.

The overall performance improvemnts over the baseline CPU version are from 5.77x and 2.04x. There are two key reasons for the improvement. First, P2P computation in DTT phase is accelerated by 1.88x to 4.46x by GPUs. Note that the acceleration ratio is not as significant as the ratio of hardware capacity (FLOP/s) because Tapas uses the identical source code for both CPU and GPU so the generated CUDA code is not optimal. In addition, we use a single θ value for both implementations and the GPU version achieves higher accuracy becuase P2P direct computation is dominant in the GPU veresion's configuration. Second, larger `ncrit` generates a smaller tree (with larger leaves), and that significantly contributes to reduce inspector overhead and communication cost.

VI. Related Work

Inspector/executor model is an old technique mainly used to improve CPU's cache locality, but some recent work use the technique to automate communication between distributed memory spaces in irregular applications. Ravishankar et al. [31, 32] presented a compiler-based code transformation to generate communication code from nested regular and irregular loops.

As mentioned in Section I, there has been a number of efforts to improve performance and productivity of hierarchical n-body applications using programming models and runtime systems [1, 13, 22, 23, 25, 30, 35, 41] or modern programming languages [28].

The most recent and related one is Zandifar et al [40]. To the best of our knowledge, it is the only existing work that directly compared a framework-based FMM and a manually-tuned state-of-the-art implementation. They built a framework for algorithmic skeletons using similar Map/Reduce concepts and ported ExaFMM onto their framework. They showed 25M bodies strong scaling over 256 cores, while we have demonstrated 310M bodies over 1536 cores. Our framework also supports GPU execution from the same source code. As they mentioned the snapshot of ExaFMM they used in the experiments seemed to have a serious issue not to be production-ready.

VII. Conclusion

We have proposed our new C++ programming framework for hierarchical algorithms on large scale heterogeneous supercomputers. The proposed framework automates shared-memory parallelism, distributed-memory parallelism, and GPU offloading from user's implicitly parallel application code. We have implemented a prototype of the framework and ported a manually tuned high-performance FMM application onto Tapas. It's performance is demonstrated on TSUBAME2.5 supercomputer. Our FMM implementation achieves 1.15x speedup over ExaFMM in serial execution. It also shows good strong scalability up to 1536 cores. A GPU-offloading version is built from the idential source code. It shows good weak scalability with 150 and 300 GPUs and 2.04x to 5.77x speedup over the corresponding CPU execution of Tapas framework.

References

[1] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based fmm for heterogenous architectures," INRIA, Tech. Rep. RR-8513, 2014.

[2] ——, "Task-based fmm for multicore architectures," *SIAM Journal on Scientific Computing*, vol. 36, no. 1, pp. C66–C93, 2014.

[3] S. Ambikasaran and E. Darve, "An O(NlogN) fast direct solver for partial hierarchically semi-seperable matrices," *Journal of Scientific Computing*, vol. 57, pp. 477–501, 2013.

[4] Y. Andoh, N. Yoshii, K. Fujimoto, K. Mizutani, H. Kojima, A. Yamada, S. Okazaki, K. Kawaguchi, H. Nagao, K. Iwahashi, F. Mizutani, K. Minami, S. Ichikawa, H. Komatsu, S. Ishizuki, Y. Takeda, and M. Fukushima, "MODYLAS: A highly parallelized general-purpose molecular dynamics simulation program for large-scale systems with long-range forces calculated by fast multipole method (FMM) and highly scalable fine-grained new parallel processing algorithms," *Journal of Chemical Theory and Computation*, vol. 9, pp. 3201–3209, 2012.

[5] J. Bédorf, E. Gaburov, M. S. Fujii, K. Nitadori, T. Ishiyama, and S. Portegies Zwart, "24.77 Pflops on a gravitational tree-code to simulate the milky way galaxy with 18600 GPUs," in *Proceedings of the 2014 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 1–12.

[6] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[7] S. Chandrasekaran, P. Dewilde, M. Gu, W. Lyons, and T. Pals, "A fast solver for HSS representations via sparse matrices," *SIAM Journal on Matrix Analysis and Applications*, vol. 29, no. 1, pp. 67–81, 2006.

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. ACM, 2005, pp. 519–538.

[9] H. Cheng, L. Greengard, and V. Rokhlin, "A fast adaptive multipole algorithm in three dimensions," *Journal of Computational Physics*, vol. 155, no. 2, pp. 468 – 498, 1999.

[10] B. Cipra, "The best of the 20th century: Editors name top 10 algorithms," *SIAM News*, vol. 3, no. 4, May 2000.

[11] Committee on the Mathematical Sciences in 2025; Board on Mathematical Sciences And Their Applications; Division on Engineering and Physical Sciences; National Research Council, *Fueling Innovation and Discovery: The Mathematical Sciences in the 21st Century*. The National Academies Press, 2012.

[12] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[13] C. Dekate, M. Anderson, M. Brodowicz, H. Kaiser, B. Adelstein-Lelbach, 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, no. 3, pp. 319–332, 2012.

[14] DOE ASCAC Subcommittee, "Top ten exascale research challenges," February 2014.

[15] W. Fong and E. Darve, "The black-box fast multipole method," *Journal of Computational Physics*, vol. 228, pp. 8712–8725, 2009.

[16] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *J. Comput. Phys.*, vol. 73, no. 2, pp. 325–348, Dec. 1987.

[17] K. Gregory and A. Miller, *Accelerated Massive Parallelism with Microsoft Visual C++*. Microsoft Press, September 2012.

[18] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess, "Stella: A domain-specific tool for structured grid methods in weather and climate models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. ACM, 2015, pp. 41:1–41:12.

[19] W. Hackbusch, "A sparse matrix arithmetic based on H-matrices, part I: Introduction to H-matrices," *Computing*, vol. 62, pp. 89–108, 1999.

[20] W. Hackbusch, B. Khoromskij, and S. A. Sauter, "On h^2-matrices," in *Lectures on Applied Mathematics*, H. Bungartz, R. Hoppe, and C. Zenger, Eds. Springer-Verlag, 2000.

[21] T. Ishiyama, K. Nitadori, and J. Makino, "4.45 Pflops astrophysical N-body simulation on K computer – The gravitational trillion-body problem," in *Proceedings of the 2012 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012.

[22] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn, "Scaling hierarchical N-body simulations on GPU clusters," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[23] M. C. Kurt and G. Agrawal, "Disc: A domain-interaction based programming model with support for heterogeneous execution," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 869–880. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.76

[24] I. Lashuk, A. Chandramowlishwaran, H. Langston, T. A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, Nov 2009, pp. 1–12.

[25] H. Ltaief and R. Yokota, "Data-driven execution of fast multipole methods," *arXiv:1203.0889v1*, 2012.

[26] D. Malhotra and G. Biros, "PVFMM: A parallel kernel independent FMM for particle and volume potentials," *Communications in Computational Physics*, vol. 18, no. 3, pp. 808–830, 2015.

[27] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. ACM, 2011, pp. 11:1–11:12.

[28] J. Milthorpe, "X10 for high-performance scientific computing," Ph.D. dissertation, The Australian National University, 2015.

[29] L. Nyland, M. Harris, and J. Prins, "Fast n-body simulation with cuda," *GPU gems*, vol. 3, no. 1, pp. 677–696, 2007.

[30] M. Pericàs, A. Amer, K. Fukuda, N. Maruyama, R. Yokota, and S. Matsuoka, "Towards a dataflow FMM using the OmpSs programming model," IPSJ SIG, Tech. Rep., 2012.

[31] M. Ravishankar, R. Dathathri, V. Elango, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Distributed memory code generation for mixed irregular/regular computations," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 65–75. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688515

[32] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Code generation for parallel execution of a class of irregular loops on distributed memory systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 72:1–72:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389094

[33] J. H. Saltz, R. Mirchandaney, and K. Crowley, "Run-time parallelization and scheduling of loops," *IEEE Transactions on Computers*, vol. 40, no. 5, pp. 603–612, May 1991.

[34] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, pp. 134:1–134:25, Apr. 2014.

[35] K. Taura, J. Nakashima, R. Yokota, and N. Maruyama, "A task parallel implementation of fast multipole methods," in *SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012, pp. 617–625.

[36] M. S. Warren and J. K. Salmon, "A parallel hashed Oct-Tree n-body algorithm," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '93. ACM, 1993, pp. 12–21. [Online]. Available: http://dx.doi.org/10.1145/169627.169640

[37] M. Winkel, R. Speck, H. Hubner, L. Arnold, R. Krause, and P. Gibbon, "A massively parallel, multi-disciplinary Barnes-Hut tree code for extreme-scale N-body simulations," *Computer Physics Communications*, vol. 183, pp. 880–889, 2012.

[38] R. Yokota, "An fmm based on dual tree traversal for many-core architectures," *Journal of Algorithms & Computational Technology*, vol. 7, no. 3, pp. 301–324, 2013.

[39] R. Yokota and L. A. Barba, "A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems," *International Journal of High Performance Computing Applications*, vol. 26, no. 4, pp. 337–346, 2012.

[40] M. Zandifar, M. Abdul Jabbar, A. Majidi, D. Keyes, N. M. Amato, and L. Rauchwerger, "Composing algorithmic skeletons to express high-performance scientific applications," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 415–424. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751241

[41] 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*, 2014.