Commun. Comput. Phys.
doi: 10.4208/cicp.230216.140416sw

Vol. **20**, No. 2, pp. 534-550
August 2016

COMPUTATIONAL SOFTWARE

RECFMM: Recursive Parallelization of the Adaptive Fast Multipole Method for Coulomb and Screened Coulomb Interactions

Bo Zhang[1,*], Jingfang Huang[2], Nikos P. Pitsianis[3,4] and Xiaobai Sun[4]

[1] *Center for Research in Extreme Scale Technologies, Indiana University, Bloomington, IN, 47404, USA.*
[2] *Department of Mathematics, University of North Carolina at Chapel Hill, Chapel Hill, NC, 27599, USA.*
[3] *Department of Electrical and Computer Engineering, Aristotle University of Thessaloniki, GR-54124, Greece.*
[4] *Department of Computer Science, Duke University, Durham, NC, 27708, USA.*

Received 23 February 2016; Accepted (in revised version) 14 April 2016

Abstract. We present RECFMM, a program representation and implementation of a recursive scheme for parallelizing the adaptive fast multipole method (FMM) on shared-memory computers. It achieves remarkable high performance while maintaining mathematical clarity and flexibility. The parallelization scheme signifies the recursion feature that is intrinsic to the FMM but was not well exploited. The program modules of RECFMM constitute a map between numerical computation components and advanced architecture mechanisms. The mathematical structure is preserved and exploited, not obscured nor compromised, by parallel rendition of the recursion scheme. Modern software system—CILK in particular, which provides graph-theoretic optimal scheduling in adaptation to the dynamics in parallel execution—is employed. RECFMM supports multiple algorithm variants that mark the major advances with low-frequency interaction kernels, and includes the asymmetrical version where the source particle ensemble is not necessarily the same as the target particle ensemble. We demonstrate parallel performance with Coulomb and screened Coulomb interactions.

AMS subject classifications: 15A06, 31C20

Key words: Fast multipole method, recursive parallelization, dynamic scheduling, Coulomb interaction, screened Coulomb interaction.

*Corresponding author. *Email addresses:* zhang416@indiana.edu (B. Zhang), huang@amath.unc.edu (J. Huang), nikos@cs.duke.edu (N. P. Pitsianis), xiaobai@cs.duke.edu (X. Sun)

Program summary

Program title: RECFMM

Nature of problem: Rapid evaluation of interactions among charged particles in 3D space, in terms of potential and force field, governed by low-frequency kernels such as for the Coulomb and screened Coulomb interactions.

Software licence: GPL 2.0

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

Distribution format: .gz

Programming language(s): C with CILK and Fortran

Computer platform: Any quipped with that described in the next three items.

Operating system: Linux, Cygwin, Mac OS X

Compilers: Intel compilers `icc` and `ifort` (15.0.4 or newer); GNU compilers (5.3.0 or newer); LLVM compiler `Cilk Plus` (3.4.1 or newer).

RAM: 32KB of storage per 1000 particles; 1.5KB (5.6KB) for each multipole or local expansion of 3-digit (6-digit) accuracy; 0.9KB (4.5KB) for each exponential expansion of 3-digit (6-digit) accuracy.

External routines/libraries: Fortran subroutines for accurate evaluation of Gamma and Bessel functions used for computing the screened Coulomb interaction.

Running time: Problem-size and computer-resource dependant.

Restrictions: 3-digit or 6-digit accuracy; low-frequency interaction kernels.

Supplementary material and references:

Additional Comments: The subroutines for multipole expansions and translated expansions can be customized to different accuracy, or replaced according to a different interaction kernel.

1 Introduction

We present RECFMM, a program representation and implementation of a recursive scheme for parallelizing the adaptive fast multipole method (FMM) on shared-memory computers. RECFMM expedites the computation of the potential ϕ and force field $\nabla\phi$ at M target locations $\{\mathbf{t}_i | i = 1,2,\cdots,M\} \subset \Omega_t$ due to charges q_j of N source particles located at $\mathbf{s}_j$, $\{\mathbf{s}_j, | j = 1,2,\cdots,N\} \subset \Omega_s$,

$$\phi_i = \sum_{j=1}^{N} K(\|\mathbf{t}_i - \mathbf{s}_j\|_2) q_j, \qquad \nabla\phi_i = \sum_{j=1}^{N} \nabla K(\|\mathbf{t}_i - \mathbf{s}_j\|_2) q_j, \tag{1.1}$$

where the target domain Ω_t and source domain Ω_s are in the 3D space $\mathbb{R}^3$, the kernel function $K(t,s)$ describes the potential at location t due to a unit charge at location s. In

the current version, RECFMM assumes and exploits the translation-invariant property. The property is possessed in particular by the Coulomb interactions [3] governed by the Laplace equation,

$$\text{Laplace Kernel:} \qquad K(\|\mathbf{t}-\mathbf{s}\|_2)=\frac{1}{\|\mathbf{t}-\mathbf{s}\|_2}, \tag{1.2}$$

and the screened Coulomb interactions by the Yukawa equation [5] with damping parameter $\lambda>0$,

$$\text{Yukawa Kernel:} \qquad K(\|\mathbf{t}-\mathbf{s}\|_2)=\frac{e^{-\lambda\|\mathbf{t}-\mathbf{s}\|_2}}{\|\mathbf{t}-\mathbf{s}\|_2}. \tag{1.3}$$

RECFMM achieves remarkable high performance using a recursive parallelization scheme implemented via CILK. In numerical computation, RECFMM is based on the FMM that is adaptive to the sparse or clustering pattern in the particle ensemble and has arithmetic complexity linear with the ensemble size [2, 3, 5, 6]. In combinatorial computation for execution control, RECFMM utilizes modern shared-memory computers equipped with software system CILK [1, 4] that provides both compilation and runtime optimization in parallel scheduling. The recursive parallelization scheme signifies a feature that is intrinsic to the FMM but had not been well exploited.

The rest of the paper is organized as follows. In Section 2, we review the adaptive FMM, as well as its main variants, from the combinatorial aspect. We extend the adaptation in spatial partition to particle sparsity to the case where the target particle ensemble is not necessarily the same as the source one. Such case arises in many applications. We introduce in Section 3 our recursive parallelization scheme. We provide an installation guide in Section 4, and demonstrate parallel efficiencies with certain test data for Coulomb and screened Coulomb interactions.

2 The dependency-concurrency graph

In this section, we review the combinatorial structure of the FMM, leaving its intrinsic connection with the numerical approximation to the references in [2, 3, 5–7] and the references therein. In the review, we make a distinction between the case where the source ensemble and target ensemble are one and the same, and the case they are not. The latter arises in many computational applications [12], such case results in a non-symmetric interaction matrix. It is beneficial to exploit the sparsity in each of the ensembles as well as the separation between the interacting ensembles.

2.1 Adaptive trees

FMM first makes a hierarchical partition of the particle domain into nested cubical boxes. Assume temporarily that the source particle ensemble and the target ensemble are one and the same. The root box is the smallest bounding box that contains all the particles. Without loss of generality, the root box is normalized to 1 in each side. The root box is

partitioned along each dimension in the middle, resulting in 2^d child boxes, $d=3$. An empty child box is pruned off. A child box is a leaf box if it contains no more than s particles, where s is a preset positive integer. Any child box with more than s particles is further partitioned in the same fashion, and recursively. The data structure for the hierarchical spatial partition is a tree. A tree node at level ℓ corresponds to a non-empty box at the ℓ level of the recursive partition. The box size at level ℓ is $1/2^\ell$. The tree edges represent the parent-child relationship. By the above criteria for branching, pruning and termination, the spatial partition is *adaptive* to the sparse or clustering patterns of the particles. The tree is not necessarily balanced. When the particles are distributed uniformly in the root box, the partition tree is likely well balanced up to many partition levels. For such uniform case, the predetermined threshold s is often replaced by a predetermined leaf level, L, i.e., the maximal level of recursive partition.

In general, the source domain Ω_s and the target domain Ω_t are not necessarily the same. They may overlap, partially overlap, or be far apart, in terms of the partitioned subdomains. To accommodate all the cases, we partition both the source domain and the target domain to obtain a source tree and a target tree. The two trees share the same root box, which is the smallest box containing all source and target particles. Each tree is adaptive to the geometry and sparsity of the corresponding particle ensemble. When the target particle ensemble is the same as the source one, the two trees become one and the same.

2.2 Interaction edges between the trees

In essence, the FMM enables rapid calculation of particle-particle interactions by accurately approximating them with box-box interactions at multiple spatial scales. By a box-box interaction, we mean the interaction between the sub-ensemble of particles residing in a source box and that in a target box. The box-box interactions are described by links or edges between the source box nodes and the target box nodes. In the FMM parlance, the box-box interaction patterns are described by the *link lists*. For the adaptive FMM, one may categorize the interactions at all scales into four types, based on the near-neighbor and far-neighbor relationships. Let B_t be a target box at level ℓ. At the same level ℓ, the *near-neighbors* of B_t are the source boxes *adjacent* to B_t, sharing a common face, edge or corner. They are also referred to simply as the *colleagues* of B_t. The source boxes that are not near-neighbors of B_t are seen as *well separated* from B_t. We are now in a position to describe the source lists according to their relation to, and influence upon, a target box B_t. Each list is a collection of source nodes/boxes designated to interact with the target box with a numerically common interaction pattern.

- $L_1(B_t)$ for a leaf box B_t consists of all leaf boxes that are adjacent to B_t. $L_1(B_t)$ for a non-leaf box B_t is empty. The interactions between particles in B_t and $L_1(B_t)$ are computed directly.

- $L_2(B_t)$ is the set containing the children of the colleagues of B_t's parent but not

including the colleagues of B_t. The multipole expansion of each box in $L_2(B_t)$ is converted into a local expansion about the center of B_t.

- $L_3(B_t)$ for a leaf box B_t contains the boxes that are not adjacent to B_t but whose parents are adjacent to B_t. $L_3(B_t)$ for a non-leaf box B_t is empty. The multipole expansion of each box in $L_3(B_t)$ is evaluated at each particle location inside B_t.

- $L_4(B_t)$ for a box B_t contains the leaf boxes that are not adjacent to B_t but are adjacent to B_t's parent. The field of each particle in $L_4(B_t)$ is converted into a local expansion about B_t's center.

$L_2(B_t)$ is the basic. For the uniform partition, only $L_2(B_t)$ lists remain, and they can be encoded in more compressive expressions.

A couple of remarks. In theory, symmetrical to the target-centric link lists, one can make the link lists source centric, i.e., make the lists contain interacting target boxes with respect to a source box. In computational execution, the two versions are not symmetrical. The source-centric link option causes conflicts in memory writes and consequential time delay in resolving the conflicts. The target-centric version incurs no such conflict and latency. Each target reads from multiple sources and writes at one exclusive target location. In matrix computation language, the target-centric version amounts to row-wise expression of matrix compression, because each row corresponds to the interaction of one target with multiple sources.

We have made a minor change in the definition for $L_4(B_t)$. Traditionally, $L_4(B_t)$ is defined as the dual list of $L_3(B_t)$ [2]. That is, if box C belongs to $L_3(B)$, then box B belongs to $L_4(C)$. The correctness of this relationship assumes that the source and target particle ensembles are one and the same. In parallel execution, it is possible that two (or more) boxes B_1 and B_2 both have box C in their L_3, and may update $L_4(C)$ concurrently, causing a data race condition. The modified definition not only permits non-coincidental particle ensembles but also eliminates this data race condition, which will be seen clearly in Algorithm 1, by clarifying the interaction direction from sources to targets. Fig. 1 is a 2D demonstration of the four types of link/interaction lists.

2.3 The graph flow and variants

The FMM graph G_{FMM} is composed of the trees and the interaction edges between the trees. The graph is directional and acyclic: all the edges are oriented from the source locations to the target locations. More specifically, the source tree edges are upward, from children to their parents, indicating M2M operation, the translation and aggregation of multipole expansion coefficients. The target tree edges are downward, from parents to their children, indicating the L2L operation, the translation and disaggregation of local expansion coefficients. The across-tree edges are from the source nodes to target nodes, denoting the M2L operation, the translation of multipole coefficients to local coefficients. Fig. 2 depicts a portion of a 2D G_{FMM} for noncoincident source and target ensembles.

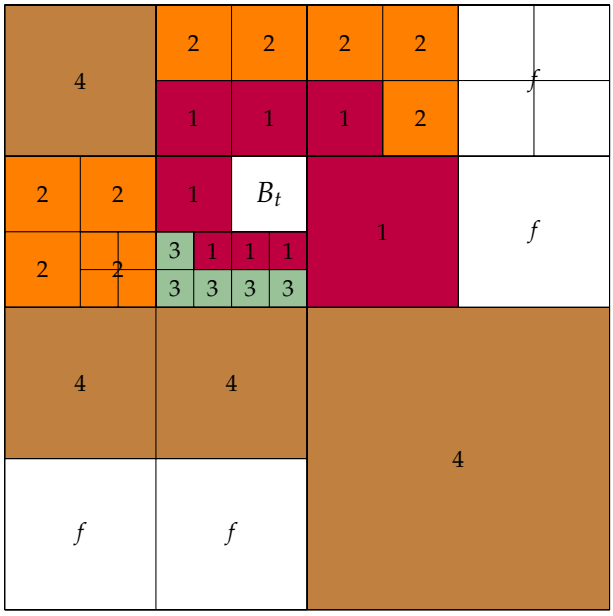

Figure 1: 2D demonstration of four types of link lists $L_i(B_t)$, $i=1,\cdots,4$ with respect to a target box B_t. Boxes with label f are well-separated from B_t's parent and their influences are passed to B_t through the lineage of the target tree.

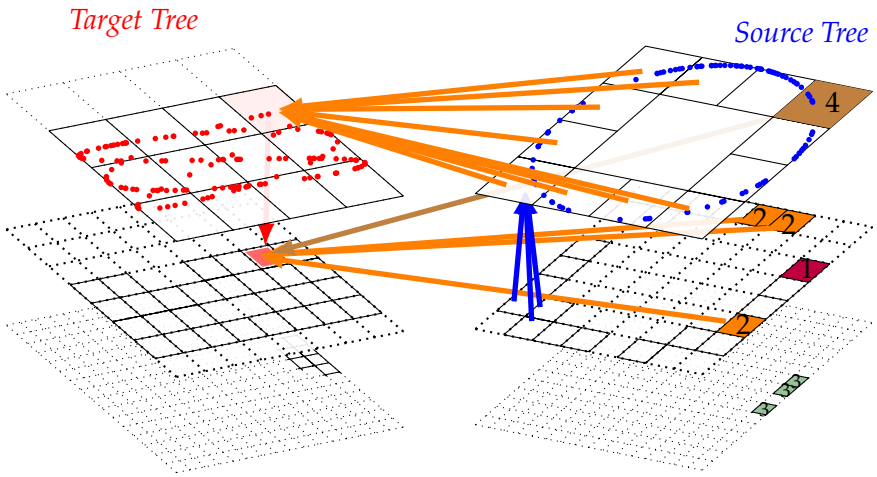

Figure 2: Illustration of a portion of the 2D graph G_{FMM}: Source particles and target particles are marked in blue and red colors, respectively. The graph nodes are the tree nodes, which are the partitioned target or source boxes with solid line boundary. All edges are oriented from the source to the target. In particular, the tree edges are of inter-scale, go upward in the source tree, via numerical computation for multipole expansion translation M2M, and downward in the target tree, via local expansion translation L2L. The inter-tree edges represent the box-box interactions, categorized in four-types.

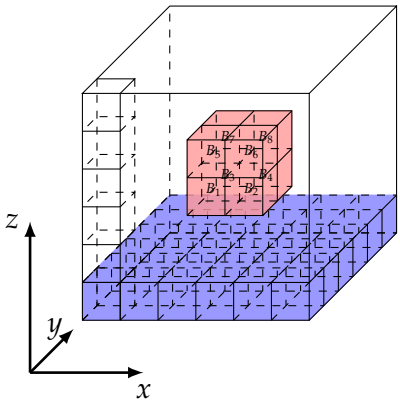

Figure 3: Illustration of the "merge-and-shift" technique for more efficient M2L operations.

The basic graph undergoes some changes with algorithmic variants. We encompass in RECFMM the major algorithm variants for interactions with a low frequency kernel. In more detail, the M2M operation (within the source tree) and the L2L operation (within the target tree) employ the "point-and-shoot" transformation technique, where one rotates the local coordinate frame so that the expansion can be efficiently shifted along the rotated z-axis [8]. The M2L operations from the source tree to the target tree use the "merge-and-shift" technique to further reduce the arithmetic complexity. The idea is to extract common components among the interacting boxes. When M_1 target boxes share M_2 source boxes on their interaction lists, instead of doing $M_1 \times M_2$ M2L translations, one can first merge the M_2 multipole expansions into one and then shift it to M_1 boxes using only $M_1 + M_2$ translations, see Fig. 3. The efficient implementation is achieved by using *exponential* expansions, where operations are performed component-wise. In terms of graph transformation, the merge-and-shift technique makes a two-layer interaction network out of the direct, one-layer interaction network from the source tree to the target one and effectively reduces the total number of interaction edges. This graph transformation is exploited by RECFMM and proven beneficial, contrary to the previous belief that this version of the FMM was not well suited for parallel implementation.

2.4 Coupled dual partitions for adaptive compression

In the description of the FMM graph, we have extended the spatial partition and interaction patterns to include all the cases where the source and target ensembles, Ω_s and Ω_t, are wholly coincident, partially overlapping, or far apart. We generate two trees, the source and target trees, sharing the same root box, with each adaptive to the sparsity in the respective particle ensemble. We elaborate in this section on how to explore the separation between the two ensembles as well.

At a first glance, it seems sufficient to consider only the coincident case $\Omega_s = \Omega_t$, which corresponds to a symmetric interaction matrix. The non-coincident case corre-

sponds to an asymmetric interaction matrix, which can be seen as a sub-matrix of the embedding symmetric matrix over the joint ensemble $\Omega_s \cup \Omega_t$. In computation, however, the sub-matrix is not readily available from the embedding symmetric matrix, due to the compression in terms of low-rank sub-matrices and the sparse structure encoded by the link/interaction lists. One simple solution is to place zero charges at non-overlapping target particles. This approach not only incurs extra cost, but also occludes potential opportunity to compress the interaction matrix even more, and reduce computation cost, without compromising numerical accuracy.

We express in RECFMM the non-symmetric matrix directly and efficiently, exploiting the sparsity in each of the particle ensembles as well as the separation between them. The latter is drawn upon the following two facts, by the approximation principles of the FMM. Consider a pair of source and target boxes (B_s, B_t).

- When B_s is well separated from B_t, no further partition is necessary, regardless of the particle population in each of the boxes. The sub-matrix of $|B_t|$ rows and $|B_s|$ columns corresponding to the box-box interaction is of rank-p, where p is the length of the multipole expansion.
- When one of the boxes has fewer than s particles, it is not necessary to partition the other box further. The rank for the sub-matrix is bounded by s.

The threshold parameter s is in fact determined to be proportional to the expansion length. The two facts suggest possibly earlier termination of the spatial partition. Specifically, we have established the following rank-based criterion.

- A target box B that contains more than s particles is not partitioned further if none of its source colleagues contains more than s particles.

The same can be applied to the source box partition. This criterion for terminating the partition couples the dual spatial partitions. The coupling can be carried out level-wise in sequential computation. Such a coupling scheme has a drawback in parallel execution. It introduces synchronization from one level to the next in the construction of the tree and graph structures, resulting to decreased degree in run-time concurrency. Instead, We use a two-pass strategy. In the first pass, the source and target trees with a common root (box) are constructed in parallel, independently, following the basic partition criteria in Section 2.1. In the second pass, we apply the above rank-based criterion to further prune the target tree, with massive, asynchronized parallelism.

3 Recursive parallelization for adaptive scheduling

3.1 The parallelization objective and previous limitation

In parallel implementation of an algorithm, the programmer embeds, intentionally or unknowingly, an algorithm parallelization model, a programming model and an execution

model, explicitly or implicitly. We have developed a recursion model that allows us to express the major algorithm variants in Section 2.3 and utilize advanced computer systems in software and hardware. The parallelization scheme signifies the recursion feature that is intrinsic to the FMM but was not well exploited, primarily, due to the lack of support in recursive programming and dynamic parallel scheduling. The major obstacles have been removed by recent advances in parallel programming systems. In particular, we employ the software system CILK for its syntax and semantics to capture the essence of the algorithm model as well as for its parallel execution model that is graph-theoretic optimal in theory and proved highly competitive in practice.

Parallel computation by the FMM, like its sequential counterpart, must follow the dependency graph G_{FMM}, namely, traverse the graph from the leaf source nodes to the leaf target nodes, along the directed edges. Unlike a sequential traversal, the time of which depends on the total number of graph edges, a parallel traversal follows as many trails simultaneously as possible. The concurrency is in the graph. For example, operations at the leaf source nodes have no incoming edges, therefore multipole expansions can start at the leaf nodes simultaneously. A target node can start M2L operations as soon as the data from the source boxes on its link/interaction lists become available. For simplicity, we do not split M2M, M2L and L2L operations. The trail length in latency, however, not only depends on the operations along the trail, but also on the execution dynamics. The objective of parallel scheduling is to minimize the traversal time, or keep all the workers busy.

We have seen two popular schemes for parallel scheduling used in existing parallel FMM on shared-memory computers. One is by pre-scheduling, based on an offline optimization, according to an idealized or simplified execution model. Such scheme is limited to certain assumptions on the distributions of particles, tree nodes and inter-tree edges, because the information of in-degrees and out-degrees across the nodes, in general, is unavailable until the partition and the link-lists are made, i.e., when the graph is constructed. Another common parallelization scheme is data driven. In a thread-programming environment on shared-memory computers, the working threads activate/spawn descendant threads when the data to the descendant threads become available. An implementation of such scheme with programmer-directed thread management is far from data driven. Threads conceived and scripted offline explicitly by the programmer, lack coordination and adaptation to the dynamics in parallel execution. In short, the previous approaches let the programmer to orchestrate parallel scheduling with little facilitation in adaptation at the system level to the dynamics in parallel execution. There are many undesirable consequences. The thread-management incurs runtime overhead that counterweights any offline optimization in scheduling by the programmer. The cycle for program development and tuning is long. In the resulting program, the mathematical or numerical structures are obscured by thread management, which may often be idiosyncratic, with numerous tantalizing performance parameters to tune. Such program is hard to maintain or modify.

3.2 Adaptive scheduling and thread-free programming

We introduce a recursive scheme for parallelizing the FMM and its implementation in the CILK programming environment. The resulting program is free of thread creation, termination and management by the programmer. The programmer translates an algorithm into a program using a few CILK primitives to describe the dependency-concurrency relationship in the data flow, such as aggregate and disaggregate, which are post-order and pre-order tree traversals in computer science language. The threads are generated, coordinated and terminated at runtime, in adaptation to execution dynamics, following the data flow structure in the algorithm described by the programmer.

The data structure, namely, the FMM graph, is constructed by Algorithm 1, shown in pseudocode. The tree pruning is based on the coupled dual spatial partition criteria introduced in Section 2.4. The link lists between the two trees are as defined in Section 2.2. The parallel construction procedure is composed of two recursive functions `partition_box` and `build_list`.

We present in actual CILK code, the FMM computation in recursion style, in order to manifest our claims about the programming style, structure and mechanism to utilize CILK. RECFMM starts the traversal of G_{FMM} from the leaf source nodes, goes through the aggregation pass along the source tree upward, across to the target tree with M2L translations, followed by the disaggregation down to target tree leaves. In the implementation, the `build_list` function in Algorithm 1 is merged with the `disaggregate` function in Fig. 4. Specifically, RECFMM does not allocate memory to store the link lists. Instead, for each list edge discovered, RECFMM spawns a task to compute the interaction.

A couple of remarks. First, the disaggregation pass follows the aggregation pass by data dependency only. The operations are not necessarily synchronized from the source nodes to the target nodes. The operations are not necessarily spawned by data availability only either. The workload balance over the entire system is taken into account by CILK as well. Secondly, the programmer may wish to express more information and pass it on to CILK, such as the change in priority on which operations to activate first among all those ready to operate, when the available resource is limited. In fact, the authors developed a dynamic prioritization technique [11] for this purpose and the technique also exploits the slack time (the time between the earliest and latest time a task can be scheduled). In [10], Zhang further investigated various runtime management and compiler options in parallel scheduling of the adaptive FMM. He found that none of the techniques could recognize and estimate the slack time. All the experiments confirmed that RECFMM is the most elegant and efficient.

3.3 Discussion

The success of RECFMM lies in the combined utilization of the recursive parallelization and the software system CILK [4]. The recursive parallelization expresses the FMM in a simple, unambiguous and graph-theoretic language that CILK understands and

Algorithm 1 Parallel Construction of the FMM Graph G_{FMM}

PARTITION_BOX(s_{root})
PARTITION_BOX(t_{root}) *//concurrent construction of source and target trees*
BUILD_LIST(t_{root}) *//making link-lists from source to target*

function PARTITION_BOX($ibox$) *//build the spatial partition tree*
  if $|ibox| > s$ **then**
    create $child(1,\ldots,8)$.
    for all $cbox \in child(1,\ldots,8)$ **do** *//Parallel loop*
      PARTITION_BOX(cbox) *//make parent-child connections as well*
    end for
  else
    mark $ibox$ as a leaf
  end if
end function

function BUILD_LIST($ibox$)
  $parentbox := parent(ibox)$.
  for $sbox \in L_1(parentbox)$ **do**
    if $sbox \in colleagues(ibox)$ **then**
      $L_1(ibox) = L_1(ibox) \cup \{sbox\}$.
    else
      $L_4(ibox) = L_4(ibox) \cup \{sbox\}$.
    end if
  end for
  PRUNE_TREE($ibox$)
  $coll\_list := colleagues(ibox)$
  if $ibox$ is nonleaf **then**
    $L_1(ibox) = L_1(ibox) \cup coll\_list$
    for all $cbox \in children(ibox)$ **do** *//Parallel loop*
      BUILD_LIST($cbox$)
    end for
  else
    Further process $coll\_list(ibox)$ to determine $L_3(ibox)$ and $L_1(ibox)$.
  end if
end function

.

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

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

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

    multipole_to_multipole(node);        // generate multipole expansion
  } else {
    source_to_multipole(node);           // generate multipole expansion
  }
  multipole_to_exponential(node);        // convert multipole into exponential
}

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

  local_to_local(node);                  // inherit parent's local expansion
  process_list4(node);
  if (node->nchild) {
    exponential_to_local(node);          // complete M2L using merge-and-shift
    cilk_for (int i = 0; i < N_CHILD; i++) // spawn tasks at a nonleaf target node
      DisAggregate(node->child[i]);
  } else {
    local_to_target(node);               // evaluate local expansion
    process_list13(node);
  }
}
```

Figure 4: Recursive parallelization of the FMM in RECFMM.

supports. The software system CILK outperforms any other existing library for multi-threaded programming due to the fundamental differences. CILK is a special-purpose compiler and runtime scheduler. At compilation time, a compiler performs data-flow analysis and establishes live variable names, identifies array access patterns, pointer analysis and loop dependencies. Optimizations are applied at many places, such as copy propagation, constant folding, common sub-expression and partial redundancy elimination, strength reduction, inline expansion, dead and unreachable code elimination. For a loop statement, the compiler will carry out invariant code motion, loop fusion or fission, unrolling, interchange, tiling. Such complexity is far beyond manual optimization by the programmer. At runtime, the CILK scheduler determines how the computational tasks are divided among *worker threads*. It is based on a decentralized, cooperative policy, referred to as *work stealing*. Every worker thread stores the program frames whose execution has been spawned but not activated in a double-ended queue data structure called

deque. The spawned and suspended frames are stored in the head of the deque and removed from the head to be resumed by the worker. The scheduler steals a suspended frame from the tail of the deque of a busy worker and inserts it in the head of the deque of another worker that has completed its tasks. The policy and its execution guarantee, within a factor of two, the best schedule, even when the dependence graph is known in advance. The programmer's task is simplified, as it shall be, to expressing an algorithm in a CILK language and utilizing the software system CILK and through which the other facilities on modern parallel computers.

4 Installation guide and numerical experiments

4.1 Installation

Download and extract the tarball to get:

- `src`: folder with source files.
- `include`: folder with header files containing the data type declarations and function prototypes.
- `test`: folder with elementary test code.
- `example`: folder with an example demonstrating the calling sequence in a time-marching setup.
- `html`: documentation folder automatically generated by `doxygen`.
- `Makefile`: the top level Makefile where the chosen compilers can be set.
- `License.txt`: the GPL license text.

RECFMM has been extensively tested with the Intel compiler toolset, although the GNU and LLVM Cilk Plus compilers may also be used. In addition, RECFMM requires the scalable memory allocator `scalable_calloc` and `scalable_free` from the *Threading Building Blocks* template library developed by Intel.

4.2 Other interaction types

RECFMM does not obscure mathematical operations by parallel rendition. Its modular design allows RECFMM to be easily changed or customized to different interaction kernels, not limited to the Coulomb and screened Coulomb interactions. To do this, one needs only to provide subroutines for kernel evaluation, expansion and translation. Depending on the kernel function, one can either provide a single M2L operator or provide operators M2E, E2E, and E2L. If a single M2L operator is provided, one needs to make the following two changes in `fmm-action.c` file inside the `src` directory:

1. Comment the `multipole_to_exponential(sbox)` line at the end of the `aggregate` function.

2. Change the `exponential_to_local(tbox)` line in the `disaggregate` function to `multipole_to_local(tbox)`.

There is one additional requirement on the operators related to M2L translations that the M2L translation is initiated by the target box. We have discussed in Section 2.2 the difference between the target-centric implementation and the source-centric implementation.

4.3 Numerical experiments

We present numerical experiments to demonstrate RECFMM performance. The experiments were carried out on a large-memory node of the Stampede cluster at Texas Advanced Computing Center, available through the Extreme Science and Engineering Discovery Environment [9]. A large memory node is equipped with four Intel E5-4650 8-core processors, running at 2.7 GHz clock rate, and 1 TB of DDR3 memory. The operating system is CentOS 6.3 with the 2.6.32 x86_64 Linux kernel. The RECFMM source code was compiled using Intel compiler version 13.1.0 with '-fast' flag and linked the scalable memory allocator `libtbbmalloc.so`.

The numerical experiments can be summarized as follows:

1. Source and target ensembles are set to be different, but have the same number of particles, denoted by N, where $N \in \{10,20,40,80,160\}$ in the unit of million.

2. Particles are uniformly distributed over two types of geometric domains: (a) a cubical box; (b) a spherical surface. Charges carried by the source points are uniformly distributed within the interval $[-0.5,0.5]$.

3. The accuracy levels are set to be of 3-digit or 6-digit in computed potentials [3, Eq. (57)].

4. For each problem size N, the parameter $s \in \{30,35,40,45,50,55\}$ for 3-digit accuracy and $s \in \{60,65,70,75,80\}$ for 6-digit accuracy. Parameter s is chosen such that the amount of time is balanced, in $1:1$ ratio, between near-field interaction (by direct and sparse matrix-vector multiplication) and far-field interaction (via multiple-scale compressed representation/expansion). The balance in time leads to the increase of s with higher accuracy because the approximate expansion takes more terms and hence more work and time. For each approximation level, we let s vary within a predicted range for time performance tuning.

5. For each triple of problem size N, data type d, and choice of s, two sequential and ten parallel executions were carried out, under the restriction on time and space allocation on the test platform, to obtain the respective average time, denoted by $t_{N,d,s,seq}$ and $t_{N,d,s,par}$, respectively. For performance evaluation, the execution time for problem size N and data type d is defined as

$$t_{N,d,seq} = \min_s \{t_{N,d,s,seq}\}, \quad t_{N,d,par} = \min_s \{t_{N,d,s,par}\}.$$

6. Speedup is defined to be the ratio of the best sequential execution time $t_{N,d,seq}$ to the best parallel execution time $t_{N,d,par}$.

Experiment results are presented in Figs. 5, 6, and 7. Accuracy levels are denoted by the prefix 3 or 6, the domain geometries are denoted by the suffix, c for cubical box and s for spherical surface. The experiment with 6c (in blue) is missing the last point due to the lack of memory space. For this case, each tree has 160 million points and 8 levels. Both trees are nearly full. Contrary to a common belief that there is more compression in this case, the memory usage is dominated by the expansions, which outnumber the tree nodes, especially, in 3D. In comparison, on the spherical surface, there are 20 levels for

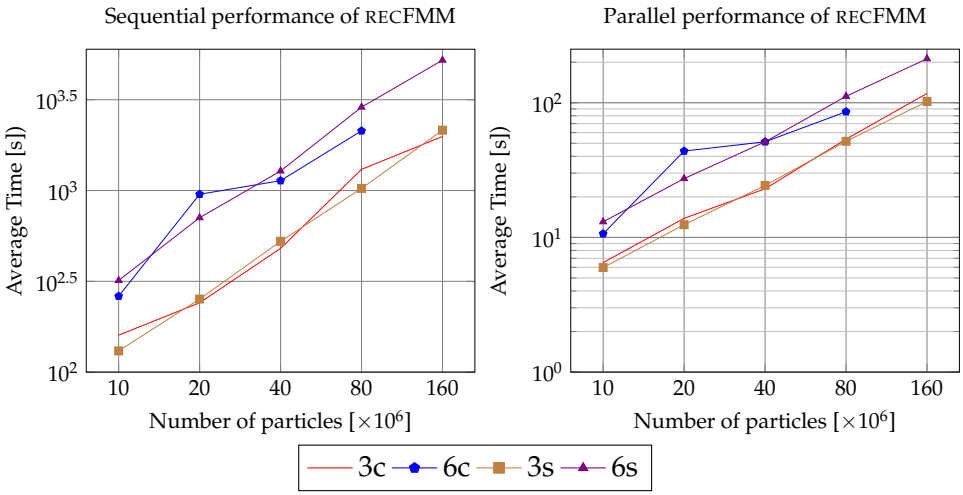

Figure 5: Execution time of RECFMM for Coulomb interaction.

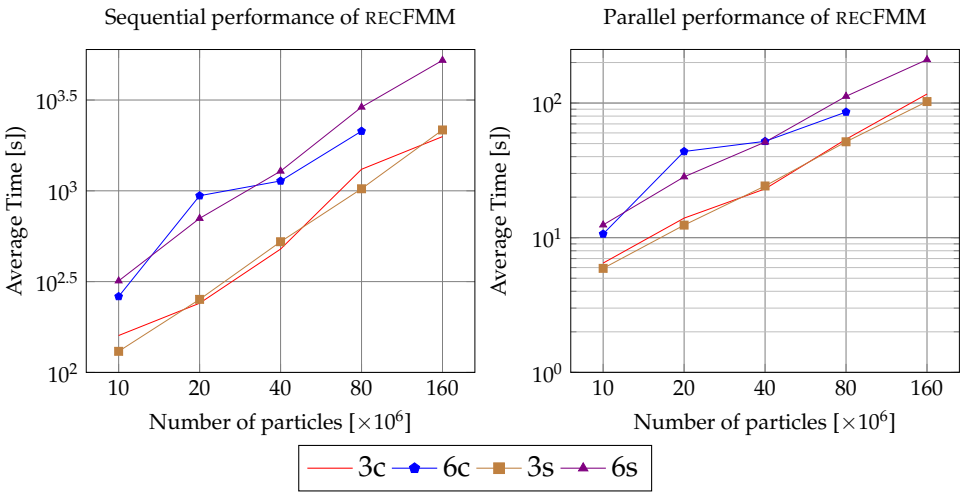

Figure 6: Execution time of RECFMM for screened Coulomb interaction.

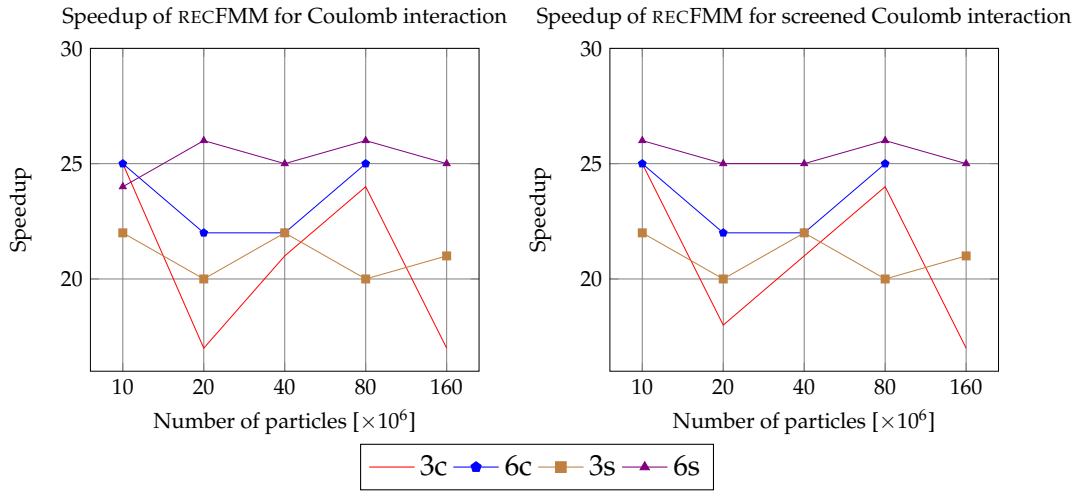

Figure 7: Speedup result of RECFMM for the Coulomb (left) and screened Coulomb (right) interactions.

the 160 million points in each of the trees. The operation compression at the sparse tree nodes and sparse source and target points shows the advantage in memory space.

Acknowledgments

The authors were supported in part by National Science Foundation grants CCF-0905164, CCF-0905473, and ACI-1440396. This experimental work used the facility Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant ACI-1053575. The authors thank Matteo Frigo for his insightful comments, and the anonymous reviewers for their suggestions for various improvements in the revised manuscript.

References

[1] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel Distr. Com.*, 37:55–69, 1996.
[2] J. Carrier, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm for particle simulations. *SIAM J. Sci. Stat. Comp.*, 9:669–686, 1988.
[3] H. Cheng, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm in three dimensions. *J. Comput. Phys.*, 155:468–498, 1999.
[4] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Notices*, 33:212–223, 1998.
[5] L. Greengard and J. Huang. A new version of the fast multipole method for screened Coulomb interactions in three dimensions. *J. Comput. Phys.*, 180:642–658, 2002.
[6] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73:325–348, 1987.

[7] L. Greengard and V. Rokhlin. The rapid evaluation of potential fields in three dimensions. *Lect. Notes Math.*, 1360:121–141, 1988.
[8] L. Greengard and V. Rokhlin. A new version of the fast multipole method for the Laplace equation in three dimensions. *Acta Numer.*, 6:229–269, 1997.
[9] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkens-Diehr. XSEDE: Accelerating Scientific Discovery. *Comput. Sci. Eng.*, 16:62 – 74, 2014.
[10] B. Zhang. Asynchronous task scheduling of the fast multipole method using various runtime systems. In *Proceedings of the Forth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, Edmonton, Canada, 2014.
[11] B. Zhang, J. Huang, N. P. Pitsianis, and X. Sun. Dynamic prioritization for parallel traversal of irregularly structured spatio-temporal graphs. In *Proceedings of 3rd USENIX Workshop on Hot Topics in Parallelism*, 2011.
[12] B. Zhang, B. Peng, J. Huang, N. P. Pitsianis, X. Sun, and B. Lu. Parallel AFMPB solver with automatic surface meshing for calculation of molecular solvation free energy. *Comput. Phys. Commun.*, 190:173–181, 2015.