THE FAST MULTIPOLE METHOD AT EXASCALE

A Dissertation
Presented to
The Academic Faculty

by

Aparna Chandramowlishwaran

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering, College of Computing

Georgia Institute of Technology
December 2013

THE FAST MULTIPOLE METHOD AT EXASCALE

Approved by:

Professor Richard Vuduc, Advisor
School of Computational Science and Engineering, College of Computing
Georgia Institute of Technology

Professor David Bader
School of Computational Science and Engineering, College of Computing
Georgia Institute of Technology

Professor George Biros
Department of Mechanical Engineering
The University of Texas, Austin

Professor Lorena Barba
Department of Mechanical Engineering
Boston University

Dr Kathleen Knobe
Software Solutions Group
Intel Corporation

Date Approved: 11/18/2013

The Fast Multipole Method at Exascale

Aparna Chandramowlishwaran

156 Pages

Directed by Professor Richard Vuduc

This thesis presents a top to bottom analysis on designing and implementing fast algorithms for current and future systems. We present new analysis, algorithmic techniques, and implementations of the *Fast Multipole Method (FMM)* for solving N-body problems. We target the FMM because it is broadly applicable to a variety of scientific particle simulations used to study electromagnetic, fluid, and gravitational phenomena, among others. Importantly, the FMM has asymptotically optimal time complexity with guaranteed approximation accuracy. As such, it is among the most attractive solutions for scalable particle simulation on future extreme scale systems.

We specifically address two key challenges. The first challenge is how to engineer fast code for today's platforms. We present the first in-depth study of multicore optimizations and tuning for FMM, along with a systematic approach for transforming a conventionally-parallelized FMM into a highly-tuned one. We introduce novel optimizations that significantly improve the within-node scalability of the FMM, thereby enabling high-performance in the face of multicore and manycore systems. The second challenge is how to understand scalability on future systems. We present a new algorithmic complexity analysis of the FMM that considers both intra- and inter-node communication costs. Using these models, we present results for choosing the optimal algorithmic tuning parameter. This analysis also yields the surprising prediction that although the FMM is largely compute-bound today, and therefore highly scalable on current systems, the trajectory of processor architecture designs, if there are no significant changes could cause it to become communication-bound as early as the year 2015. This prediction suggests the utility of our analysis approach, which

directly relates algorithmic and architectural characteristics, for enabling a new kind of highlevel algorithm-architecture co-design.

To demonstrate the scientific significance of FMM, we present two applications namely, direct simulation of blood which is a multi-scale multi-physics problem and large-scale biomolecular electrostatics. MoBo (Moving Boundaries) is the infrastructure for the direct numerical simulation of blood. It comprises of two key algorithmic components of which FMM is one. We were able to simulate blood flow using Stokesian dynamics on 200,000 cores of Jaguar, a peta-flop system and achieve a sustained performance of 0.7 Petaflop/s. The second application we propose as future work in this thesis is biomolecular electrostatics where we solve for the electrical potential using the boundary-integral formulation discretized with boundary element methods (BEM). The computational kernel in solving the large linear system is dense matrix vector multiply which we propose can be calculated using our scalable FMM. We propose to begin with the two dielectric problem where the electrostatic field is calculated using two continuum dielectric medium, the solvent and the molecule. This is only a first step to solving biologically challenging problems which have more than two dielectric medium, ion-exclusion layers, and solvent filled cavities.

Finally, given the difficulty in producing high-performance scalable code, productivity is a key concern. Recently, numerical algorithms are being redesigned to take advantage of the architectural features of emerging multicore processors. These new classes of algorithms express fine-grained asynchronous parallelism and hence reduce the cost of synchronization. We performed the first extensive performance study of a recently proposed parallel programming model, called Concurrent Collections (CnC). In CnC, the programmer expresses her computation in terms of application-specific operations, partially-ordered by semantic scheduling constraints. The CnC model is well-suited to expressing asynchronous-parallel algorithms, so we evaluate CnC using

two dense linear algebra algorithms in this style for execution on state-of-the-art multicore systems. Our implementations in CnC was able to match and in some cases even exceed competing vendor-tuned and domain specific library codes. We combine these two distinct research efforts by expressing FMM in CnC, our approach tries to marry performance with productivity that will be critical on future systems. Looking forward, we would like to extend this to distributed memory machines, specifically implement FMM in the new distributed CnC, distCnC to express fine-grained parallelism which would require significant effort in alternative models.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

First and foremost, I want to thank Rich Vuduc, for his tremendous support, patience, and unwavering belief in me even when I didn't believe in myself. His research guidance and encouragement shaped me into what I've become today. His ability in articulating even the most complex ideas in an elegant simple form is astonishing. I will miss my advisor and always cherish the memories of my interactions with him.

Secondly, I'd like to thank my thesis committee members David Bader, George Biros, Kath Kobe, and Lorena Barba for agreeing to serve on my dissertation committee and for their valuable feedback, not just on my thesis but throughout my graduate career. David was also my masters advisor and I'm deeply gratefully to him for giving me the first opportunity and taste of research. He gave me direction and guidance when I was a total novice and has always been extremely supportive of my research. My current thesis wouldn't have been possible without George who introduced me to the world of Fast Multipole Method. He is one of the most energetic, enthusiastic, and optimistic person I've met. He always made me feel as though my work was important and made time to listen to my ideas even when he had a flight to catch.

I'm fortunate to have had the opportunity to intern at wonderful places and work alongside extremely knowledgeable researchers. Kath Knobe has always been my female role model and mentor since my first year. She gave me my first opportunity outside Georgia Tech and the internship I did with her was the most memorable one. She is one of the kindest and most determined person I've met throughout my career and she always had time for me – I'm deeply indebted to her for her support and hours spent trying to make sense of my incoherent ideas. I thank Leonid Oliker and Sam Williams at Lawrence Berkeley National Labs for their introduction and early

guidance into the world of performance and architecture. I gained a tremendous amount of experience and knowledge working with them which has been invaluable for my career. Finally, I thank Pradeep Dubey and Victor Lee at Intel Labs for my first experience working on a research prototype.

For the numerous years spent working side by side in a huge lab and countless hours playing boardgames, exploring the restaurants of Atlanta, and pointless discussion about god knows what including god, I am especially grateful to Jee Choi, Karl Jiang, Seunghwa Kang, Oded Green, Ivan Antonov, and Abtin Rahimian. I'm additionally thankful to the rest of HPC lab and HPC garage for being wonderful officemates and fellow researchers, making my academic life memorable.

Most of all, I am deeply grateful to Kamesh Madduri for his constant emotional support, understanding, and companionship. He was like a second advisor to me and I cannot thank him enough for the numerous hours he spent patiently explaining concepts and ideas. He gave me the confidence and faith, without which I couldn't have finished my thesis.

Finally, I am indebted to my parents for giving me the opportunity to build my career and for their endless love. My sister, Pavithra followed my footsteps to Georgia Tech and made my last couple of years in Atlanta a very enjoyable and memorable time. I will never forget the countless days she made me dinner because I was scrambling to make a deadline. She made leaving Atlanta very difficult and I'm fortunate to have such a loving family.

This research was supported in part by the National Science Foundation (NSF) under award number 0833136, NSF Tera-Grid allocation CCR-090024, NSF CAREER award 0953100, joint NSF 0903447 / Semiconductor Research Corporation (SRC) Award 1981, and the U.S. Department of Energy (DOE) under Contract No. DE-AC02-05CH11231 and grants from the Defense Advanced Research Projects Agency (DARPA), and a PhD fellowship from Intel Corporation. Any opinions, findings and

conclusions or recommendations expressed in this material does not necessarily reflect those of NSF, SRC, DARPA, DOE, or Intel. We would like to thank the Georgia Tech College of Computing and University of California Parallel Computing Laboratory for access to machines.

CHAPTER I

INTRODUCTION

This dissertation presents a top to bottom analysis on designing and implementing fast algorithms for current and future systems. We present new analysis, algorithmic techniques, and implementations of the *Fast Multipole Method (FMM)* for solving N-body problems. We target the FMM because it is broadly applicable to a variety of scientific particle simulations used to study electromagnetic, fluid, and gravitational phenomena, among others. It is also regarded as one of the most important algorithms in scientific and engineering computing [24, 43]. The two main challenges we address in designing and implementing fast algorithms are,

1. how to *engineer* fast code, and

2. *understand* performance and scalability on current and future systems.

We present the first extensive study of single-node performance optimization, tuning, and analysis of the FMM on modern multicore systems. We consider single- and double-precision with numerous performance enhancements, including low-level tuning, numerical approximation, data structure transformations, OpenMP parallelization, and algorithmic tuning. Section 3.3 shows that optimization and parallelization

can improve double-precision performance by 25× on Intel's quad-core Nehalem, 9.4× on AMD's quad-core Barcelona, and 37.6× on Sun's Victoria Falls (dual-sockets on all systems).

However, applying these optimizations by hand, also called *hand-tuning* is becoming increasingly difficult as architectures change over time. This difficulty along with the lack of a systematic approach for producing scalable code is the motivation for trying to sketch out such a process. Also, hand-tuning every single code for every new architecture requires tremendous skilled manpower. We sketch the process in such a way that a tool or compiler could automate atleast part of the process alleviating the burden on the programmer. We decompose this process into three stages. In the first stage, we treat the program and hardware as a black box, limiting analysis and tuning to simple modeling and measurement techniques (Section 4.3). In the second stage, we assume more knowledge of the computation and machine, enabling deeper inferences and at least a limited set of code transformation techniques (Section 4.4). In Section 4.5, we describe the final stage where we assume deep knowledge of the code and machine, and therefore not only arrive at the deepest insights, but also apply the most aggressive transformations. At each stage, we show by example what models, insights, hypotheses, and performance-enhancing optimizations—including aggressive asynchronous scheduling and reordering optimizations for explicit locality and bandwidth management—might be discovered and applied.

The ultimate goal is to be able to scale particle simulations on current and future exa- and extreme-scale systems. To that end, we also develop theoretical models for the different phases of the FMM to understand the absolute limits of performance that can be achieved on different architectures that exist today and hypothetical ones that might be built in the future. We present a new algorithmic complexity analysis of the FMM that considers both computation and communication costs. Not only does the model try to capture algorithmic parameters of the FMM such as number

of particles, depth of the tree, accuracy, etc,. but also key architectural features such as peak processor speed, memory bandwidth, and memory hierarchy design. Such a performance model is powerful enough to answer critical questions such as,

1. Can the optimal value of the algorithmic tuning parameter(s) be estimated purely analytically?

2. Assuming future systems will also be heterogeneous, what is the optimal work partitioning and scheduling strategy? Can we statically before runtime estimate it?

3. FMM is largely compute-bound today. Given technology trends, will it still be compute-bound at exascale?

4. Were it possible to design an ideal system for FMM, what might it look like?

Our predictions run counter to what is expected by the community conducting research in N-body methods. The analysis yields the surprising prediction that although the FMM is largely compute-bound today, and therefore highly scalable on current systems, the trajectory of processor architecture designs, if there are no significant changes could cause it to become communication-bound as early as the year 2015 (Section 8.1). This prediction suggests the utility of our analysis approach, which directly relates algorithmic and architectural characteristics, for enabling a new kind of highlevel algorithm-architecture co-design.

The remainder of this chapter presents a summary of our contributions so far (Section 1.1) and a more detailed context to the problem being addressed in this thesis (Section 1.2).

1.1 Thesis Contributions

The following are the main contributions of this thesis.

- **Algorithm engineering for FMM:** We present an extensive study of single-node performance analysis, optimization, and tuning of a tree-based classical physics N-body method called the FMM on modern multicore systems. These optimizations were then incorporated in a hybrid implementation to support parallelism at all levels, including inter-node distributed memory parallelism (MPI), intra-node shared memory parallelism (OpenMP), data parallelism (SIMD vectorization), and accelerated using GPUs resulting in a highly scalable code (Chapter 3).

- **Systematic approach to tuning:** We describe a systematic process of transforming a conventionally parallelized code to a highly tuned one. Our study lays solid foundation for scaling FMM on future extreme-scale systems. Not only that, our study also sheds new light on the form of a more general performance analysis and tuning process that other multicore/manycore tuning practitioners and automated performance analysis and tuning tools could themselves apply (Chapter 4).

- **New communication analysis:** We present the first in-depth models for compute and memory costs for FMM in Chapter 5. Our analysis refines the estimates of the constants, normally ignored in traditional asymptotic analyses, with calibration against our state-of-the-art implementation. The result is an analytical performance model with three important properties. First, the model predicts the optimal setting of one of the FMMs tuning parameters, which in practice had previously required manual experimentation. Secondly, the model can solve practical performance engineering problems, such as how to schedule the computation for heterogeneous (e.g., CPU+GPU) systems. Thirdly, since the analysis includes important high-level architectural parameters, such as last-level cache capacity, the resulting models can be used to estimate whether the

FMM will scale or not on future architectural designs. Our analysis suggests that we need to re-think the design of future architectures and focus on minimizing communication costs when designing algorithms.

- **Practical applications of FMM:** One of the target applications is the direct simulation of blood, which we model as a mixture of a Stokesian fluid (plasma) and red blood cells (RBCs). We were able to simulate up to 200 million deformable RBCs, which improves upon prior state-of-the-art by four orders of magnitude and the optimized scalable FMM is one of the main components of this infrastructure (Chapter 7).

- **Productivity with performance:** We perform the first extensive performance study of a novel general-purpose parallel programming model, called Concurrent Collections (CnC) for HPC applications using two dense linear algebra algorithms (an asynchronous dense Cholesky factorization and a novel partly-asynchronous dense symmetric eigensolver) on state-of-the-art multicore systems (Chapter 6). Given a well-tuned sequential BLAS, our implementations match or exceed competing multithreaded vendor-tuned codes and manually tuned libraries by up to 2.6×.

1.2 Problem Context and History

N-body methods were identified as one of the original seven dwarfs or motives [15] and are believed to be important in the next decade. In fact, FMM, the focus of this thesis made the list for the top 10 algorithms having the greatest influence on the development of science and engineering in the 20th century [43]. Below, we describe a brief history of the sequential and parallel algorithms to give the reader an overview of the research in this area.

1.2.1 Trends in N-body methods

Sequential algorithms The sub $\mathcal{O}(N^2)$ algorithm that uses approximations to reduce the complexity was independently developed by Appel [13] in 1985 and Barnes and Hut [18] in 1986. In 1987, Greengard and Rokhlin proposed the first linear time algorithm that improved upon the Barnes-Hut algorithm and was based on the theme of multipole expansions and translations [49]. A year later in 1988, the original authors with Carrier extended the idea to adaptive particle distributions [30].

Anderson in 1992 first introduced the idea of equivalent sources, a different approximation to achieve the same running time as the multipole based FMM [12]. In 2003, Ying et al.proposed the kernel independent FMM algorithm which is based on Anderson's idea of equivalent sources and replaces the analytical expansions with equivalent density representations. We discuss this algorithm in detail in Section 2.4.4. There are also other variations of kernel independent FMM such as the black box FMM [45] which uses a Chebyshev interpolation scheme.

Parallel algorithms There has been a lot of effort parallelizing tree codes for both traditional distributed memory clusters and GPU based hybrid systems. Warren and Salmon in 1992 presented the fastest distributed memory parallel implementation for the Barnes-Hut algorithm [95]. The key idea in this paper was the use of local essential trees (LET), which can also be extended to FMM as was later shown by others [65]. The same authors also introduced the idea of hashed octrees along with using space filling curves to improve the efficiency of tree codes in 1993 [96]. One of the highly scalable distributed memory implementations that also handles adaptive distributions is by Lashuk et al. [65].

In the past few years, there has been several attempts on mapping FMM onto GPUs and heterogeneous systems. One of the first attempts on the GPU was in 2008 by Gumerov et al. [50]. Several authors have since then presented results on cluster of GPUs and considered various work partitioning schemes to divide work between

CPUs and GPUs [53, 58, 64, 90, 101].

1.2.2 Case for analytical performance modeling via co-design

Traditional algorithm analysis based on asymptotic complexity has been a key metric in understanding the efficiency of an algorithm. But, abstract algorithm complexity analysis does not capture key parameters of an architecture, such as the number of cores or cache sizes or bandwidth to memory or memory latency. Since we are ultimately interested in running the computation on some machine, such analysis can be misleading since it does not capture the parameters of the system.

To give a more concrete example as to why traditional asymptotic complexity analysis is not sufficient in today's computing world, consider *sorting* which is a classical fundamental problem. Two popular in-memory sorting algorithms are *merge* and *radix* sorts. It is well known that radix sort has a computational complexity of $\mathcal{O}(N)$ as opposed to comparison based sorts such as merge sort which have a lower bound of $\Omega(N \log N)$. But, it has been shown that even though radix sort is asymptotically more efficient than merge sort, because of its high bandwidth utilization and inherent difficulty to vectorize due to simultaneous updates to the same memory location, merge sort delivers higher performance in practice [85]. As a result, ignoring architecture can result in incorrect choice of algorithms and we make a case for *algorithm architecture co-design*, a new model for analyzing algorithms [40].

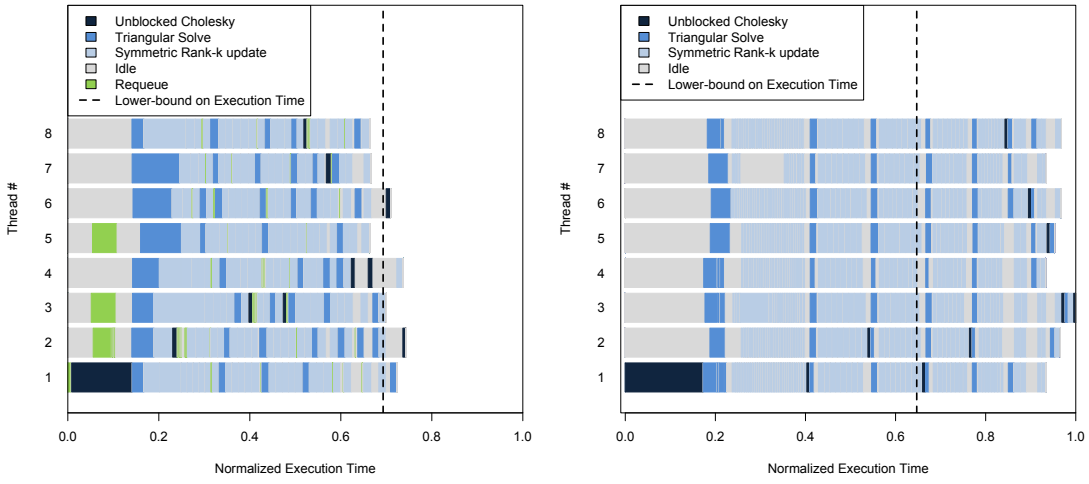
This approach is largely analytical and at a higher-level than cycle accurate performance analysis and modeling but that gives us the freedom to tweak the parameters in the model to make future predictions and inferences given the rapidly changing hardware market. It is intended to be at a higher level and act as a guide for algorithm designers and also architects rather than have the fidelity of traditional hardware/code specific methods involving simulators, benchmarks, traces, etc..

Chapter 5 explains in detail the value we derive from applying such an analysis for

the FMM. We have chosen time to solution as the metric for optimization throughout this thesis. Given the growing emphasis on reduction in power and energy, we acknowledge them as important factors but *race to halt* is still practically the best technique today. We hope to factor in these metrics as part of future work.

1.2.3 Case for programming models inspired by asynchronous style of execution

In high performance computing, focus is not just on attaining high performance but also on productivity. It is not an easy task for domain scientists and practitioners to produce high performing implementations across the spectrum of architectures ranging from multicores and manycores to accelerators and heterogeneous systems. To that end, we discuss productive performance solutions that maximize programmer productivity given the fast changing landscape of parallel computing. Specifically, we sketch an argument motivating the need for *asynchronous-parallel programming models*.



(a) CnC (asynchronous) **(b)** Cilk++ (bulk synchronous)

Figure 1: Scheduling for Cholesky factorization on CnC and Cilk++ (matrix size = 1000)

Figure 1 (left) shows the scheduling timeline for Cholesky factorization of a matrix

of size 1000 using an asynchronous programming language, CnC. We can see that the different inner kernels are executing asynchronously and the vertical dotted line which is the lower bound on execution time reveals that CnC is performing close to its lower bound.

Instead, if we were to synchronize at the end of every task, we would end up with a timeline that looks like Figure 1 (right). This is our Cilk++ based recursive implementation which spends a lot of time idle waiting for slower threads to finish execution compared to the asynchronous schedule of CnC and ultimately performing much below its potential.

The above experiment makes a compelling case for asynchronous style of execution and there has been numerous similar efforts by various groups that encourage this style of execution demonstrated across various application domains. We give an overview of few interesting ones here apart from CnC which is explained in detail in Chapter 6. The underlying idea in all these models is a task-based execution model and an intelligent runtime that tries to extract parallelism.

- **StarSs:** It introduces **#pragma** annotations and relies on a source-to-source compiler to generate tasks [19, 79]. It is the task of the programmer to identity which tasks should be offloaded to the different computing environments.

- **OmpSs:** The key idea here is to extend an already existing popular programming language OpenMP with new directives to support asynchronous parallelism [44]. It derives from StarSs and builds on top of it. It works on different architectures including multicores, GPUs, and hybrid environments.

- **StarPU:** It is based on integrating a data management library that enforces a coherent view between heterogeneous memory (*e.g.*CPU main memory and GPU memory) and a tasking API [16]. It implements a greedy *list scheduling* where tasks are inserted into queues when the last dependency of the task is

executed.

- **QUeuing And Runtime for Kernels (QUARK):** QUARK is a dynamic runtime system that asynchronously schedules tasks on multicore multi-socket shared memory systems [98]. The main idea is to decipher the data dependencies between tasks using runtime analysis of data hazards such as Read After Write (RAW), Write After Read (WAR), and Write After Write (WAW) based on the data usage. Although its focus has been to support dynamic linear algebra algorithms, it is more generally applicable to other domains.

- **DAGuE:** It is a directed acyclic graph (DAG) scheduling engine where the nodes are sequential tasks and the edges denote the data dependencies [25]. It consists of a distributed multi-level dynamic scheduler, an asynchronous communication engine, and a engine for detecting data dependencies. The runtime is responsible for automatically distributing data between various resources efficiently.

Though we try to argue that programmer productivity is an important metric, it is hard to quantify this but we believe all the above efforts are tending in the right direction in programming model/language and runtime designs.

CHAPTER II

SURVEY OF N-BODY METHODS

This chapter provides an overview of popular N-body methods and is intended to serve as a survey of the key ideas behind these methods. Section 2.1 defines the problem we are trying to solve. Section 2.2 describes the tree data structure and Sections 2.3 and 2.4 present two popular tree-based algorithms including analysis of their computational complexity and correctness.

2.1 Problem Overview

The N-body problem can be defined as the problem of simulating the movement of points or particles or bodies under the influence of some type of force. Depending on the type of force, there are numerous applications ranging from astrophysics, molecular dynamics, fluid dynamics, to computer graphics and machine learning.

Mathematically, given a system of N *source* particles, with positions given by $\{y_1, \ldots, y_N\}$, and N *targets* with positions $\{x_1, \ldots, x_N\}$, we wish to compute the N sums,

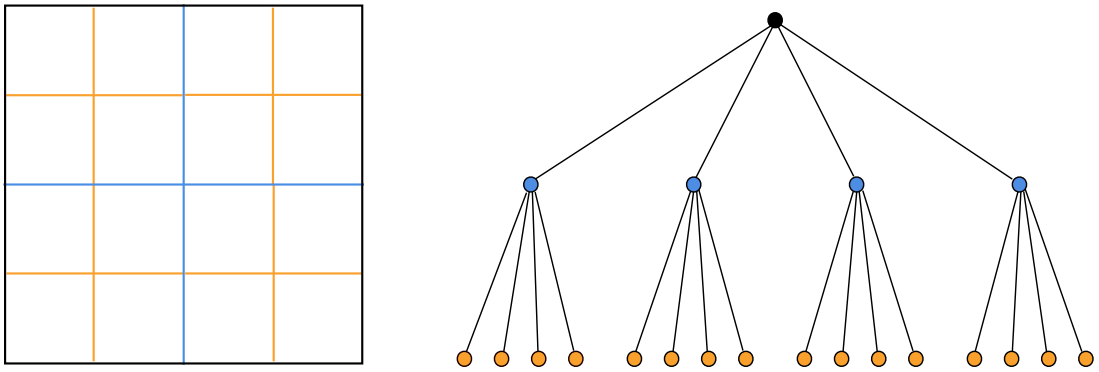


Figure 2: A uniform complete quadtree in 2-D with 3 levels.

$$f(x_i) = \sum_{j=1}^{N} K(x_i, y_i) \cdot s(y_j), \quad i = 1, \ldots, N \tag{1}$$

where $f(x)$ is the desired *potential* at target point x; $s(y)$ is the *density* at source point y; and $K(x, y)$ is an *interaction kernel* that specifies "the physics" of the problem. For instance, the single-layer Laplace kernel, $K(x, y) = \frac{1}{4\pi}\frac{1}{||x-y||}$, might model electrostatic or gravitational interactions.

There are a number of algorithms for computing the potential and it's derivative force exerted on the target particles by the source particles. They can be broadly classified into two categories namely, direct and approximation algorithms. Evaluating these sums using a direct algorithm appears to require $\mathcal{O}(N^2)$ operations. However, there are computationally less expensive algorithms which reduce the complexity to $\mathcal{O}(N \log N)$ and even $\mathcal{O}(N)$. One such class of approximation algorithms are called *tree-methods*, which use a tree data structure to hierarchically decompose the particles. We will focus on these tree-based algorithms and other N-body methods are outside the scope of this thesis.

2.2 Data Structure: Quadtrees and Octress

Given the input points and a user-defined parameter q, we construct an oct-tree T (or quad-tree in 2-D) by starting with a single box representing all the points and

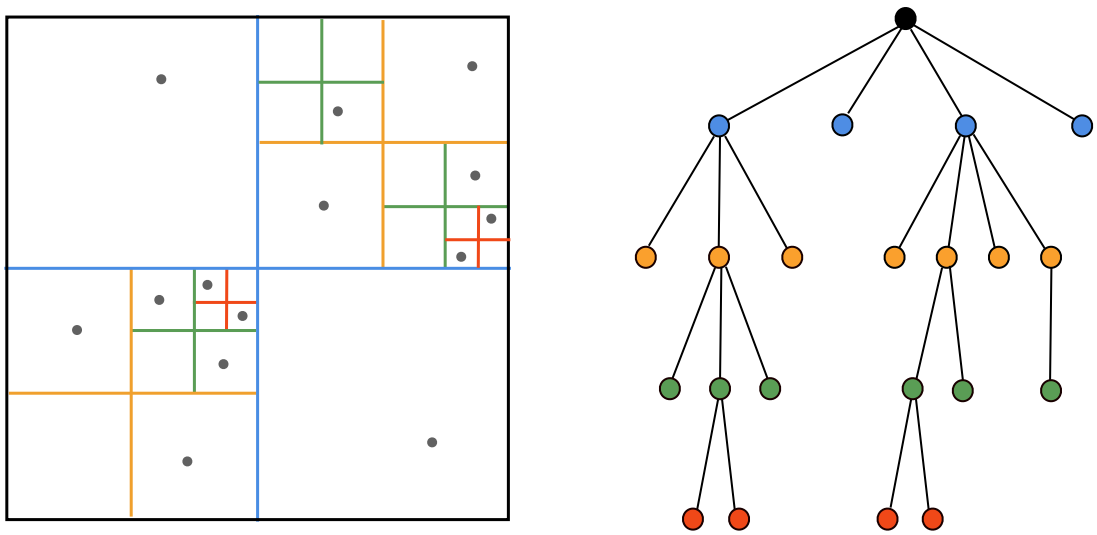


Figure 3: An adaptive quadtree in 2-D with $q = 1$.

recursively subdividing each box if it contains more than q points. Each box (octant in 3-D or quadrant in 2-D) becomes a tree node whose children are its immediate sub-boxes.

We illustrate with the example of a quadtree since it is easier to visualize in 2-D, the octree in 3-D is analogous. Figure 2 shows two visualizations of a uniform quadtree where each node is subdivided into four children. Each level of the tree is denoted by a different color and the tree is fully balanced. However, in most real applications, the particles are not uniformly distributed and constructing a balanced tree in that case would result in storing unnecessary information. Instead, we only subdivide the boxes that contain more than q particles. This results in a non-uniform or *adaptive* quadtree as shown in Figure 3. Adaptive trees are more complex and hence more difficult to deal with than non-adaptive trees. Therefore, for most of our analytical discussions, we will assume a non-adaptive tree but we present results for both types of input distributions.

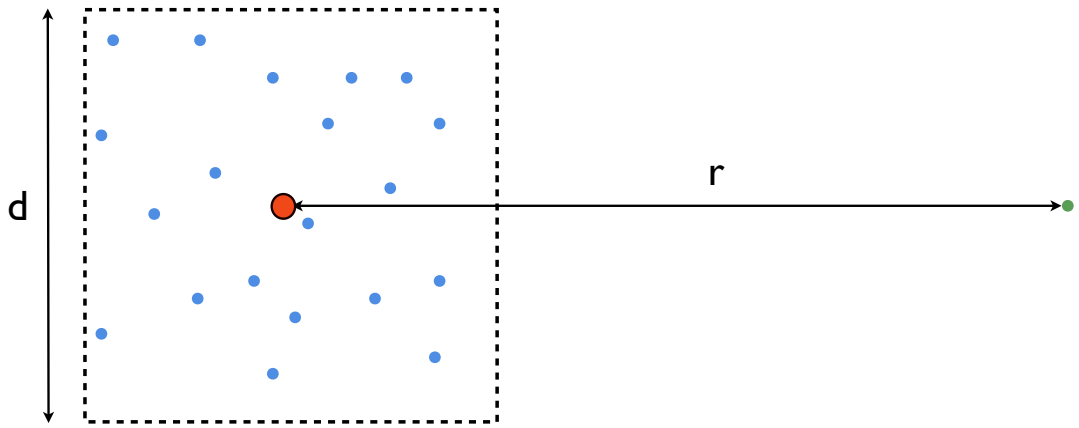


Figure 4: The potential due to a set of particles enclosed by a box of size d on a single particle at a distance r can be approximated by the center of mass if the MAC is satisfied.

2.3 Barnes-Hut Algorithm: $\mathcal{O}(N \log N)$ method

The Barnes-Hut algorithm published by J. Barnes and P. Hut in 1986 [18] is a tree-based approximation algorithm which reduces the computational complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$.

Basic Idea The key idea in developing fast potential calculation is the notion of approximating the potential due to a set of particles. We say a particle is *far away* from a square region as shown in the Figure 4 if the distance r between the particle and the center of mass of the box denoted by the red dot is larger than some constant times the side length of the box, d. This ratio between d and r is called the *Multipole Acceptance Criterion (MAC)*.

$$\theta = \frac{d}{r} \tag{2}$$

In such cases, we can approximate the set of particles by a single point located at its *center of mass* with a mass equal to the total mass of all particles. The center of mass is shown as a red dot in Figure 4. The approximation is more accurate if the point at which the potential is being evaluated is far away from the box containing the particles. In other words, smaller θ results in better accuracy.

Algorithm At a high-level, we can describe the Barnes-Hut algorithm as follows.

- **Step 1: Tree Construction** Choose a q value and construct the tree as described in Section 2.2.

- **Step 2: Far-field Construction** For all nodes in the tree, compute the center of mass and total mass of the particles in each node. This is usually done by traversing the tree bottom-up or *post order* traversal of the tree, starting with the leaf nodes and computing the far-field representation of the non-leaf nodes from its children in constant time.

- **Step 3: Evaluation** We traverse each node of the tree; if a node is *far away* from a target particle then, we compute the potential due to all the particles in the node by just using its center of mass and total mass. We compare the computed MAC to a user-defined threshold to decide whether to approximate or continue to traverse the children of the node if it is a non-leaf. If the node is a leaf, the potential due to the particles in the node is computed using a direct evaluation.

Complexity Analysis If the depth of the tree is $\mathcal{O}(\log N)$, then Step 1 can be done in $\mathcal{O}(N \log N)$ time. Step 2 performs a post order traversal of the tree and has a cost of $\mathcal{O}(N \log N)$ since every particle contributes to the calculation of the center of mass and total mass. In Step 3, for each particle, we traverse the tree to compute the potential on it. The total cost of this step if $\mathcal{O}(N \log N)$. The dominant term is $\mathcal{O}(N \log N)$ and is the running time of the Barnes-Hut algorithm.

2.4 FMM: $\mathcal{O}(N)$ method

The Fast Multipole Method was published by L. Greengard and V. Rokhlin in 1987 [49] is another tree-based approximation algorithm which shares the tree structure with

Barnes-Hut but FMM instead computes *approximations* of all of these sums in optimal $\mathcal{O}(N)$ time with a guaranteed user-specified accuracy, where the desired accuracy changes the complexity constant [49].

This section provides an overview of the Fast Multipole Method (FMM), summarizing the key components that are relevant to this thesis. For more in-depth algorithmic details, see Greengard, et al. [49, 100].

The FMM is based on two key ideas: (i) a *tree representation* for organizing the points spatially; and (ii) *fast approximate evaluation*, in which we compute summaries at each node using a constant number of tree traversals with constant work per node. Before diving into the algorithm, we will introduce some key ideas used in FMM.

2.4.1 Key Ideas

In this subsection, we discuss three key ideas employed for reducing the complexity to $\mathcal{O}(N)$. We discuss the two-dimensional case since it is simpler to understand for the Laplace equation in $\mathbb{R}^2$.

Multipole Expansion The first idea is to use multipole expansions to reduce the number of calculations. The multipole expansion allows us to approximate the potential due to the particles in a node of a quadtree or octree if it is *far away* from any other node or particle. This is similar in spirit to Barnes-Hut where we used the center of mass and total mass to approximate the influence of the particles in a node. We will define *far away* for FMM more rigorously later on but this definition is sufficient for now to understand the basic intuition behind this method.

Theorem 2.4.1. Multipole Expansion

Suppose that m charges of strengths $q_i, i = 1, ..., m$ are located at points $z_i, i = 1, ..., m$, with $|z_i| < r$. Then, for any z with $|z| < r$, the potential $\phi(z)$ induced by the charges is given by

$$\phi(z) = Q\log(z) + \sum_{k=1}^{\infty} \frac{a_k}{z^k}, \tag{3}$$

where

$$Q = \sum_{i=1}^{m} q_i \quad \textit{and} \quad a_k = \sum_{i=1}^{m} \frac{-q_i z_i^k}{k}. \tag{4}$$

Local Expansion Multipole expansion allows us to compute the potential *far away* from a node, due to the particles *inside* the node. Local expansion on the other hand, lets us compute the potential *inside* the node due to the particles *far away* from it.

Translations In order to understand the algorithm, we describe three transformations used on the expansions described above which is key to explaining the method.

1. **Multipole to Multipole Translation (M2M)**: The M2M translation transforms the mutlipole expansion of a node's children to its own multipole expansion. It is mathematically defined in Theorem 2.4.2.

   Theorem 2.4.2. Multipole to Multipole Translation

   Suppose that

   $$\phi(z) = a_0\log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z - z_0)^k} \tag{5}$$

   is a multipole exapansion of a potential due to a set of m charges of strengths $q_1, q_2, ..., q_m$, all of which are located inside the circle D of radius R with center at z_0. Then for z outside the circle D_1 of radius $(R + |z_0|)$ and centered at the origin,

   $$\phi(z) = a_0\log(z) + \sum_{l=1}^{\infty} \frac{b_l}{z^l}, \tag{6}$$

   where

$$b_l = -\frac{a_0 z_0^l}{l} + \sum_{k=1}^{l} a_k z_0^{l-k} \begin{pmatrix} l-1 \\ k-1 \end{pmatrix}, \tag{7}$$

with $\binom{l}{k}$ *the binomial coefficients.*

2. **Multipole to Local Translation (M2L)**: The M2L translation transforms the multipole expansion of a node into the local expansion of a node in its interaction list.

   Theorem 2.4.3. Multipole to Local Translation

   Suppose that m *charges of strengths* $q_1, q_2, ..., q_m$ *are located inside a circle* D_1 *with radius* R *and center at* z_0*, and that* $|z_0| > (c+1)R$ *with* $c > 1$*. Then the corresponding multipole expansion Equation* (5) *converges inside the circle* D_2 *of radius* R *centered about the origin. Inside* D_2*,*

$$\phi(z) = \sum_{l=0}^{\infty} b_l . z^l, \tag{8}$$

   where

$$b_0 = a_0 \log(-z_0) + \sum_{k=1}^{\infty} \frac{a_k}{z_0^k} (-1)^k, \tag{9}$$

   and

$$b_l = -\frac{a_0}{l . z_0^l} + \frac{1}{z_0^l} \sum_{k=1}^{\infty} \frac{a_k}{z_0^k} \begin{pmatrix} l+k-1 \\ k-1 \end{pmatrix} (-1)^k, \quad \textit{for } l \geq 1. \tag{10}$$

3. **Local to Local Translation (L2L)**: The L2L translation transforms the local expansion of a node's parent to its own local expansion.

   Theorem 2.4.4. Local to Local Translation

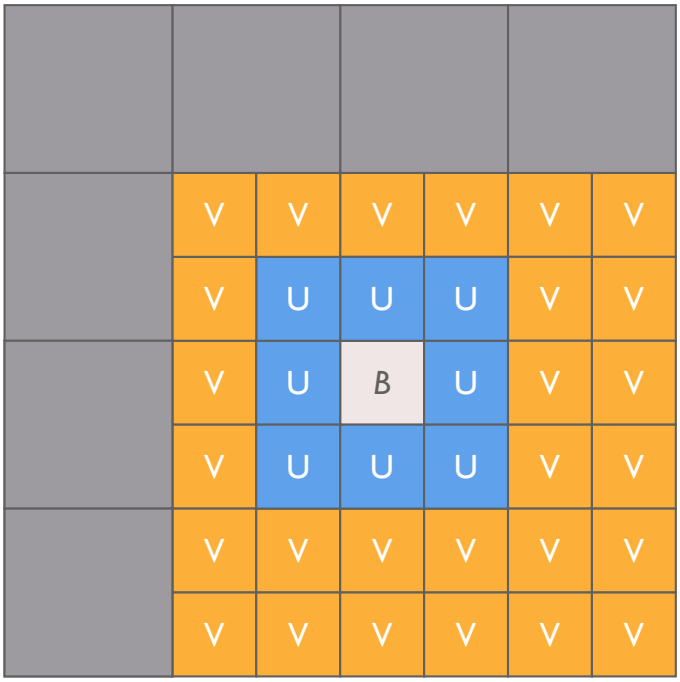


Figure 5: U and V lists of a tree node B for a uniform quadtree in 2-D.

Translation of a local expansion centered about z_0 into an expansion centered about the origin,

$$\sum_{k=0}^{n} a_k (z - z_0)^k = \sum_{l=0}^{n} b_l . z^l, \tag{11}$$

where

$$b_l = \sum_{k=1}^{n} a_k \begin{pmatrix} k \\ l \end{pmatrix} (-z_0)^{k-l}. \tag{12}$$

2.4.2 Non-adaptive Algorithm

Algorithm We can describe the FMM algorithm for as follows.

- **Step 1: Tree Construction** Choose a q value and construct the tree as described in Section 2.2. During construction, we associate with each node two neighbor *lists* namely U and V. Each list has bounded constant length and

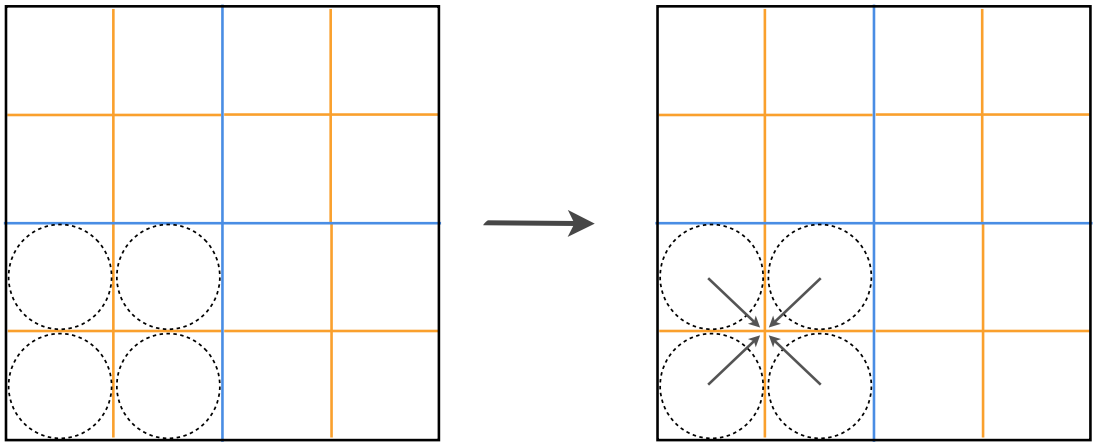


Figure 6: M2M translation. To compute the multipole expansion of the parent, translate the multipole expansion of the four children in 2D and accumulate them to get the multipole expansion of the parent.

contains (logical) pointers to a subset of other tree nodes. For example, every leaf box $B \in \text{leaves}(T)$ has a U *list*, $U(B)$, which is the list of all leaves adjacent to B. Figure 5 shows a quad-tree example, where neighborhood list nodes for B are labeled accordingly.

Tree construction has $O(N \log N)$ complexity, and so the $O(N)$ optimality refers to the evaluation phase (below). However, tree construction is typically a small fraction of the total time; moreover, many applications build the tree periodically, thereby enabling amortization of this cost over several evaluations.

- **Step 2: Far-field Construction (Upward Step)** For each leaf node in the tree, construct the multipole expansion due to all the particles inside the node. For all the non-leaf nodes at each level in the tree, construct the multipole expansion by combining the expansions of its children as shown in Figure 6. This is usually done by traversing the tree bottom-up or *post order* traversal of the tree, starting with the leaf nodes and computing the multipole expansions of the non-leaf nodes from its children in constant time.

- **Step 3: Direct Evaluation (U list step)** For each leaf B in the tree, we

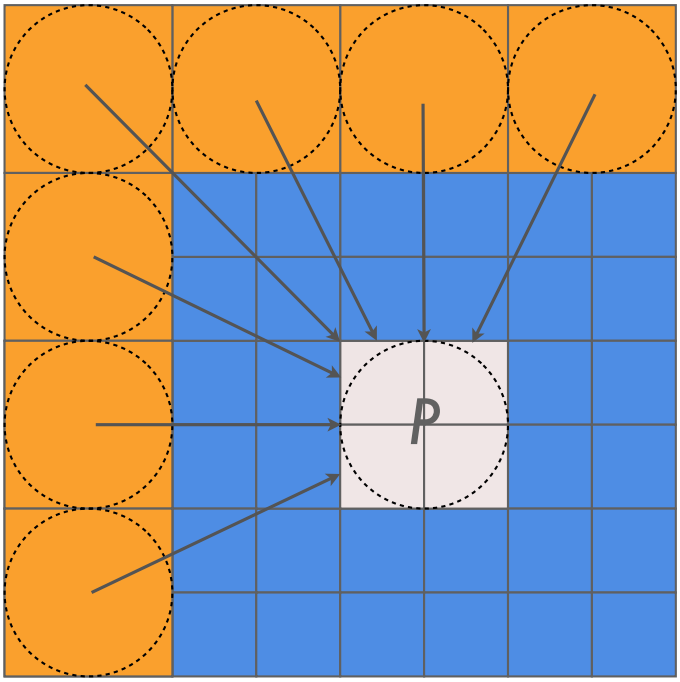


Figure 7: M2L translation. To compute the local expansion of the parent node P, we translate the multipole expansion of the nodes in the interaction list of P denoted by orange shaded region and accumulate them to get the local expansion of P.

perform a direct evaluation between B and the leaves that are adjacent to it which are its nearest neighbors. These are the blue nodes in Figure 5 and are denoted by the letter U for a given leaf node B.

- **Step 4: Far-field to Near-field Conversion (V list step)** For each node B, we want to compute the potential inside the node due to all the particles. Step 3 computes the potential due to all the adjacent nodes. Hence, we are left with all the other nodes (orange and gray nodes in Figure 5). For each node B, we begin by converting the multipole expansion of all the boxes in its interaction list into local expansions about the center of the node B. For a given box B, the nodes in its interaction list are denoted by orange color and the letter V in Figure 5. This leaves the particles in the gray shaded boxes but this would fall in the interaction list of B's parent as shown in Figure 7.

- **Step 5: Near-field Evaluation (Downward Step)** This phase is analogous

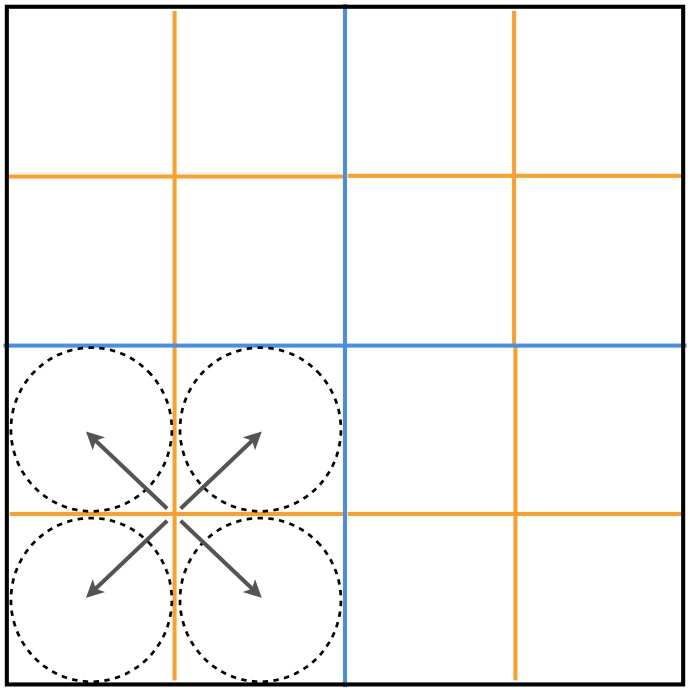


Figure 8: L2L translation. To compute the local expansion of the four children in 2D, translate the local expansion of the parent to the child nodes.

to the upward phase. We combine the partial near-field representations computed in Step 4 to form the complete near-field representation and complete the evaluation. This is done by traversing the tree top-down or *pre order* traversal of the tree, where we compute parents before children as shown in Figure 8. Let us denote the parent of a node B by the letter P. The near field representation of P includes the contribution of all the particles in the gray shaded region. Local to local translation converts the expansion from the parent P to the child node B. Combining this with the local expansion computed in Step 4 gives us the complete near-field representation. Local expansion at the finest level is now available and can be used to generate the potential due to all the particles other than near neighbors. The local expansions at the particle positions are evaluated and the result is added to the potential due to near neighbors computed using direct evaluation in Step 3 for every particle.

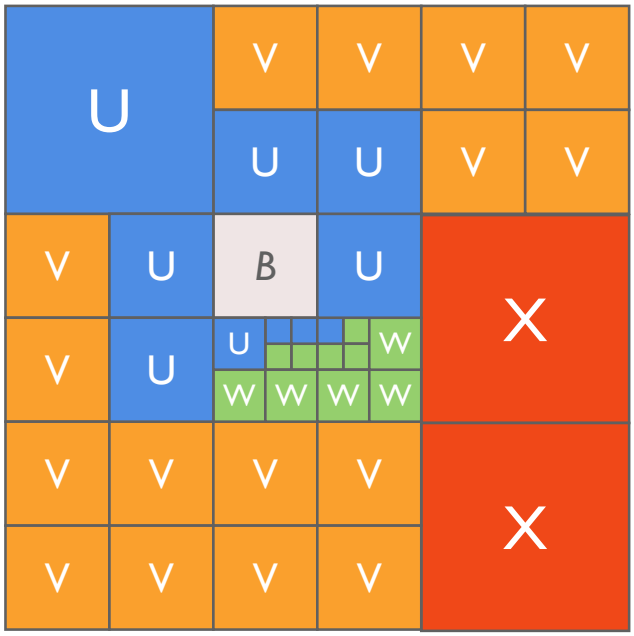


Figure 9: U, V, W, and X lists of a tree node B for an adaptive quadtree in 2-D.

2.4.3 Adaptive Algorithm

When the input distribution is not uniformly distributed, we switch from an uniform tree structure to an adaptive data structure. It is possible to still construct an uniform tree even if the input distribution is not uniform, but it will result in a number of empty nodes resulting in additional storage which is not optimal. We will now detail how to deal with adaptivity in FMM.

Tree Construction

Recall from the tree construction for non-adaptive distributions, that we associate with each node one or more neighbor *lists*. In addition to U and V lists, in the adaptive case, we introduce two additional lists canonically knows as W and X. Figure 9 shows a quad-tree example, where neighborhood list nodes for B are labeled accordingly. For example, every leaf box $B \in \text{leaves}(T)$ has a W *list*, $W(B)$, which is the list of all descendants of B's neighbors whose parents are adjacent to B, but they are not adjacent to B themselves. The X *list*, $X(B)$ consists of all nodes A such that $B \in W(A)$.

Evaluation To account for adaptivity, we add two stages to the non-adaptive algorithm to make it generic.

- **Step 3a: W list step** For each leaf B in the tree, we evaluate the multipole expansion of all the boxes in $W(B)$ at every particle position in B. The result is added to the potential computed in Step 5 of the non-adaptive algorithm. $W(B)$ are the green nodes in Figure 9 and are denoted by the letter W for a given leaf node B.

- **Step 4a: X list step** For each node B, construct the local expansion due to all the particles in $X(B)$ and add the resulting expansion to the partial near-field representation computed in Step 5. $X(B)$ are the red nodes in Figure 9 and are denoted by the letter X for a given leaf node B.

2.4.4 Kernel-Independent FMM

The *kernel independent variant* of the FMM, or KIFMM [100] has the same structure as the classical FMM [49]. Its main advantage is that it avoids the mathematically challenging analytic expansion of the kernel, instead requiring only the ability to evaluate the kernel.

The key idea is to replace analytical expansions and translations of the kernel with *equivalent density* representations. Before describing the representations and translations in the kernel independent version, we will define some notation in Table 1.

Equivalent density and check potential In KIFMM, the equivalent operation to computing the multipole expansion is to compute the upward equivalent density (the potential from the source densities ϕ_i is represented as the potential from the density distribution $\phi^{B,u}$, also called the *upward equivalent density* at locations $y^{B,u}$, the *upward equivalent surface* of a box B). In other words, it involves the solution of an integral equation, the potential induced by the source densities and the upward

Table 1: Notation used in KIFMM

B	node in the tree
q	maximum particles per source or target leaf node
N	number of source or target particles
I_s^B	set of indices of source particles in B
I_t^B	set of indices of target particles in B
$\mathbb{N}^B$	near range of B in $\mathbb{R}^d$
$\mathbb{F}^B$	far range of B in $\mathbb{R}^d$
$y^{B,u}$	upward equivalent surface of B
$\phi^{B,u}$	upward equivalent density of B
$x^{B,u}$	upward check surface of B
$q^{B,u}$	upward check potential of B
$y^{B,d}$	downward equivalent surface of B
$\phi^{B,d}$	downward equivalent density of B
$x^{B,d}$	downward check surface of B
$q^{B,d}$	downward check potential of B

equivalent density satisfy a second-order linear elliptic PDE. The solution of an exterior Dirichlet problem for this PDE is unique and the two potentials are equal in the far range of B if they coincide at the boundary or the intermediate surface between the far range and the upward equivalent surface. This intermediate surface is called the *upward check surface* and denoted by $x^{B,u}$ and the potential computed at this surface is called the *upward check potential*, $q^{B,u}$ as shown in Figure 10.

$$\int_{y^{B,u}} G(x,y)\,\phi^{B,u}\,\mathrm{d}y = \sum_{i \in I_s^B} G(x,y_i)\,\phi_i = q^{B,u} \quad \text{for any } x \in x^{B,u}. \tag{13}$$

Likewise, we represent the potential inside B due to the sources in the far range as the potential induced by the density distribution, $\phi^{B,d}$, called the *downward equivalent density* at locations $y^{B,d}$ called the *downward equivalent surface* in $\mathbb{N}^B$ as shown in Figure 10. The solution of the interior Dirichlet problem for the given PDE is also unique and we only need to match the potentials on the *downward check surface*, $x^{B,d}$ and the matched potential is called the *downward check potential*, $q^{B,d}$.

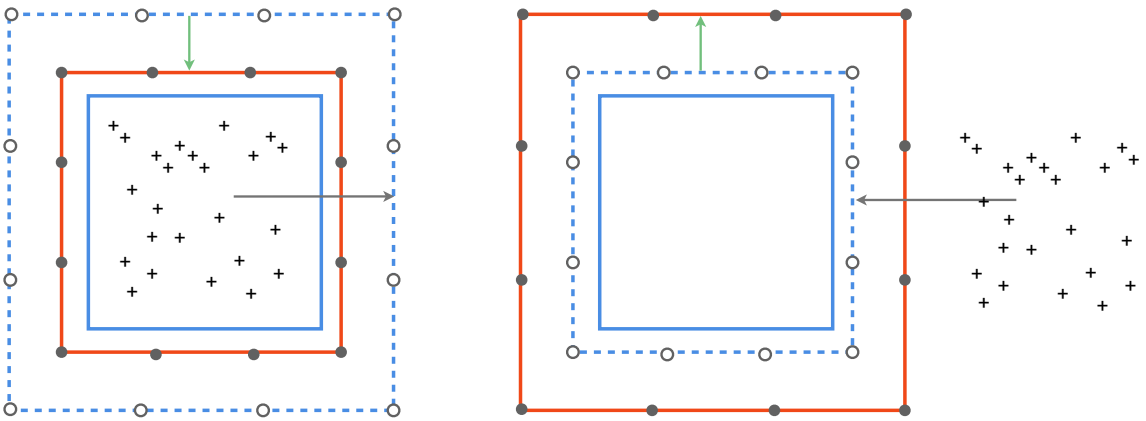


Figure 10: The cross section of the equivalent and check surfaces in 3D. Left: Computation of upward equivalent density. Right: Computation of downward equivalent density. The solid red square denotes the equivalent surface and the blue dotted line represents the check surface. On both surfaces, discretization points are equally spaced and marked with $\bullet$ and $\circ$ respectively. The source densities are marked with $+$. The computation of the equivalent density is shown by the arrows. The gray arrows denotes the evaluation of the check potential using the source particles and the green arrows denote the inversion of the integral equation to compute the equivalent density.

$$\int_{y^{B,d}} G(x,y)\,\phi^{B,d}\,\mathrm{d}y = \sum_{i\in I_s^{\mathbb{F}B}} G(x,y_i)\,\phi_i = q^{B,d}. \tag{14}$$

In KIFMM, we use trapezoidal rule to discretize the integral equations (trapezoidal rule is known to have super-algebraic convergence for smooth functions) on spheres in 2D. In 3D, we discretize on the surface of a cube and the p quadrature points are equally spaced on the six faces. The cross section of the cube in 3D is shown in Figure 10. The choice of the value of p determines the accuracy.

M2M translation The upward equivalent density of the non-leaf nodes are computed by traversing the tree bottom-up and translating the equivalent density from the child node A to its parent node B. We solve the following equation for $\phi^{B,u}$ and the process is illustrated in Figure 11.

$$\int_{y^{B,u}} G(x,y)\,\phi^{B,u}\,\mathrm{d}y = \int_{y^{A,u}} G(x,y)\,\phi^{A,u}\,\mathrm{d}y \quad \text{for all } x \in x^{B,u}. \tag{15}$$

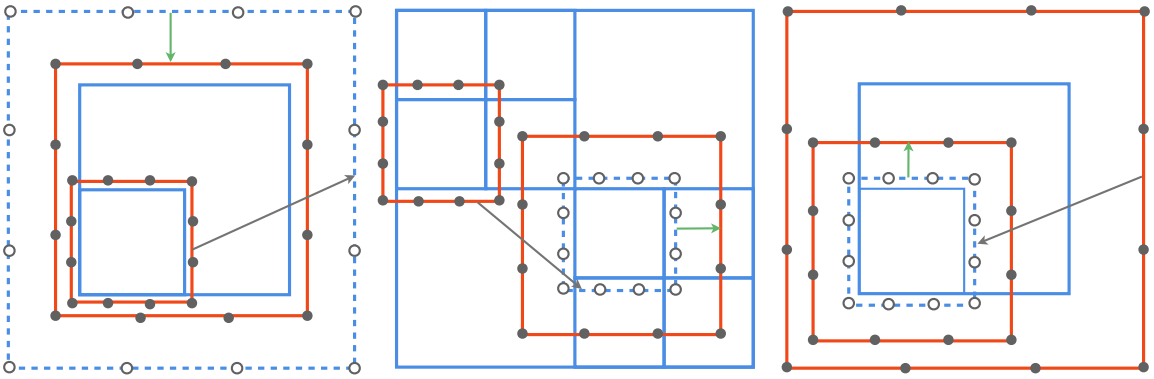


Figure 11: Translations in 3D. Left: M2M translation. Middle: M2L translation. Right: L2L translation. The solid red square denotes the equivalent surface and the blue dotted line represents the check surface. On both surfaces, discretization points are equally spaced and marked with • and ∘ respectively.

M2L translation M2L translates the upward equivalent density into downward equivalent density similar to the classical algorithm. Suppose a node A is in the far range of B, $\mathbb{F}^B$, we solve the following equation for $\phi^{B,u}$.

$$\int_{y^{B,d}} G(x,y)\,\phi^{B,d}\,\mathrm{d}y = \int_{y^{A,u}} G(x,y)\,\phi^{A,u}\,\mathrm{d}y \quad \text{for all } x \in x^{B,d}. \tag{16}$$

L2L translation L2L translation are computed by traversing the tree top-down and translating the downward equivalent density of a parent A to compute that of its child B. The downward check potential of the child box B, $\phi^{B,d}$ satisfies the following equation.

$$\int_{y^{B,d}} G(x,y)\,\phi^{B,d}\,\mathrm{d}y = \int_{y^{A,d}} G(x,y)\,\phi^{A,d}\,\mathrm{d}y \quad \text{for all } x \in x^{B,d}. \tag{17}$$

2.4.5 Complexity Analysis

We summarize the asymptotic complexity and main algorithmic characteristics of each phase of KIFMM in Table 2. Even though KIFMM has the same overall linear complexity as the classical algorithm, it does more work which is the trade-off for using kernel evaluations as opposed to analytical expansions of the kernel. For the

Table 2: Asymptotic complexity and characteristics of the computational phases in KIFMM. N is the number of source particles, the number of boxes is $M \sim N/q$ and p denotes the number of expansion coefficients. The user chooses p to trade-off time and accuracy, and may tune q to minimize time. †Size is determined by the chosen accuracy, generally smaller than q.

Phase	Computational Complexity	Algorithmic Characteristics
Upward	$O(Np + Mp^2)$	postorder tree traversal, small† matvecs
U-list	$O(27Nq)$	direct computation as in Equation 1 (matvecs on the order of q)
V-list	$O(Mp^{3/2}logp + 189Mp^{3/2})$	consists of small FFTs, pointwise vector multiplication (convolution)
X-list	0 uniform distribution $O(Nq)$ non-uniform distribution	matvecs
W-list	0 uniform distribution $O(Nq)$ non-uniform distribution	matvecs
Downward	$O(Np + Mp^2)$	preorder tree traversal, small† matvecs

rest of this thesis, we will focus on the kernel independent FMM which has the same stages as the classical algorithm described above but at the same time, allows one to leverage our techniques and apply them to new kernels and problems.

CHAPTER III

INTRA-NODE ALGORITHM ENGINEERING FOR FMM

This chapter presents the first extensive study of single-node performance optimization, tuning, and analysis of the Fast Multipole Method (FMM) [49] on state-of-the-art multicore processor systems. This chapter focuses on single-node performance since it is a critical building-block in scalable multi-node distributed memory codes and, moreover, is less well-understood.

Specifically, we consider implementations of the *kernel-independent FMM* (KIFMM) algorithm [100], which simplifies the integration of FMM methods in practical applications (Chapter 2). The KIFMM itself is a complex computation, consisting of six distinct phases, all of which we parallelize and tune for leading multicore platforms (Section 3.3). We develop both single- and double-precision implementations, and consider numerous performance enhancements, including: low-level instruction selection, SIMD vectorization and scheduling, numerical approximation, data structure transformations, OpenMP-based parallelization, and tuning of algorithmic parameters. Our implementations are analyzed on a diverse collection of dual-socket multicore systems, including those based on the Intel Nehalem, AMD Barcelona, Sun Victoria Falls, and NVIDIA GPU processors. (Section 3.4).

3.1 Background and Related Work

For parallel FMM, most of the recent work of which we are aware focuses on distributed memory codes with GPU-based acceleration [7,50,65,82]. Indeed, the present study builds on our own state-of-the-art parallel 3-D KIFMM implementation, which uses MPI+CUDA [65]. However, these works have not yet considered conventional multicore acceleration and tuning. In Section 3.4, we compare our multicore optimizations to this prior use of GPU acceleration, with the perhaps surprising finding

Table 3: Architectural Parameters. All power numbers, save the GPU, we obtained using a digital power meter. *reciprocal square-root approximate. †shared among cores on a socket. ‡max server power (of which the 2 active CPUs consume 160W) plus max power for two GPUs.

Architecture	Intel X5550 (Nehalem)		AMD 2356 (Barcelona)		Sun T5140 (Victoria Falls)		NVIDIA T10P (S1070)	
Frequency (GHz)	2.66 GHz		2.30 GHz		1.166 GHz		1.44 GHz	
Sockets	2		2		2		2 (+2 CPUs)	
Cores/Socket	4		4		8		30 (GPU)	
Threads/Core	2		1		8		8 (GPU)	
SIMD (DP, SP)	2-way	4-way	2-way	4-way	1	1	1	8-way
GFlop/s (DP, SP)	85.33	170.6	73.60	146.2	18.66	18.66	N/A	2073.6
rsqrt/s* (DP, SP)	0.853	42.66	0.897	73.60	2.26	—	N/A	172.8
L1/L2/L3 cache	32/256/8192† KB		64/512/2048† KB		8/4096† KB		—	
local store	—		—		—		16 KB	
DRAM Bandwidth	51.2 GB/s		21.33 GB/s		64.0 GB/s		204 GB/s	
Power	375W		350W		610W		325W+400W‡	
Compiler	icc 10.1		icc 10.1		cc 5.9		nvcc 2.2	

that a well-tuned multicore implementation can match a GPU code. Coulaud, et al., propose Pthreads- and multithreaded BLAS-based multicore parallelization within node [37]. However, we use a larger set of optimizations and provide cross-platform performance and power analysis.

There are numerous non-GPU studies of single-core distributed memory FMM implementations [63, 76] (see also references in Ying, et al. [99]), most based on the classical tree-based N-body framework of Warren and Salmon [96], including our own prior KIFMM work [65, 99]. Researchers have considered a variety of data structures with attractive communication properties, again in the distributed context [54]. To our knowledge, the present study is the first to consider extensive multicore-centric optimizations, data structures, and cross-platform analysis.

For direct $O(N^2)$ methods, tuning, and special-purpose hardware (*e.g.* MDGRAPE), see the references in related papers [14, 75].

3.2 Experimental Setup

We explore FMM performance as we vary architecture, floating-point precision, and initial particle distribution. To facilitate comparisons to prior work, we select a commonly used kernel K (Laplace kernel in Chapter 2). The desired accuracy is fixed to a typical minimum setting that is also sensible for single-precision (yielding 4–7 decimal digits of accuracy). Moreover, because tuning can dramatically change the requisite number of floating-point operations, we define and defend our alternate performance metrics. These aspects are discussed in detail below.

3.2.1 Architectures

This section summarizes the key differences, as they pertain to the FMM, among the three dual-socket multicore SMPs used in this study: Intel's quad-core Xeon (Nehalem), AMD's quad-core Opteron (Barcelona), and Sun's chip-multithreaded, eight-core UltraSparc T2+ (Victoria Falls). Our final analysis references our prior GPU-only accelerated results [65]. The key parameters of these systems appear in Table 5.

Basic microarchitectural approach:

Nehalem and Barcelona are x86, superscalar, out-of-order architectures with large per-thread caches and hardware prefetchers. Victoria Falls, by contrast, employs fine-grained chip multithreading (CMT) and smaller per-thread caches. Consequently, Victoria Falls requires the programmer to express roughly an order of magnitude more parallelism than the x86 systems in order to achieve peak performance or peak bandwidth. Luckily, there is ample fine-grained thread-level parallelism in the FMM.

Computational peak:

The two x86-based systems have similar peak floating-point performance, but 4× higher than Victoria Falls partly due to the SIMD units on x86. SIMD enables up to to 4 flops (multiply and add) per cycle per core in double-precision (DP) and 8

in single-precision (SP). Because the FMM has high computational intensity in at least one of its major phases (U-list), we may expect superior performance on the x86 systems compared to Victoria Falls.

Unfortunately, many kernels $K(x, y)$ also require square root and divide operations, which on all three systems are not pipelined and therefore are extremely slow. For example, on Nehalem, double-precision divide and square root run at 0.266 GFlop/s (5% of peak multiply/add performance) and 0.177 GFlop/s (3%), respectively. To address this deficiency, both x86 systems (but not Victoria Falls) have a low latency and pipelined single-precision approximate reciprocal square-root operation ($\frac{1}{\sqrt{x}}$) that we can exploit to accelerate double-precision computations [75].

Memory systems:

Nehalem has a much larger L3 cache and much higher peak DRAM bandwidth. This should enable better performance on kernels with large working sets. However, Nehalem also has smaller L1 and L2 caches, yielding a per-thread cache footprint that is $\frac{1}{4}$ that of Barcelona, suggesting performance will be similar for computations with small working sets and high computational intensity. The FMM phases exhibit a mix of input-dependent behaviors, and so the ultimate effects are not entirely clear a priori.

Comparisons to GPU:

Our prior work applied GPU acceleration to KIFMM on the NCSA Lincoln Cluster [65], where each node is a dual-socket × quad-core Xeon 5410 (Harpertown) CPU server paired with two NVIDIA T10P GPUs. We use the CPUs only for control, and run all phases (except tree construction) on the GPUs. That is, there is one MPI process on each socket, and each process is assigned to one GPU; processes communicate via message passing and to their respective GPUs via PCIe. For our energy comparisons, we bound power using two configurations: aggregate peak GPU power plus zero CPU power, and aggregate peak GPU power plus the *peak* CPU power. With

12× the compute capacity and over 5× the bandwidth, one would naïvely expect the GPU implementation to considerably outperform all other platforms.

3.2.2 Kernel, Precision, and Accuracy

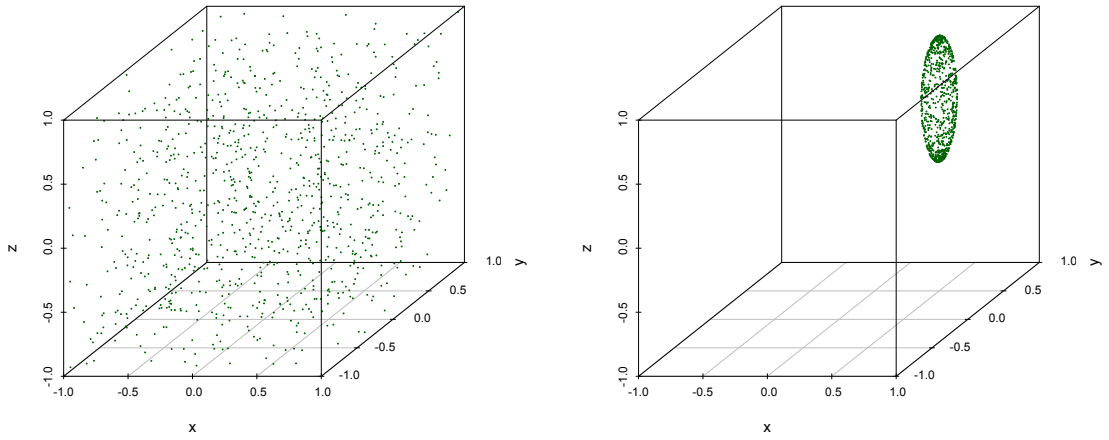
In this section, we describe the interaction kernel, precision, and desired accuracy, and their implications for implementation and optimization.

Kernel: Following prior work, we use the single-layer Laplacian kernel (Chapter 2) owing to its widely-recognized importance [100].

Precision: We consider both single and double-precision in our study.

Single-precision is an interesting case for a variety of reasons. First, an application may have sufficiently low accuracy requirements, due to uncertainty in the input data or slow time-varying behavior. In this case, using single-precision can yield significant storage and performance benefits. Next, on current x86 architectures, SIMD instructions are 2-wide in double precision and 4-wide in single. However, forthcoming x86 Advanced Vector Extensions (AVX) will double these widths. As such, using single precision SIMD on today's Nehalem is a proxy for double precision performance on tomorrow's Sandy Bridge. Finally, current architectures provide fast reciprocal square-root methods in single-precision, but not double. By exploring the benefits in single, we may draw conclusions as to potential benefit future architectures may realize by implementing equivalent support in double.

Accuracy: One of the inputs to FMM is numerical accuracy desired in the final outcome, expressed as the desired "size" of the multipole expansion. In our experiments, we choose the desired accuracy to deliver the equivalent of 6 decimal digits in double-precision and 4 digits in single. We verify the delivered accuracy of our all of our naïve, optimized, parallel, and tuned implementations.



(a) (b)

Figure 12: Distribution of particles inside a unit cube. Left: uniform random distribution. Right: ellipsoidal distribution with an aspect ratio of 1:1:4.

3.2.3 Particle Distributions

We examine two different particle distributions namely, a spatially uniform and a spatially non-uniform (elliptical or ellipsoidal) distribution as shown in Figure 12. The uniform case is analyzed extensively in prior work; the non-uniform case is where we expect tree-based methods to deliver performance and accuracy advantages over other numerical paradigms (*e.g.*, particle-mesh methods). In both cases, our test problems use 4 million source particles plus an additional 4 million target particles.

Uniform: In this case, we distribute points uniformly at random within the unit cube. In 3D, for boxes not on the boundary, the U-list (neighbor list) contains 27 boxes and the V-list (interaction list) contains 189 boxes. The X- and W-lists are empty since the neighbors of a box are adjacent boxes in the same level. Thus, the time spent in the various list computations will differ from the non-uniform case and tuning will favor a different value for q, the maximum points per box.

Elliptical: In this case, particles are angularly-uniformly (in spherical coordinates) distributed on the surface of an ellipsoid with an aspect ratio 1:1:4. For an

uniform distribution, a regular octree is constructed. However, the elliptical case requires an adaptively refined octree. As such, the depth of the computation tree could be quite large, resulting in high tree construction times as shown in Figure 16.

3.2.4 Performance Metrics

Since our optimizations and tuning of q can dramatically change the total number of floating-point operations, we use time-to-solution (in seconds) as our primary performance metric rather than GFlop/s In our final comparison, we present relative performance (evaluations per second), where higher numbers are better.

This choice has ramifications when assessing scalability. In particular, rather than examining GFlop/s/core or GFlop/s/thread to assess per-core (or per-thread) performance, we report **thread-seconds**: that is, the product of execution time by the number of threads. When it comes to energy efficiency, we present the ratio relative to the optimized and parallelized Nehalem energy efficiency (evaluations per Joule).

3.3 Optimizations

We applied numerous optimizations to the various computational phases (Section 2). Beyond optimizations traditionally subsumed by compilers, we apply numerical approximations, data structure changes, and tuning of algorithmic parameters. Table 4 summarizes our optimizations and their applicability to the FMM phases and our architectures. Note that not all optimizations apply to all phases.

Figure 13 presents the cumulative benefit as each optimization is successively applied to the serial reference KIFMM implementation [100]. We will refer to this figure repeatedly as we describe each optimization. If an optimization has associated tuning parameters (*e.g.*unrolling depth), we tune it empirically.

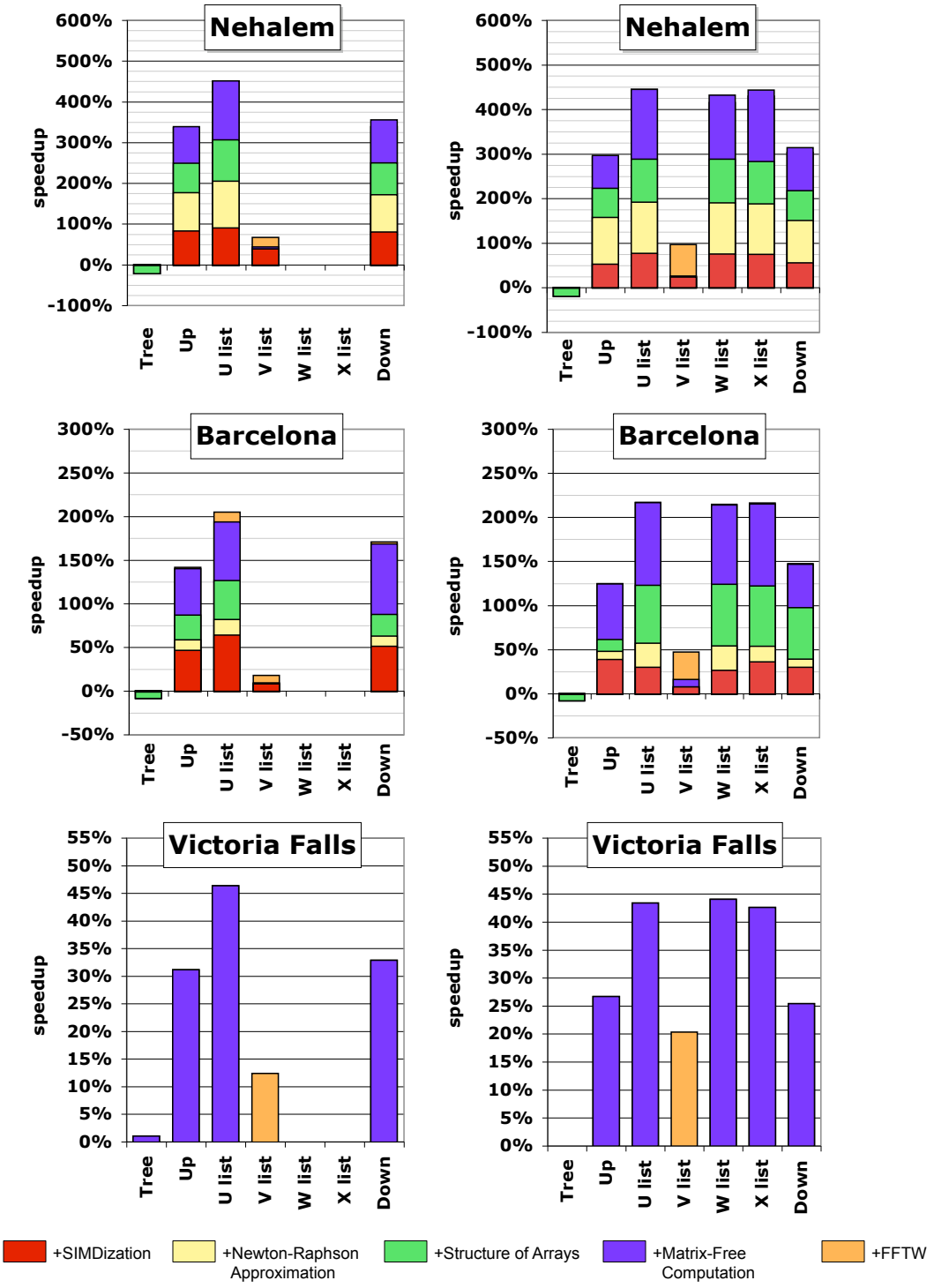


Figure 13: Speedup over the double-precision reference code. Left: Uniform distribution. Right: Elliptical distribution. Note, W- and X- lists are empty for the uniform case. SIMD, Newton-Raphson, and data structure transformations were not implemented on Victoria Falls.

Table 4: FMM optimizations attempted in our study for Nehalem and Barcelona (x86) or Victoria Falls (VF). *Structures of arrays (SOA) layout. [1]double-precision only. A "✓" denotes all architectures, all precisions.

	Tree	Up	U-list	V-list	W-list	X-list	Down
SIMDization	—	x86	x86	—	x86	x86	x86
Newton-Raphson	N/A	x86[1]	x86[1]	N/A	x86[1]	x86[1]	x86[1]
SOA* layout	x86	x86	x86	x86	x86	x86	x86
Matrix-free	N/A	✓	✓	✓	✓	✓	✓
FFTW	N/A	N/A	N/A	✓	N/A	N/A	N/A
OpenMP	—	✓	✓	✓	✓	✓	✓
Tuning for best q	✓	✓	✓	✓	✓	✓	✓

3.3.1 SIMDization

We found it necessary to apply SIMD vectorization manually, as the compiler was unable to do so. All Laplacian kernel evaluations and point-wise matrix multiplication (in the V-list) are implemented using SSE intrinsics; specifically, in double-precision, we use SSE instructions like `addpd`, `mulpd`, `subpd`, `divpd`, and `sqrtpd`. Note that the Laplacian kernel performs 10 flops (counting each operation as 1 flop) per pairwise interaction, and includes both a square-root and divide.

In single-precision on x86, there is a fast (pipelined) approximate reciprocal square-root instruction: `rsqrtps`. As such, with sufficient instruction- and data-level parallelism, we may replace the traditional scalar `fsqrt`/`fdiv` combination with not simply a `sqrtps`/`divps` combination, but entirely with one `rsqrtps`. Doing so enables four reciprocal square-root operations per cycle without compromising our particular accuracy setting.

Figure 13 shows the speedup from SIMDization. The top three figures in Figure 13 use an uniform particle distribution and the bottom three use an elliptical distribution. SIMD nearly doubles Nehalem performance for all kernels except V-list, where FFTW (see Section 3.3.5) is already SIMDized. The benefit on Barcelona was much

smaller (typically less than 50%), which we will investigate in future work. As there are no double precision SIMD instructions in SPARC/Victoria Falls, SIMD related optimizations are not applicable.

3.3.2 Fast Reciprocal Square Root

A conventional double-precision SIMDized code would perform the reciprocal square-root operation using the intrinsics `sqrtpd` and `divpd` as above. Unfortunately, these instructions have long latencies (greater than 20 cycles) and cannot be pipelined, thus limiting performance. As we have abundant instruction-level parallelism, we can exploit x86's fast single-precision reciprocal square-root instruction to accelerate the double-precision computations [75]. That is, we replace the `sqrtpd/divpd` combination with the triplet, `cvtpd2ps` (convert double to single)/`rsqrtps`/`cvtps2pd` (single to double). To attain the desired accuracy, we apply an additional Newton-Raphson refinement iteration. This approach requires more floating-point instructions, but they are low latency and can be pipelined.

Figure 13 shows that the Newton-Rapshon approach improves Nehalem performance by roughly 100% over SIMD. Since there are relatively few kernel evaluations in the V-list, we don't see an appreciable benefit. Surprisingly, the benefit on Barcelona is relatively modest; the cause is still under investigation.

3.3.3 Structure-of-Arrays Layout

Our reference implementation uses an array-of-structures (AOS) data structure where all components of a point are stored contiguously in memory (*e.g.* $x_1, y_1, z_1, x_2, y_2, z_2,$ ..., x_n, y_n, z_n). This layout is not SIMD-friendly as it requires a reduction across every point and unrolling the inner loop twice (or 4 times in single-precision).

Instead, we explore using the structure-of-arrays (SOA) or structure splitting layout [106] in which the components are stored in separate arrays. This transformation simplifies SIMDization since we can load two (or four) components into separate

SIMD registers using a single instruction.

Changing the data layout further improved the overall U-list performance on Nehalem up to 300% speedup over the reference. The transformation does not affect the V-list phase due to its relatively low computational intensity. Moreover, the data layout change substantially improved Barcelona performance on most phases.

Unfortunately, the data layout change increased tree construction time due to lack of spatial locality. This tradeoff (dramatically reduced computational phase time for slightly increased tree construction time) is worthwhile if tree construction time is small compared to the total evaluation execution time.

3.3.4 Matrix-free calculations

To use tuned vendor BLAS routines, our reference code explicitly constructs matrices to perform matrix-vector multiplies (*matvecs*), as done by others [37]. However, we can apply what is essentially interprocedural loop fusion to eliminate this matrix, instead constructing its entries on-the-fly and thereby reducing the cache working set and memory traffic.

For example, recall that the U-list performs a direct evaluation like Equation 1, which is a matvec, between two leaf boxes. Rather than explicitly constructing the kernel matrix K and performing the matvec, we can fuse the two steps and never store this matrix, reducing the memory traffic from $O(q^2)$ to $O(q)$, if the maximum points per box is q. The idea applies to the *Up*(leaf), *W-list*, *X-list*, and *Down*(leaf) phases as well; the matrices arise in a different way, but the principle is the same.

As seen in Figure 13, this technique improves Nehalem performance by an additional 25%, and often improved Barcelona and Victoria Falls performance by better than 40%. Cumulatively including this optimization (aside from V-list) improved Nehalem performance by more than 400%, Barcelona by more than 150%, and Victoria Falls by 40%.

3.3.5 FFTW

In our KIFMM implementation, the V-list phase consists of (i) small forward and inverse FFTs, once per source/target box combination; and (ii) pointwise multiplication I times for each target box, where I is the number of source boxes in said target box's V-list.

Local FFTs are performed using FFTW [47]. Typically, one executes an FFTW plan for the array with which the plan was created using the function `fftw_execute`. As the sizes and strides of the FFTs in V-list are not only quite small but are also identical, we may create a single plan and reuse it for multiple FFTs, using the `fftw_execute_dft_r2c` function. We ensure alignment by creating the plan with the `FFTW_UNALIGNED` flag coupled with an aligned `malloc()`.

FFTW only benefits V-list computations. Nevertheless, on both x86 machines, FFTW substantially improved V-list performance for elliptical distributions. The benefit on uniform distributions was less dramatic since the relative time spent in the V-list tends to be smaller. Unfortunately, Victoria Falls saw little benefit from the plan reuse within FFTW; this will be addressed in future work.

3.3.6 Tree Construction

There are numerous studies of parallel tree construction [7, 54, 89]. In this paper, we focus on accelerating the evaluation phases of FMM for two main reasons. First, tree construction *initially* constituted a small fraction of execution time. Second, in many real simulation contexts, particle dynamics may be sufficiently slow that tree reconstruction can be amortized. Future work will involve parallelization of tree construction keeping in mind both uniform and non-uniform distributions.

3.3.7 Parallelization and Tuning

After applying serial optimization, we parallelize all phases via OpenMP. There are multiple levels of concurrency during evaluation: across phases (*e.g.*, the upward and

U-list phases can be executed independently), within a phase (*e.g.*, each leaf box can be evaluated independently during the U-list phase), and within the per-octant computation (*e.g.*, vectorizing each direct evaluation). For the *Upward* and *Downward* phases, which both involve tree traversals, there is a obvious dependency between a parent and its child boxes. However, the children themselves are independent and can be computed concurrently with the amount of work per level increasing toward the leaves.

We apply inter-box parallelization for all phases except *Upward* and *Downward*. That is, we assign a chunk of leaf boxes to each thread and exploit parallelism within each phase. For *Upward* and *Downward* which has dependencies across the levels of the tree, we exploit the concurrency at each level. By convention, we exploit multiple sockets, then multiple cores, and finally threads within a core.

Algorithmically, the FMM is parameterized by the maximum number of particles per box, q. For instance, in the U list phase, we traverse all leaf nodes, where for each leaf node $B \in \text{leaves}(T)$ we perform a *direct evaluation* between B and each of the nodes $B' \in U(B)$. For each leaf B, this direct evaluation operates on $O(q)$ points and has a flop cost of $O(q^2)$ for each box. By contrast, the V-list operates on $O(q)$ points and performs $O(q \log q)$ flops for each box, and so has lower computational intensity. Generally, we expect the cost of the U-list evaluation phase to dominate other phases when q is sufficiently large.

As expected, when q grows, the U-list phase quickly increases in cost even as other phases become cheaper as the tree height shrinks; this dependence is non-trivial to predict, particularly for highly non-uniform distributions. We exhaustively tune q for each implementation on each architecture, as discussed in Section 3.4. Auto-tuning q will be the subject of our future work.

3.4 Performance Analysis

The benefits of optimization, threading, and tuning are substantial. When combined, these methods delivered speedups of 25×, 9.4×, and 37.6× for Nehalem, Barcelona, and Victoria Falls, respectively, in double-precision for the uniform distribution; and 16×, 8×, and 24×, respectively, for the elliptical case. In this section we first tune our parallel implementation for the FMM's key algorithmic parameter, q, the maximum particles per box. We then analyze the scalability for each architecture. Finally, we compare the performance and energy efficiency among architectures.

3.4.1 Tuning particles per box

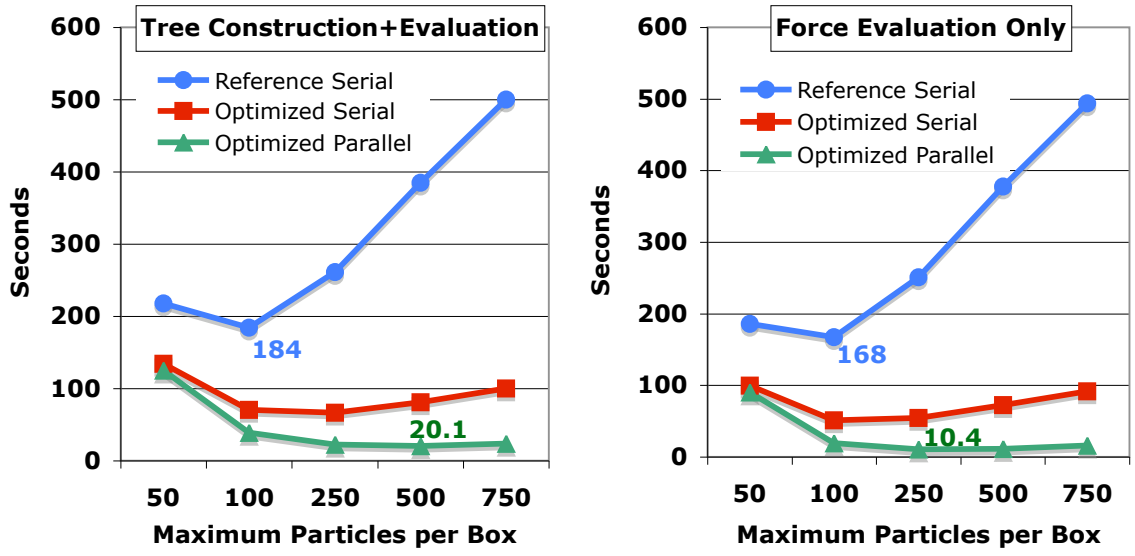
Figure 14(a) presents the FMM execution time as a function of optimization, parallelization, and particles per box q on Nehalem with an elliptical particle distribution. Although we performed this tuning for all architectures, particle distributions, and precisions, we only present Nehalem data due to space limitations.

The optimal setting of q varies with the level of optimization, with higher levels of optimization enabling larger values of q. Since we did not parallelize tree construction, we consider just the evaluation time in Figure 14(b). Thus, one should only tune q (or other parameters affected by parallelization) after all other optimizations have been applied and tuned.

Figure 14(c) decomposes evaluation time by phase. The Upward traversal, Downward traversal, and V-list execution times decrease quickly with increasing q. However, execution time for the other lists, especially U-list, grow with increasing q. Observe a crossover point of $q = 250$ where time saved in Up, Down, and V-list can no longer keep pace with the quickly increasing time spent in U-list.

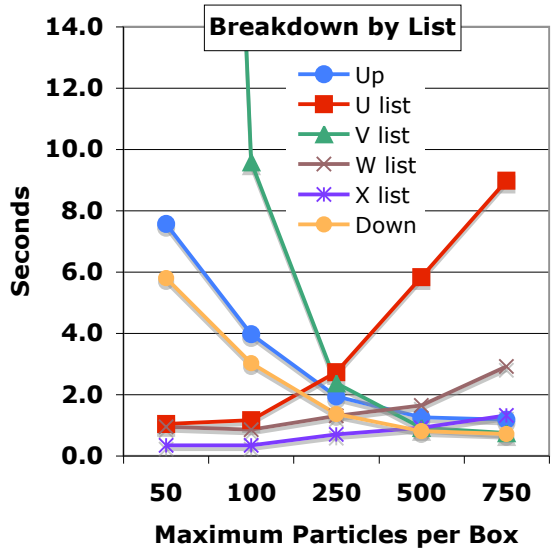
3.4.2 Scalability

Exploiting multicore can be challenging as multiple threads share many resources on a chip like caches, bandwidth, and even floating-point units. Thus, the benefit of



(a)

(b)



(c)

Figure 14: Tuning q, the maximum number of particles per box. Only Nehalem, elliptical distribution data is shown. There is contention between decreasing Up, Down, and V-list time, and increasing U-list time. Note, tree construction time scales like Up, Down, and V-list times.

thread-level parallelism may be limited.

Figure 15 presents the performance scalability by architecture as a function of thread-level parallelism for the double-precision, uniform distribution case (thus, has no W or X list phases). Threads are assigned first to multiple sockets before multiple cores, and multiple cores before simultaneous multithreading (SMT). For clarity, we highlight the SMT region explicitly. On all machines, the 2 thread case represents one thread per socket.

The top figures show overall times for two cases: (i) assuming tree construction before evaluation, and (ii) the asymptotic limit in which the tree is constructed once and can be infinitely reused. (Recall that tree construction was the only kernel not parallelized.) Observe that Nehalem delivers very good scalability to 8 threads at which point HyperThreading provides no further benefit. Unfortunately, the time required for tree construction becomes a substantial fraction of the overall time. Thus, further increases in core count will deliver sublinear scaling due to Amdahl's Law.

In the bottom figures, we report thread-seconds to better visualize the scalability of the code for each phase. A flat line denotes perfect scalability and a positive slope denotes sublinear scaling. The U-list initially dominates the overall evaluation time and delivers very good scalability to 8 cores. This observation is not surprising since this phase has high arithmetic intensity. However, the V-list computations, which initially constitute a small fraction of overall time, show relatively poorer scalability, eventually becoming the bottleneck on Nehalem. The V-list has the lowest arithmetic intensity and is likely suffering from bandwidth contention. The other two kernels show good scalability to 4 cores, but remain a small fraction of the overall time.

Barcelona shows good scalability to four cores (2 per socket) but little thereafter. The bottom figure shows the problem: U-list scalability varies some but is reasonably good, while the V-list scales poorly and eventually constitutes 50% more time than U-list. Given Barcelona's smaller L3 cache and diminished bandwidth, the effects

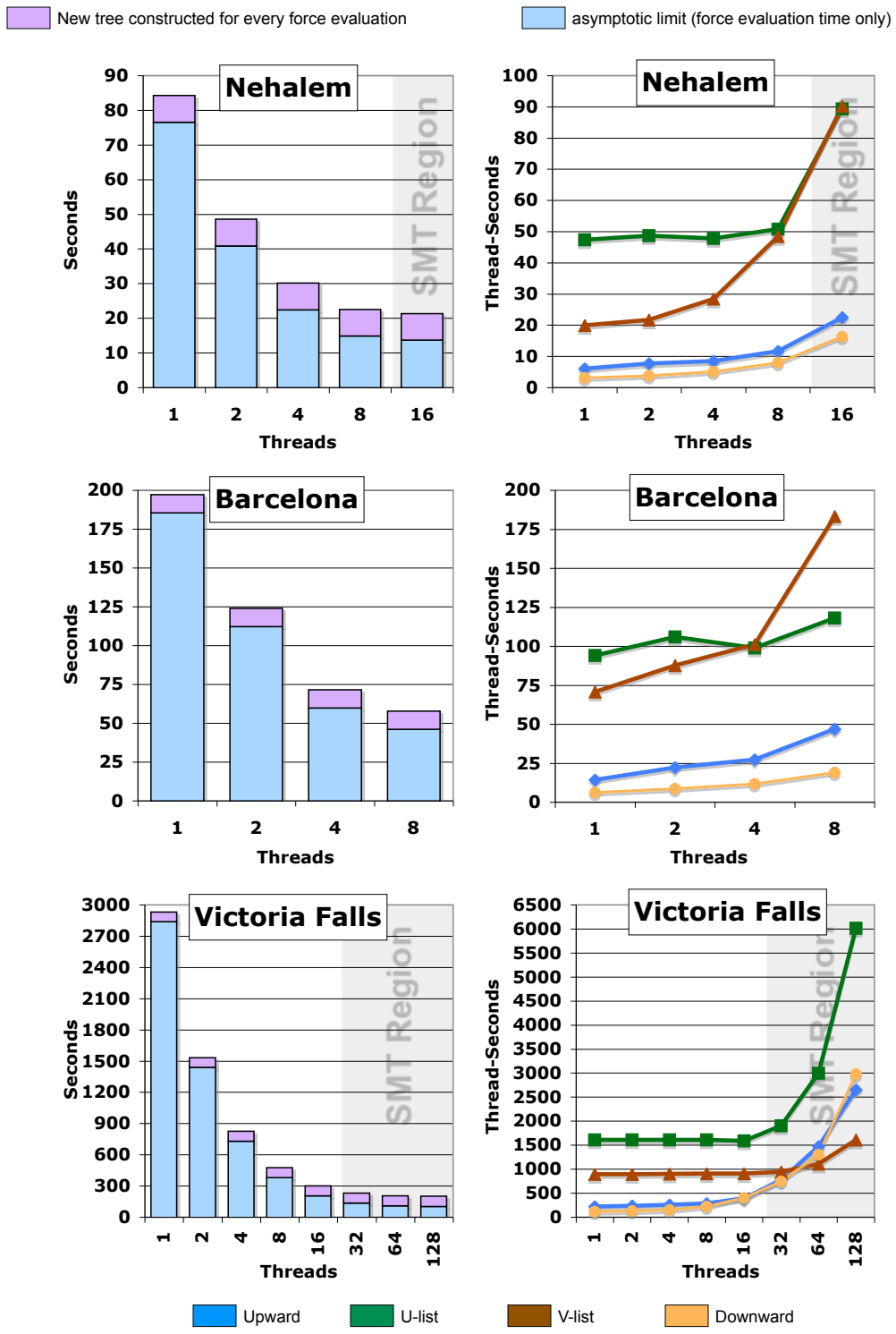


Figure 15: Double-precision, multicore scalability using OpenMP for a uniform particle distribution. Left: time as a function of thread concurrency showing relative time between list evaluations and tree construction. Right: break down of evaluation time by list. Note: "Thread-seconds" is essentially the inverse of GFlop/s/core, so flat lines denote perfect scaling:

seen on Nehalem are only magnified.

Victoria Falls, with its ample memory bandwidth and paltry floating-point capability, scales perfectly to 16 cores (lines are almost perfectly flat). Once again, performance is dominated by the U-list. As multithreading within a core is scaled, the time required for U-list skyrockets. Although, we manage better than a 2× speedup using four threads per core, we see none thereafter.

Figure 16 extends these results to the non-uniform elliptical distribution, which will now include W- and X-list phases. On x86, for the same number of particles, the elliptical time-to-solution is less than the uniform case. This occurs because more interactions can be pruned in the non-uniform case. However, as a consequence, tree construction also becomes a more severe impediment to scalability.

As with the uniform distribution, evaluation time scales well on Nehalem up to 8 threads (one thread per core). Thereafter, it reaches parity with tree construction time. Additional cores or optimization yield only an additional 2× speedup. Also observe that the relative time of each phase changes. U-list time dominates at all concurrencies, and V-list time scales well since the V-lists happen to have fewer boxes for this distribution.

The behavior on Barcelona is similar except, the U-list time scales somewhat more poorly with the elliptical distribution than in the uniform case. Interestingly, most phases scale poorly in the multicore region on Victoria Falls, though U-list time still dominates.

3.4.3 Architectural Comparison

Beyond the substantial differences among architectures in serial speedups, tuning, and multicore scalability, we observe that the raw performance among processors differs considerably as well. Moreover, performance varies dramatically with floating-point precision.

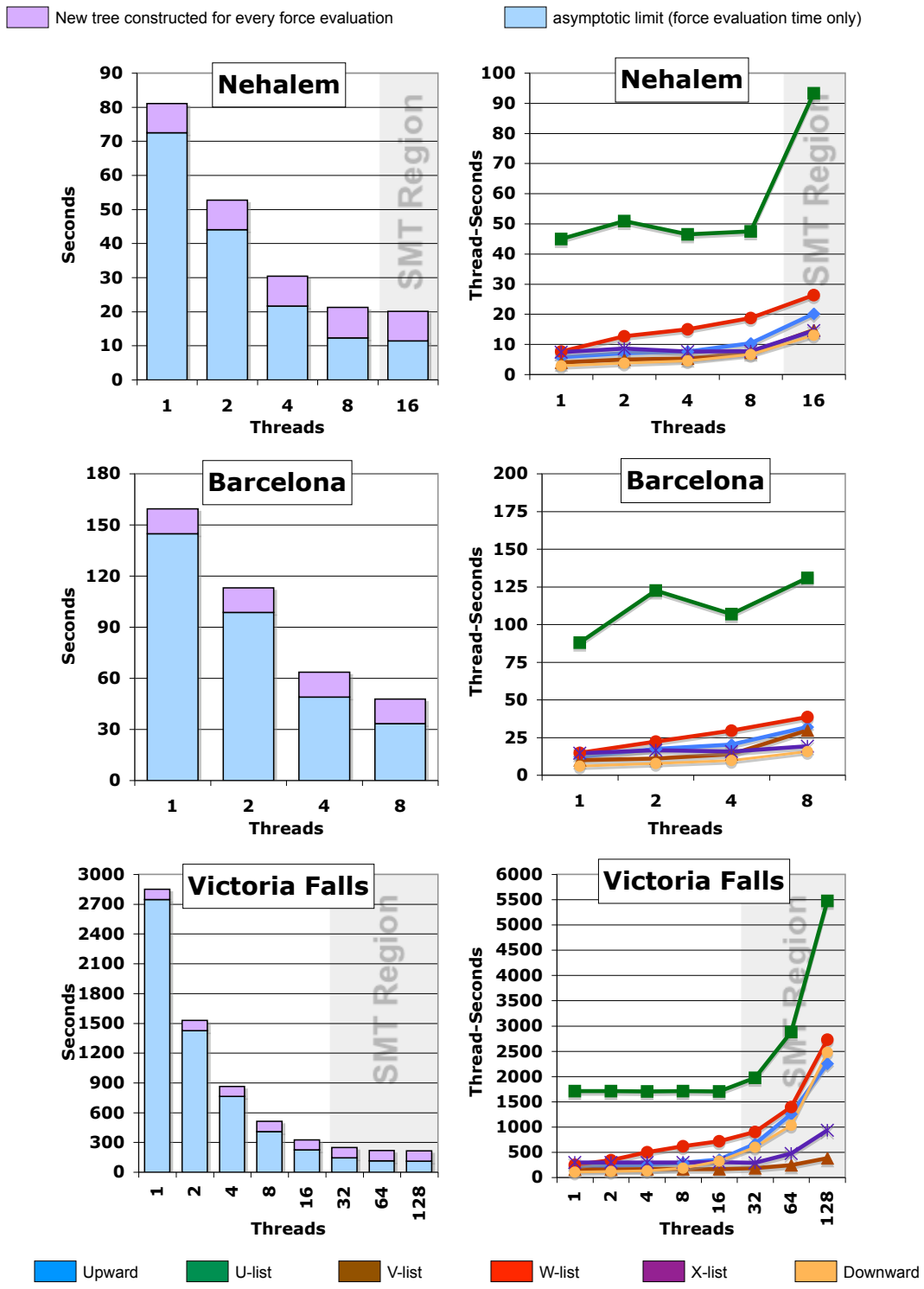


Figure 16: Double-precision, multicore scalability using OpenMP for an elliptical particle distribution. Left: Time as a function of thread concurrency, comparing evaluations and tree construction components. Right: Breakdown of evaluation time by list. Note: "Thread-seconds" is essentially the inverse of GFlop/s/core, so flat lines indicate perfect scaling.

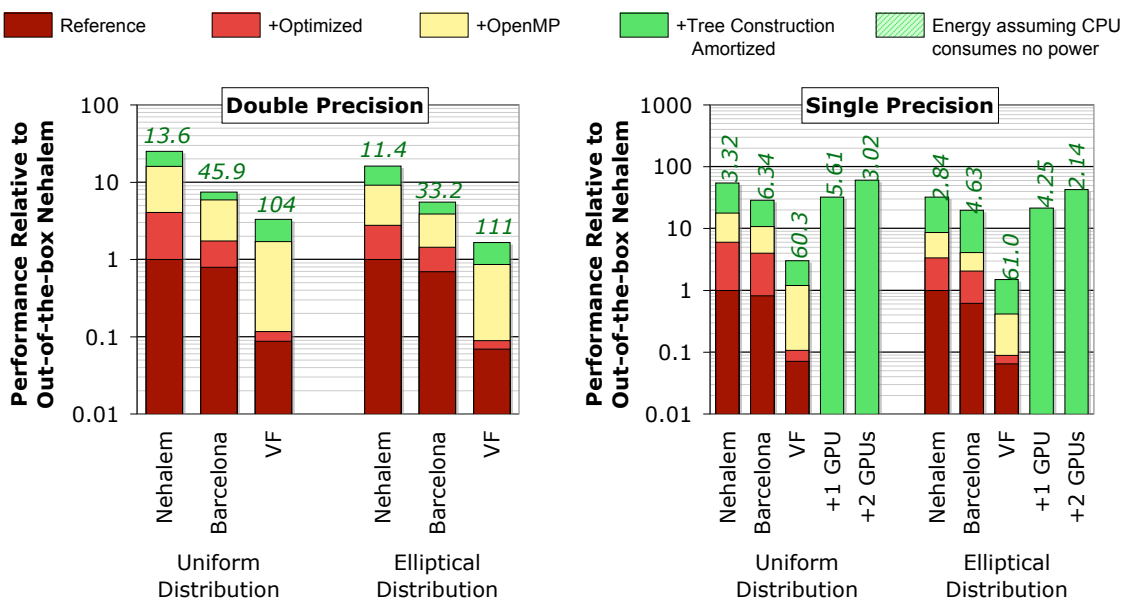


Figure 17: Performance relative to out-of-the-box Nehalem for each distribution. Note, performance is on a log scale. Labels show the final execution time (secs) after all optimizations.

Figure 17 compares the double-precision performance for our three machines for both particle distributions. For each distribution, performance is normalized to the *reference* Nehalem implementation. Initially, Nehalem is only about 25% faster than Barcelona; however, after optimization, parallelization, and tuning, Nehalem is more than 3× faster. Although the initial performance differences are expected given the differences in frequency, the final difference is surprising as Nehalem has comparable peak performance and only about twice the bandwidth. Moreover, and unexpectedly, with only 4.5× the peak flops Nehalem is as much as 10× faster than Victoria Falls. Victoria Falls' small per-thread cache capacity may result in a large number of cache misses. Across the board, we observe the benefit of tree construction amortization is nearly a factor of two.

Unlike the scalar Victoria Falls, x86 processors can more efficiently execute single-precision operations using 4-wide SIMD instructions. GPUs take this approach to the extreme with an 8-wide SIMD-like implementation. To that end, we repeated

all optimizations, parallelization, and tuning for all architectures and distributions in single-precision. We then re-ran the GPU-accelerated code of prior work by Lashuk, et al. [65].

For the GPU comparison, we consider 1 node (2 CPU sockets) with either 1 GPU or 2 GPUs. The GPUs perform all phases except tree construction, with the CPU used only for control and thus largely idle. This experiment allows us to compare not only different flavors of homogenous multicore processors, but also compare them to heterogeneous computers specialized for single-precision arithmetic. Note that the GPU times *include* host-to-device data structure transfer times.

These single-precision results appear in Figure 17. Barcelona saw the most dramatic performance gains of up to 7.2× compared with double-precision. This gain greatly exceeds the 2× increase in either the peak flop rate or operand bandwidth. Nehalem's gains, around 4×, were also surprisingly high. Victoria Falls typically saw much less than a factor of two performance increase, perhaps due to nothing more than the reduction in memory traffic. We attribute this difference to x86's single-precision SIMD advantage, as well as the ability to avoid the Newton-Raphson approximation in favor of one `rsqrtps` (reciprocal square root approximation) instruction without loss of precision.

Although Nehalem's performance advantage over Victoria Falls increased to as much as 21×, its advantage over Barcelona dropped to as little as 1.6× — which is still high given the architectural similarities.

Perhaps the most surprising result is that with optimization, parallelization, and tuning, Nehalem is up to 1.7× faster than one GPU and achieves as much as $\frac{3}{4}\times$ the 2-GPU performance. Where Nehalem's optimal particles per box was less than 250, GPUs typically required 1K to 4K particles per box. In order to attain a comparable time-to-solution, the GPU implementation had to be configured to prioritize the computations it performs exceptionally well — the computationally intense regular

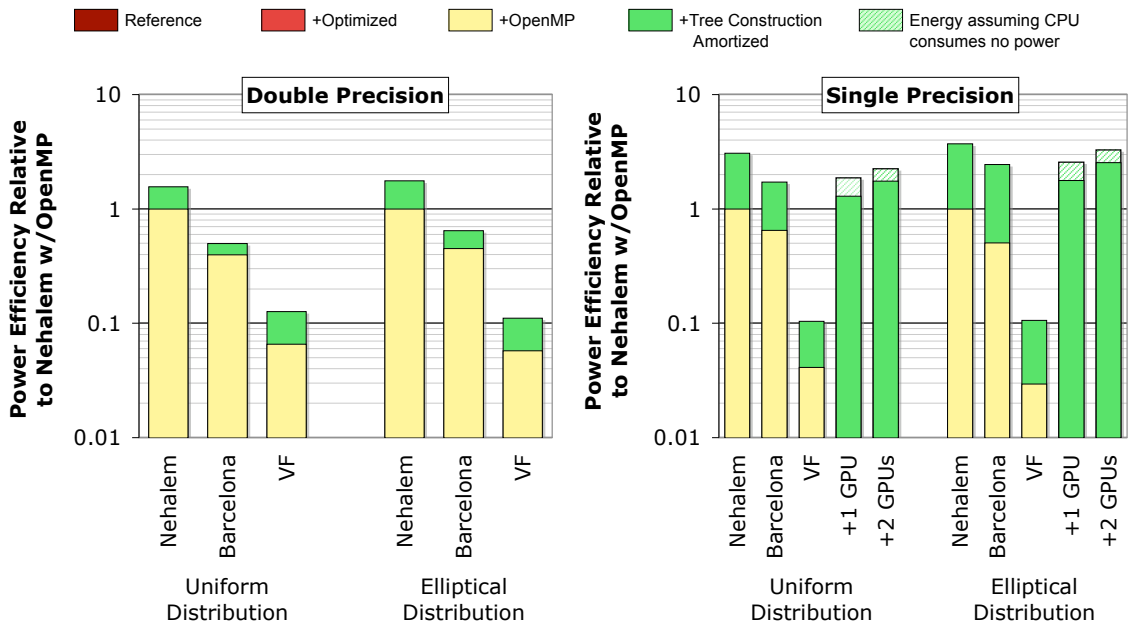


Figure 18: Energy efficiency relative to optimized, parallelized, and tuned Nehalem for each distribution. Note, efficiency is on a log scale. (higher is better).

parallelism found in U-list.

3.4.4 Energy Comparison

We measured power usage using a digital power meter for our three multicore systems. As it was not possible to take measurements on the remote GPU-based system, we include two estimates: peak power (assuming full GPU and full CPU power) and maximum GPU-only power (assuming the CPUs consume zero power). We report the resultant power efficiency relative to parallelized and tuned Nehalem (higher is better). Figure 18 shows this relative energy-efficiency as a function of architecture, distribution, and precision.

Nehalem still manages a sizable energy efficiency over all other CPU architectures, although its energy advantage over Barcelona is less than its performance advantage. Conversely, the FBDIMM-based Victoria Falls consumes at least 66% more power than any other CPU-based machine. As such, for FMM, Nehalem is as much as 35× more energy-efficient than Victoria Falls.

The GPU-based systems, by our estimates, consumes as much as 725W. Thus, Nehalem is as much as 2.37× and 1.76× more energy-efficient than systems accelerated using 1 or 2 GPUs, respectively. Even under the optimistic assumption that the largely idle CPUs consumed no power, then Nehalem's energy-efficiency is still between 0.97× and 1.65×.

3.5 Future Work

Looking forward, we see numerous opportunities.

3.5.1 Asynchronous Parallelism

We relied largely on bulk-synchronous parallelism in optimizing the single node performance of FMM. Although this approach resulted in significant performance improvement and scalability, it also resulted in working sets that did not fit in cache. Alternatively, dataflow and work-queue approaches may mitigate this issue. FMM has abundant fine-grained asynchronous parallelism which we currently do not exploit. As the number of cores per node are only expected to increase in the future, this alternate paradigm would also reduce the time spent in synchronization. We will revisit this topic in detail in Chapter 6.

3.5.2 Parallelizing Tree Construction

Although the tree construction phase constituted only a small fraction of the overall execution time for the baseline code, it becomes a bottleneck once all the phases have been parallelized. Even though for many applications, the tree only needs to be re-built after several iterations, it could still be a serious impediment to overall scalability with increasing number of cores.

3.5.3 Algorithmic Tuning Knob

Optimal value of algorithmic parameters like particles per box will vary not only with architecture, but also optimization and scale of parallelism. Hence, it has always been

empirically tuned at runtime since offline serial tuning is insufficient. Although the optimal particles per box is distribution dependent, often the distribution is known apriori. With this knowledge, one could design a performance model which captures both the algorithmic parameters such as number of particles, distribution, depth of the tree, etc,. and architectural parameters such as cache size, cache line size, memory bandwidth, peak floating point performance, etc,. to predict the optimal algorithmic tuning parameter. We will attempt to design this model in Chapter 5.

3.5.4 Compiler Optimizations

Finally, our manual SIMD transformations and their interaction with data layout was a significant performance win, and most compilers/programming models fail to exploit SIMD, Newton-Raphson approximations, or TLP. We believe that it should be a priority for new compiler and/or programming model efforts since it will benefit a number of scientific applications.

3.6 Summary

Given that single-node multicore performance and power efficiency will be critical to scalability on next-generation extreme scale systems, we believe our extensive study of parallelization, low-level *and* algorithmic tuning at run-time (*i.e.* the input-dependent maximum points per box, q), numerical approximation, and data structure transformations contributes a solid foundation for the execution of FMM on such machines.

Surprisingly, given a roughly comparable implementation effort, a careful multi-core implementation can deliver performance and energy efficiency on par with that of a GPU-based approach, at least for the FMM [65]. We believe this finding is a significant data point in our collective understanding of the strengths and limits of using heterogeneous computers.

CHAPTER IV

GUIDELINES AND SYSTEMATIC APPROACH FOR ALGORITHM ENGINEERING

This study was prompted by the following question: What is the *process* by which one might start with a program and then *systematically* analyze and tune its single-node multisocket multicore performance? Though there are many principles, techniques, and promising tools [8, 17, 46, 48, 52, 66, 71, 72, 78, 88, 91, 93, 97, 104, 105], defining a general and practical process is exceedingly difficult, owing both to the diversity of programs and to the complexity of modern multicore systems. The practitioner (end-user programmer), whose job it is to identify, to understand, and to fix within-node performance bottlenecks, often has little or no guidance on how to

proceed.

This chapter attempts to document just such a process, within the specific context of improving the within-node scalability of a distributed memory implementation of the *fast multiple method (FMM)* [36,49,65,100]. We decompose this process into three stages. In the first stage, we treat the program and hardware as a black box, limiting analysis and tuning to simple modeling and measurement techniques (Section 4.3). In the second stage, we assume more knowledge of the computation and machine, enabling deeper inferences and at least a limited set of code transformation techniques (Section 4.4). In Section 4.5, we describe the final stage where we assume deep knowledge of the code and machine, and therefore not only arrive at the deepest insights, but also apply the most aggressive transformations. At each stage, we show by example what models, insights, hypotheses, and performance-enhancing optimizations—including aggressive asynchronous scheduling and reordering optimizations for explicit locality and bandwidth management—might be discovered and applied.

4.1 Background and Related Work

Implications and limitations. At first glance, the main limitation of this study may be its seemingly narrow focus on the FMM, which raises the natural question of how well this process could generalize. We mitigate this issue in two ways. First, we use the FMM because of its inherent implementation complexity. The FMM consists of many phases of varying computational intensity and memory behavior, thereby making it exhibit many of the general computational characteristics and features of full-scale applications while remaining compact enough to study in detail. Entire program tuning will be more complicated, but we believe algorithm or component-level tuning in the style we describe will be a useful starting point. Secondly, we choose to characterize the overall process by "level of practitioner," where the analysis and optimization techniques that require the least expertise are likely to be the simplest

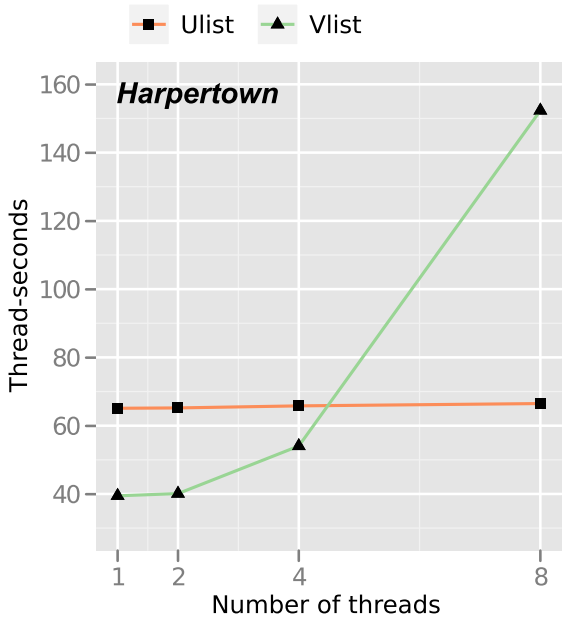


Figure 19: U list and V list parallel scaling (with OpenMP static scheduling) for an 8 million point problem instance (uniform particle distribution in a cube). Unless specified otherwise, this is the default problem instance used in the paper. The y-axis shows parallel cost, $p \cdot T_p$, so flat lines indicate perfect scaling.

to generalize and to apply to other programs; and, more importantly, the easiest to automate and to incorporate into existing performance analysis tools [4, 5, 67, 72–74, 77, 86, 87, 91].

Baseline code. This baseline has extensive single-core performance enhancements already, including manual SIMD vectorization and "smarter" low-level numerical tricks. It is parallelized using OpenMP, applying the basic `omp parallel for` with static scheduling at the outermost loop of each of the six phases of the FMM (Section 2). That is, imagine the FMM being written as the sequence of phases, "upward" followed by "U list" followed by "V list," and so on, where each phase is (roughly) a set of (imperfectly nested) loops; then, our parallel baseline uses `#pragma omp parallel for schedule(static)` at the outermost loop of each phase, respecting dependences. This approach is the minimum level of parallelization we might reasonably expect of any multicore parallelization. Refer to Chapter 3 for details.

A scalability bottleneck. Although this multicore implementation performs

Table 5: Architectural details of parallel systems used in our study. †shared among cores on a socket. ‡shared among 2 cores on a socket.

System	Intel E5405	AMD Opteron 2356	Intel X5550	Intel X7560
Core Architecture	**Harpertown**	**Barcelona**	**Nehalem-EP**	**Nehalem-EX**
Sockets×cores×threads	$2 \times 4 \times 1$	$2 \times 4 \times 1$	$2 \times 4 \times 2$	$4 \times 8 \times 2$
# threads	8	8	16	64
Clock (GHz)	2.00	2.30	2.66	2.27
DP (SP) GFlop/s	64 (128)	73.6 (146.2)	85.33 (170.6)	290 (58)
L1/L2/L3 cache (KB)	32/6144‡/-	64/512/2048†	32/256/8192†	32/256/24576†
DRAM Capacity (GB)	4	16	12	64
Bandwidth (GB/s)	21.33	21.33	51.2	170.6
Compiler	Intel C v11.0	GNU C v4.4.1	Intel C v11.0	Intel C v11.0

well, it also exhibits a scalability bottleneck as core count increases. We can observe this bottleneck as follows. First, recall that the FMM consists of several phases (Section 2). For one common input, a uniform point distribution, the dominant phases are the U list and V list computations. Figure 19 shows, for each of these two phases, its parallel cost, $p \cdot T_p$, where p is the number of threads and T_p is the phase's parallel execution time in seconds, on the Intel Harpertown system. If a phase scales linearly, then its $p \cdot T_p$ is a constant. This behavior holds for the U-list (squares) but not for the V-list (triangles), which becomes the performance bottleneck at four or more cores. Indeed, there is actually a slowdown at eight threads compared to four. Thus, it is in this context that, for the remainder of the chapter, we wish to systematically investigate whether and how to improve scalability.

4.2 Architectural Summary

Table 5 provides a summary of our evaluation architectures. These systems have a range of characteristics of particular interest to multicore performance analysis and optimization.

First, we consider both dual-socket quad-core (8 cores total) and quad-socket oct-core (32 cores) systems. Relative to prior FMM multicore studies, our use of a

32-core multisocket multicore configuration is among (if not the) largest single-node configuration tested to date in terms of core counts [36,38]. Our study sheds light on what algorithmic or architectural features may be necessary to continue scaling the FMM on future systems.

Secondly, the systems span a range of cache capacity and cache sharing configurations. The last-level cache capacities range from as little as 512 KB per core (Barcelona) to as high as 3 MB per core (Harpertown and Nehalem-EX). Thus, we expect to be able to explore the effect of cache capacity. Moreover, the Harpertown system has on each socket two L2 caches, each of which is shared by a pair of cores. We will see how this configuration permits insight into the code.

Thirdly, we consider platforms both with and without non-uniform memory architectures (NUMA). Thus, we can evaluate the effects of data placement for the FMM, as well as strategies for diagnosing and improving NUMA-related performance issues.

Finally, we consider both simultaneous multithreading (SMT) and non-SMT processors. Thus, we will be able to see the extent to which existing x86 SMT designs provide the right kind of support for an FMM style computation.

4.3 Stage I: Diagnosis and Initial Tuning

The process of improving the V-list scaling problem in the baseline code (Chapter 3) begins with an exploratory analysis. Initially, we assume

- high-level knowledge of the application and algorithm and full access to the source code, but with perhaps limited knowledge of the code;
- a basic understanding of the architecture, such as the properties listed in Table 5, including whether the system has a uniform or non-uniform memory architecture;
- knowledge of several "rules-of-thumb" analytical techniques, such as Amdahl's

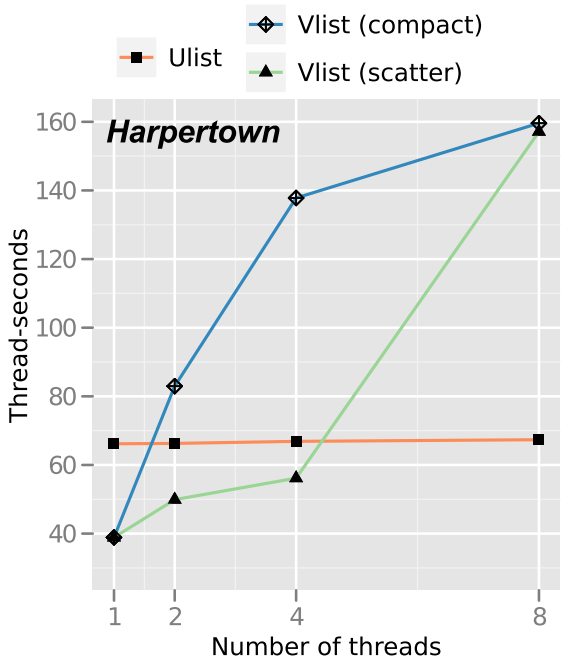


Figure 20: U list and V list performance scaling on Harpertown with compact and scatter thread pinning schemes.

Law, Little's Law, notions of arithmetic or computational intensity, and the roofline model [10, 97];

- full access to all of the available performance measurement, analysis, and compilation tools, such as VTune, HPCToolkit, TAU, and Open|SpeedShop, among others [4, 5, 67, 86, 91].

4.3.1 First Measurements and Analysis

Using standard tools, such as VTune, we profile the code to make the initial observation of poor V-list scaling, as outlined in Section 4.1 and Figure 19. Again through performance counter measurement, we estimate the computational or arithmetic intensity (ratio of flops to *main memory* bytes) of these two phases, finding that the U-list is compute-bound while the V-list is memory-bound. Thus, one would expect U-list to scale and suspect memory system contention as the culprit for poor V-list scaling. We set out to discover the cause.

First, we ask whether there is any load imbalance, given our initial choice of static scheduling. Using performance tools, we check the per-thread time and determine that each thread takes nearly exactly the same amount of time. Thus, we rule out load imbalance as a problem.

Given that the V-list is memory bound, we then ask what part of the memory system the V-list may be stressing as threads increase. We recognize that our platform, a dual-socket quad-core Intel Harpertown, has pairs of cores sharing the L2 cache. To gain some additional insight and intuition into what effect this cache architecture has, we conduct two experiments. Importantly, these experiments are simple as neither requires modifying the code.

In the first experiment, we use the OpenMP thread affinity option to "scatter" consecutively numbered threads first across sockets, then across cores not sharing L2, and then within pairs sharing L2. We observe precisely Figure 20. In particular, the code only achieves 3× scaling on four cores, when there is no cache sharing, suggesting that bandwidth contention is at least one specific issue. We also note that the precipitous drop occurs at 8 threads when pairs of threads share the L2 cache. This observation suggests a second experiment, in which we change the affinity policy to "compact assignment," where consecutively numbered threads are first assigned to cores sharing L2, then within the core, then across sockets. If L2 cache sharing is an issue, then we expect to see the scalability issue with even just 2 threads, since they will under compact assignment be mapped to cores that share L2. Indeed, we make exactly this observation, as shown in Figure 20. Thus, we conclude that in addition to bandwidth contention when there is no cache sharing, that there is also the potential for cache contention, most likely due either to capacity or conflict issues. These observations do not yet suggest how to improve V-list scaling, but they do lead to an interesting inference, described next.

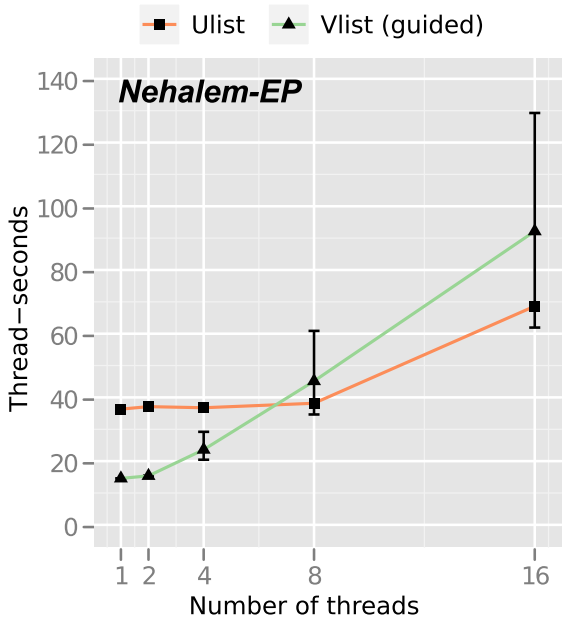


Figure 21: *U* list and *V* list parallel scaling on Nehalem-EP. The black error lines indicate variation in execution time between the fastest and slowest thread with static scheduling.

4.3.2 Tuning Exploration and Results

Since cache sharing leads to poor scalability, there may be some natural affinity of threads to *independent* pieces of data. That is, if two threads operate on independent data, we would expect no improvement from assigning them to cores that share a cache. We then ask the following question: although Harpertown platform uses a uniform memory architecture, what would happen if we moved to a *NUMA* one? Our observations about affinity imply that we will see even more performance problems there, since data placement will be critical on a NUMA platform. Thus, even though we are not yet ready to fix the V-list scalability problem, we can "future-proof" our code for a NUMA architecture using either of two potential optimizations, one which does not require modifying the code substantially, and one which involves some modest modifications.

Guided scheduling. We run our code on a Nehalem-EP system, which has a NUMA architecture. Whereas we previously observed no load imbalance issues, here

we observe one! Figure 21 shows the issue, where, for each thread configuration, we also show both the maximum *and* the minimum execution time of any V-list thread. The ratio between these two is as high as 2×. Checking the flop counts, we see they are roughly the same across threads. As such, the imbalance must be coming from an increase in the cost of some other operation, and for a memory-bound computation, that culprit is most likely the memory operations themselves.

One simple way to cope with any load imbalance is to switch the work sharing scheduling policy, here from static to *guided* scheduling.[1] This "fix" does not address the root cause of improper data placement but does mitigate the problem somewhat, as seen in Figure 21.

NUMA optimizations. Though guided scheduling helped on Nehalem-EP, it still does not address the fundamental problem of data placement. To do so requires modifying the code. In particular, to exploit NUMA, we use the first-touch page allocation policy of the operating system to also parallelize the data initialization loops, a still relatively non-invasive change. The results are shown in Figure 22. Indeed, performance improves by 1.4–3.2×. However, on both Harpertown and Nehalem-EP, the fundamental V-list scalability issue remains.

4.4 Stage II: Intermediate-level Tuning

In Stage II, we assume more knowledge of the computation and machine, thereby enabling deeper inferences and more complex performance-enhancing transformations. In particular, we assume in addition to Stage I,

- a more detailed understanding of the processor microarchitecture;
- a deeper understanding of the application's data and control dependency structure;

[1] Although we chose guided, one can try others.

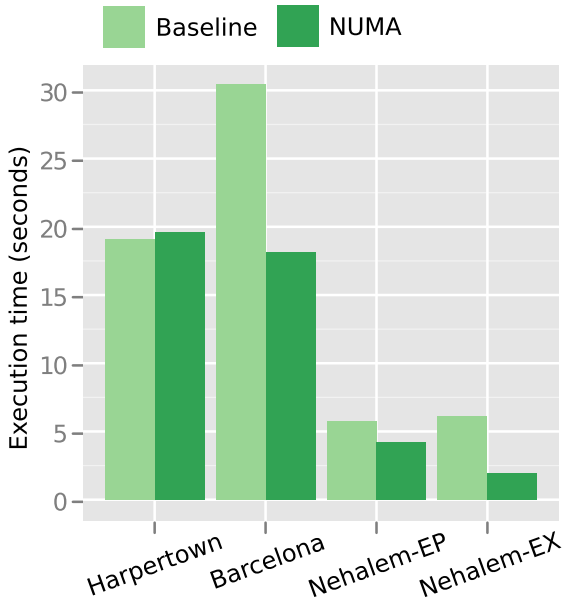


Figure 22: The impact of NUMA-aware allocation: V list performance at full concurrency and explicit static thread scheduling.

- consequently, a willingness to try some "riskier" optimization strategies.

Based on the Stage I analysis, we know that the U-list computation is compute-bound and nearly perfectly scalable, whereas the V-list is memory-bound. Although the NUMA optimizations enhanced performance significantly, there could still be room for improvement.

4.4.1 Analysis and Implied Optimizations

The simplest way to avoid the scalability "cliff" is to use at most the number of threads for which the computation still scales, provided the remaining threads can scalably do other useful work. Indeed, in Stage II, being more informed about the code, we discover that the U- and V-list computations are actually independent of one another and may therefore run concurrently. The trade-off is that the code, which was parallelized in the straightforward bulk-synchronous style from its sequential counterpart (Chapter 3), will require more extensive modification than that of the NUMA-aware optimizations.

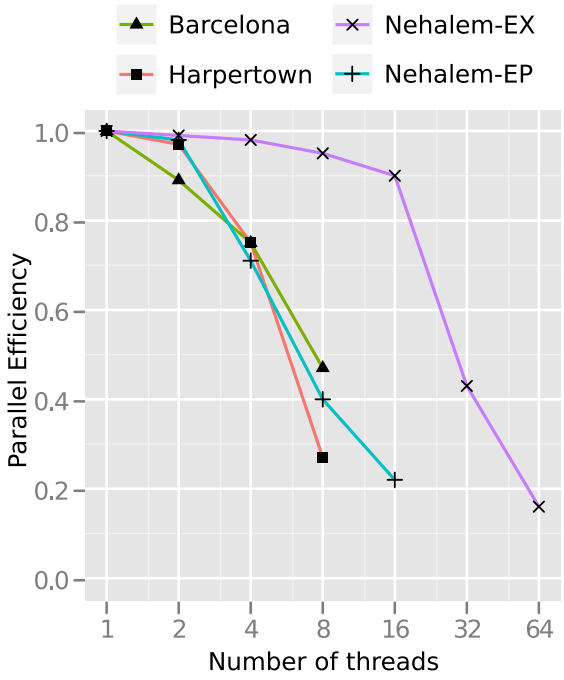


Figure 23: The parallel efficiency of V list computation after NUMA-aware memory allocation on various platforms.

In the best case, we make a quick back-of-the-envelope estimate of the best possible improvement from this technique. In the current code, let T_U be the U-list time and T_V the V-list time. If we can successfully hide the entire cost of the V-list, we might achieve a $\frac{T_U+T_V}{\max\{T_U,T_V\}} \leq 2$ speedup. For the four platforms featured in this study, we collect this data and estimate that a 1.56–1.96× speedup might be possible.

Under this assumption, we generate two ideas for exploiting this concurrency through *asynchronous-parallel* execution.

Idea 1: Mixed phase execution. We consider specific schedules of the U- and V-list components in which some threads operate on V-list work, up to the V-list's scalability limit as seen from Figure 23; the remaining threads work on the U-list, which scales well on any number of cores. The degree of work partitioning can be a runtime parameter subject to (automated) tuning.

Idea 2: Explicit SMT assignment. Since the arithmetic intensities of U- and V-list components differ markedly, we hypothesize that we might profitably exploit the

simultaneous multithreading (SMT) features of the Nehalem platforms. The intuition is simply that the U-list computations will make heavy use of the computational units, whereas the V-list is mostly idle waiting for memory. Thus, concurrently executing U- and V-list components on the same core could in principle benefit from SMT, assuming sufficient within-core functional units.

4.4.2 Results

We apply these ideas to the NUMA-aware code of Section 4.3. In Figure 24, we summarize the improvement over the NUMA-aware code of the preceding section. Recall that the best possible performance according to our back-of-the-envelope estimate, shown here the highest (green) bar, and ranges from 1.56–1.96×.

For mixed phase execution (blue bar), we see significant improvements of 73% and 60% on Harpertown and Nehalem-EX, respectively. The benefit is somewhat smaller ($\approx 1.2\times$) though still appreciable on Barcelona and Nehalem-EP. This observation lends support to the notion of investing in programming, compile-time, and run-time mechanisms that can readily facilitate this style of execution.

On the two SMT-enabled Nehalem platforms, our intuition about explicit SMT assignment does in fact lead to a pay-off that nearly matches that of mixed phase execution, as shown by the red bars in Figure 24. Still, the achieved performance falls short of our estimated ideal in both cases. In fact, although the V-list is memory bound, it is evidently not strictly idle and is likely still competing with the co-scheduled U-list thread for processor-core functional units. We expect that further investigation of the U- and V-list resource requirements could help inform the balance of functional units in future SMT designs.

4.5 Stage III: Advanced Tuning, Algorithm Redesign

We now detail some optimizations that one cannot recommend or implement without an in-depth analysis of the computation. We envision that practitioners at this stage

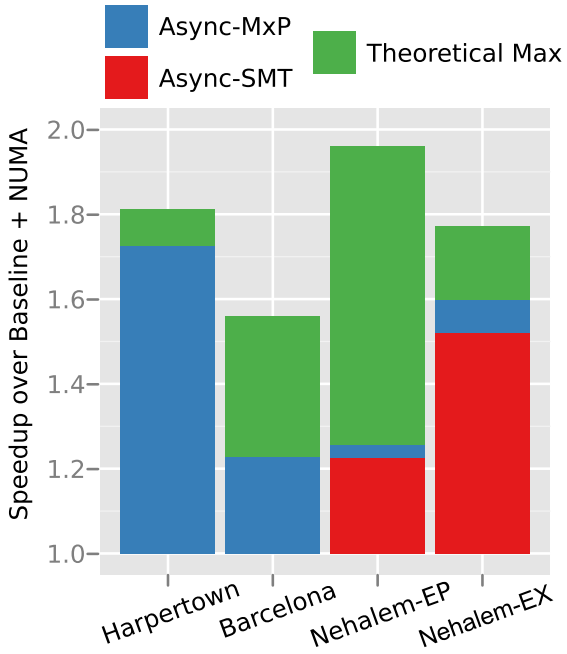


Figure 24: The performance improvement achieved using asynchronous implementations (red indicates explicit SMT assignment and blue depicts additional improvement with mixed phase execution). The green bar is the an indicator of the theoretical maximum possible speedup (based on U list and V list execution times at full concurrency).

of the performance-tuning hierarchy have a detailed understanding of the application, and are also experts in programming multicore architectures.

In the context of our KIFMM code tuning, we attempt to model the computation, analytically validate and reason about the VTune performance counter data (collected in Stage I, as discussed in Section 4.3), and finally evaluate if we can achieve a further performance by investing effort in a complete redesign or restructuring of the algorithm.

4.5.1 Modeling U list and V list computations

In Section 4.3, we classified U list and V list as *compute-intensive* and *memory-intensive* respectively, based on performance counter data gathered for a few problem instances. We now describe our methodology of source code inspection and analytic estimation to determine the floating point computation performed and the memory

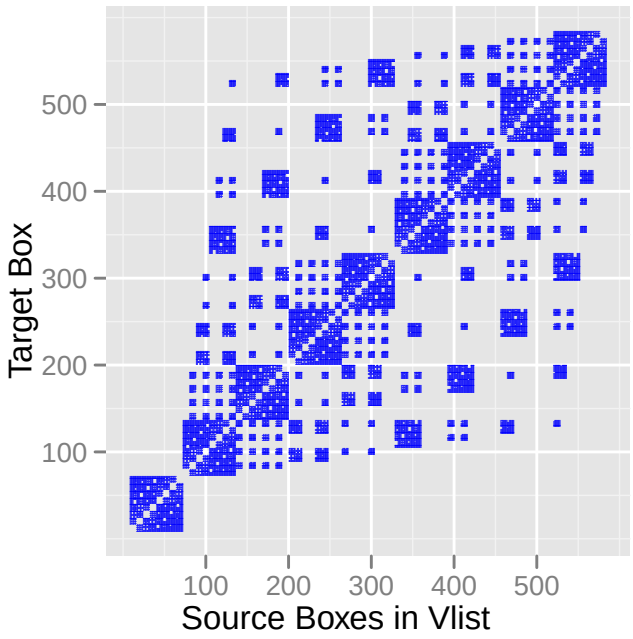


Figure 25: A plot of the sparsity pattern observed in the V list target box–source box implicit mapping matrix (200K points, uniform random distribution).

traffic sustained, and validate performance counter data. While the analyses here may not be as generally applicable as the ones discussed in the earlier sections, we hope that the following outline of efforts would give the reader a sense of the complexity involved at this stage.

U **list computation.** From Table 2 and from prior analysis by Ying et al. [100], we have that U list's complexity is $\mathcal{O}(27Nq)$. We provide a short explanation for this term. Asymptotically, for a uniform random distribution in a three-dimensional space and rectilinear discretization, we can assume that each target box has 27 neighbors in its U list (we can easily visualize this by counting the number of unit cubes in a $3\times3\times3$ cube). Thus, we perform a total of $27q^2$ pairwise kernel computations per target box. We have roughly $M = N/q$ such boxes, and so we get to the cost term of $27Nq$. This term ignores the fact that in our experiments, the cube is of a finite size. Accounting for the boundary cases (boxes on the surface), we refine our counting argument to come up with a better approximation: $(3M^{1/3}-2)^3q$. Next, we inspect our source code

and generated assembly instructions to approximately count the number of floating point operations performed, which would be a multiplicative factor to this pairwise box computations. In our case, we count 19 instructions, which we expect to take roughly 17 cycles after accounting for possible instruction level parallelism (ILP), instruction latencies, and throughputs [1]. Our count agrees very closely with the statistically-sampled VTune performance numbers for Harpertown. We could also verify that the execution time is indeed dominated by floating point computations. Next, the expected number of memory words transferred can be similarly determined based on the average U list neighbor count, and it closely correlates with the observed bus transactions (memory) value. This corresponds to a very small fraction of the total execution time, and hence any further memory optimizations will only be of limited benefit. Our inspection of the inner loop and manual counting of floating point instructions gives us an estimate of any further performance improvements that could be realized for U list computation, either with improved pseudo-code or better architectural support for the higher latency instructions.

V list computation. Based on scalability analysis in prior sections, we suspect that V list performance is likely hindered by cache capacity misses and saturation of available memory bandwidth. Another indicator of this problem is the relative increase in performance counter values as we increase thread-level concurrency, due primarily to bus transactions (memory) and last-level cache misses. Thus, we now attempt to estimate the bus transaction counts and cache working set requirement per thread in terms of FMM problem parameters. Recall that V list's asymptotic computational complexity is given by $O(Mp^{3/2} \log p + 189Mp^{3/2})$, where M is the number of target boxes and p denotes the number of expansion coefficients. The first term corresponds to computation of FFT's to transform a convolution into multiple element-wise vector products in Fourier space. The 189 multiplicative constant denotes the asymptotic average V list size for a target box, assuming a uniform random

distribution ($6^3 - 3^3$; visualize by counting the number of unit cubes within a $6 \times 6 \times 6$ cube, minus the V list box count). In practice, we observe that $p^{3/2} \log p$ is an over-estimate for each FFT computation. We link to tuned FFTW [47] routines, and this step only constitutes 4% of the total V list execution time.

The vector size after FFTs is determined by the number of expansion coefficients, which is chosen to set the accuracy desired for the KIFMM summations. For four digits of accuracy, the FFT size is 640, and for six-digit accuracy, this is typically 2016. Our code is currently configured for six digits of accuracy. The element-wise vector products are performed between source boxes (i.e., the vectors corresponding to source boxes after FFTs) and vectors chosen from a set of what are referred to as *translation vectors*. The appropriate translation vector chosen for a particular source-target combination depends, among other aspects, on the depth of the source and target boxes under consideration. To summarize, given six digits of precision, for each target box vector, we perform approximately 189 element-wise vector products (source box – translation vector combinations, each of size 2016 elements) in the Fourier space and update then the target box vector. This gives us a better estimate of the computation than the $p^{3/2}$ term in the asymptotic worst-case analysis. The performance counter data accurately matches our estimate.

The count for memory references is slightly more involved. Figure 25 denotes the sparsity pattern of the source boxes that appear in the V list of each target box, for a small problem instance of 200K random particles in a cube. We access target boxes in the order of their identifier (i.e., 1 to N), and the source box FFTs for each target box are contiguously stored in memory. Thus, in the implicit matrix shown in the figure, the memory accesses are row-wise. Each dot in the figure also corresponds to a vector of size 2016. We observe substantial temporal reuse (note the blocked structures along the diagonal of the matrix). Based on geometry arguments, we can provide a weak upper bound estimate of 86% reuse between every consecutive pair of

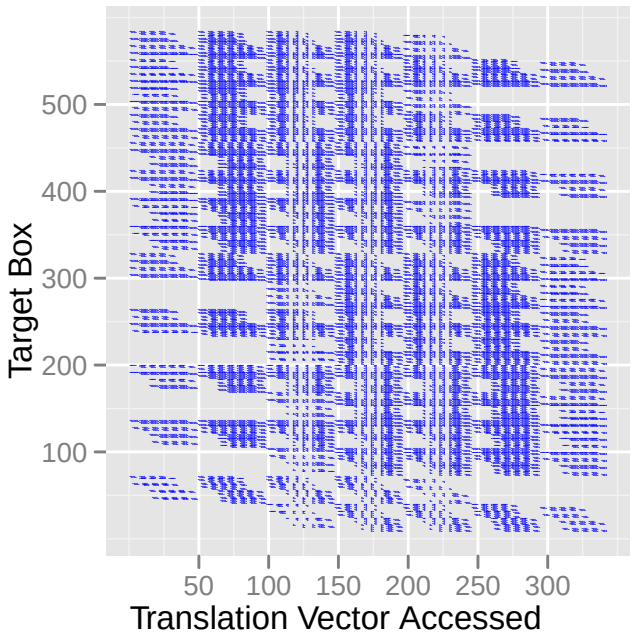


Figure 26: A plot indicating translation vector accesses in V list computation (200K points, uniform random distribution).

target boxes. (The proof is based on considering unit stride translations of a $5 \times 5 \times 5$ cube in all directions and estimating overlap.) Also, on closer inspection, we see that this matrix pattern is reminiscent of a stencil-like traversal, with approximately 189 points per row of the matrix. We further refine our reuse estimates by manually computing the pairwise overlap. Next, we empirically estimate the reuse distance of every source box as a fraction of the total number of boxes. This number further refines our visual estimate that roughly half the boxes are lumped together along the diagonal, and other analytic arguments based on the geometric structure. We thus conclude that the source box accesses are bandwidth-bound asymptotically, but with a reuse pattern along the diagonal. If we are able to fit these blocked structures (roughly half the average number of boxes per target box, per row, 2016 doubles each) in the last-level cache, we can substantially reduce memory traffic.

The other source of memory references is accesses to the translation vectors. Note that the translation vector count is independent of the number of particles or boxes. In

this case, we have roughly 343 translation vectors, and for each target box, about 189 unique ones are accessed. As seen in Figure 26, this pattern is much more complicated to analyze. However, note that the degree of reuse is considerably higher in this case, given that we choose 189 out of 343 for every iteration. This suggests that one may require a minimal cache size for ensuring all of the translation vectors stay close to the processor, and to avoid fetches from memory for every target box evaluation.

For the 8 million point problem on Harpertown, we are able to match the performance counter data for memory bus transactions by up to 99%, essentially by combining both of the preceding arguments and refining our counting estimates to consider the boundary conditions (the average source boxes turns out to be 161 rather than 189 for the 8 million point case, and an even smaller 90 for the 200K-point instance). Further, we can determine precisely the cause for a number of empirical observations in prior sections. We observed an increase in execution time on the Harpertown system when two concurrently-running threads were sharing the L2 cache, due to insufficient per-thread cache size for exploiting the available blocked structure pattern for the source vectors. The bandwidth-bound nature of streaming source boxes is also apparent in all the performance results. For the 8 million-point problem size, we estimate a per-thread cache working set of 5.2MB for the source boxes and translation vectors. This explains the relatively-good scaling observed on the Nehalem-EX system.

4.5.2 Optimization and Results

Based on the preceding analysis, we now suggest algorithmic improvements. One approach to reduce capacity misses is to increase temporal locality, if any. We note that the relatively high access rate of the translation vectors (Figure 26) is a promising avenue to pursue. Next, we observe that threads are executing concurrently rather than cooperatively when streaming the source boxes, and this leads to a p-way pressure on

cache utilization.

Based on these observations, we experiment with two ideas for further improving performance.

Idea 1: Blocking of the translation vector. A common optimization in sparse matrix computations is to block a matrix (i.e., stride through it in regular chunks), such that the blocks fit in cache. We could block the translation matrix (essentially the virtual matrix in Figure 26) since it is reused by all the threads. The blocking size can be a runtime parameter or a parameter that can be determined based on the last-level cache size available.

Idea 2: Hybrid scheduling. Since the current static scheduling strategy of assigning fixed chunks of target box computation to each thread results in a loss of temporal locality between consecutive target boxes, we devise an improvement to exploit this locality. Instead of static scheduling throughout, we employ hybrid scheduling. That is, we perform static scheduling across different sockets, but within a socket, if the threads are sharing the cache, we switch to cyclic scheduling. This way, we avoid fetching potentially entirely different source box vectors into the cache, thus alleviating capacity misses.

We apply these techniques individually to the NUMA-aware code of Section 4.3. Figure 28 summarizes the performance when we block the translation vectors across all four platforms considered. We achieve significant speedups from 2.1× to 4× when running at full concurrency. To fit all the translation vectors and all the source boxes in one target box's V-list in cache, we would require 5.5 MB and 2.6 MB respectively. This amount would consume the entire L3 cache of a Nehalem-EP socket (8 MB). Adding an additional thread doubles contention whereas blocking significantly mitigates it.

Similarly, we can first determine whether hybrid scheduling would provide a performance boost before going on to implement it. On the Nehalem-EP, when running

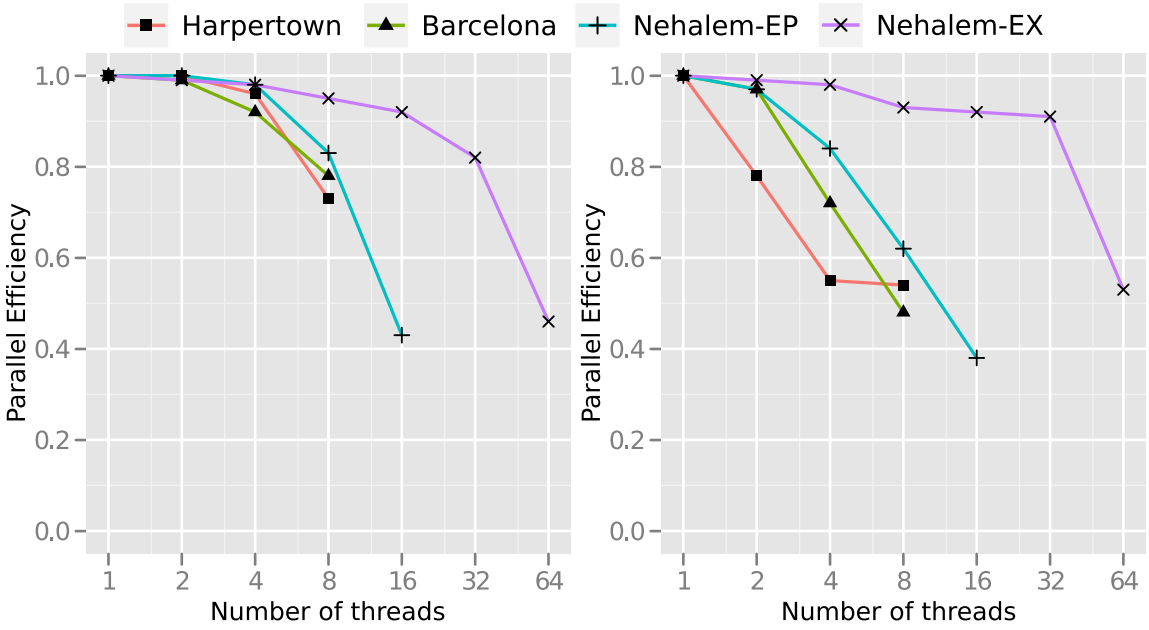


Figure 27: V list efficiency plots for the optimized implementations with blocking (left) and hybrid scheduling (right) optimizations.

at full concurrency, the working set size for the source boxes alone would be 2.6×8 MB per socket. This again exceeds the L3 cache capacity and would result in capacity and conflict misses. Hence we expect this scheduling scheme to help. Figure 28 summarizes the benefit across all the platforms. Performance improves by 1.7× to 3.4×, except on Barcelona. This can be explained by the significantly smaller cache size, where even the working set of a single thread does not fit entirely in cache. Across the board, we observe better scalability after applying the above optimizations as seen from Figure 27 compared to Figure 23.

4.6 Future Work

Our case study for the processs is FMM and despite the fact that FMM has multiple phases which have different compute and memory characteristics, it does not test the generality of the approach. Repeating the process to improve the performance and scalability of another algorithm would test its applicability beyond FMM.

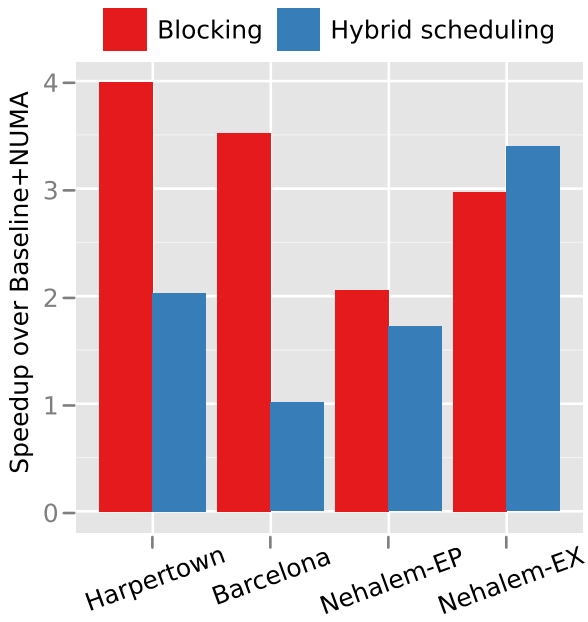


Figure 28: V list parallel performance on all platforms with blocking and hybrid scheduling optimizations at full concurrency.

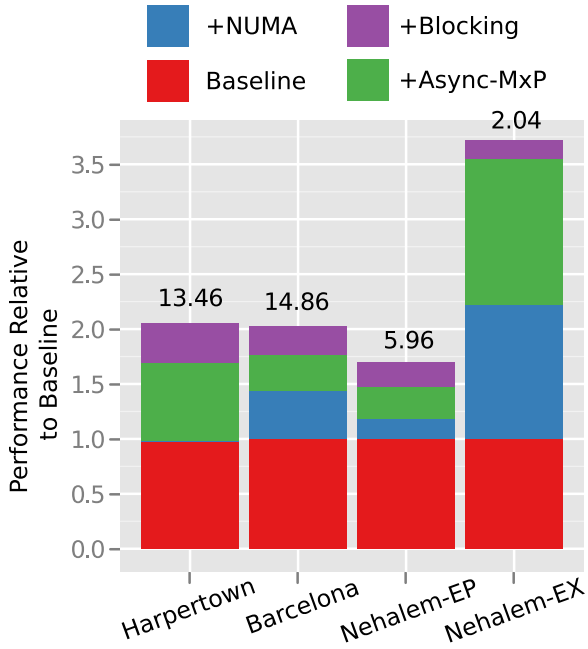


Figure 29: A summary chart depicting the impact of performance optimizations on each architecture, with speedup given with respect to the baseline implementation (U-list + V-list) at full concurrency. Labels show the final execution time (secs) after all optimizations.

4.7 Summary

Our study builds on the considerable collective wisdom on the principles, techniques, and tools needed for enhancing multicore scalability [8, 17, 46, 52, 66, 71, 72, 78, 88, 88, 91, 93, 97]. Indeed, it is inspired by the detailed examples of memory hierarchy transformations for regular computations (matrix multiply) on single-core systems [48, 104, 105]. In our case, each transformation is motivated by a model, measurement, and/or analysis, resulting in the overall sequence of performance improvements over a state-of-the-art baseline as shown in Figure 29. In terms of within-node FMM performance, our code now represents the new state-of-the-art.

Although our decomposition into stages may seem simplistic, we nevertheless believe that from the perspective of building tools, it might be useful to try to decompose tuning into processes that vary by the amount of information and aggressiveness of program transformation permitted.

Our "Stage I" and "Stage II" analyses could in principle be incorporated into "performance wizards" and "what-if scenarios" for existing or new analysis, tuning, and prediction tools. Except for the code-specific NUMA transformation, the Stage I analyses required only measurement and exploration of tuning parameters that the programming model exposed directly (e.g., choice of OpenMP schedule). As for the NUMA transformation, even if it could not be fully automated, it may be possible that a "smart" performance analysis tool could at least suggest it based on program observation.

The Stage II transformations, which apply concurrent or "asynchronous-parallel" scheduling and execution, is another example of a trend in multicore performance exploitation [28,32,33]. As a technique for effectively managing bandwidth and locality, as well as exploiting SMT, this approach will become only more important over time, and warrant additional research and development on programming models and systems support for this style of execution.

Stage III constitutes the deepest and most FMM-specific analyses and optimizations. Nevertheless, it also shows the feasibility and utility of analytical modeling, dependency analysis, and transformation for indirect and semi-irregular programs like the FMM. Such programs remain extremely challenging cases.

CHAPTER V

COMMUNICATION ANALYSIS/MODEL FOR FMM

This chapter presents a new communication-centric algorithmic analysis of the FMM. The FMM is widely believed to be among the most important algorithms for a variety of current and future scientific simulations in electromagnetic, fluid, and gravitational phenomena, among others [24], and is hypothesized to be of increasing importance at exascale [102]. Our analysis helps to develop a deeper understanding of the FMM's performance and scaling characteristics as they relate to key features of the underlying architecture, such as peak processor speed, memory bandwidth, and memory hierarchy design.

Our new analysis enables precise answers to three critical questions about the FMM as we approach exascale. (1) Assuming an exascale node will be a hybrid system comprising CPUs and GPUs, what is the best work partitioning/scheduling strategy? (2) Given technology trends, will the FMM—largely compute-bound on

today's systems—still be compute-bound at exascale? (3) Can we statically before runtime determine the optimal value of the algorithmic tuning parameter?

Section 5.2 presents lower bounds on cache complexity for key phases of the FMM and uses these bounds to derive analytical performance models. Our novel performance model is the first for FMM to capture the precise relationship among the FMM's algorithmic tuning knobs—both of the data structure and the desired accuracy—and key architectural parameters, making it practical and usable. Using the performance model we developed for the two systems (CPUs and GPUs), we determine the optimal work partitioning scheme on heterogeneous systems (Section 5.3). Section 5.4 uses the performance model to derive the optimal value of the algorithmic tuning parameter. Lastly in Section 8.1, we use these performance models to make predictions about FMM's scalability on possible exascale system configurations, based on current technology trends.

5.1 Background and Related Work

There is extensive literature on the FMM and its parallelization, from fast shared memory implementations to highly scalable distributed memory codes, to GPU-based accelerated implementations [7, 39, 45, 50, 53, 54, 58, 63, 65, 76, 82, 96, 99]. Within-node, the current state-of-the-art FMM implementation is our own prior multicore code [35, 36, 65, 83]. Therefore, we take this implementation as the baseline CPU code for our study.

For hybrid-FMM on CPU-GPU systems, Hu et al. [58] discuss a work partitioning scheme where all computation on the leaf nodes is done on the GPU and everything else is handled by the CPU. Moreover, this partitioning scheme is fixed for all input sizes and desired accuracy. To our knowledge, our work is the first to discuss a systematic model-based work division, which results in optimal scheduling decisions that run *counter* to the Hu et al. scheme.

5.2 Analytical Performance Model for FMM

In this section, we present lower bounds for the two key phases of FMM. We assume a uniform random distribution of source and target points for the rest of the analysis.

We assume a simple two-level memory hierarchy, consisting of an infinite memory and a cache of size Z. Data is transferred between the memory and cache in cache lines of size L.

5.2.1 Upward step

Algorithm 1 Upward computation.

Input: Source leaf boxes $\mathcal{L}$; Source non-leaf boxes $\mathcal{N}$; Source density vectors s_α of size q, $\alpha \in \mathcal{L} \cup \mathcal{N}$;

Output: Upward densities u_α of size p, $\alpha \in \mathcal{L} \cup \mathcal{N}$.

1: Pre-compute source to upward translation matrix S_α
2: **for all** $\alpha \in \mathcal{L}$ **do**
3: Evaluate kernel matrix $K(\gamma, \alpha)$
4: $f_\alpha \leftarrow K(\gamma, \alpha) s_\alpha$
5: Solve for u_α; $S_\alpha u_\alpha \leftarrow f_\alpha$
6: Pre-compute upward to upward translation matrix U_α
7: **for all** $\alpha \in \mathcal{N}$ **do**
8: $f_\alpha \leftarrow f_\alpha + U_\alpha u_{C(\alpha)}$
9: Solve for u_α; $S_\alpha u_\alpha \leftarrow f_\alpha$

The Upward step of KIFMM consists of two phases:

1. Source to Multipole (S2M): compute multipole expansions (ME) of the source boxes at the leaf level. The ME captures the effect of the particles within a box B at a distant point.

2. Multipole to Multipole (M2M): perform a postorder traversal of the tree, combining child ME's to compute the ME for the parent.

Algorithm 1 outlines the upward computation. Given b leaf boxes, p coefficients, and q points per leaf box, the asymptotic complexity of S2M is $\mathcal{O}(np + bp^2)$ and M2M

stage is $\mathcal{O}(bp^2)$. The time to compute is the total number of floating point operations divided by the peak computation throughput C_0, in flops per unit time.

The total compute time spent in S2M is given by,

$$T_{comp\_s2m} = \frac{C^1_{up}.b.p.q + C^2_{up}.b.p^2}{C_0} \tag{18}$$

The total compute time spent in M2M is given by,

$$T_{comp\_m2m} = \frac{C^3_{up}.(2^d + 1).b.p^2}{C_0} \tag{19}$$

C^1_{up}, C^2_{up}, and C^3_{up} are kernel-dependent constant.

To account for the memory hierarchy communication, we describe a communication-optimal algorithm as illustrated in Figure 30. The algorithm can be viewed as a two-step process. In the first step, $\frac{Z}{4q}$ source leaf boxes are loaded into cache and we compute its S2M (For each point, the position (x, y, and z coordinates) and density are maintained, resulting in a cumulative size of $4q$ machine words per source leaf box). Once the S2M of the $\frac{Z}{4q}$ leafs have been computed, we can compute the M2M of it's ancestors in a level-by-level manner. When we have processed all the nodes in the sub-tree, we load the next $\frac{Z}{4q}$ sources and repeat the above process until all the leaf nodes have been computed on. After the first step, the following iterative process is applied, as shown in Figure 30. The M2M of the non-leaf boxes $\frac{Z}{Kp}$ at a level i are loaded into the cache and the M2M of its ancestors computed until we reach the root node, where K is the number of children.

5.2.2 Near field Interactions (U list step)

For each target leaf box, this phase of the FMM algorithm performs a direct summation of potentials due the source boxes in its immediate neighborhood. The neighborhood of a box B is defined to be the set of all the source leaf boxes adjacent to B, and contains B as well. This list of boxes L^B_U is called the U list, and we refer to

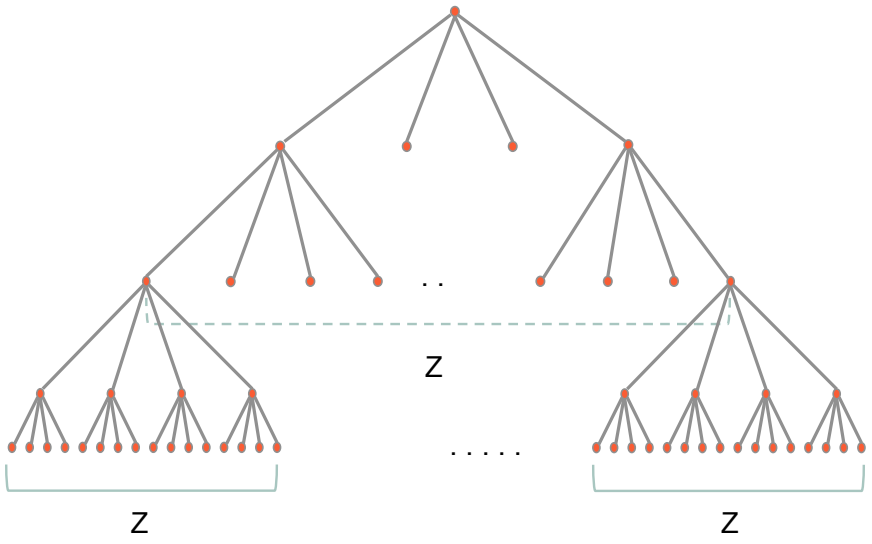


Figure 30: Illustration of upward computation algorithm.

Algorithm 2 Near field interactions (U list).

Input: Target leaf boxes $\mathcal{L}$, source boxes $\mathcal{S}$; U list L_U^β of every target leaf box $\beta \in \mathcal{L}$; Source density vectors s_α of size q, $\alpha \in S$;

Output: Target leaf box potentials f_β of size q, $\beta \in L$.

1: **for all** $\beta \in \mathcal{L}$ **do**
2: $\quad f_\beta \leftarrow \phi$
3: $\quad$ **for all** $\alpha \in L_U^\beta$ **do**
4: $\quad\quad$ Evaluate kernel matrix $K(\alpha, \beta)$
5: $\quad\quad f_\beta \leftarrow f_\beta + K(\alpha, \beta) s_\alpha$

this near field interactions evaluation phase as the U list step. In 2D, the U list of a non-boundary leaf box would be of size 9. Similarly, in 3D, the U list would be of size 27.

Algorithm 2 outlines the U list step. For each target-source pair, a dense matrix of kernel evaluations is created, and the target potential vector is updated with a dense matrix and vector multiplication. Given b leaf boxes and assuming q points per leaf box, the computational complexity of the near field interaction phase is $\mathcal{O}(bq^2)$. In 3D, the operation count is more precisely $27bq^2$. This estimate can be further refined to account for boundary boxes, and we have $(3b^{\frac{1}{3}} - 2)^3 q^2$.

The time spent performing floating-point operations in the U list step is the total

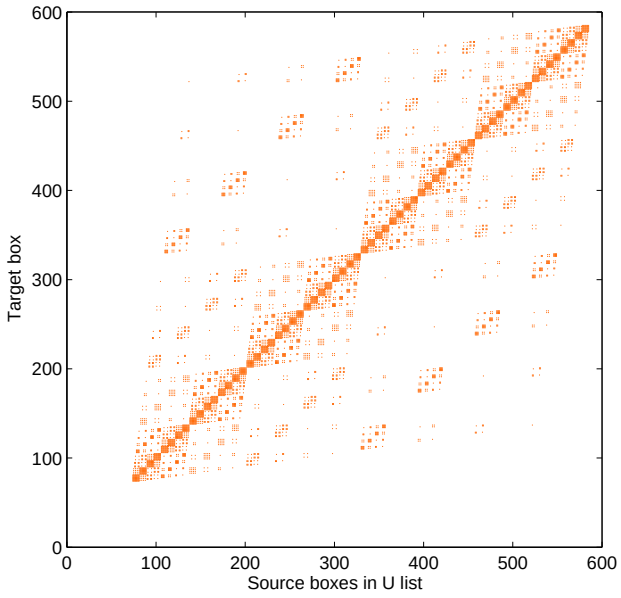


Figure 31: U list: A plot of the sparsity pattern observed in the target box–source box implicit mapping matrix (uniform random distribution of points, octree depth $l_{max} = 4$).

number of floating point operations, divided by the peak computation throughput (C_0) in floating-point operations per unit time.

$$T_{comp,u} = \frac{C_u^1 \,.\, (3b^{1/3} - 2)^3 \,.\, q^2}{C_0} \tag{20}$$

C_u^1 is a kernel- and implementation-dependent constant.

To account for memory costs in accessing the source and target box data structures, we observe that the outer loops of the computation (lines 1 and 3 of Algorithm 2) can be modeled as a sparse matrix vector multiply (SpMV). Figure 31 is a visualization of the source boxes in the U list of every target leaf box for an octree of depth 4. Each source box contains q points on an average. For each point, the position (x, y, and z coordinates) and density are maintained, resulting in a cumulative size of $4q$ machine words per source or target box.

Blelloch et al. [20, 21] present a cache-oblivious algorithm for SpMV that is based on a separator-based reordering of the matrix. They show that if the support graph of a matrix satisfies the n^ϵ edge-separator theorem [68], then such a matrix, when laid

out in row-major format after reordering, would incur at most $\mathcal{O}\left(\frac{m}{L} + \frac{n}{Z^{1-\epsilon}}\right)$ (where m is the number of non-zeros in the matrix) cache misses for SpMV. Figure 31 shows that the U list implicit dependency matrix is indeed structured, and this is a result of the spatial sorting of the boxes during tree construction. The source boxes are also stored contiguously in row-major format, and so we can adapt the SpMV bounds for near interaction computation.

The memory access costs for this step are comprised of read accesses to the source boxes, the U lists for each target box, and updates to the target leaf box potentials. The rows of the kernel matrix K are constructed on-the-fly for each source-target pair prior to matrix vector multiplication, and so we do not consider accesses to this matrix. The U list upper bounds for the number of cache lines fetched are as follows:

$$
\begin{aligned}
Q_u &= Q_{u\_src} + Q_{u\_trg} + Q_{u\_lists} \\
Q_{u\_src} &= \mathcal{O}\left(k_u \cdot \frac{N}{q} \cdot \frac{4q}{L}\right) = \mathcal{O}\left(\frac{4k_u N}{L}\right) \\
Q_{u\_trg} &= \mathcal{O}\left(\frac{N}{q} \cdot \frac{4q}{L}\right) = \mathcal{O}\left(\frac{4N}{L}\right) \\
Q_{u\_lists} &= \mathcal{O}\left(\frac{k_u \frac{N}{q}}{L}\right)
\end{aligned}
$$

Here, k_u is the average number of source boxes in the U list of a target leaf box. The above bound for Q_{u_src} assumes that there is no reuse of source boxes. The cache complexity for U list is thus dominated by the time to read source boxes.

Utilizing the SpMV bounds from [20] and assuming $\epsilon = 2/3$ for 3D, we get a tighter bound on Q_{u_src}, which is the dominant term in the overall cache complexity bound. Since each non-zero in the matrix corresponds to a source box of size $4q$, we scale the fast memory capacity Z by a factor of $4q$.

$$
Q_u = \mathcal{O}\left(\frac{4N}{L} + \frac{k_u \frac{N}{q}}{L} + \frac{4N}{L} + \frac{\frac{N}{q}}{\left(\frac{Z}{4q}\right)^{\frac{1}{3}}}\right) \tag{21}
$$

The dominant memory access time in this step is modeled as the total data fetched into fast memory, divided by the peak rate at which data is fetched into memory (*i.e.*, memory bandwidth β_{mem}).

$$T_{mem,u} = \frac{C_u^2 N}{\beta_{mem}} + \frac{C_u^3 NL}{\beta_{mem}(Z^{\frac{1}{3}} q^{\frac{2}{3}})} \tag{22}$$

C_u^2 and C_u^3 are implementation- and machine-dependent constants that we determine empirically by fitting the execution times to the model.

5.2.3 Far field Interactions (V list step)

Algorithm 3 Far field interactions (V list).

Input: Target boxes $\mathcal{T}$, source boxes $\mathcal{S}$; V list L_V^β of every target box $\beta \in \mathcal{T}$; Source upward densities u_α of size p, $\alpha \in S$;
Output: Target downward potentials g_β of size p, $\beta \in T$.

```
1: Pre-compute up to down translation matrix E(α, β)
2: for all α ∈ S do
3:    Compute FFT of upward densities; u_α ← FFT(u_α)
4: for all β ∈ T do
5:    for all α ∈ L_V^β do
6:       g_β ← g_β + E(α, β) u_α
7:    Compute Inverse FFT of downward potentials; g_β ← IFFT(g_β)
```

For each target box in the tree, this phase accumulates the multipole expansions of the source boxes in its V list into a local expansion. This step is also called multipole to local (M2L) translation. The V list of a box B is defined to be the set of all source boxes that are children of the neighbors of box B's parent, but not adjacent to B itself. The computation performed in the V list is 3D convolution. We implement this in 3 steps, namely, (a) 3D FFT, (b) complex pointwise multiplication in the frequency domain, and (c) 3D inverse FFT. Assuming b_s source boxes, the computational complexity of the FFT phase is $\mathcal{O}(b_s \,.\, p^{\frac{3}{2}} \log p)$ where p is a constant determined by the desired accuracy ($p = \mathcal{O}(\gamma^2)$, γ is the number of digits of precision). The inverse FFT's are done once for each target box, resulting in a complexity of

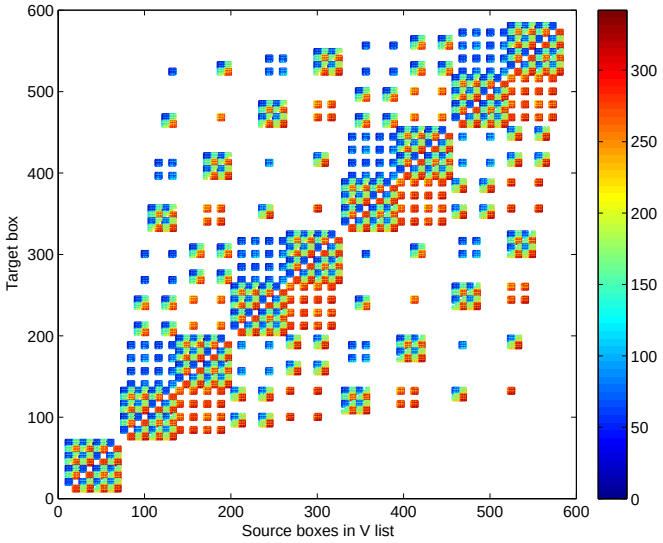


Figure 32: V list: A plot of the sparsity pattern observed in the target box–source box-translation implicit mapping matrix (uniform random distribution of points, octree depth $l_{max} = 4$).

$\mathcal{O}(b_t \,.\, p^{\frac{3}{2}} \log p)$, assuming b_t target boxes. Each target box performs k_v pointwise multiplications ($k_v = 189$ for an interior box for an uniform distribution), and has an asymptotic complexity of $\mathcal{O}(b_t \,.\, k_v \,.\, p^{\frac{3}{2}})$.

Refining these estimates, the computational time for V list if given by

$$T_{comp,v} = \frac{C_v^1 \ (b_s \ + \ b_t + \ 343) \ p^{\frac{3}{2}} \log p}{C_0} + \frac{C_v^2 \ b_t \ k_v \, p^{\frac{3}{2}}}{C_0} \tag{23}$$

C_v^1 and C_v^2 are implementation-dependent constants.

Figure 32 shows the implicit dependency between target boxes, source boxes, and translation operators. For each point in the matrix, we perform a pointwise multiply between a source box of size $p^{\frac{3}{2}}$ with a translation operator. Since the number of translation operators (343) is fixed, we assume they fit in the shared cache Z. Hence, the effective cache size becomes $Z^{'} = Z - 343 p^{\frac{3}{2}}$.

Similar to U list, re-applying the SpMV bounds from [21], we get an upper bound on the cache complexity for this phase:

$$Q_v = \mathcal{O}\left(\frac{(b_t + b_s)\ p^{\frac{3}{2}}}{L} + \frac{k_v\ b_t}{L} + \frac{b_t}{\left(\frac{Z'}{p^{\frac{3}{2}}}\right)^{\frac{1}{3}}}\right) \tag{24}$$

Considering the higher order terms, the memory access time of V list can be approximated by,

$$T_{mem,v} = \frac{C_v^3 N p^{\frac{3}{2}}}{q\beta_{mem}} + \frac{C_v^4 N p^{\frac{1}{2}} L}{(Z'^{\frac{1}{3}} q)\beta_{mem}} \tag{25}$$

5.2.4 Downward step

Algorithm 4 Downward computation.

Input: Target leaf boxes $\mathcal{L}$; Target non-leaf boxes $\mathcal{N}$; Target downward potential g_β of size p, $\beta \in \mathcal{L} \cup \mathcal{N}$;

Output: Target potential f_β of size q, $\beta \in \mathcal{L}$.

1: Pre-compute down to down translation matrix D_β
2: Pre-compute down to target translation matrix T_β
3: **for all** $\beta \in \mathcal{L} \cup \mathcal{N}$ **do**
4: $\quad g_\beta \leftarrow g_\beta + D_\beta d_{P(\beta)}$
5: $\quad$ Solve for d_β; $T_\beta d_\beta \leftarrow g_\beta$
6: **for all** $\beta \in \mathcal{L}$ **do**
7: $\quad$ Evaluate kernel matrix $K(\gamma, \beta)$
8: $\quad f_\beta \leftarrow K(\gamma, \beta) d_\beta$

Downward step consists of two stages namely, (a) Local to Local (L2L): translate the local expansion (LE) from parent to child, and (b) Local to Target (L2T): Translate LE from box to individual targets.

The LE's are computed by traversing the tree top-down and translating the LE's of the parent to compute the LE of its children. LE's represent the effect of all distant particles on the points within B. Algorithm 4 outlines the downward computation.

$$T_{comp\_l2l} = \frac{C_d^1\, 2^d.p^2 + C_d^2\, b.p^2}{C_0} \tag{26}$$

The asymptotic complexity of this step is $\mathcal{O}(bp^2)$.

If L is the leaf level, there there are b leaf nodes, where $b = 2^{Ld}$ or $b = \frac{n}{q}$. We perform L2T once for each target leaf node. This step essentially translates the

contribution due to all the sources apart from the near neighbors to individual target points.

The main computational step is a dense matrix-vector multiply, where the matrix is of size $p \times q$ and its entries are computed on the fly.

$$T_{comp\_l2t} = \frac{C_d^3 \, b.p.q}{C_0} \tag{27}$$

The asymptotic complexity of this step is $\mathcal{O}(np)$.

Table 6: Description of the main computational steps in KIFMM.

KIFMM step	Description	Computational Complexity	Cache Complexity
Upward	Algorithm 1	$\mathcal{O}(np + bp^2)$	
Near field	Algorithm 2	$\mathcal{O}(nq)$	$\mathcal{O}\left(\frac{4N}{L} + \frac{k_u \frac{N}{q}}{L} + \frac{4N}{L} + \frac{\frac{N}{q}}{\left(\frac{Z}{4q}\right)^{\frac{1}{3}}}\right)$
Far field	Algorithm 3	$\mathcal{O}(bk_v p^{\frac{3}{2}} + bp^{\frac{3}{2}} \log p)$	$\mathcal{O}\left(\frac{(b_t+b_s)\, p^{\frac{3}{2}}}{L} + \frac{k_v \, b_t}{L} + \frac{b_t}{\left(\frac{Z'}{p^{\frac{3}{2}}}\right)^{\frac{1}{3}}}\right)$
Downward	Algorithm 4	$\mathcal{O}(np + bp^2)$	

5.2.5 V list memory access complexity assuming cache blocking

We can alternatively use geometrical arguments to model the V list memory access complexity of our implementation more accurately. Due to spatial reordering, there is substantial locality in V list box identifiers. There is maximum reuse amongst 'siblings'. For a given cache size Z, we can maximize reuse by blocking sibling boxes and all their V list neighbors, along with all the translation vectors. Consider, for example, the sibling boxes in 3D. At the level of leaf nodes, the union of all V list neighbors for eight siblings is a solid cube encompassing the eight sibling sources ($2 \times 2 \times 2$), and all the boxes that are $+2$ units away in each direction, for all dimensions. If we go up the tree and view the leaf siblings as a single sibling, and consider eight of these leaf siblings sets, we again need to block a solid cube of

boxes consisting of $(2 \times 2 \times 2) \times (2 \times 2 \times 2)$ boxes and boxes that are +2 away in each dimension, for a total of 512 boxes, along with the appropriate targets and the translation vectors. This can be viewed as progressively gathering larger-sized siblings and their neighbors until we can no longer fit them in the given cache. This idea is summarized by Equation 28 for a tree of depth l and dimension d. The $p^{\frac{3}{2}}$ term indicates the number of words per vector, determined by the precision p. k refers to the number of levels in the tree away from l. This determines the amount of data we are trying to block in the cache. The three main terms in the equation refer to the number of targets, sources, and the translations. The min term for the sources is needed to take care of boundary conditions where the full set of V list neighbors are not available.

$$Z = (2^{kd} + \min((2^k + 4)^d, 2^{ld}) + (7^d - 3^d))p^{\frac{3}{2}} \tag{28}$$

Given cache size Z, we can solve for k to determine an optimal blocking size. By iterating over the entire tree using this blocking scheme, we can determine the total memory traffic cost and estimate $T_{mem,v}$.

5.3 Optimal Scheduling

In this section, we first analyze the performance of the different phases of FMM based on the analytical model discussed in Section 5.2 to predict the optimal scheduling strategy on a hybrid CPU-GPU system. We then compare the performance of KI-FMM on three architectures, namely dual socket sixcore Intel Xeon X5650 (CPU-only), NVDIA Tesla M2090 "Fermi" GPU (GPU-only), and on a hybrid CPU-GPU system with the above two processors (hybrid).

We consider two different particle distributions, namely a uniform random distribution and an elliptical (non-uniform) distribution. In the uniform distribution, particles are uniformly distributed within a unit cube as seen in Figure 12. In the

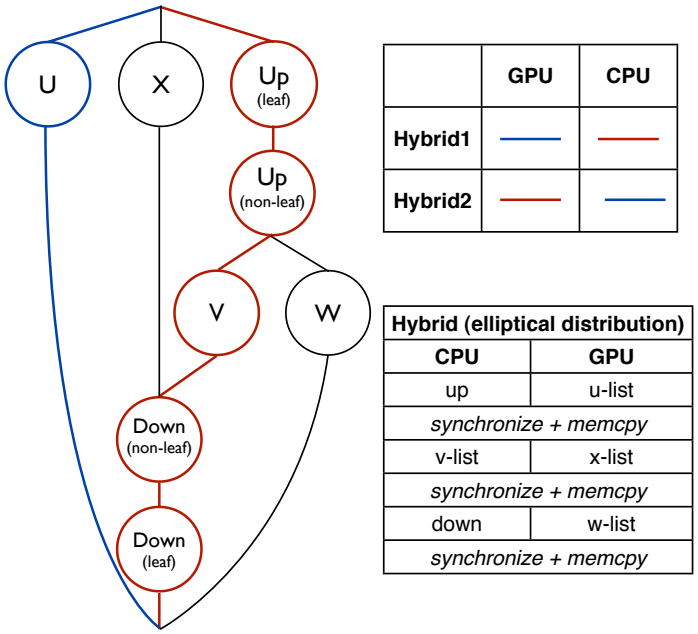


Figure 33: Directed Acyclic Graph for FMM.

elliptical distribution, particles are distributed on the surface on an ellipsoid with an aspect ratio of 1:1:4. We use the analytical performance model to estimate the optimal scheduling of the different phases of FMM for the uniform case.

Strategy

The dependencies between the various phases of FMM can be represented as a directed acyclic graph (DAG) as shown in Figure 33. The DAG for the uniform distribution is identical, bearing no contribution from X and W lists. The figure also shows the distribution of work between CPU and GPU for various hybrid scheduling strategies, for both uniform and non-uniform distributions.

For uniform distribution of points we have two scheduling strategies, *hybrid1* and *hybrid2*. The DAG for uniform distribution has two independent paths – the U list calculation path and the up-V list-down path – with no communication between the two paths until the very end. In *hybrid1*, we have the compute-intensive U list step running on the GPU and the other data-intensive path executing on the CPU. This is reversed for *hybrid2*. There is also an extra cost at the end when the result from the

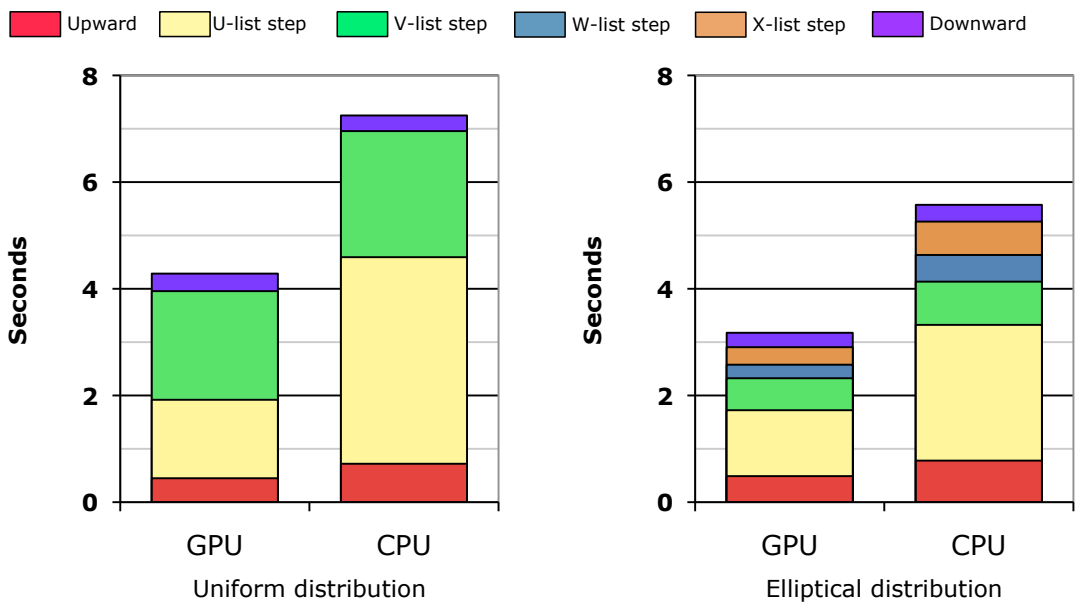


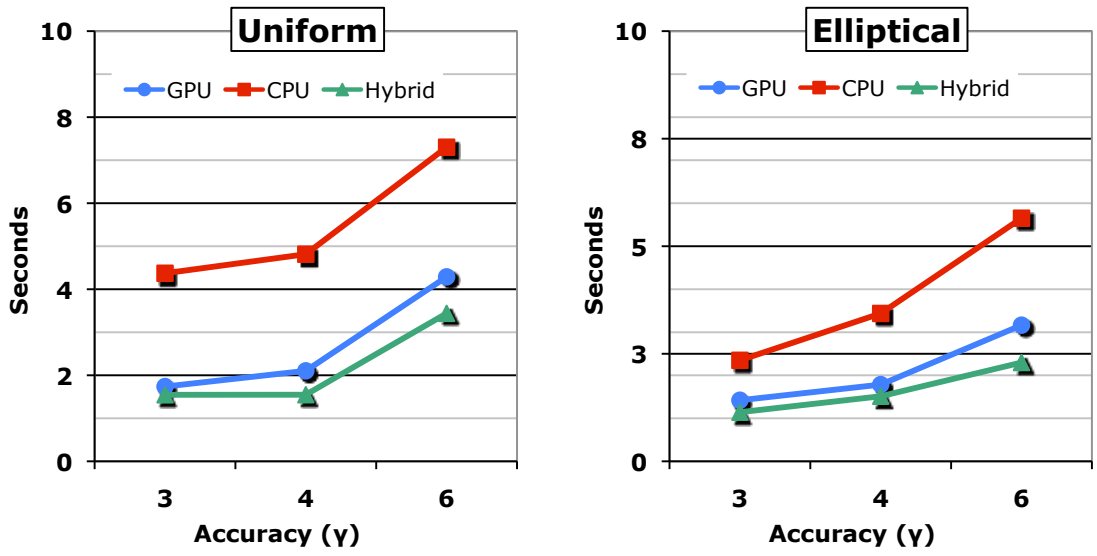
Figure 34: Breakdown of running time for a Laplace kernel potential calculation and uniform and elliptical particle distributions ($N = 4M$, $\gamma = 6$, double precision).

GPU is sent back to the host over PCI-e, which we include in our overall execution time. Although these are not the only possible schedules for the given DAG, they are the most intuitive, as U list and V list account for approximately 90% of the overall execution time. Using the models in Section 5.2, we can analytically predict which of the two schedules will yield the best performance.

Results

Figure 34 compares the double-precision performance of FMM on CPU and GPU systems for both uniform and elliptical distributions. The computation is broken down into different phases for $N = 4M$ particles with γ set to 6 digits of accuracy. For the uniform case, we observe that majority of the time is spent in the U- and V-list steps (which compute the near- and far-field evaluations). The time distribution is more spread out for the elliptical distribution. But, in both cases, the GPU achieves an overall performance improvement of $1.7\times$ over the CPU.

To understand the impact of hybrid scheduling, we vary γ and Figure 35 shows the performance of the three variants. Hybrid scheduling performs the best and the



(a) (b)

Figure 35: Comparison of run time on different system configurations for $N = 4M$ for uniform and elliptical distributions for varying γ.

improvement over GPU increases as we move to larger accuracy requirements.

5.4 Algorithmic Tuning Parameter

Based on the analytical execution time estimates, the total time T is, in the uniform case, given by

$$T = T_u + T_v.$$

This expression in turn gives us the optimal value of q, our main algorithmic performance tuning parameter:

$$q = \frac{\gamma^{3/2}}{C_1}\sqrt{C_2 + C_3\frac{C_0}{\beta_{mem}}}. \tag{29}$$

The constants C_1, C_2, and C_3 can be estimated given a kernel and an implementation. The fraction $\frac{C_0}{\beta_{mem}}$ is the processor's balance, in units of flops per word. For our current state-of-the-art multicore implementation, $\frac{C_2}{C_3} \approx 50$ and on a single socket Intel Westmere node $\frac{C_0}{\beta_{mem}} = 2.6$, which results in the optimal $q \approx 250$. This value matches our experimental results.

If we further assume that $q = \mathcal{O}(\gamma^{\frac{3}{2}})$, then T can be simplified as the expression below.

$$T = \frac{N\gamma^{3/2}}{C_0}\left(C' + C''\frac{C_0}{\beta_{mem}}\right), \tag{30}$$

where C' and C'' are constants defined appropriately in terms of the previous constants. Equation (30) has a striking feature: if a processor becomes increasingly *imbalanced*, meaning the ratio C_0/β_{mem} increases, we can compensate by decreasing our accuracy requirement, which has a *superlinear* impact ($\gamma^{3/2}$). Thus, understanding an application's minimum accuracy requirement becomes a profound scaling technique.

5.5 Summary

Our analysis refines the estimates of the constants, normally ignored in traditional asymptotic analyses of the FMM, with calibration against our state-of-the-art implementation [35, 36]. Our detailed memory hierarchy communication analysis for the kernel-independent FMM has a number of significant practical implications. One is the first *a priori* estimate of the maximum points per leaf, q, which had previously been regarded as a purely empirical tuning parameter. Another is the precise analytical characterization of how reducing the desired accuracy, γ, can *superlinearly* compensate for processor imbalance, a seemingly inevitable technology trend. In fact, our model suggests a new kind of high-level analytical *co-design* of the algorithm and architecture. For instance, classical analyses of *balance* [22, 29, 56, 57, 62, 73] relate algorithmic properties, such as intensity (intrinsic ratio of useful operations to bytes transferred), to a processor's balance (its peak ops/sec divided by peak bandwidth). Thirdly, we can accurately prediction the optimal static computation schedules for heterogeneous systems, yielding a nearly 2× speedup from a previously unexpected scheduling scheme.

CHAPTER VI

PRODUCTIVE PARALLEL PROGRAMMING

We study the use of a novel general-purpose parallel programming model, called

Concurrent Collections (CnC) [27,60,61]. In CnC, the programmer expresses her computation in terms of high-level application-specific components, partially-ordered only by minimal semantic scheduling (data- and control-flow) constraints (Section 6.2). This model encourages the programmer to focus on expressing the computation at a high-level without unnecessary serialization and gives the run-time system flexibility in scheduling operations.

Our central contribution is the first extensive study of the performance potential for HPC applications using the CnC model, based on Intel's v0.3 Linux CnC implementation for shared memory multicore systems [2]. We discuss which aspects of the CnC language and run-time could be improved. Our study is essential to establishing what the potential is for achieving high performance using CnC.

We express and analyze a prior asynchronous-parallel variant of dense Cholesky factorization when written using CnC. When coupled with a well-tuned BLAS, CnC can closely match or exceed the performance and scalability of the vendor-tuned Intel Math Kernel Library (MKL). Our CnC-based code also compares favorably to PLASMA 2.0, a state-of-the-art domain-specific library-based approach (Section 6.3). Both MKL and PLASMA use an asynchronous-parallel approach, and so constitute the current state-of-the-art.

For additional comparison, we provide results in alternative programming models, including the "off-the-shelf" solution of ScaLAPACK with a shared memory implementation of MPI (MPICH2+nemesis); OpenMP; and Cilk++. The principal difference between CnC and these models is CnC's natural support for asynchronous execution.[1] Our findings quantify the gap between asynchronous-parallel and bulk-synchronous execution (Section 6.4).

[1]We do not use the most advanced features of OpenMP 3.0 and Cilk++, nor do we compare directly to the FLAME/SuperMatrix and library-based SMPSs approaches [32, 80]. Nevertheless, for Cholesky, we would expect at best comparable performance to PLASMA.

Finally, we develop a complete CnC implementation of a novel and partly asynchronous-parallel generalized eigensolver for dense symmetric matrices, of which Cholesky is a small component (Section 6.3). This non-trivial computation is, as far as we know, the first of its kind. As such, we show that it is feasible to express a complex algorithm within CnC. Our implementation outperforms the Intel MKL equivalent by 1.1–2.6× (Section 6.4).

6.1 Background and Related Work

Although there are several papers about various aspects of the CnC model, to date there have been no performance demonstrations or evaluations to assess its viability, particularly for high-performance computing (HPC) applications. This chapter presents the first such performance study. In particular, we ask whether CnC delivers competitive performance on computations with well-defined performance targets and challenging algorithmic characteristics. For our evaluation, we select dense linear algebra computations written for multicore systems in an *asynchronous-parallel* style, by which we mean bulk-synchronous parallel behavior is replaced by more fine-grained task-level parallelism and localized synchronization [28, 32, 80]. This approach (a) is naturally suited to cores with relatively smaller cache or local-store memories, and (b) reduces the degree of synchronization, whose cost may reasonably be expected to increase with increasing core counts. There are numerous successful demonstrations of this approach for dense linear algebra on current multicore systems [28, 32, 69], meaning there are clear and rigorous performance targets.

Scope.

Importantly, this study is about the *performance potential* of CnC. Such studies are essential for any new parallel programming model to show value for HPC. Our positive findings show there is potential in CnC as far as performance is concerned.

As an evaluation of CnC for HPC, our use of dense linear algebra limits the

findings' generalizability to one class of computations. Still, this class has challenging properties (e.g., our eigensolver), and so we believe that the basic CnC approach could still be an appropriate starting point for similarly asynchronous-parallel algorithms in other areas.

Equally important to questions of performance are those of *productivity*. We argue, qualitatively, that CnC is suitable for these computations. However, we stress that a *true assessment* is a human-factors question, requiring a separate and carefully controlled experiment, and as such is beyond the scope of the present study.

Related Work. Existing work on asynchronous-parallel algorithms for dense linear algebra covers Cholesky, LU, and QR factorization, as well as so-called "two-sided" transformations, Hessenberg, tridiagonal and bidiagonal reduction [28, 32, 69]. The implementations are based on some combination of schedulers and APIs based on domain-specific abstraction (*e.g.*, SuperMatrix [32]), or hand-coded or pragma-directed schemes (*e.g.*, SMPSs [69, 80]). The present study contributes experience working in a novel model and a novel asynchronous-parallel implementation of a different algorithm (the eigensolver).

The CnC programming model itself has rich influences from the long history of concurrent programming models, including tuple-spaces, streaming languages, and dataflow languages [31, 42, 92]. CnC's key distinction is its treatment of both control and dataflow, thereby making CnC more general than pure streaming or classical data flow approaches. Also, item collections in CnC allows for more general indexing than dataflow arrays. While both CnC and tuple space languages like Linda specify computation using tags, they differ in a number of aspects. In CnC there is a clear separation between tags and values, while there is no distinction between the two in Linda. Moreover items are accessed by value and not by location, and adhere to dynamic single assignment form, as noted by Budimlic, et al. [27].

6.2 Overview of CnC

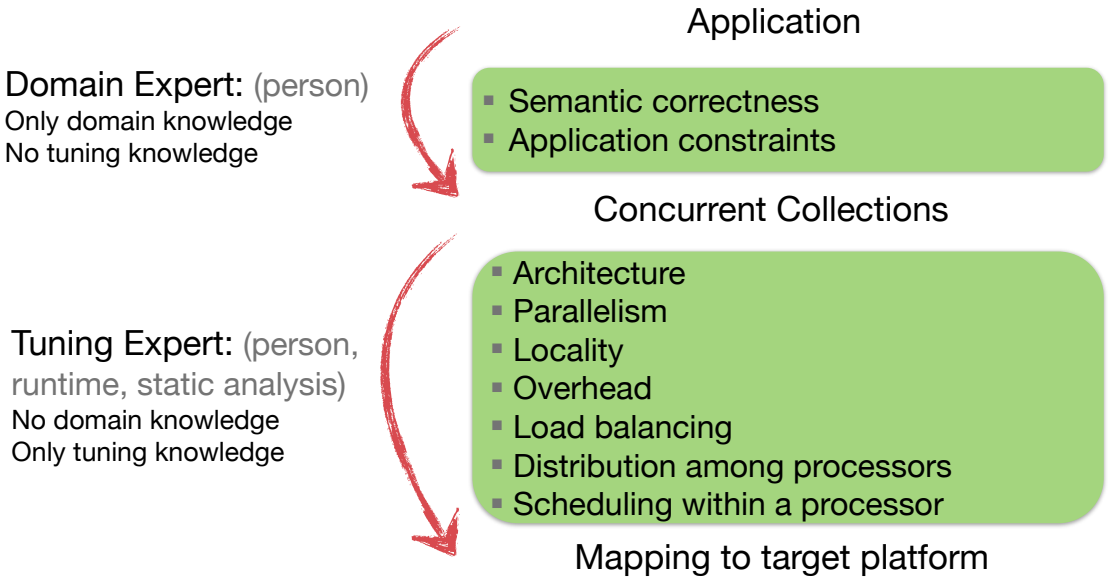


Figure 36: Outline of CnC's model of computation.

This section provides a cursory overview of the basic CnC concepts relevant to our implementation and experimental results. Portions of this material are taken from existing detailed summaries [2, 27, 60].

The CnC model as illustrated in Figure 36 separates the specification of the computation from the expression of its parallelism. This design can simplify the tasks of a domain expert, who is responsible for expressing the computation, from the tasks of a parallelization and tuning expert (possibly the same person, a different person, or some software/compiler), who identifies the parallelism and performs scheduling/distribution and manages communication/synchronization. CnC combines ideas from earlier language work on tuple-spaces, streaming, and data flow models [31,42,92] (see Section 6.1).

6.2.1 Computation Specification

We summarize the basic CnC model by an example. Consider the dense outer product computation, $Z \leftarrow x \cdot y^T$, where x and y are two column vectors and Z is a matrix

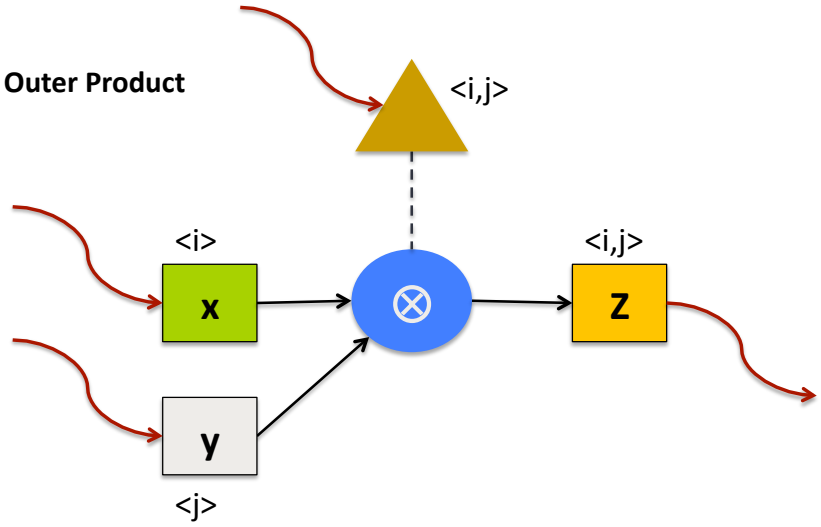


Figure 37: CnC graphical representation of the outer product operation, $Z \leftarrow x \cdot y^T$.

of appropriate dimension. Algorithmically, we compute $Z_{i,j} \leftarrow x_i \cdot y_j$ for all pairs, (x_i, y_j).

The domain expert specifies the computation in a form that can be represented by a graph, as shown in Figure 37 for the outer product. This graph has 3 kinds of nodes: computational *steps*, data *items*, and control *tags*. Directed edges show producer-consumer relationships among these nodes.

A *step* is the basic unit of execution, which for the outer product is pairwise element multiplication.[2] The blue oval in Figure 37 is a *step collection*, which statically represents the set of dynamic instances of these multiplications.

Data is represented using *item collections*. Here, x, y, and Z are the three item collections, shown by boxes in Figure 37. Each item collection comprises *item instances*, which in this case are the elements of the x, y, and Z objects. These items serve as the basic unit of storage, communication, and synchronization.

Steps may consume items (item $\rightarrow$ step) or produce them (step $\rightarrow$ item), shown by directed edges in Figure 37.

Each instance of a step or item has a unique application-specific identifier, or *tag*,

[2]We consider this very fine granularity for example only, as in practice one might wish to choose a larger grain, such as a block.

which is a tuple of *tag components*. For the outer product, it is natural to use element indices as tags. In Figure 37, we denote the tag for x by $< i >$, y by $< j >$, and Z by $< i, j >$.

Tag collections (also called *tag spaces*) specify exactly which instances of a step will execute. A step collection is associated with exactly one tag collection/space; a step instance executes only if a matching tag instance exists. For the outer product, we show the tag space by a triangle and denote it by $< i, j >$. For instance, only if the tag collection has $< 3, 10 >$ does the corresponding pairwise multiply for $Z_{3,10} \leftarrow x_3 \cdot y_{10}$ execute. We say that a tag collection *prescribes* a step collection, and show that visually with a dashed undirected edge connecting the tag collection to the step collection. Multiple step collections may be prescribed by the same tag collection. Importantly, tags indicate *whether* a step will execute, but nothing about *when* it executes. This distinction shows in part how CnC separates scheduling decisions from the computation's specification.

Though not shown here, a step may produce tags. In this way, a step may control what other steps execute. This facility is part of what makes CnC a more flexible and general model than, say, a pure streaming language.

Lastly, Figure 37 contains "squiggly" lines that are missing either a source or a sink. These lines mean that the item or tag comes from or goes to the *environment*, which is the external code that invokes this computation. For the outer product, the environment provides the data items and control tags. (There are other designs; for instance, we could have added an additional step that consumes data containing, say, the dimensions of the x and y vectors, and then produces the control tags that prescribe the pairwise multiply step.)

6.2.2 Textual Notation

There is a formal textual representation of this graph. We illustrate this representation in Section 6.3, when we describe the CnC implementations of our target dense linear algebra computations. In the current implementation, a translator converts this specification into C++ code, generating subroutine stubs corresponding to the steps. The programmer must implement these stubs (presumably as purely sequential code).

When the run-time calls the sequential step code, it provides the tag and data item instances. The step code calls an API to *get* the input tags and, if it produces tags, *put* them back "into" the graph. We refer the interested reader to the CnC documentation [2].

6.2.3 Semantics and Execution

If a step executes and produces an item or a tag, that item or tag becomes *available*. If a tag collection prescribes a step collection and a particular tag becomes available, then the step is *prescribed*. If all items for a particular step are available, the step becomes *inputs-available*. If a step is both inputs-available and prescribed, then it is *enabled* and may execute. The program terminates when no step is executing and no unexecuted step is enabled. This termination is valid if all prescribed steps have been executed.

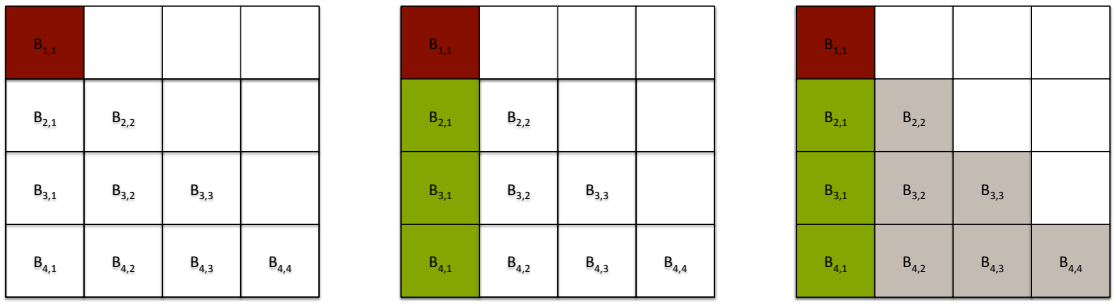
The CnC model permits many run-time system designs, including those for distributed memory systems using MPI as well as shared memory versions [27]. We use Intel CnC 0.3, which is based on the Intel Thread Building Blocks (TBB) [2]. The TBB run-time system is based on a Cilk-style work stealing scheduler, with work queues implemented to use last-in first-out (LIFO) order.

In the Intel CnC, there are four types of *events*: start, complete, idle, and requeue. *Start* signals the beginning of execution of a step, while *complete* signals its successful

completion. An *idle* event is the time spent between the end of one step and start of the next, when the thread is waiting to be scheduled or waiting for data to become available.

The *requeue* event is specific to the Intel CnC. The run-time may start executing a step as soon as the prescribing tag is available. Thus, if some of the step's inputs are not yet available, the step may be requeued and tried again later. We revisit requeuing in Section 6.4.

6.3 Dense Linear Algebra in CnC



(a) Initial factorization of the $(1, 1)$ tile.

(b) Triangular solve using the $(1, 1)$ tile.

(c) Symmetric rank-k update using the $(2 : 4, 1)$ tiles.

Figure 38: An illustration of asynchronous Cholesky factorization.

Algorithm 5 Tiled Cholesky factorization algorithm of Buttari, et al. [28].

Input: Input matrix: B, Matrix size: nxn where $n = p * b$ for some b which denotes the tile size

Output: Lower triangular matrix: L

1: **for** $k = 1 \rightarrow p$ **do**
2: Conventional Cholesky (B_{kk}, L_{kk});
3: **for** $j = k + 1 \rightarrow p$ **do**
4: Triangular Solve (L_{kk}, B_{jk}, L_{jk});
5: **for** $i = k + 1 \rightarrow j$ **do**
6: Symmetric Rank-k Update (L_{jk}, L_{ik}, B_{ij});

In this section, we discuss the CnC implementations of the asynchronous-parallel variant of dense Cholesky factorization and a novel asynchronous-parallel dense generalized eigensolver.

6.3.1 Asynchronous-Parallel Cholesky

A Cholesky factorization of a symmetric positive definite matrix B is the product $L \cdot L^T$, where L is a (lower) triangular matrix. We specifically consider Algorithm 5, which is the *tiled Cholesky* algorithm of Buttari, et al. [28]. This algorithm is based on decomposing B into blocks (or tiles), and computes L in an asynchronous parallel manner suitable for multicore hierarchical memory platforms.

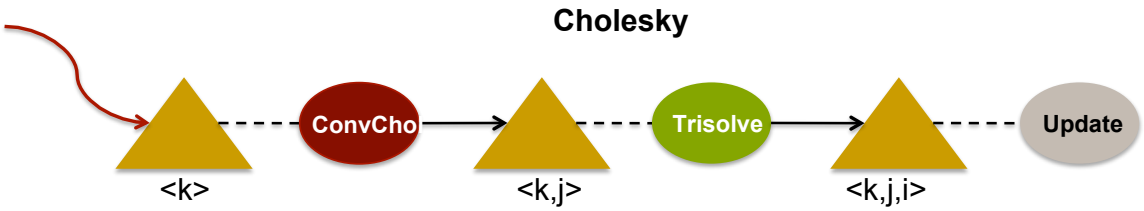
Figure 38 illustrates how asynchronous-parallelism arises in Algorithm 5. We first factor $B_{11} = L_{11} \cdot L_{11}^T$ (line 2 of Algorithm 5 for $k = 1$), using a conventional sequential Cholesky algorithm. Then, lines 3–4, which operate on blocks B_{21}, B_{31}, and B_{41} can execute in parallel. Moreover, lines 5–6 suggest that once we complete operating on the B_{21} block, we can do a symmetric rank-k update of block B_{22} in one thread while another thread is still, say, performing the triangular solve on block B_{31}. Hence, there is a lot of task- and data-level parallelism in Cholesky.

6.3.2 Cholesky in CnC

This asynchronous-parallel behavior maps naturally to the CnC constructs seen in Section 6.2. Lines 2, 4, and 6 in Algorithm 5 map to *steps* in CnC. The index iteration variables of Algorithm 5 constitute a natural choice for *tags* which helps distinguish between different data *items* (tiles in this case).

Figure 39 shows the graphical computation specification of tiled Cholesky in CnC. For simplicity, we omit the data items from this graph. Below the graph, we show the textual representation of the graph that the programmer might write. CnC translates the textual representation into C++ code containing stubs for the programmer to fill in code, as illustrated in Figure 40. That is, at this point, all the programmer does is input the appropriate tags and data items along with the serial logic of the step.

For the serial step implementation, we call tuned sequential vendor BLAS routines. This allows us to couple CnC with an optimized serial library to obtain an efficient




```
// Item: Matrix L, tagged by <k, j, i>
// Suppose 'BlockedMatrix<T>' is a C++ class
// that encapsulates the matrix data,
// dimension 'n' and block size 'b'.
[BlockedMatrix<double>* L: int, int, int];

// Tag declarations
<C_tag : int>;             // <k>
<TS_tag: int, int>;        // <k, j>
<U_tag : int, int, int>;   // <k, j, i>

// Step Prescriptions
<C_tag>  :: (ConvChol);
<TS_tag> :: (Trisolve);
<U_tag>  :: (Update);

// Input from the environment
env -> [L], <C_tag>;

// Step executions
[L] -> (ConvChol);
(ConvChol) -> [L], <TS_tag>;

[L] -> (Trisolve);
(Trisolve) -> [L], <U_tag>;

[L] -> (Update);
(Update) -> [L];

// Output to environment
[L] -> env;
```

Figure 39: Top: CnC graphical notation of Cholesky factorization. The red oval is the conventional Cholesky step; the green oval is the triangular solve step; and the grey oval is the symmetric rank-k update. Bottom: Textual notation of Cholesky factorization. Includes one statement for each relation in the graph.

parallel implementation with minimal coding effort. Figure 40 shows the actual step code with the call to *dtrsm* which performs triangular solve. The API calls (Get/Put) before and after the BLAS function call reads in the input tile(s) identified by the tag, performs the computation and outputs tile(s) with the corresponding tag identifier. The input/output and computation performed might vary across different steps, but the basic principle is the same.

Note that the choice of tags is important, as it determines the amount of parallelism exposed. Tag choice is largely natural for dense linear algebra but a poor choice of tags could impede performance.

Once, the data is available and the step inputting the data is prescribed by a valid tag identifer, the CnC run-time schedules that particular step instance for execution. Given the freedom to schedule steps, CnC schedules them in a way to expose asynchronous-parallel execution.

As with related approaches, this CnC-based implementation contains a tuning parameter, the block size, which we have assumed the domain expert introduces and selects. This parameter is critical to performance in that it implicitly controls the degree of available asynchronous-parallelism.

6.3.3 Generalized Symmetric Eigensolver

Though illustrative, the Cholesky example is fairly compact. To better understand and assess CnC, we also developed a tiled and partly asynchronous-parallel symmetric generalized eigensolver, which is considerably larger than Cholesky. This is one of the first efforts on designing and implementing an asynchronous eigensolver and most importantly a model like CnC allows us to express the asynchronous-parallel behavior naturally with relatively modest effort.

Algorithm.

To compute the eigenvalues, λ we solve the linear algebra equation $Az = \lambda Bz$.

```
StepReturnValue_t  Trisolve(cholesky_graph_t& graph, const Tag_t& TS_Tag)
{
  char uplo = 'l', side = 'r';
  char transa = 't', diag = 'n';
  double alpha = 1;

  const int k = (TS_Tag[0]);
  const int j = (TS_Tag[1]);

  // For each input item in this step
  // retrieve the item using the proper tag

  // User code to create item tag here
  BlockedMatrix<double>* A_block  = graph.L.Get(Tag_t(j, k, k));
  BlockedMatrix<double>* Li_block = graph.L.Get(Tag_t(k, k, k+1));

  // Get block size
  int b = A_block->getBlockSize ();

  // Step implementation logic goes here
  dtrsm(&side, &uplo, &transa, &diag, &b, &b, &alpha, Li_block, &b,
  A_block, &b);

  // For each output item for this step
  // put the new item using the proper tag

  // User code to create item tag here
  graph.L.Put(Tag_t(j, k, k+1), A_block);

  return CNC_Success;
}
```

Figure 40: CnC code for the triangular solve step of the Cholesky algorithm. The black and gray text in this code snippet denote the stubs that are generated automatically using the inputs and outputs defined in the graph. The code fragments filled in by the user are indicated in bold (blue color text). Note: We call tuned BLAS for the sequential step implementation (*dtrsm* in this example).

Here, A is symmetric and B is symmetric positive definite matrix. Although this implementation is based on the LAPACK routine *dsygvx*, we note that unlike the original algorithm our implementation in CnC is in fact asynchronous-parallel.

The basic algorithm implemented by *dsygvx* has four components. First, we compute the Cholesky factorization $B \rightarrow LL^T$. Next, we reduce the real symmetric-definite generalized eigenproblem to so-called standard form, $(L^{-1} * A * L^{-T})z = \lambda z$. Methods exist for computing the eigenvalues directly from the generalized form. The third component is reduction of the symmetric matrix to symmetric tridiagonal form, using an orthogonal similarity transformation, $T = Q' * A * Q$. This step can be decomposed into a number of kernels, including matrix-vector multiplication, symmetric matrix vector multipliction, and dot product, among others. Finally, we extract the eigenvalues from the tridiagonal matrix using a modified QR algorithm. This step is not compute intensive and may be computed by a single thread.

Asynchronous-parallelism.

Figure 41 is a directed acyclic graph (DAG) of the first two steps of the eigensolver for a matrix partitioned into a 3×3 grid of blocks. Nodes represent computation and edges represent dependencies among them. Each block is labeled by the appropriate submatrix block coordinates. Note that all nodes at any level of the DAG (highlighted by a grey oval) have no dependencies among themselves. Although initially the computation is sequential until root node finishes execution, there is abundant parallelism thereafter. As is evident from the figure, different tasks, denoted by different colors, can execute concurrently.

In the eigensolver, we not only execute steps in Cholesky asynchronously, but also interleave them with steps of the reduction to standard form (e.g., right triangular solve, symmetric matrix multiplication). The CnC code easily expresses this concurrency, and the run-time exploits that concurrency naturally through its scheduling as discussed in Section 6.4.

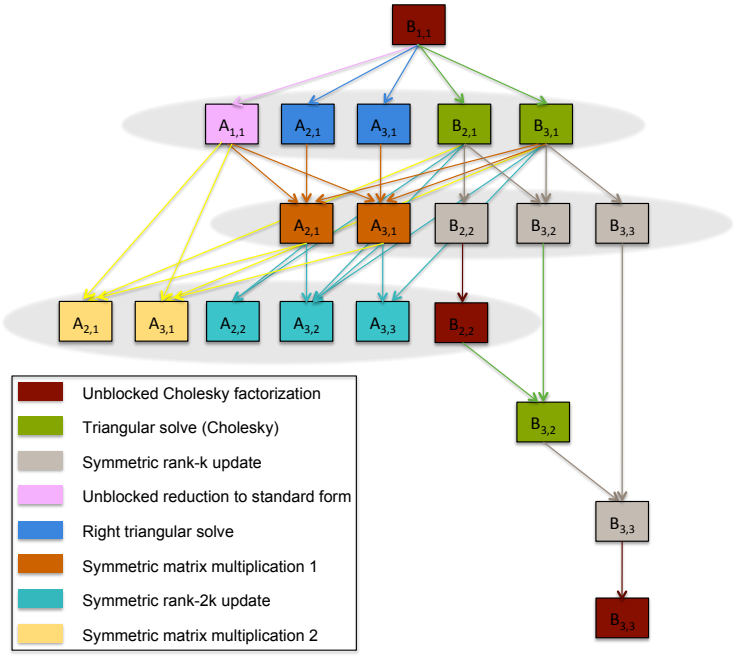


Figure 41: DAG representation of the eigensolver.

Although it is possible to extract parallelism using CnC from the third component of the eigensolver (reduction to tridiagonal form), the inherent dependencies inhibit asynchronous-parallel execution. We would need a different algorithmic approach altogether.

Parallelization.

Once the dependencies between the steps are laid out, it is possible to extract parallelism more efficiently. To that end, we parallelize all sub-kernels within the first three phases. Unfortunately, one of the most compute-intensive kernel is the symmetric matrix-vector multiply (dsymv), which was not efficiently implemented in the BLAS. Hence, for this kernel alone, we manually parallelize the computation in CnC. This trade-off is worth while as we observe a dramatic increase in performance for a slight increase in programmer effort.

Table 7: Description of the main computational steps in KIFMM.

Vendor	AMD	Intel	Intel
Proc. Model	Opteron 8350	Xeon E5405	Xeon X5560
Proc. Name	Barcelona	Harpertown	Nehalem
Clock(GHz)	2	2	2.8
# Sockets	4	2	2
Cores(Threads)/Socket	4(4)	4(4)	4(8)
L1 Data Cache	64 KB/core	32 KB/core	32 KB/core
L2 Data Cache	512 KB/core	6 MB/2cores	256 KB/core
Shared L3 Cache	2 MB/socket	–	8 MB/socket
DRAM Capacity	32 GB	4 GB	12 GB
DRAM Bandwidth (GB/s)	21.3	21.3	51.2
DP Peak Performance (GFlop/s)	128	64	89.6

6.4 Results and Discussion

In this section, we first evaluate our CnC-based Cholesky and symmetric generalized eigensolver implementations on the three state-of-the-art multicore platforms shown in Table 7. We then compare their performance to six other implementations (double-precision) and execution-time bounds based on critical path length. For the non-CnC implementations, we make a "best effort" to do some tuning, and always use sequential MKL when possible. Finally, we examine CnC's scheduling compared to bulk-synchronous strategies.

We compare the following implementations.

Baseline – Sequential MKL: The Intel Math Kernel Library (MKL) implementation of Cholesky factorization, `dpotrf`, run in sequential mode with the input matrix in column-major storage. This baseline is highly tuned by Intel. We also measure multithreaded MKL (see below). [In the plots, we use sequential MKL implementation as the baseline and show this by *hollow circles*.]

Blocked iterative OpenMP + sequential MKL: We implemented the tiled Cholesky Algorithm 5, where we (1) distribute the loop in line 3 to get two 'j' loop nests, *i.e.*, one for triangular solve and one for symmetric rank-k update; and then

(2) use OpenMP to parallelize the 'j' loops. We then use the highly-tuned sequential MKL for each block operation. We tune the block size by exhaustive search for each input size, and report the best performance. [*plus signs*]

Cilk++ 1.0.3 block recursive + sequential MKL: We implemented a blocked recursive Cilk++ implementation. In particular, the entire algorithm including the triangular solve and rank-k update steps are performed recursively [11], so as to be able to easily use the Cilk++ thread `spawn` keyword. The recursive form of each step partitions the matrix into roughly half in each dimension. We stop recursion at a tunable block size, determined by exhaustive search for each input matrix dimension. We report the best performance. We use sequential MKL for the leaf kernels. [*crosses*]

Multithreaded MKL: We use the multithreaded MKL implementation of Cholesky factorization, `dpotrf`. We report performance on the the number of cores that delivers the highest performance, up to the maximum available cores. The input matrix is in column-major storage. [*hollow diamonds*]

ScaLAPACK + shared memory MPI: We use ScaLAPACK 1.8.0 with an MPI "tuned" for shared memory. In particular, we use MPICH2 1.0.8 compiled with the Nemesis device. We tune the processor grid, trying all valid configurations for a given number of MPI tasks, trying all numbers of tasks, and report the best performance. [*up-pointing triangles*]

PLASMA + sequential MKL: We use the Cholesky implementation that is part of freely available PLASMA 2.0.0 package. There is a block size parameter, which we tune for each problem size. Since PLASMA currently does not solve eigenvalue problems, we compare only against our Cholesky. [*downward pointing triangles*]

CnC + sequential MKL: The CnC implementation of Cholesky using sequential MKL for the steps. The data is stored in blocked data layout [11, 28, 32, 51]. The block size used in the layout is determined by an exhaustive search over all possible values for a given input matrix size. The block size that achieves the highest

performance is chosen. [*filled squares*]

DGEMM Peak: The peak performance (GFlop/s) of double-precision dense matrix multiplication measured using all the cores on the system. Rather than benchmarking all sizes, we show a representative GFlop/s number for $n = 10,000$, which gave the best DGEMM performance on all three platforms among all values of n that we considered for Cholesky and the eigensolver. [*dashed lines*]

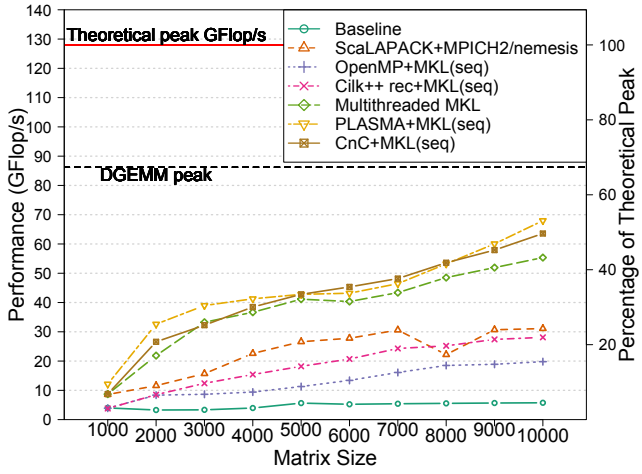
Theoretical Peak: The theoretical machine-specific upper-bound on double-precision GFlop/s achievable. [*solid lines*]

Compilers: For our CnC Cholesky factorization, eigensolver, OpenMP and ScaLAPACK implementations, we use the Intel v11.0 and v10.1 compilers on the Intel and AMD platforms respectively. We use MKL v10.0.3.020 on Barcelona, v10.2 on Nehalem, and v10.1.0.019 on Harpertown. For our Cilk++ implementation, we use the gcc 4.2 compiler that ships with it.

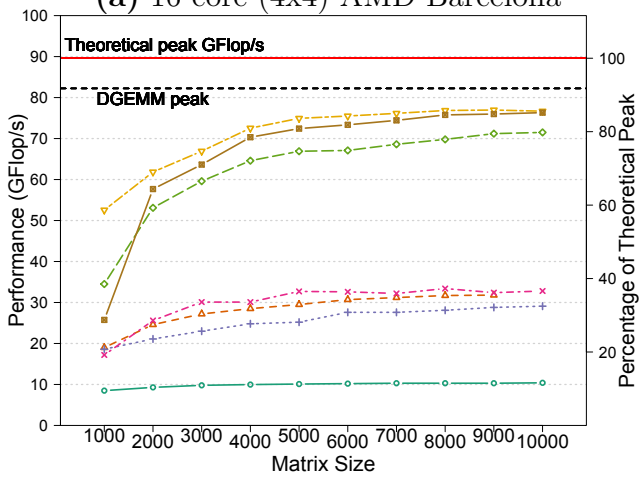
6.4.1 Cholesky factorization

Figure 42 presents the performance scalability by architecture as a function of matrix size for double precision Cholesky factorization. The baseline performance results correspond to sequential MKL values. On all machines, we use the number of cores that delivers highest performance for each matrix. We use the theoretical flop count of $n^3/3$ when reporting performance.

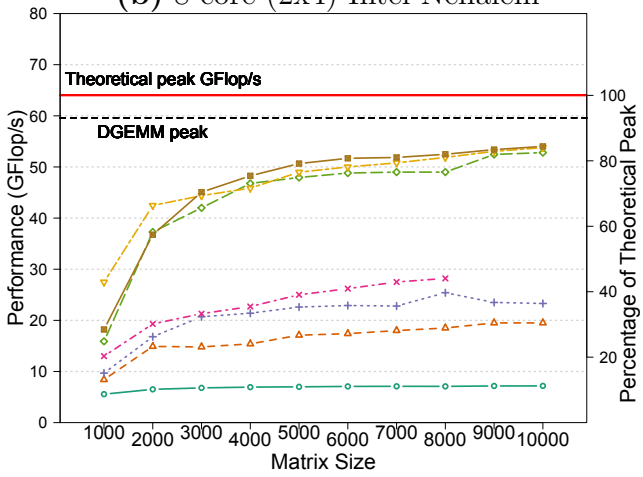
On Nehalem, our asynchronous-parallel CnC Cholesky compares favorably to PLASMA and MKL. The sequential MKL baseline runs at over 10 GFlop/s. By contrast, OpenMP with sequential MKL is 2.8× faster than the baseline; and recursive Cilk++ with sequential MKL and ScaLAPACK using shared memory MPI provides only an additional 10% over that. We observe that our fully asynchronous-parallel CnC implementation using sequential MKL delivers very good scalability (speedup of nearly 7.3× in comparison to the baseline), upto 8 threads. Beyond this,



(a) 16-core (4x4) AMD Barcelona

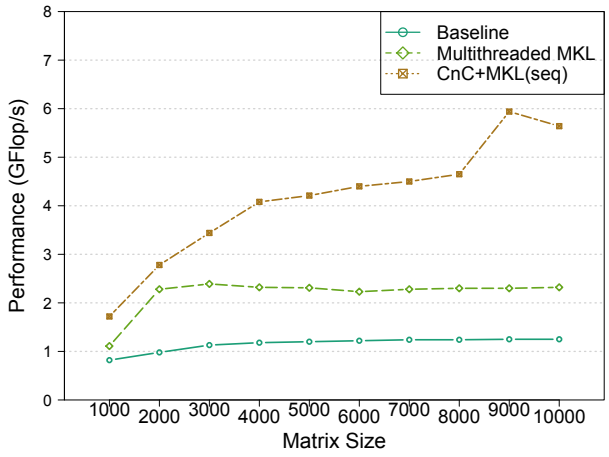


(b) 8-core (2x4) Intel Nehalem

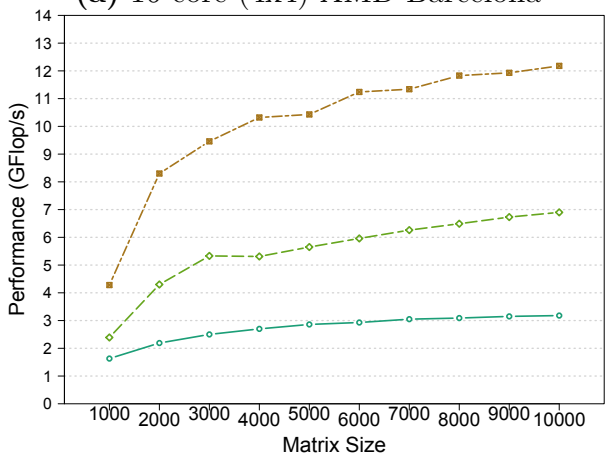


(c) 8-core (2x4) Intel Harpertown

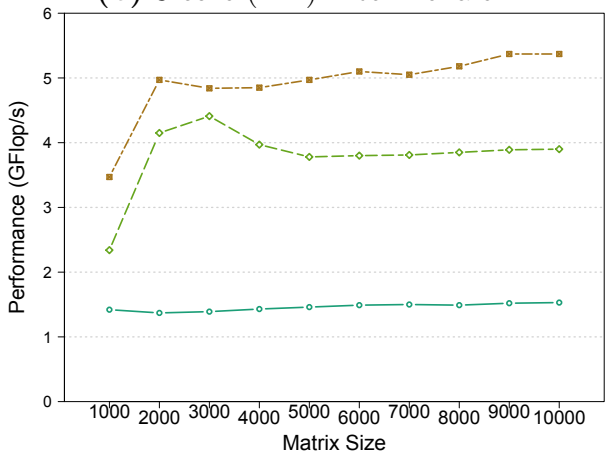
Figure 42: Performance summary of double precision Cholesky factorization: Performance in GFlop/s (left y-axis) and percentage of theoretical peak (right y-axis) as a function of matrix size, comparing seven implementations discussed.



(a) 16-core (4x4) AMD Barcelona



(b) 8-core (2x4) Intel Nehalem



(c) 8-core (2x4) Intel Harpertown

Figure 43: Performance summary of double precision Eigensolver: Performance (GFlop/s) for the three implementations discussed. The flop count used is measured using PAPI performance counters.

HyperThreading yields no added benefit on all three competiting implementations (MKL, PLASMA and CnC). Nevertheless, CnC Cholesky on Nehalem achieves more than 85% of the theoretical peak performance for the largest matrix size ($n = 10,000$) where DGEMM is at 92%.

We make similar observations on Harpertown whose Core architecture is similar to Nehalam. Unlike Nehalem, there is no simultanous multithreading (SMT) on Harpertown. Once again, our CnC implementation achieves near perfect scaling, a speedup of 7.5× on 8 cores, competing well with MKL and PLASMA implementations. Moreover, we achieve more than 80% of the theoretical peak performance.

The data on Barcelona also follow similar trends except, interestingly, Cholesky factorization achieves only half the theoretical peak performance. Nevetheless, our CnC implementation delivers performance on par with the state-of-the-art PLASMA and exceeds multithreaded MKL for large problem sizes. Barcelona performance also shows good scalability, nearly 11× on 16 cores. (Note: We did check AMD's BLAS, which was slower than MKL.)

In summary, these results show the potential of CnC to exploit the available parallelism, achieving competitive performance with reasonable programming effort.[3]

6.4.2 Eigensolver performance

Figure 43 compares the double precision eigensolver performance on our three machines. For all platforms, we compare: (i) the baseline sequential MKL implementation; (ii) the multithreaded MKL implementation; and (iii) our CnC code. Though all three codes can compute both eigenvalues and eigenvectors, we compute just the eigenvalues since it is generally recognized that `dgsyvx` is best suited to that case.

We observe that CnC delivers significantly higher performance than multithreaded

[3]Although we do not use the most recent version of MKL on Barcelona and Harpertown, we believe the comparisons made are fair in that we use the same MKL for all implementations on a given platform.

MKL on all three systems, with speedups of 1.9×, 2.6×, and 1.5× in the best case for Nehalem, Barcelona, and Harpertown, respectively.

Three factors contribute to this improvement. First, the critical path of the asynchronous-parallel CnC implementation is smaller than multithreaded MKL due to a reduced number of synchronizations. Secondly, the symmetric matrix-vector multiply kernel is not parallelized in MKL, as we confirmed by testing. Finally, on Barcelona and Nehalem, NUMA effects likely play an additional role as well, causing the MKL eigensolver implementation to not scale beyond 1 socket. Hence we compare against the 4-thread and 8-thread runs only on Barcelona and Nehalem respectively, which was the best result we could obtain when searching over all numbers of threads up to 16 (to match the 16 cores on Barcelona and 8 cores hyperthreaded on Nehalem).

Unlike MKL, in our CnC implementation we manually parallelized the symmetric matrix-vector multiply routine, thereby enabling the computation to scale up to the maximum number of cores/threads.

Interestingly, we do not see scalability issues on Harpertown. The MKL eigensolver scales up to 8 threads even though the symmetric matrix-vector multiply is sequential. Our partly asynchronous-parallel CnC implementation is up to 1.5× faster than multithreaded MKL. Even though the eigensolver implementation is non-trivial and much more complex than Cholesky factorization, our CnC implementation achieved a high level of performance, showing that at least the basic model and run-time system have good potential. We refer interested readers to [34] which contains additional details on comparison of CnC against other asynchronous approaches, detailed scalability results, which we omit in this paper due to space constraints.

6.4.3 Scheduling

The current run-time system is characterized by a static grain size, dynamic schedule, and dynamic distribution. It is built on top of the TBB [3]. TBB controls the

scheduling and execution of steps in a CnC program. TBB implements a Cilk-like work-stealing scheduler that supports fine-grained task parallelism [23].

Tag generation.

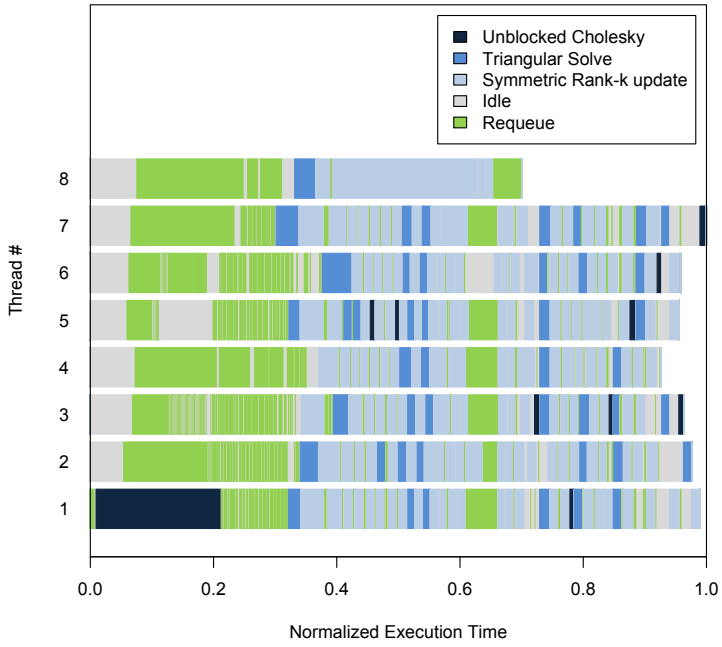
In CnC, we have the option either to pre-generate tags or generate them on-the-fly. Figure 44 depicts an execution timeline for the Cholesky factorization of a matrix of size 1000, for two approaches to tag generation. Figure 44(a) shows the approach in which we pre-generate all tags. Owing to the run-time's LIFO queuing (Section 6.2.3), the last tag generated will be scheduled first. That is, for Cholesky, first-in-first-out (FIFO) is preferable. We typically want the first tag value in tag space $< k >$ to be scheduled first since all other steps are stalled until this step finishes execution. One solution is to generate tags with data on-the fly shown in Figure 44(b).

We layout the execution profile of each thread along the y-axis (one "row" per thread); and on the x-axis show execution time, normalized in both charts to the time taken by longest executing thread. The different color-coded regions represent the different step instances.

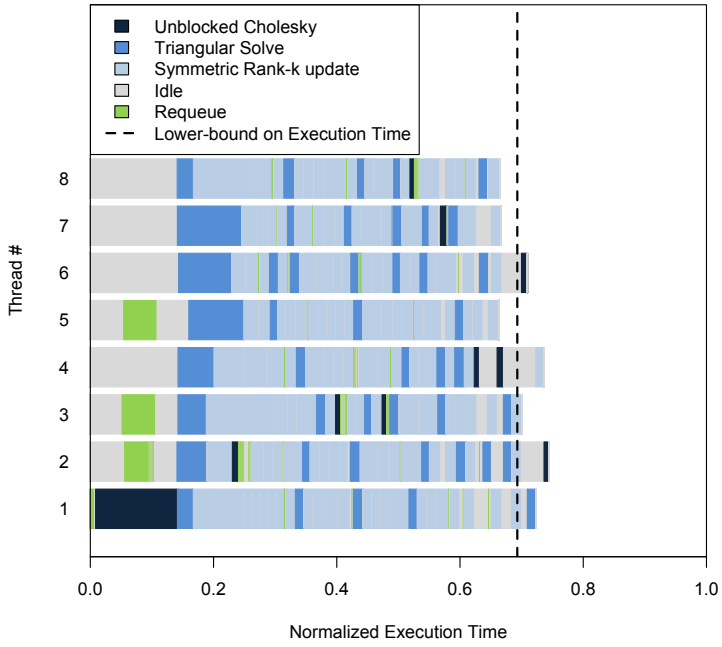
We make a number of observations. First, the dynamic tag approach takes only 75% of the time taken by the pre-generated tags approach. When pre-generating tags, 30.6% of the overall execution time is spent on requeue events for the matrix size 1000 (sum of green regions). The computation is also not load balanced, with thread 8 completing much earlier than the other threads. When we explicitly generate tags only when data becomes available, we observe a marked decrease in the number of requeue events, thereby yielding the 25% improvement in time. Moreover, the computation is better load balanced.

By reducing requeue events, we increase the number of *Get* and *Put* operations, but the overhead due to these operations is much less than the requeue delays, as shown by the decrease in the overall running time.

However, we also observed for larger n, requeue delays were actually not that



(a) Tags pre-generated; requeue time = 30.6%.



(b) Tags generated when data becomes available.

Figure 44: Scheduling for Cholesky factorization (matrix size = 1000)

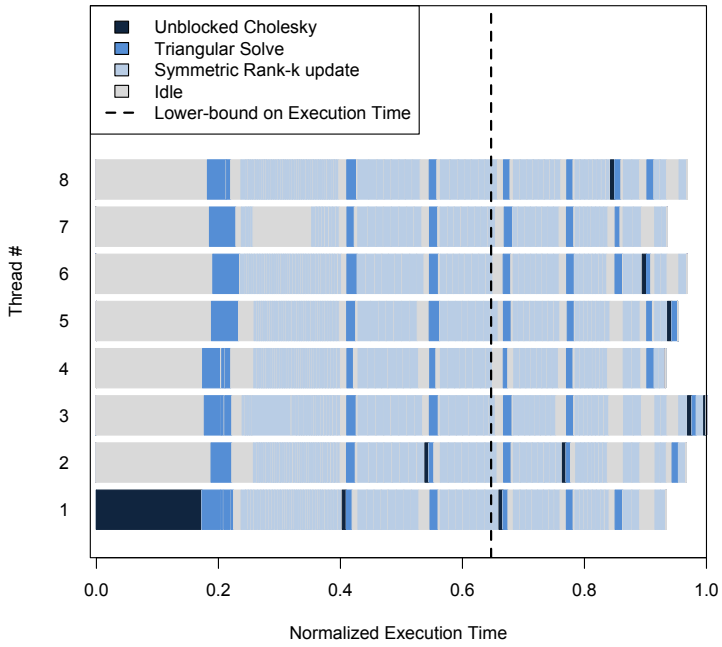


Figure 45: Scheduling timeline for Cholesky factorization using Cilk.

significant. In particular, recall from Figure 44 that the time spent on the factorization of the first block B_{11} is about 20% of the entire execution time. Hence, all other threads waiting for the input from the first step are requeued. When $n = 6000$, less than 1% of the time was spent on factoring the first block, and so the time spent on requeue events was only 0.56% of the overall execution time. Thus, the on-the-fly approach does not pay-off in all instances and, in fact, becomes a "tunable" parameter.

Comparison to bulk-synchronous approaches.

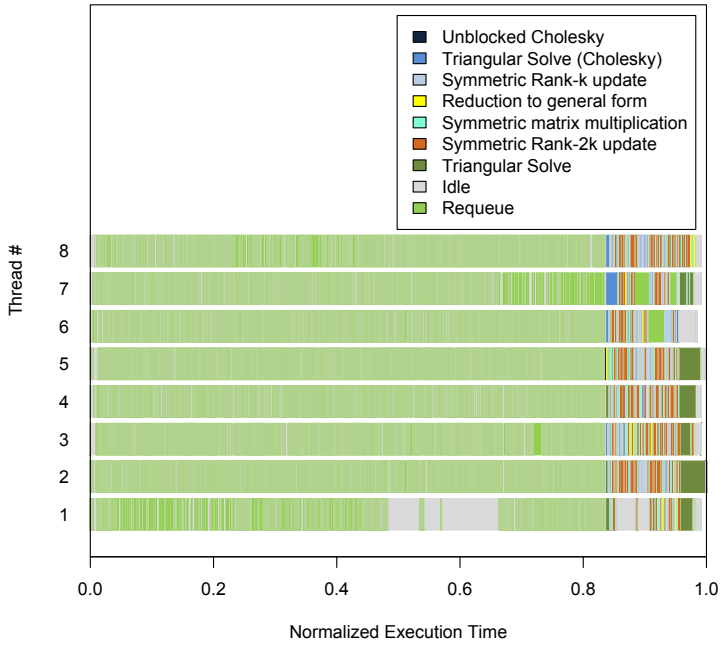
It is well-established that asynchronous scheduling can eliminate the idle times present in bulk-synchronous approaches. For example, Figure 45 shows the scheduling of Cholesky factorization in our Cilk++-based recursive implementation. There is a synchronization stage at the end of every task. Thus, the idle time increases because fast threads are waiting for the slower ones to complete execution. This behavior is a

consequence of our choice of a recursive implementation, which is encouraged by the Cilk++ model, as well as the model's nested DAG parallelism.

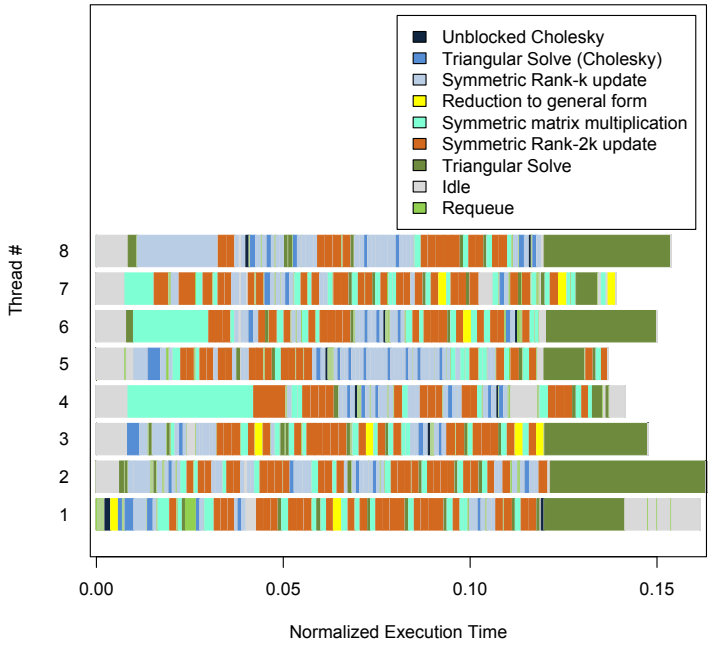
In the CnC implementation, there is no synchronization of tasks and the execution driven by tags imposes only one condition on preserving the data dependency between the steps. This eliminates idle time, as is evident when comparing Figures 6.4.3 and 45. Also, these figures show a vertical dotted line which is the estimated lower-bound on execution time. Given a weighted DAG (node weights measured as the time spent in the corresponding step, and the edge weights to be 0), the lower bound is computed by finding the longest path from the start to end (sequential steps along the critical path). We observe that CnC performs extremely well, within 10% of its lower bound; by contrast, the bulk-synchronous code performs well-below its potential.

Figure 46 shows scheduling of the eigensolver on Harpertown. The scheduling figure only shows the asynchronous-parallel portion of the eigensolver, which has the Cholesky and reduction to standard form components. Figure 46(a) shows how the scheduling unfolds when all the tags have been generated before the start of computation. In this part of the computation, more than 80% of the execution time is spent on requeue events. This behavior is due to the LIFO scheduling, where tags are scheduled last-in-first-out. Since the number of tag spaces and tags in each tag space are much larger compared to Cholesky factorization, the amount of requeue is significantly higher at start. However, the number of requeue events for the entire computation is only 33.3% of the execution time. Figure 6.4.3 shows an 85% reduction in overall execution time of the zoomed portion by generating tags only when input becomes available.

In short, we can achieve high performance in CnC, but there is still scope for additional improvements and tuning in the run-time system with respect to scheduling of steps, locality, and data movement.



(a) Pre-generated tags result in 33.3% of time spent on requeue events.



(b) Tags generated on-the-fly.

Figure 46: Scheduling for Eigensolver (matrix size = 1000)

6.5 FMM in CnC

In Chapter 3, we discussed a bulk-synchronous parallelization strategy for FMM. Using a model like CnC, we could exploit the fine grained asynchronous-parallelism in FMM effectively. Figure 33 shows the dependencies between the different phases of the FMM. A model like CnC will not only be able to schedule the different independent phases at the same time but at a much finer granularity than we have done before. For example, Figure 33 shows that there is a dependency between Up (leaf) and Up (non-leaf). But, the DAG fails to show that a non-leaf box depends only on a subset of the leaf boxes. Once these leaf boxes have been processed, the non-leaf box can be evaluated without waiting for the entire Up (leaf) phase to finish execution. CnC can under the hood handle the scheduling of these independent tasks which otherwise the programmer has to handle in our previously adopted OpenMP programming model.

This is the first attempt at mapping an irregular prototype of a real scientific application in CnC. Our goal in this study is to see if a general purpose programming model such as CnC can achieve performance on par with hand-tuned and hand-parallelized code with relatively low programmer effort. This raises a number of challenges compared to our previous work on mapping dense linear algebra in CnC mainly due to the data access pattern and complexity inherent in the FMM algorithm. We will discuss this in detail in the following section.

6.5.1 Mapping FMM in CnC

In the traditional OpenMP approach to parallelizing FMM, we adopt a bulk-synchronous strategy. We simply parallelize each stage of the FMM algorithm by adding a *parallel for* construct which loops over all the boxes or nodes in the tree. This approach is easy to implement but the biggest drawback is that it requires synchronization at the end of each stage along with redundant loads and stores of intermediate data. This is unscalable especially moving to future many-core platforms where the cost of

synchronization is expected to grow and cost of moving data more expensive.

CnC overcomes both of these drawbacks by scheduling tasks as soon as their dependencies are satisfied, thereby resulting in asynchronous execution with no explicit synchronization and also alleviating the need for storing intermediate values. A straightforward approach to converting a traditionally parallelized or sequential code to CnC is to assign a tag for each node in the tree. This approach exposes abundant parallelism and we discuss below how this works for each phase of the algorithm.

Up: The leaves of the *Up* phase are independent and can be scheduled as soon as the CnC graph is initialized. The parent nodes are dependent on the respective children and can be scheduled as soon as the 8 children in 3D (or 4 children in 2D) finish execution.

U-list: As the DAG in Figure 33 illustrates, the *U-list* phase is independent of all phases and is also ready to execute as soon as the CnC graph is initialized.

V-list: Each target box in this phase depends on 189 source boxes in its far-field which are produced by the *Up* phase and therefore, there is a data dependency between *V-list* and *Up* phases. Since we have parallelized at the granularity of boxes, each task cannot execute until all of its 189 dependencies are satisfied.

Down: This phase depends on the result from *V-list* and children at each level depend on the parent. Once the parent finishes execution, all of its children can be scheduled simultaneously.

Figure 47 shows the scheduling timeline generated using Intel's Trace Analyzer and Collector (ITAC) tool on a dual-socket quad-core Intel Sandybridge. The scheduling figure shows the different phases running asynchronously and in parallel on all the 8 cores. We don't observe any idle time. It also shows that the *V-list* phase of the computation gets requeued multiple times before finishing execution as seen by the number of calls to this step. This behavior is due to the current implementation of the CnC runtime where steps are scheduled as soon as the tags are available. Hence,

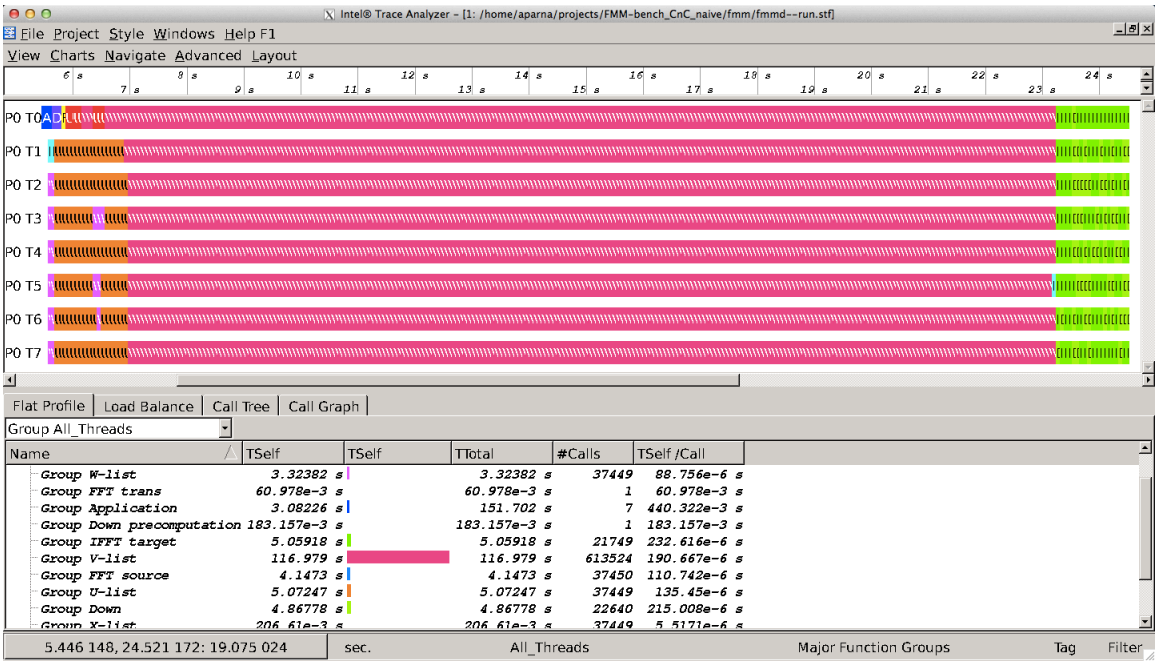


Figure 47: Scheduling timeline using ITAC for FMM (Naive) for an uniform distribution of 1 million points and $q = 100$.

if the data dependencies are not satisfied, the step gets requeued and this leads to significant performance degradation. We got around this bottleneck in dense linear algebra by generating tags only when the data became available. But, this becomes a challenge in the *V-list* phase of FMM because we are dependent on 189 distinct data items which are produced by 189 distinct instances of the *Up* step. Moreover, due to the CnC's execution model of decoupling *when* a step actually executes from its semantics, the application programmer has no knowledge of when all the dependencies are satisfied during the dynamic execution of the program.

6.5.2 Task-parallel V-list

As a result of these issues in the runtime system, the naive CnC implementation for the problem setup in Figure 47 was 4× slower than the corresponding hand-tuned OpenMP implementation for an identical configuration. To improve the performance of *V-list* in CnC, we enumerate a number of optimizations in this section that can be implemented at the application level.

1-D Blocking: One of the issues that causes performance degradation due to requeues is the number of dependencies that need to satisfied for the step to finish successful execution. The first optimization attempts to alleviate this problem by reducing the number of dependencies by blocking. The idea is to block along one dimension of the dependency tensor. This reduces the number of dependencies from 189 in the naive case to a chosen block size.

Figure 48 shows the scheduling timeline of the 1-D blocked implementation. Even though the total number of requeues remain relatively the same as in the naive case, we observe a speedup of 1.4× over the previous implementation. This is because we only need to satisfy a smaller number of dependencies compared to the previous case at any given time to complete the step, for instance, block size number of dependencies.

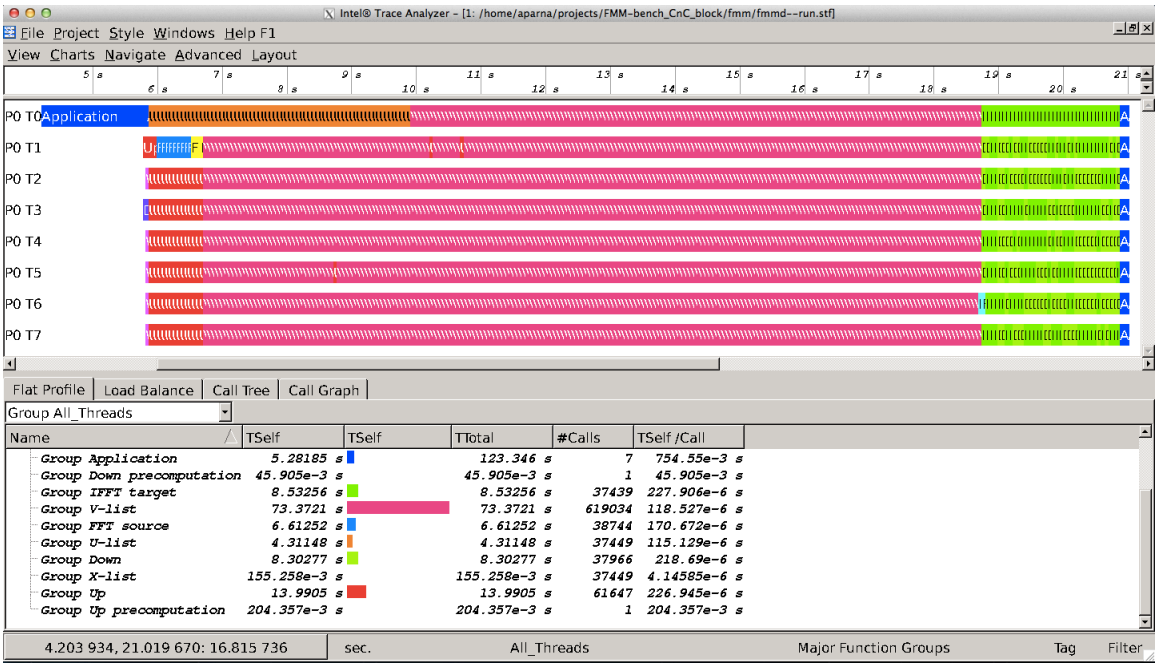


Figure 48: Scheduling timeline using ITAC for FMM with 1-D blocking for an uniform distribution of 1 million points and $q = 100$.

Geometrical blocking: The previous method reduces the time spent in checking if the dependencies are satisfied since we check a smaller number than the naive case. But, it doesn't reduce the total number of dependencies. One way to reduce the total

number of dependencies is to increase the task granularity.

In order to reduce the number of dependencies, we perform *V-list* at the grain of *clusters* instead of boxes. A *cluster* is defined as a group of 8 boxes in 3-D (or 4 boxes in 2-D) which have the same parent. The 8 or 4 boxes are siblings of one another. So, instead of computing 1-1 interaction between a target and source box, we now perform 8x8, a total of 64 interactions.

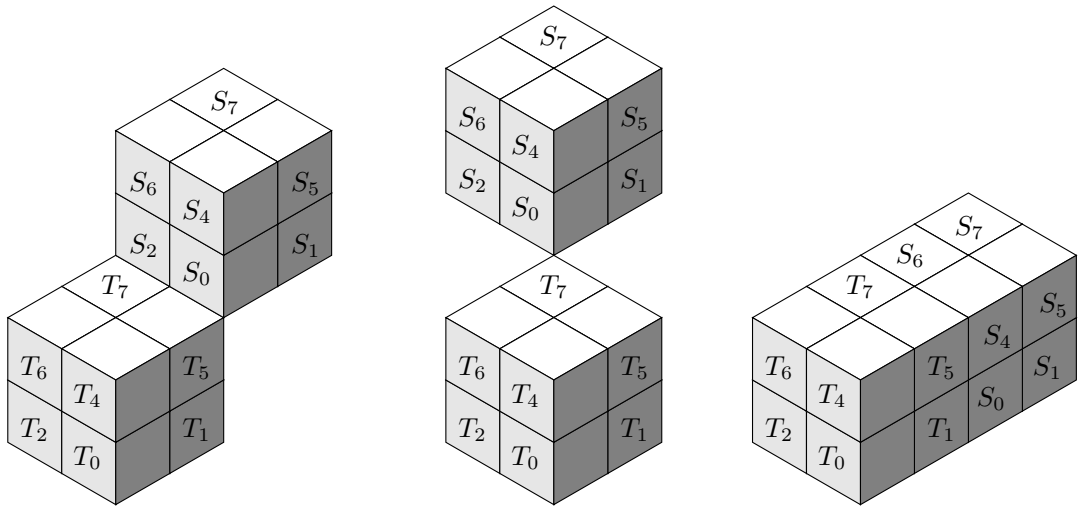


Figure 49: Geometrical Blocking: Diagram of three target-source clusters, namely cluster indexes 23, 26, and 14 along with their orientation in 3-D.

Figure 49 shows three example cluster orientations between pairs of source-target clusters. In 3-D there are 27 possible interactions between any two clusters as enumerated in Table 50. These include both near- and far-field interactions but the *V-list* only includes far-field interactions. So, specific to each source-target cluster, certain interactions listed in the table are not allowed and shouldn't be factored in the computation of *V-list*. Also, note that all the interactions in any row of the table have the same translation operator. This allows us to reuse translation operators and this blocking scheme exploits the geometrical properties of the algorithm, thereby increasing locality.

Ultimately, this reduces our total number of dependencies and the number of

Interaction-kind	Interactions (A pair 'i-j' denotes the interaction between T_i and S_j)
0	0-7
1	1-6
2	2-5
3	3-4
4	4-3
5	5-2
6	6-1
7	7-0
8	0-6, 1-7
9	0-5, 2-7
10	0-3, 4-7
11	1-4, 3-6
12	1-2, 5-6
13	2-4, 3-5
14	2-1, 6-5
15	3-0, 7-4
16	4-2, 5-3
17	4-1, 6-3
18	5-0, 7-2
19	6-0, 7-1
20	0-4, 1-5, 2-6, 3-7
21	0-2, 1-3, 4-6, 5-7
22	0-1, 2-3, 4-5, 6-7
23	1-0, 3-2, 5-4, 7-6
24	2-0, 3-1, 6-4, 7-5
25	4-0, 5-1, 6-2, 7-3
26	0-0, 1-1, 2-2, 3-3, 4-4, 5-5, 6-6, 7-7

Figure 50: Table of all possible interactions between any two target-source cluster in 3-D. The target and source clusters are indexed from $T_0 - T_7$ and $S_0 - S_7$ respectively.

dependencies for each step to 27, since each target cluster has 27 neighboring clusters. Figure 51 shows the scheduling timeline and we achieve a further speedup of 1.3× over the 1-D blocking. Unfortunately, we still have not completely eliminated requeues and hence slower than the equivalent OpenMP implementation by 2×.

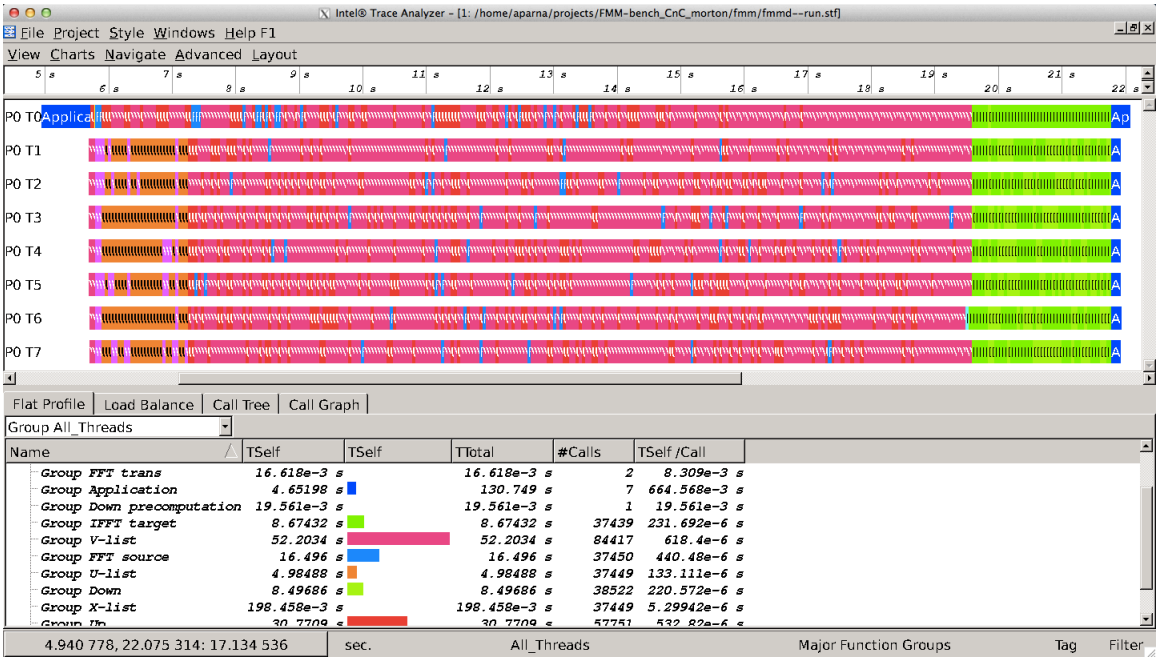


Figure 51: Scheduling timeline using ITAC for FMM with geometrical blocking for an uniform distribution of 1 million points and $q = 100$.

There is also a trade-off in this approach since coarsening the task restricts the amount of parallelism. There is still abundant parallelism in FMM on today's platforms and this optimizations is favorable. Ideally, we would like to expose all the parallelism and one could imagine a sophisticated version of the runtime system which dynamically varies grain size.

Fuse steps: The above optimization increases grain size, thereby reducing the number of steps in CnC. It also increases locality and reduces requeues. It is hard to quantify the performance improvement from each of these and to further understand the impact of runtime overhead, we fuse steps to reduce the total number of steps to be scheduled. If there is overhead in the runtime system due to scheduling steps, we

should see a performance benefit from fusing steps. Figure 52 shows the scheduling timeline with fewer steps and we infact see a speedup of 35% from fusing steps which indicates that there might be runtime overhead in scheduling and managing tasks.

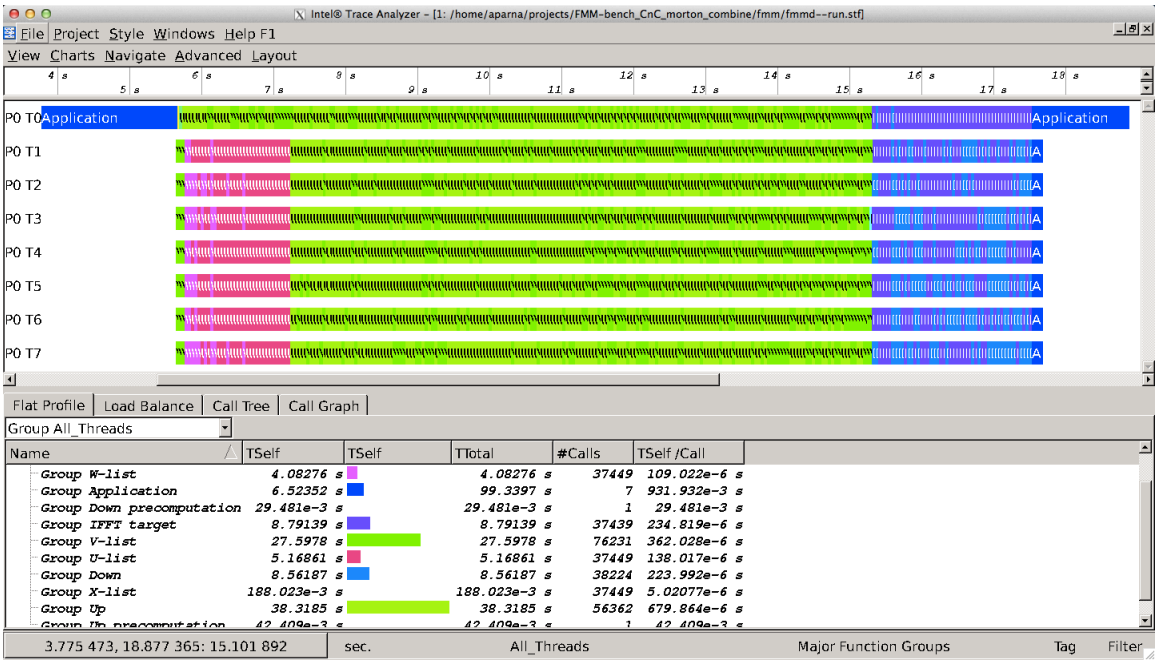


Figure 52: Scheduling timeline using ITAC for FMM after fusing steps for an uniform distribution of 1 million points and $q = 100$.

Input counter: To completely eliminate requeues, we implement *input counts* on the application side of CnC. The idea is similar to *get counts* in CnC where we maintain a global counter of the number of data dependencies for each step of *V-list*. Whenever a step produces data, it decrements the counter *atomically*. When the last data item is produced and the counter value becomes zero, a tag is created for the new dependent step to execute. As a result, a tag is produced only when all the data it is dependent on already exists, completely eliminating requeues.

We currently implement *input counts* using atomics, specifically using the function **__sync_fetch_and_sub (value, 1)**. It atomically fetches the value and decrements it by 1. Since there is proof that *get counts* can be efficiently implemented in the CnC runtime system, we could potentially also implement this in the runtime. Figure 53

shows the scheduling timeline with *input counts* implemented and we can see each step being executed only once. We also see a further speedup of 6% over the previous optimization. At this point, we are only a factor of less than 50% slower than OpenMP as opposed to the "naive" mapping which was 4× slower. Although, we have implemented all these optimizations in the user space, there is no fundamental reason why it cannot be translated into the runtime system making it transparent to the application programmer.

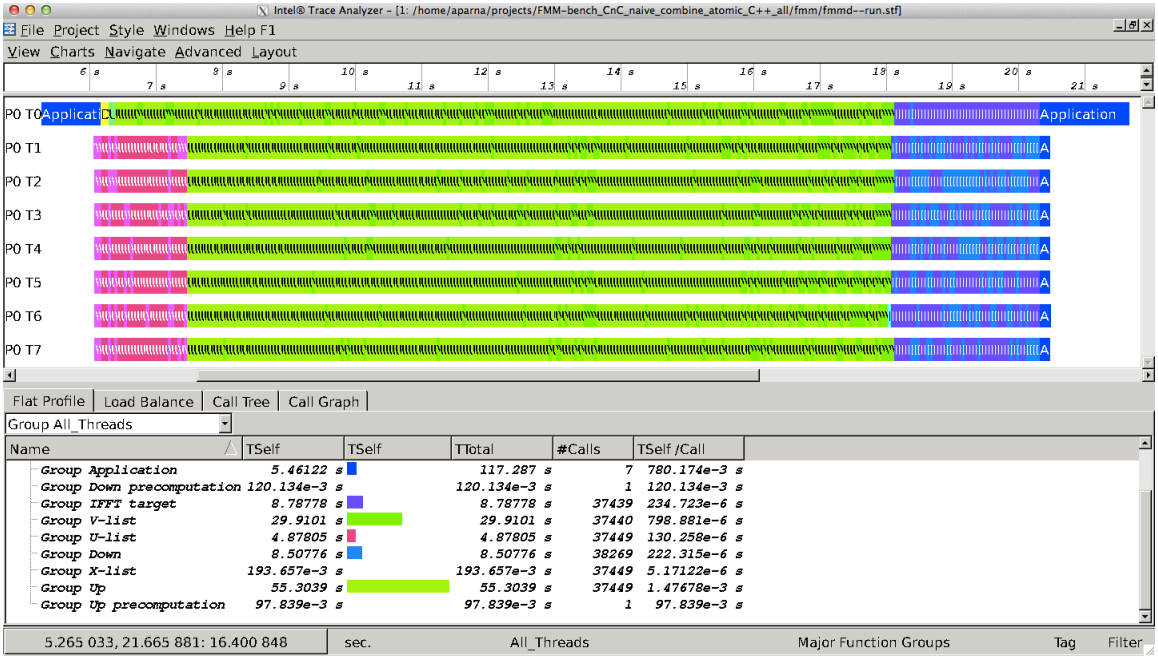


Figure 53: Scheduling timeline using ITAC for FMM with *input counter* implemented for an uniform distribution of 1 million points and $q = 100$.

Given that a future version of a CnC runtime system includes a *tuning expert* which incorporates all these tuning ideas and optimizations, we have demonstrated that it is possible to achieve performance close to hand-tuned code using a general purpose high-level programming model. The performance we observe is still slower than OpenMP, although intuitively we would expect an ideal asynchronous implementation to be faster than its equivalent bulk-synchronous variant. With respect to FMM, this could be due to the fact that our asynchronous implementation is not

ideal. NUMA is critical to FMM as observed in Chapter 4 and currently CnC has no notion of locality. There are current efforts on incorporating NUMA and above described optimizations in the CnC runtime and we hope to revisit this again in future.

6.6 Future Work

6.6.1 Extending CnC

Our experience reveals ways in which to improve CnC further. First, additional work queue scheduling policies (besides LIFO) are needed. Secondly, we can avoid run-time inefficiencies by exploiting additional dependence information available in the specification itself (textual notation). In our case, when the same tag collection prescribes multiple dependent step collections, we can reduce requeuing by not scheduling those collections. Thirdly, there are a number of ways in which tag management could be tuned, perhaps automatically. Finally, there are ways to enhance the textual notation and API; we are currently looking at adding new abstractions as well as new syntax for easily composing CnC components.

6.6.2 FMM in distCnC

Distributed CnC (DistCnC) is an extension of Intel's shared memory CnC to target distributed memory systems which is now part of the Intel's v0.7 CnC implementation [6]. It supports multiple communication backends including MPI. The parallel execution model is heterogeneous where each address space has a full shared-memory runtime to take advantage of multicore. DistCnC also supports a tuner that provides the programmer with a way to control the execution and communication of a program. For example, the programmer can specify *where* (on which core/processor) a step should execute and *when* a step should execute. A step can be delayed execution until all the programmer specified data items are available. We propose to implement FMM in distCnC since it seems promising as a productive high performance

alternative to current programming paradigms.

6.7 Summary

This study constitutes the first performance evaluation of the CnC model, with compelling results on a challenging pair of computations from parallel dense linear algebra. CnC complements existing approaches for expressing and scheduling asynchronous-parallel computations, by providing novel abstractions that enable a variety of control flow and dataflow constructs to be expressed in a way that enables effective parallelization. For our target computations, we can both (a) match or exceed a highly-tuned vendor library for Cholesky and (b) extend these results to significant speedups (1.1–2.6×) on a complicated eigensolver. Indeed, the CnC model *enabled* our novel asynchronous-parallel eigensolver implementation.

CHAPTER VII

PRACTICAL APPLICATIONS OF FMM

7.1 Direct Numerical Simulation of Blood

One of the target applications is the direct simulation of blood, which is a challenging multi-scale, multi-physics problem which we model as a mixture of a Stokesian fluid (plasma) and red blood cells (RBCs). We have designed an infrastructure called *MoBo*, Moving Boundaries, which was the first at this scale to capture the physics of cell deformation. Deformation is what enables cells to actually flow in very small capillaries, for instance; ignoring deformation leads to unrealistic simulations. We were able to simulate up to 200 million deformable RBCs, which improves upon prior state-of-the-art by four orders of magnitude and the optimized scalable FMM is one of the main components of this infrastructure [83].

7.1.1 Algorithm and Problem Formulation

Before diving into the algorithm and description of the role of FMM in MoBo, we will briefly describe the problem setup. The fluid flow model is a Stokes (creeping) flow, slow-moving and viscous fluid, in our case also assumed to be Newtonian. Differential equation is solved for resolving the forces on the membrane of an RBC and all RBCs

are assumed to be filled with the same Stokesian fluid as the surrounding plasma (requires evaluation of a single-layer potential; use double-layer potential for more general case).

The numerical method is based on a boundary integral formulation, in which the fluid is represented implicitly and we only need to discretize the cell boundary.[1]

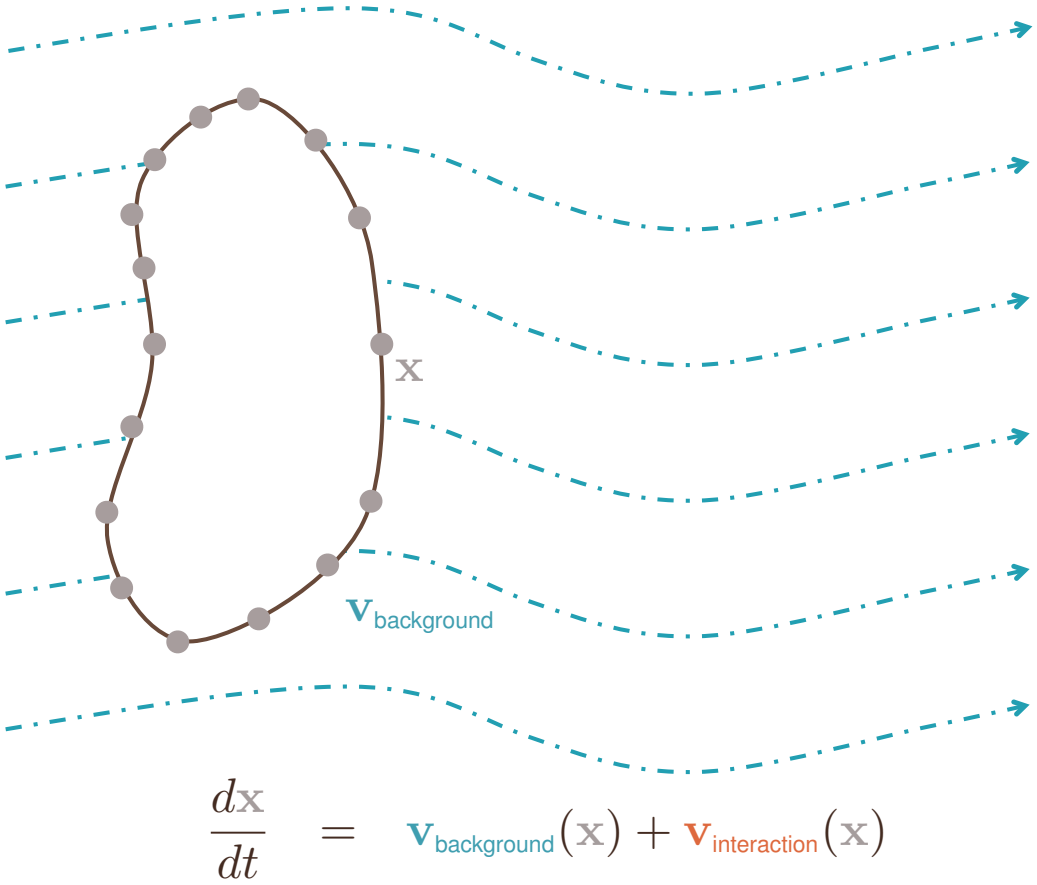


Figure 54: Vesicle flow: Model a red blood cell (RBCs) as fluid-filled deformable and inextensible sac in viscous solution.

There are two main challenges in implementing boundary integral methods. Firstly, stiffness, due to high-order derivatives required to accurately capture RBC deformations. We represent RBC's n a spherical harmonics basis, which permits accurate

[1]In contrast to Lattice Boltzman or finite-element methods, in which we discretize the entire domain.

high-order derivative computations. Second challenge is resolving and efficient evaluation of long-range interactions. We address this by evaluating long range interactions using our scalable FMM.

Mathematical formulation: Mathematical formulation for dynamics of the membrane is based on the Stokes equations, which model very low Reynolds number flows and incompressible linearly elastic solids. The equation for the evolution of the interface is given by

$$\frac{dx}{dt} = v(x), \quad \text{for } x \in \gamma_k, \text{ and all } k, \tag{31}$$

where x is the position vector of the points on γ_k, which denotes the interface between the k^{th} RBC and the surrounding fluid as shown in Figure 54.

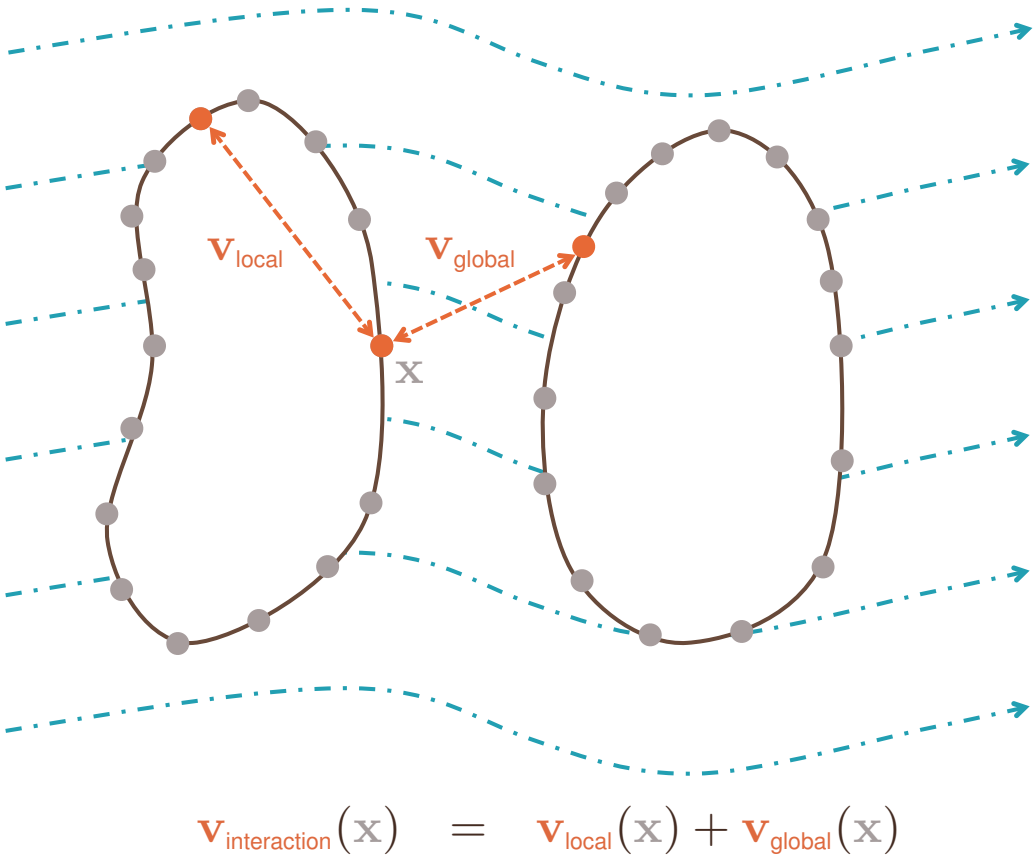


Figure 55: Local and global particle interactions between two RBCs.

The velocity v at a point x can be decomposed as follows.

$$v(x) = v_{background}(x) + v_{interaction}(x) \tag{32}$$

The first term $v_{background}$ is the velocity of the imposed background flow and the second term $v_{interaction}$ can be further decomposed into two terms as shown in Figure 55.

$$v_{interaction}(x) = v_{local}(x) + v_{global}(x) \tag{33}$$

The local velocity, v_{local} at a point x depends on the shape of the individual RBC. The global velocity is derived from the evaluation of the Stokes kernel.

Basic Algorithm: Given a set of n RBCs, where the k^{th} RBC is represented by the set γ_k of its surface points, the steps of the algorithm can be described as follows.

1. Compute the local velocity, $v_{local}(x)$ for all $x \in \gamma_k$ and all k.
2. Compute the global velocity, $v_{global}(x)$ using FMM.
3. Compute the background velocity, $v_{background}(x)$ using a user-supplied method analytically.
4. Update the position to the new position $x_{new} = x + \Delta t(v_{local}(x) + v_{global}(x) + v_{background}(x))$.
5. Periodically load re-balance or repartition.

We use an explicit Euler time-marching scheme and Δt is the time-step. The near field interaction or inter-cell computation given by the first step is pleasingly parallel over RBCs and the computation itself consists of several kernels described in detail in [83]. The time is primarily dominated by many small matrix multiplies. The global interaction or the second step of the algorithm involves an N-body calculation with the Stokes kernel. Using the KIFMM algorithm, we only need to evaluate the kernel at the given points.

7.2 Future Direction: Biomolecular Electrostatics

One interesting application for future work is biomolecular electrostatics. Electrostatics play a crucial role in biomolecular interactions. The challenge in understanding these interactions is the molecules are always in a solution (water molecules and dissolved ions). Molecular dynamics (MD) methods are the classical approach to studying these systems and are implemented in libraries such as CHARMM [26] and NAMD [81], but one could model the system using continuum electrostatics. The electric potential can be described by a Poission-Boltzmann equation which can be solved directly using a boundary-integral formulation discretized with boundary element method (BEM) [9, 59, 70].

The challenge in BEM is the computational cost of solving a large linear system. Krylov subspace iterative methods such as conjugate gradient (CG) [55], generalized minimum residual method (GMRES) [84], etc,. are efficient ways of solving the linear system. The computational kernel in these methods is a dense matrix vector multiply which can be calculated using the fast multipole method (FMM) in O(N) time enabling calculations with millions and billions of unknowns. We have already implemented FMM for the Laplace kernel, $\frac{1}{|r|}$ (Chapter 3) and extending it to include the Yukawa kernel, $\frac{\exp(-\kappa|r|)}{|r|}$ required for solving the linear system described below is straightforward since we use the kernel independent FMM.

7.2.1 Electrostatics in the Continuum Model

The continuum electrostatics based on the Debye-Huckel theory presented in 1923 describes the behavior of electric field in an ionic solution. The continuum electrostatic model we use in this thesis assumes a mixed dielectric medium – the molecule's interior denoted as region I in Figure 56 is modeled as a low dielectric with permittivity ϵ_1 (typical values between 2 and 10) and the solvent surrounding the molecule denoted as region II has permittivity ϵ_2. The solvent usually has a high dielectric

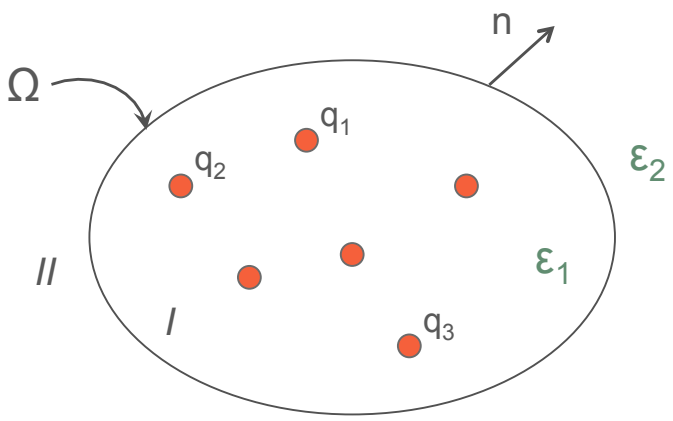


Figure 56: Sketch of a solvent molecule system with two continuum dielectric mediums. The interior of the molecule is a dielectric with permittivity ϵ_1 and the outside solvent region has permittivity ϵ_2.

constant (≈ 80), such as water. If there are no ions in the solution, the electrostatic potential satisfies a Poisson equation.

$$\nabla^2\phi_2 = -\frac{1}{\epsilon_2}4\pi\rho \tag{34}$$

If there are ions dissolved in the solvent medium, they will rearrange themselves according to the applied electric field. This causes the ions to be distributed according to the Boltzmann distribution and since the applied field that causes the ionic movement is electrostatic, we can model this as a Poisson-Boltzmann equation.

$$\nabla^2\phi_2(r) = -\frac{4\pi}{\epsilon_2}\sum_i c_i z_i q \exp\left(\frac{-z_i q\phi(r)}{k_b T}\right) \tag{35}$$

Here, c_i is the concentration of the ion i, z_i is the charge of the ion, q is the charge of the proton, k_b is the Boltzmann constant, and T is the temperature.

The Poisson-Boltzmann equation is usually used in the linearized form called the Linearized Poisson Boltzmann Equation (LPBE),

$$\nabla^2\phi_2 = \kappa^2\phi_2 \tag{36}$$

where $\frac{1}{\kappa}$ is known as the Debye length and is given by,

$$\kappa^2 = 4\pi \frac{\sum_i c_i z_i^2 q^2}{\epsilon k_b T} \tag{37}$$

The molecular charge distribution in the interior is modeled as n_c discrete point charges at the atom centers. Each of the i^{th} charges have value q_i at position r_i. The electrostatic potential in this region satisfies a Poisson equation.

$$\nabla^2 \phi_1(r) = -\sum_{i=1}^{n_c} \frac{q_i}{\epsilon_1} \delta(r - r_i) \tag{38}$$

At the dielectric boundary, Ω, the potentials and electric displacements are continuous and result in the following boundary conditions.

$$\begin{aligned} \phi_1(r_\Omega) &= \phi_2(r_\Omega) \\ \epsilon_1 \frac{\partial \phi_1}{\partial n}(r_\Omega) &= \epsilon_2 \frac{\partial \phi_2}{\partial n}(r_\Omega) \end{aligned} \tag{39}$$

Here, n is the outward pointing unit normal, pointing into region II from region I. Mathematically, we have represented this as a system of 4 equations; Poisson equation in region I (38), LPBE on the solvent side in region II (36), and boundary conditions on potential and electric displacement given by Equation (39).

7.2.2 Boundary Integral Formulation

A numerical approach commonly used is based on the integral formulation of the coupled system in the previous section [103]. The integral formulation of Equation (38) is given by the following equation.

$$-\frac{\phi_1(r)}{2} + \oint_\Omega \frac{\partial \phi_1(r')}{\partial n(r')} G_1(r, r') d\Omega' - \oint_\Omega \frac{\partial G_1(r, r')}{\partial n(r')} \phi_1(r') d\Omega' = -\frac{1}{\epsilon_1} \sum_i q_i G_1(r, r') \tag{40}$$

A very similar expression is obtained for Equation (36) which differs in the Green's function and the signs since the normals are pointing from region I into region II as shown in Figure 56.

$$-\frac{\phi_1(r)}{2} - \oint_\Omega \frac{\partial \phi_2(r')}{\partial n(r')} G_2(r,r')d\Omega' + \oint_\Omega \frac{\partial G_2(r,r')}{\partial n(r')}\phi_2(r')d\Omega' = 0 \tag{41}$$

Applying the boundary conditions in Equation (39) to the above equation, we get the formulation below which is valid for the potential, ϕ_1 inside the molecule.

$$2\pi\phi_1(r) - \oint_\Omega \frac{\partial \phi_1}{\partial n}(r')\frac{1}{|r-r'|}d\Omega' + \oint_\Omega \frac{\partial}{\partial n}\left[\frac{1}{|r-r'|}\right]\phi_1(r')d\Omega' = \sum_{i=1}^{n_c}\frac{q_i}{\epsilon_1}\frac{1}{|r-r_i|}$$

$$2\pi\phi_1(r) + \frac{\epsilon_2}{\epsilon_1}\oint_\Omega \frac{\partial \phi_1}{\partial n}(r')\frac{e^{-\kappa|r-r'|}}{|r-r'|}d\Omega' + \oint_\Omega \frac{\partial}{\partial n}\left[\frac{e^{-\kappa|r-r'|}}{|r-r'|}\right]\phi_1(r')d\Omega' = 0 \tag{42}$$

If we assume that there are no dissolved ions in the solvent, we can get a simpler formulation to get a better intuition into the physics of the process. The presence of a solvent with high dielectric constant (≈ 80), such as water, causes the external electric field to influence the orientation of the solvent molecules as a result of the *reaction field*. The solvent polarization appears as a layer of induced charge $\sigma(r)$ at the dielectric boundary.

$$\left(1-\frac{\epsilon_1}{\epsilon_2}\right)\left(\frac{\partial}{\partial n}\oint_\Omega \frac{\sigma(r')}{|r-r'|}d\Omega' + \frac{\partial}{\partial n}\sum_{i=1}^{n^c}\frac{q_i}{|r-r'|}\right) = \sigma(r) \tag{43}$$

The surface charge induces a potential in the molecule called the reaction potential, ϕ_{reac}.

$$\phi_{reac}(r) = \oint_\Omega \frac{\sigma(r')}{|r-r'|}d\Omega \tag{44}$$

If the molecule is in a vacuum, the electric potential would be just due to the bare Coulomb potential induced by the point charges. But, in the presence of a solvent, the solute charges polarize the solvent, creating a reaction potential in the solute. Hence the total potential, ϕ_1 is the sum of the reaction potential, ϕ_{reac} and the Coulomb potential. The electrostatic energy associated with the reaction potential

is equivalent to the work required to bring the molecule from vacuum into a solvent. This energy is called the solute's electrostatic *solvation energy*, ΔG_{solv}.

$$\Delta G_{solv} = \frac{1}{2}\sum_{i=1}^{n_c} q_i \phi_{reac}(r_i) \tag{45}$$

Some of the applications of the solvation energy are to estimate quantities such as electrostatic contributions to protein stability, calculating the binding affinity between two molecules, etc,.

7.2.3 Boundary Element Method

The first step in using BEM to solve Equation (42) is to the discretize the boundary Ω into n_p discrete non-overlapping panels or boundary elements. The integral equation in Equation (42) is solved for every panel Ω_i and is approximated as a sum of integrals over planar triangles, a common practice used in representing complicated geometries. We get a $2n_p \times 2n_p$ linear system of equation as shown below and we solve for ϕ_1.

$$\begin{bmatrix} 2\pi I + A & -B \\ 2\pi I - C & \frac{\epsilon_1}{\epsilon_2} D \end{bmatrix} \begin{bmatrix} \phi_1 \\ \frac{\partial \phi_1}{\partial n} \end{bmatrix} = \begin{bmatrix} Q \\ 0 \end{bmatrix} \tag{46}$$

The $n_p \times n_p$ matrices A, B, C, and D and n_p vector Q are defined as follows:

$$\begin{aligned} A_{ij} &= \int_{\Omega_j} \frac{\partial}{\partial n}\left(\frac{1}{|r_{ij}|}\right) d\Omega \\ B_{ij} &= \int_{\Omega_j} \frac{1}{|r_{ij}|} d\Omega \\ C_{ij} &= \int_{\Omega_j} \frac{\partial}{\partial n}\left(\frac{\exp(-\kappa|r_{ij}|)}{|r_{ij}|}\right) d\Omega \\ D_{ij} &= \int_{\Omega_j} \frac{\exp(-\kappa|r_{ij}|)}{|r_{ij}|} d\Omega \\ Q_i &= \sum_{k=0}^{n_c} \frac{q_k}{\epsilon_1} \frac{1}{|r_{ik}|} \end{aligned} \tag{47}$$

We consider a Galerkin approach and the integrals in Equation (47) are calculated using a simple Gauss quadrature rule, a single point located at the center of the panel. Since the BEM integral equation has a double layer potential which cannot guarantee a symmetric matrix, we currently use GMRES for solving the linear system. The matrix vector multiplication of GMRES which is the integral over each panel multiplied by the weights, can be written as a summation of the Laplace and LPBE Green's functions. In this formulation, we can use FMM to accelerate the matrix vector product resulting in linear runtime. Since we use the kernel independent FMM, implementing the Yukawa kernel, $\frac{\exp(-\kappa|r|)}{|r|}$ required for solving the linear system just requires evaluating the kernel. Moreover, one might potentially reuse all the optimizations used for Laplace kernel described in Chapter 3 .

A simple approach would be to begin with the two dielectric problem described in this chapter where the electrostatic field is calculated using two continuum dielectric medium, the solvent and the molecule. This is only a first step to solving biologically challenging problems which have more than two dielectric medium, ion-exclusion layers, and solvent filled cavities. Using FMM, one could potentially enable calculations with millions and billions of unknowns and present another practical scientific application of FMM.

CHAPTER VIII

CONCLUSIONS

This dissertation presents a top to bottom process for designing and implementing algorithms on current and future architectures, taking the Fast Multipole Method as a case study.

8.1 Summary

The main contributions and results of this thesis can be summarized as follows.

- **Practice: Algorithm engineering for FMM:** As discussed in Chapter 3, we presented the first extensive single-node performance study for FMM. This includes cross-platform evaluations of performance and scalability incorporating various compute- and memory-centric optimizations carefully tailored for the various phases of FMM. The benefits of optimizations, parallelization, and tuning are substantial and when combined, they delivered speedups of 25×, 9.4×, and 37.6× for Nehalem, Barcelona, and Victoria Falls respectively, in double precision for the uniform distribution. For a non-uniform, elliptical distribution, we observed speedups of 16×, 8×, and 24× respectively. This single-node study lays a solid foundation and is a building block for ultra-scalable FMM implementations on current and future systems. We describe the process of transforming

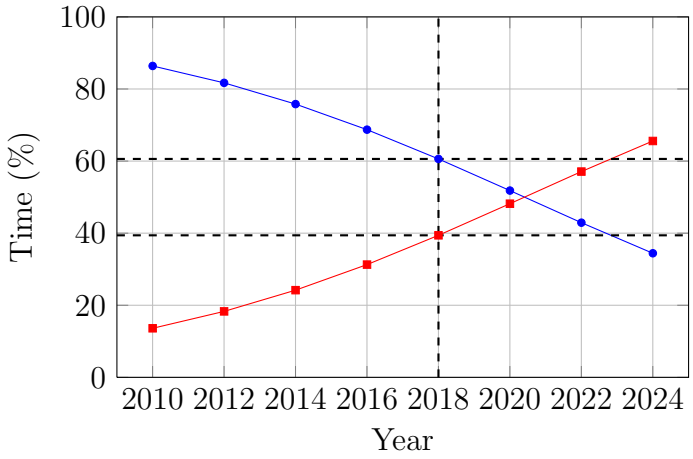
a conventionally parallelized code to a highly tuned one in Chapter 4.

- **Theory: Modeling and communication analysis:** The goal of performance modeling and analysis are two-fold – (a) to derive a more realistic performance model which includes both high-level architectural parameters, such as last-level cache capacity, bandwidth, peak floating point performance and also algorithmic parameters, such as number of particles, depth of the tree, to accurately evaluate the performance of the code, and (b) to gain insights into choosing the right parameters, schedule, and distribution on different architectures.

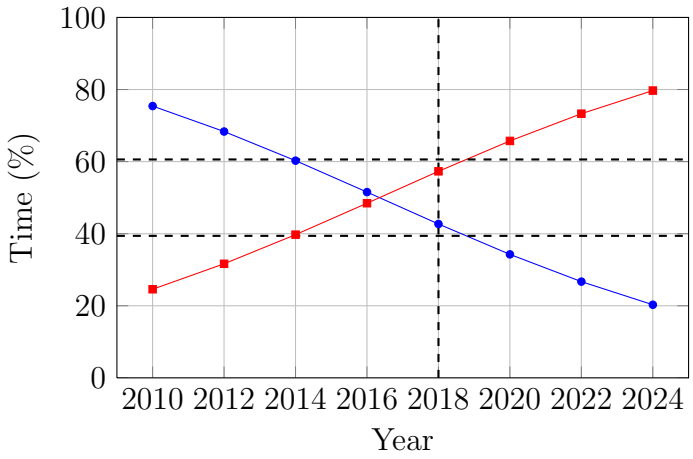
  To that end, we present the first in-depth models for compute and memory costs for FMM in Chapter 5. Our model predicts the optimal setting of one of the FMMs tuning parameters, which in practice had previously required manual experimentation. The model also provides solutions for practical performance engineering problems, such as how to schedule the computation for heterogeneous (e.g., CPU+GPU) systems.

Using both the above theory for the analytic expression for execution time and practice for a highly optimized scalable code, we predict the execution time for large-scale problem instances on possible future CPU-based exascale systems. The machine characteristics of the exascale system are based on extrapolating historical technology trends [94]. Figure 57 shows the execution time split into computational and memory access time for three different systems namely, (a) CPU-based system, (b) GPU-based system, and (c) a hybrid CPU-GPU system. We observe that the crossover point when the memory access time T_{mem} matches the compute time T_{comp} occurs at different time frames for each of these system configurations. This implies that the more *imbalanced* the system, the sooner we will observe the crossover.

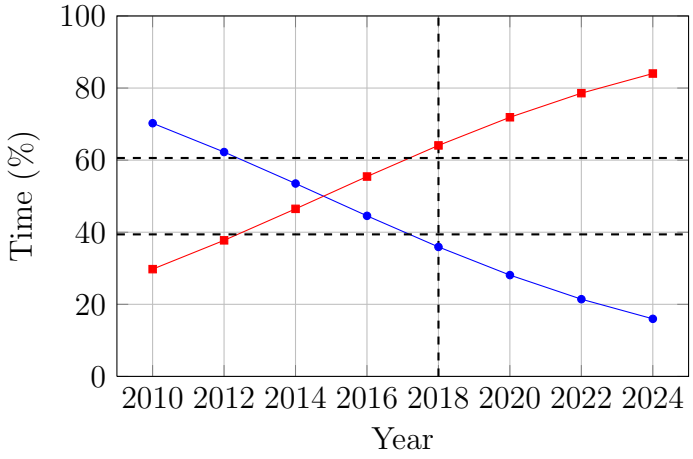
The hybrid computation with direct evaluation on the GPU (compute-bound) and far-field evaluation on the CPU (memory-bound) is highly imbalanced, and could



(a) CPU



(b) GPU



(c) Hybrid

Figure 57: A depiction of the KIFMM computational and memory costs for parallel execution on extrapolated systems. The problem size N starts at 4 million points in 2010, and is scaled at the same rate as the cache size Z.

become memory bound as early as year 2015, barring radical shifts in node balance. For problems that must be distributed across many processors, node balance is likely to play a significant role [41]; we will be analyzing and addressing the distributed case as part of our future work. In particular, this observation raises the question of what the architecture of an *ideal* multinode FMM processing system might look like.

If this prediction is true and the FMM will become memory bound, then it suggests that our memory-centric optimizations will only become more important in the future, while others may become less important. Specifically, optimizations such as re-organizing the data layout, inter-procedural loop fusion to compute matrices on-the-fly, NUMA-aware data allocation, cache and register blocking described in Section 3.3, Section 4.3, Section 4.5, and Section 6.5 become increasingly more critical compared to optimizing for floating point operations. This suggests that we need to re-think the design of future architectures and focus on minimizing communication costs when designing algorithms.

8.2 Future Work

Our model helps us answer some interesting questions that arise as we move towards exascale but it is only the tip of the ice-berg. There are other interesting questions that remain to be answered and we discuss them in this section.

8.2.1 Non-uniform Distributions

We limited our analysis to the uniform distribution which is a reasonable starting point. But for many practical applications, the distribution of points infact results in an adaptive tree. Modeling non-uniform distributions in general is a hard problem. Instead of a purely analytical model, a combination of offline micro-benchmarks to extract features of the distribution and a modification of our current model could result in a semi-analytical model for other unknown distributions.

For a non-uniform distribution, there are also a larger number of possible scheduling paths to choose from as shown in Figure 33. First, there are more ways to schedule the DAG itself, which makes the search space for the *best* schedule larger. Secondly, we can tune the number of points per box, q, which allows us to vary the execution time for the different phases of FMM to achieve maximum overlap. This lets us better utilize resources, but at the same time, we can no longer solely tune for U list and V list phases, since they may no longer be the dominant part of the computation. We present one chosen scheduling strategy which works relatively well in practice for the given architecture and implementation. There is definitely a strong need for a model-driven hybrid scheduling framework for non-uniform distributions.

8.2.2 Better Bounds

We currently make the assumption of an unified cache of size Z whereas, the GPU has private caches and our model does not capture this difference. In future work, we hope to extend our analysis to include different system configurations, for example, shared versus private caches, etc,.

8.2.3 Mixed precision

A popular optimization technique well known in literature is mixed precision arithmetic. We have not explored mixed precision in this thesis and as part of future work, we would like to profile the error of different phases of the FMM to understand how they contribute to the overall accuracy. Based on this analysis, we could experiment running the phases that have a lower contribution to overall accuracy in lower precision.

Our projections also indicate that we will become memory bound in the future because memory bandwidth is not expected to scale at the same rate as all the other system parameters. The traditional mixed precision technique changes the precision of the computation without modifying the precision of the data. This will become

insufficient in the future and we could further improve performance by storing data in lower precision too, thereby saving bandwidth consumption. This raises the question of how this affects the overall accuracy and if we could compensate by doing the computation in higher precision. We hope to explore these questions as part of future work.

REFERENCES

[1] "Intel® 64 and IA-32 architectures software developer's manuals." `http://www.intel.com/products/processor/manuals/`.

[2] "Intel®Concurrent Collections for C/C++: User's Guide, v0.3." `http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/`, 2009.

[3] "Intel®Threading Building Blocks." `http://www.threadingbuildingblocks.org/documentation.php`, 2009.

[4] "Open|SpeedShop™, version 1.9.3." `http://www.openspeedshop.org/wp/`, October 2009.

[5] "Intel® VTune™Performance Analyzer." `http://software.intel.com/en-us/intel-vtune/`, April 2010.

[6] "Intel®Concurrent Collections for C++: User's Guide, v0.7." `http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/`, April 2012.

[7] Ajmera, P., Goradia, R., Chandran, S., and Aluru, S., "Fast, parallel, GPU-based construction of space filling curves and octrees," in *Proc. Symp. Interactive 3D Graphics (I3D)*, (Redwood City, CA, USA), 2008. (*poster*).

[8] Alam, S. R., Bhatia, N., and Vetter, J. S., "An exploration of performance attributes for symbolic modeling of emerging processing devices," in *Proc. High Performance Computation Conference (HPCC)*, (Houston, TX, USA), September 2007.

[9] Altman, M. D., Bardhan, J. P., White, J. K., and Tidor, B., "Accurate solution of multi-region continuum biomolecule electrostatic problems using the linearized poisson–boltzmann equation with curved boundary elements," *Journal of Computational Chemistry*, vol. 30, pp. 132–153, January 2009.

[10] Amdahl, G. M., "Validity of the single processor approach to achieving large-scale computing capabilities," in *Proc. AFIPS Joint Computer Conf.*, vol. 30, (Atlantic City, NJ, USA), pp. 483–485, April 1967.

[11] Andersen, B. S., Gustavson, F., Karaivanov, A., Marinova, M., Waśniewski, J., and Yalamov, P., "LAWRA: Linear algebra with recursive algorithms," in *Appl. Par. Comput.*, vol. LNCS 1947, pp. 38–51, Springer, 2001.

[12] Anderson, C. R., "An implementation of the fast multipole method without multipoles," *SIAM Journal on Scientific and Statistical Computing*, vol. 13, pp. 923–947, July 1992.

[13] Appel, A. W., "An efficient program for many-body simulation," *SIAM Journal on Scientific and Statistical Computing*, vol. 6, January 1985.

[14] Arora, N., Shringarpure, A., and Vuduc, R., "Direct n-body kernels for multicore platforms," in *Proc. Int'l. Conf. Par. Proc. (ICPP)*, September 2009.

[15] Asanovic, K., Bodik, R., Catanzaro, B., and et al, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[16] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A., "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, February 2011.

[17] Barnes, B. J., Rountree, B., Lowenthal, D. K., Reeves, J., de Supinski, B., and Schulz, M., "A regression-based approach to scalability prediction," in *Proc. ACM Int'l. Conf. Supercomputing (ICS)*, pp. 368–377, June 2008.

[18] Barnes, J. and Hut, P., "A hierarchical $\mathcal{O}(n \log n)$ force-calculation algorithm," *Nature*, vol. 324, December 1986.

[19] Bellens, P., Perez, J. M., Badia, R. M., and Labarta, J., "Cellss: a programming model for the cell be architecture," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, (New York, NY), 2006.

[20] Blelloch, G. E., Chowdhury, R. A., Gibbons, P. B., Ramachandran, V., Chen, S., and Kozuch, M., "Provably good multicore cache performance for divide-and-conquer algorithms," in *Proc. Nineteenth Annual ACM-SIAM symposium on Discrete algorithms (SODA '08)*, (Philadelphia, PA, USA), pp. 501–510, Society for Industrial and Applied Mathematics, 2008.

[21] Blelloch, G. E., Gibbons, P. B., and Simhadri, H. V., "Low depth cache-oblivious algorithms," in *Proc. ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, (Thira, Santorini, Greece), July 2010.

[22] Blelloch, G. E., Maggs, B. M., and Miller, G. L., "The hidden cost of low bandwidth communication," in *Developing a Computer Science Agenda for High-Performance Computing* (Vishkin, U., ed.), pp. 22–25, New York, NY, USA: ACM, 1994.

[23] Blumofe, R. D., Jörg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y., "Cilk: An efficient multithreaded runtime system," in *Proc. PPoPP*, pp. 207–216, 1995.

[24] Board, J. and Schulten, K., "The fast multipole algorithm," *Computing in Science and Engineering*, vol. 2, pp. 76–79, January/February 2000.

[25] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J., "DAGuE: A generic distributed dag engine for high performance computing," Tech. Rep. ICL-UT-10-01, University of Tennesse Innovative Computing Laboratory, April 2010.

[26] Brooks, B. R., Bruccoleri, R. E., Olafson, B. D., States, D. J., Swaminathan, S., and Karplus, M., "Charmm: A program for macromolecular energy, minimization, and dynamics calculations," *Journal of Computational Chemistry*, vol. 4, no. 2, pp. 187–217, 1983.

[27] Budimlić, Z., Chandramowlishwaran, A., Knobe, K., Lowney, G., Sarkar, V., and Treggiari, L., "Multi-core implementations of the Concurrent Collections programming model," in *Proc. Wkshp. Compilers for Parallel Computing (CPC)*, (Zürich, Switzerland), January 2009.

[28] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J., "A class of parallel tiled linear algebra algorithms for multicore architectures.," Tech. Rep. UT-CS-07-600, Univ. of Tenn. Knoxville, Sep. 2007.

[29] Callahan, D., Cocke, J., and Kennedy, K., "Estimating interlock and improving balance for pipelined architectures," *J. Parallel Distrib. Comput.*, vol. 5, pp. 334–358, August 1988.

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

[31] Carriero, N. and Gelernter, D., "Linda in context," *Comm. ACM*, vol. 32, no. 4, pp. 444–458, 1989.

[32] Chan, E., Quintana-Ortí, E. S., Quintana-Ortí, G., and van de Geijn, R., "SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures," in *Proc. SPAA*, pp. 116–125, 2007.

[33] Chandramowlishwaran, A., Knobe, K., and Vuduc, R., "Performance evaluation of Concurrent Collections on high-performance multicore computing systems," in *Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, (Atlanta, GA, USA), April 2010. Winner, Best Paper (software track).

[34] Chandramowlishwaran, A., Knobe, K., and Vuduc, R., "Performance evaluation of Concurrent Collections on high-performance multicore computing systems," Tech. Rep. GT-CSE-10-01, Georgia Institute of Technology, Atlanta, GA, USA, February 2010.

[35] Chandramowlishwaran, A., Madduri, K., and Vuduc, R., "Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, (New Orleans, LA, USA), November 2010.

[36] Chandramowlishwaran, A., Williams, S., Oliker, L., Lashuk, I., Biros, G., and Vuduc, R., "Optimizing and tuning the Fast Multipole Method for state-of-the-art multicore architectures," in *Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, (Atlanta, GA, USA), April 2010.

[37] Coulaud, O., Fortin, P., and Roman, J., "Hybrid MPI-thread parallelization of the fast multipole method," in *Proc. ISPDC*, (Hagenberg, Austria), 2007.

[38] Coulaud, O., Fortin, P., and Roman, J., "High performance BLAS formulation of the multipole-to-local operator in the fast multipole method," *J. Comp. Phys.*, vol. 227, pp. 1836–1862, January 2008.

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

[40] Czechowski, K., Battaglino, C., McClanahan, C., Chandramowlishwaran, A., and Vuduc, R., "Balance principles for algorithm-architecture co-design," in *Proc. USENIX Wkshp. on Hot Topics in Parallelism (HotPar)*, (Berkeley, CA, USA), May 2011.

[41] Czechowski, K., McClanahan, C., Battaglino, C., Iyer, K., Yeung, P.-K., and Vuduc, R., "On the communication complexity of 3D FFTs and its implications for exascale," in *Proc. ACM Int'l. Conf. Supercomputing (ICS)*, (San Servolo Island, Venice, Italy), June 2012.

[42] Dennis, J. B., "First version of a data flow procedure language," in *Programming Symp.: Proc.*, vol. LNCS 19, pp. 362–376, Springer, 1974.

[43] Dongarra, J. and Sullivan, F., eds., *Computing in Science and Engineering: Special issue on The Top 10 Algorithms*, vol. 2, pp. 22–79. IEEE, 2000.

[44] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J., "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, pp. 173–193, June 2011.

[45] Fong, W. and Darve, E., "The black-box fast multipole method," *J. Comp. Phys.*, vol. 228, pp. 8712–8725, December 2009.

[46] Fowler, R. J., Gamblin, T., Porterfield, A. K., Dreher, P., Huang, S., and Joó, B., "Performance engineering challenges: The view from RENCI," *Proc. SciDAC, J. Physics: Conf. Ser.*, vol. 125, pp. 1–6, July 2008.

[47] Frigo, M. and Johnson, S. G., "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, 2005.

[48] Goto, K. and van de Geijn, R. A., "Anatomy of high-performance matrix multiplication," *ACM Trans. Mathematical Software (TOMS)*, vol. 34, p. 25pp, May 2008.

[49] Greengard, L. and Rokhlin, V., "A fast algorithm for particle simulations," *J. Comp. Phys.*, vol. 73, pp. 325–348, 1987.

[50] Gumerov, N. A. and Duraiswami, R., "Fast multipole methods on graphics processors," *J. Comp. Phys.*, vol. 227, pp. 8290–8313, 2008.

[51] Gustavson, F. G., "New generalized data structures for matrices lead to a variety of high performance dense linear algebra algorithms," in *Appl. Par. Comput.*, vol. LNCS 3732, pp. 11–20, 2006.

[52] Hager, G., Zeiser, T., and Wellein, G., "Data access optimization for highly threaded multi-core CPUs with multiple memory controllers," in *Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, pp. 1–7, April 2008.

[53] Hamada, T., Narumi, T., Yokota, R., Nitadori, K. Y. K., and Taiji, M., "42 TFlops hierarchical n-body simulations on GPUs with applications in both astrophysics and turbulence," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, (Portland, OR, USA), November 2009.

[54] Hariharan, B. and Aluru, S., "Efficient parallel algorithms and software for compressed octrees with applications to hierarchical methods," *Parallel Computing (ParCo)*, vol. 31, pp. 311–331, March–April 2005.

[55] Hestenes, M. R. and Stiefel, E., "Methods of conjugate gradients for solving linear systems," *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, 1952.

[56] Hillis, W. D., "Balancing a Design," *IEEE Spectrum*, 1987.

[57] Hockney, R. W. and Curington, I. J., "f1/2: A parameter to characterize memory and communication bottlenecks," *Parallel Computing*, vol. 10, pp. 277–286, May 1989.

[58] Hu, Q., Gumerov, N. A., and Duraiswami, R., "Scalable fast multipole methods on distributed heterogeneous architectures," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, (Seattle, WA, USA), November 2011.

[59] Juffer, A., Botta, E. F., van Keulen, B. A., van der Ploeg, A., and Berendsen, H. J., "The electric potential of a macromolecule in a solvent: A fundamental approach," *Journal of Computational Physics*, vol. 97, pp. 144–171, November 1991.

[60] Knobe, K., "Ease of use with Concurrent Collections (CnC)," in *Proc. USENIX HotPar*, 2009.

[61] Knobe, K. and Offner, C. D., "TStreams: A model of parallel computation," Tech. Rep. HPL-2004-78R1, HP Labs, 2004.

[62] Kung, H. T., "Memory requirements for balanced computer architectures," in *Proc. ACM Int'l. Symp. Comp. Arch. (ISCA)*, (Tōkyō, Japan), 1986.

[63] Kurzak, J. and Pettitt, B. M., "Massively parallel implementation of a fast multipole method for distributed memory machines," *J. Parallel Distrib. Comput.*, vol. 65, pp. 870–881, July 2005.

[64] Lashuk, I., Chandramowlishwaran, A., Langston, H., Nguyen, T.-A., Sampath, R., Shringarpure, A., Vuduc, R., Ying, L., Zorin, D., and Biros, G., "A massively parallel adaptive Fast Multipole Method on heterogeneous architectures," *Communications of the ACM (CACM)*, vol. 55, pp. 101–109, May 2012.

[65] Lashuk, I., Chandramowlishwaran, A., Langston, H., Nguyen, T.-A., Sampath, R., Shringarpure, A., Vuduc, R., Ying, L., Zorin, D., and Biros, G., "A massively parallel adaptive Fast Multipole Method on heterogeneous architectures," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, (Portland, OR, USA), November 2009. Finalist, Best Paper.

[66] Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C., *Quantitative system performance: Computer system analysis using queueing network models*. Upper Saddle River, NJ, USA: Prentice-Hall, 1984.

[67] Levon, J., "OProfile manual." `http://oprofile.sourceforge.net/doc/index.html`, 2004.

[68] Lipton, R. and Tarjan, R., "A separator theorem for planar graphs," *SIAM Journal on Applied Mathematics*, vol. 36, no. 2, 1979.

[69] Ltaeif, H., Kurzak, J., and Dongarra, J., "Scheduling two-sided transformations using algorithms-by-tiles on multicore architectures," Tech. Rep. UT-CS-09-637, Univ. of Tenn. Knoxville, 2009.

[70] Lu, B., Cheng, X., Huang, J., and McCammon, J. A., "Order n algorithm for computation of electrostatic interactions in biomolecular systems," *National Academy of Sciences*, vol. 103, no. 51, pp. 19314–19319, 2006.

[71] Mandal, A., Fowler, R., and Porterfield, A., "Modeling memory concurrency for multi-socket multi-core systems," in *Proc. IEEE Int'l. Symp. Performance Analysis of Systems and Software (ISPASS)*, March 2010.

[72] Mathis, H., Mericas, A., McCalpin, J. D., Eickemeyer, R., and Kunkel, S., "Characterization of simultaneous multithreading (SMT) efficiency in POWER5," *IBM Journal of Research and Development*, vol. 49, pp. 555–564, July/September 2005.

[73] McCalpin, J., "Memory bandwidth and machine balance in high performance computers," *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.

[74] McVoy, L. and Staelin, C., "lmbench: Portable tools for performance analysis," in *Proc. USENIX Ann. Technical Conf.*, (San Diego, CA, USA), January 1996.

[75] Nitadori, K., Makino, J., and Hut, P., "Performance tuning of n-body codes on modern microprocessors: I. Direct integration with a Hermite scheme on x86_64 architecture," *New Astron.*, vol. 12, pp. 169–181, 2006.

[76] Ogata, S., Campbell, T. J., Kalia, R. K., Nakano, A., Vashishta, P., and Vemparala, S., "Scalable and portable implementation of the fast multipole method on parallel comptuers," *Computer Phys. Comm.*, vol. 153, pp. 445–461, July 2003.

[77] Pase, D., "pChase benchmark." `http://pchase.org`, March 2008.

[78] Peng, L., Peir, J.-K., Prakash, T. K., Staelin, C., Chen, Y.-K., and Koppelman, D., "Memory hierarchy performance measurement of commercial dual-core desktop processors," *J. Sys. Arch.*, vol. 54, pp. 816–828, August 2008.

[79] Perez, J. M., Badia, R. M., and Labarta, J., "A flexible and portable programming model for smp and multi-cores," Technical Report 03, Barcelona Supercomputing Center - CNS, Barcelona, Spain, 2007.

[80] Perez, J. M., Badia, R. M., and Labarta, J., "A dependency-aware task-based programming environment for multicore architectures," in *Proc. IEEE CLUSTER*, pp. 142–151, 2008.

[81] Phillips, J. C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R. D., Kalé, L., and Schulten, K., "Scalable molecular dynamics with namd," *Journal of Computational Chemistry*, vol. 26, pp. 1781–1802, December 2005.

[82] Phillips, J. C., Stone, J. E., and Schulten, K., "Adapting a message-driven parallel application to GPU-accelerated clusters," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, (Austin, TX, USA), November 2008.

[83] Rahimian, A., Lashuk, I., Malhotra, D., Chandramowlishwaran, A., Moon, L., Sampath, R., Shringarpure, A., Veerapaneni, S., Vetter, J., Vuduc, R., Zorin, D., and Biros, G., "Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, (New Orleans, LA, USA), November 2010.

[84] Saad, Y. and Schultz, M. H., "Gmres: a generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, pp. 856 – 869, July 1986.

[85] Satish, N., Kim, C., Chhugani, J., Nguyen, A. D., Lee, V. W., Kim, D., and Dubey, P., "Fast sort on CPUs and GPUs: a case for bandwidth oblivious simd sort," in *Proceedings of the ACM SIGMOD International Conference on Management of data*, pp. 351–362, 2010.

[86] Shende, S. and Maloney, A. D., "The TAU parallel performance system," *Int'l. J. High Performance Computing Applications (IJHPCA)*, vol. 20, no. 2, pp. 287–311, 2006.

[87] Snavely, A., Carrington, L., Wolter, N., Labarta, J., Badia, R., and Purkayastha, A., "A framework for performance modeling and prediction," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, (Baltimore, MD, USA), November 2002.

[88] Song, F., Moore, S., and Dongarra, J., "Analytical modeling and optimization for affinity based thread scheduling on multicore systems," in *Proc. IEEE Int'l. Conf. Cluster Computing (CLUSTER)*, October 2009.

[89] Sundar, H., Sampath, R. S., and Biros, G., "Bottom-up construction and 2:1 balance refinement of linear octrees in parallel," *SIAM J. Sci. Comput.*, vol. 30, no. 5, pp. 2675–2708, 2008.

[90] Takahashi, T., Cecka, C., Fong, W., and Darve, E., "Optimizing the multipole-to-local operator in the fast multipole method for graphical processing units," *J. Numer. Meth. Engng.*, vol. 89, pp. 105–133, January 2012.

[91] Tallent, N., Mellor-Crummey, J., Adhianto, L., Fagan, M., and Krentel, M., "HPCToolkit: Performance tools for scientific computing," *Proc. SciDAC, J. Physics: Conf. Ser.*, vol. 125, 2008.

[92] Thies, W., Karczmarek, M., and Amarasinghe, S., "StreamIt: A language for streaming applications," in *Proc. Int'l. Conf. Compiler Construction (CC)*, vol. LNCS 2304, pp. 49–84, Springer, 2002.

[93] Tikir, M. M., Carrington, L., Strohmaier, E., and Snavely, A., "A genetic algorithms approach to modeling the performance of memory-bound computations," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, no. 47, November 2007.

[94] Vuduc, R. and Czechowski, K., "What gpu computing means for high-end systems," *IEEE Micro*, July/August 2011.

[95] Warren, M. S. and Salmon, J. K., "Astrophysical n-body simulations using hierarchical tree data structures," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, pp. 570–576, 1992.

[96] Warren, M. S. and Salmon, J. K., "A parallel hashed oct-tree n-body algorithm," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, (Portland, OR, USA), pp. 12–21, November 1993.

[97] Williams, S., Waterman, A., and Patterson, D., "Roofline: An insightful visual performance model for multicore architectures," *Comm. ACM (CACM)*, vol. 52, pp. 65–76, April 2009.

[98] YarKhan, A., Kurzak, J., and Dongarra, J., "Quark users' guide: Queueing and runtime for kernels," Tech. Rep. ICL-UT-11-02, University of Tennesse Innovative Computing Laboratory, April 2011.

[99] Ying, L., Biros, G., Zorin, D., and Langston, H., "A new parallel kernel-independent fast multipole method," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, (Phoenix, AZ, USA), November 2003.

[100] Ying, L., Zorin, D., and Biros, G., "A kernel-independent adaptive fast multipole method in two and three dimensions," *J. Comp. Phys.*, vol. 196, pp. 591–626, May 2004.

[101] Yokota, R., Narumi, T., Sakamaki, R., Kameoka, S., Obi, S., and Yasuoka, K., "Fast multipole methods on a cluster of gpus for the meshless simulation of turbulence," *Computer Physics Communication*, vol. 180, pp. 2066–2078, November 2009.

[102] Yokota, R. and Barba, L. A., "A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems," *Int. J. High-perf. Comput.*, 2011.

[103] Yoon, B. J. and Lenhoff, A. M., "A boundary element method for molecular electrostatics with electrolyte effects," *Journal of Computational Chemistry*, vol. 11, no. 9, pp. 1080–1086, 1990.

[104] Yotov, K., Li, X., Ren, G., Garzarán, M. J., Padua, D., Pingali, K., and Stodghill, P., "Is search really necessary to generate high-performance BLAS?," *Proc. IEEE*, vol. 93, pp. 358–386, February 2005.

[105] Yotov, K., Roeder, T., Pingali, K., Gunnels, J., and Gustavson, F., "An experimental comparison of cache-oblivious and cache-conscious programs," in *Proc. ACM Symp. Parallelism in Algorithms and Architectures (SPAA)*, (San Diego, CA, USA), pp. 93–104, June 2007.

[106] Zhong, Y., Orlovich, M., Shen, X., and Ding, C., "Array regrouping and structure splitting using whole-program reference affinity," *ACM SIGPLAN Notices*, vol. 39, pp. 255–266, May 2004.