UNIVERSITY OF CALIFORNIA,
IRVINE

A High-Performance Domain-Specific Language and Code Generator for General N-body Problems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Laleh Aghababaie Beni

Dissertation Committee:
Professor Aparna Chandramowlishwaran, Chair
Professor Alexander Ihler
Professor Brian Demsky

2019

DEDICATION

To my beloved Family.

"... a man who keeps company with glaciers comes
to feel tolerably insignificant by and by."

– Mark Twain, *A Tramp Abroad*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

Page

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Aparna Chandramowlishwaran, for her help during the course of my PhD. Also, I'd like to thank my thesis committee members Professor Alexander Ihler and Professor Brian Demskey for agreeing to serve on my candidacy exam committee and defence committee, for their invaluable time, dedication, contributions, and helpful rigorous feedbacks. I would also like to thank many other researchers and anonymous reviewers for their very helpful discussion and feedback, which have helped improve my research.

I am deeply grateful to Rosario Cammarota for his help and mentorship. I can not thank him enough for the numerous number of hours that he spent on discussion with me on several topics in research. Also, I am thankful to Professor Paul Dourish, Professor Nader Bagherzadeh, Andrea Bannigan, Melanie Sanders, and Kris Bolcer. Finishing this Ph.D. wouldn't be possible with the tremendous help and support of Dr. Phong Luong.

I am also grateful to my friends who were always there for me throughout my journey. I would like to thank Forough Arabshahi, Mahdi Abbaspour Tehrani, Hosein Mohimani, Maral Amir, Tahere Jabbari, Bahareh Mostafazadeh davani, Behnam Pourghasemi, Mehdi Sadri, Mehdi Ganji, Arash Gholami Davoodi, Korosh Vatanparvar, Omid Assare, Kasra Moazzemi, Sadjad Sedighi, Milad Asgari, Mehrdad Khatir, Ferran Marti, Anastasia Shuba and my amazing lab mates in HPC Forge, Hengjie Wang, Shu-mei Tseng, Rohit Zambre, and Octavi Obiols Sales.

I would like to thank Lars Bregstrom and Jack Moffitt, for giving me the two internship opportunities at Mozilla Research and their invaluable advice, guidance and patience during my internships period.

Most importantly, I cannot begin to express my sincere thanks to my family. My Mom, Dad, and siblings, without the unconditional love and support of whom I would have never been where I am today. I am forever grateful to them for providing me with the opportunity to pursue my education and for always encouraging me to seek a better life. I am deeply grateful to: Mahdi for teaching me how to think logically from a young age, Fahime for teaching me how to be brave and strong, Majid for teaching me how to be rational, Muhammad for teaching me how to do my best, Meysam for teaching me to never give up and never lose hope, MuhammadJavad for teaching me how to be kind. I am truly blessed to have MuhammadHosein, Elena, and Sara, as endless sources of joy through the darkest moments. I'd like to give special thanks to Mahmoud, who has been the source of encouragement and motivation. I could not have finished this work without his constant support, confidence, understanding, patience, and faith in me.

This dissertation was supported through by the U.S. National Science Foundation under award number 1533917. I would like to thank UCI HPC computing facility for providing access to their cluster and their technical staff for assistance. I would like to thank ICS department at UCI for supporting me with a dean's fellowship.

I would like to acknowledge IEEE for giving me permission to include Chapter 5 of this dissertation parts of which was originally published in IEEE Xplore. I would also like to thank the proceedings of Parallel Processing for giving me permission to include Chapter 3 in my dissertation which was originally published therein.

CURRICULUM VITAE

Laleh Aghababaie Beni

EDUCATION

Doctor of Philosophy in Computer Science **2019**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2014**
University of California, Irvine *Irvine, California*

Bachelor of Science in Computer Engineering **2012**
Sharif University of Technology *Tehran, Iran*

WORK EXPERIENCE

Research Engineer Intern **June 2014 – September 2014**
Mozilla Research, Mozilla Corporation *San Francisco, California*

Research Assistant Intern **June 2015 – September 2015**
Mozilla Research, Mozilla Corporation *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant:
Data structure implementation & Analysis (I&C Sci 46) **Spring 2019**
University of California, Irvine *Irvine, California*

Engineering Data structure & Algorithm (EECS 114) **Winter 2019**
University of California, Irvine *Irvine, California*

Data Structure Implementation & Analysis (I&C Sci 46) **Spring 2015**
University of California, Irvine *Irvine, California*

Programming with Software Libraries (I&C SCI 32) **Winter 2015**
University of California, Irvine *Irvine, California*

Programming with Software Libraries (I&C SCI 32) **Fall 2014**

University of California, Irvine *Irvine, California*

Fundamental Algorithms (COMPSCI 260) **Fall 2013**

University of California, Irvine *Irvine, California*

REFEREED CONFERENCE PUBLICATIONS

Portal: A High-Performance Language and Compiler for Parallel N-body Problems by Aghababaie Beni L., Ramanan S., Chandramowlishwaran A., *In Proceeding of International Parallel and Distributed Processing Symposium(IPDPS'19), 2019.*

PASCAL: A Parallel Algorithmic SCALable Framework for N-body Problems by Aghababaie Beni L., Chandramowlishwaran A., *In Proceeding of International European Conference on Parallel and Distributed Computing (Euro-Par'17), 2017.*

Parallel Performance-Energy Predictive Modeling of Browsers: Case Study of Servo by Zambre R., Bergstrom L., Aghababaie Beni L., Chandramowlishwaran A., *In Proceeding IEEE International Conference on High-Performance Computing, Data, and Analytics (HiPC'16), 2016.*

WebRTCBench: A Benchmark for Performance Assessment of WebRTC Implementations by Taheri, S, Aghababaie Beni L., Nicolau A., Veidenbaum A. V., Cammarota R., Haghighat M.R., *In Symposium on Embedded Systems for Real-Time Multimedia(ESTIMedia'15), 2015.*

A Compilation and Run-time Framework for Maximizing Performance of Self-scheduling Algorithms by Wang, Y, Aghababaie Beni L., Nicolau A., Veidenbaum A. V., Cammarota R., *In Proceeding International Conference on Network and Parallel Computing, 2014.*

Optimizing Program Performance via Similarity, Using a new Feature-agnostic Characterization Approach by Cammarota R., Aghababaie Beni L., Nicolau A., Veidenbaum A. V., *In The International Conference on Advanced Parallel Processing Technique, and as a Chapter in LNCS, Springer Verlang., 2013.*

Effective Performance Evaluation of Multi-core Based Systems by Cammarota R., Aghababaie Beni L., Nicolau A., Veidenbaum A. V., *In Proceeding IEEE International Symposium on Parallel and Distributed Computing, 2013.*

POSTER PRESENTATIONS

Parallel Performance-Energy Predictive Modelling of Browsers: Case Study of Servo 2016

by Zambre R., Bergstrom L., Aghababaie Beni L., Chandramowlishwaran A.,
In Proceeding ACM/IEEE Conference Supercomputing

PEAK: Parallel EM Algorithm using Kdtree 2015

by Aghababaie Beni L., Chandramowlishwaran A.,
In Proceeding ACM/IEEE Conference Supercomputing

ABSTRACT OF THE DISSERTATION

A High-Performance Domain-Specific Language and Code Generator for General N-body Problems

By

Laleh Aghababaie Beni

Doctor of Philosophy in Computer Science

University of California, Irvine, 2019

Professor Aparna Chandramowlishwaran, Chair

General N-body problems are a set of problems in which an update to a single element in the system depends on every other element. N-body problems are ubiquitous, with applications in various domains ranging from scientific computing simulations in molecular dynamics, astrophysics, acoustics, and fluid dynamics all the way to computer vision, data mining and machine learning problems [1, 2, 3, 4, 5, 6, 7, 8, 9]. Different N-body algorithms have been designed and implemented in these various fields. However, there is a big gap between the algorithm one designs on paper and the code that runs efficiently on a parallel system. It is time-consuming to write fast, parallel, and scalable code for these problems. On the other hand, the sheer scale and growth of modern scientific datasets necessitate exploiting the power of both parallel and approximation algorithms where there is a potential to trade-off accuracy for performance [10]. The main problem that we are tackling in this thesis is how to automatically generate asymptotically optimal N-body algorithms from the high-level specification of the problem. We combine the body of work in performance optimizations, compilers and the domain of N-body problems to build a unified system where domain scientists can write programs at the high level while attaining performance of code written by an expert at the low level.

In order to generate a high-performance, scalable code for this group of problems, we take the following steps in this thesis; first, we propose a unified algorithmic framework named PASCAL in order to address the challenge of designing a general algorithmic template to represent the class of N-body problems. PASCAL utilizes space-partitioning trees and user-controlled pruning/approximations to reduce the asymptotic runtime complexity from linear to logarithmic in the number of data points. In PASCAL, we design an algorithm that automatically generates conditions for pruning or approximation of an N-body problem considering the problem's definition. In order to evaluate PASCAL, we developed tree-based algorithms for six well-known problems: k-nearest neighbors, range search, minimum spanning tree, kernel density estimation, expectation maximization, and Hausdorff distance. We show that applying domain-specific optimizations and parallelization to the algorithms written in PASCAL achieves 10× to 230× speedup compared to state-of-the-art libraries on a dual-socket Intel Xeon processor with 16 cores on real-world datasets.

Second, we extend the PASCAL framework to build PASCAL-X that adds support for NUMA-aware parallelization. PASCAL-X also presents insights on the influence of tuning parameters. Tuning parameters such as leaf size (influences the shape of the tree) and cut-off level (controls the granularity of tasks) of the space-partitioning trees result in performance improvement of up to 4.6×.

A key goal is to generate scalable and high-performance code automatically without sacrificing productivity. That implies minimizing the effort the users have to put in to generate the desired high-performance code. Another critical factor is the adaptivity, which indicates the amount of effort that is required to extend the high-performance code generation to new N-body problems. Finally, we consider these factors and develop a domain-specific language and code generator named Portal, which is built on top of PASCAL-X. Portal's language design is inspired by the mathematical representation of N-body problems, resulting in an intuitive language for rapid implementation of a variety of problems. Portal's back-end is

designed and implemented to generate optimized, parallel, and scalable implementations for multi-core systems. We demonstrate that the performance achieved by using Portal is comparable to that of expert hand-optimized code while providing productivity for domain scientists. For instance, using Portal for the k-nearest neighbors problem gains performance that is similar to the hand-optimized code, while reducing the lines of code by 68×. To the best of our knowledge, there are no known libraries or frameworks that implement *parallel asymptotically optimal algorithms* for the class of general N-body problems and this thesis primarily aims to fill this gap. Finally, we present a case study of Portal for the real-world problem of face clustering. In this case study, we show that Portal not only provides a fast solution for the face clustering problem with similar accuracy as the state-of-the-art algorithm, but also it provides productivity by implementing the face clustering algorithm in only 14 lines of Portal code.

Chapter 1

Introduction

The main problem that we are tackling in this thesis is how to automatically generate asymptotically optimal N-body algorithms from a high-level specification of the problem that is natural for domain scientists. This is especially useful in rapidly growing fields such as machine learning and data mining where new models are created at a much faster rate than their optimal algorithms and implementations for the models.

1.1 Motivation

The critical role of N-body problems in a vast spectrum of scientific applications brings the need for optimized and scalable solutions. For example, one N-body problem, named k-nearest neighbors, has been used by researchers for early detection of Alzheimer disease. Alzheimer's Disease (AD) is a pathological form of dementia that degenerates brain structures. AD affects millions of older adults over the world and the number of people with AD doubles every year. While treatments focus on slowing the progression of this disease and controlling its symptoms, early diagnosis is vital. Today, researchers can detect AD using the Magnetic Resonance Imaging (MRI) of the brain by examining the biomarkers found in structural MRI. Techniques such as k-nearest neighbors have been employed to predict and discriminate AD and mild AD from healthy examples [4, 11, 12]. Also, the k-nearest

neighbors method has been used in the diagnosis of breast cancer in digitized mammograms. Breast cancer is one of the most dangerous cancers in the world, and the k-nearest neighbors classifier is one of the well-known methods used to distinguish between normal and abnormal tissues to classify tumors as malignant or benign [5, 13, 14].

Another influential N-body problem, named k-means, has been used widely in image processing and computer vision [15, 16, 17, 18]. For example, dealing with medical images, k-means clustering can provide efficient image segmentation for pre-surgery and post-surgery decisions helping in the recovery process. k-means is a useful method for automatic brain tumor segmentation for the extraction of tumor tissues from MRI images [8]. In another case, k-means clustering provides automatic detection of Malaria parasite tissues. Manual diagnosis of Malaria parasites by the pathologists is considered cumbersome; however, usage of k-means produces an efficient and accurate detection of Malaria parasite tissues [19].

Kernel Density Estimation (KDE) is another N-body problem which influences many applications such as crime prediction. The KDE method improves crime prediction performance by considering linguistic analysis to identify discussion topics and crimes automatically [9]. Moreover, KDE has been used in detecting violence and in general, detecting aggressive behaviors in videos, which is extremely useful in some video surveillance systems such as those used in psychiatric or elderly centers [20]. The adoption of KDE algorithm expands in applications such as profiling road accident hotspots [21]. Finding the road accident hotspots has a crucial role in deciding effective strategies for the reduction of high-density accidents' areas.

Above, I presented a few examples of N-body problems in different fields and the practical use cases of these problems in people's lives. Moreover, N-body problems have many more use cases and are prevalent in many other disciplines. For instance, k-nearest neighbors, expectation maximization, k-means, and naïve Bayes are four of the *top ten algorithms* having

the highest impact in data mining research according to a survey conducted in 2006 [22]. Furthermore, N-body methods were identified as one of the original seven dwarfs which are believed to be the computational kernels of many applications (A dwarf is an algorithmic method that captures a pattern of computation and communication) [23].

On the other hand, modern machines are becoming more and more complex causing even the most advanced compilers to fail in generating optimized code. Moreover, Proebsting's Law [24] states that improvements in compiler technology double the performance of *typical programs* every *18 years*. This leads to the trend of high-performance expert programmers who desire the best possible performance to write hand-tuned and hand-optimized code that outperforms compiler generated code. Unfortunately, this typically doesn't scale beyond implementing a single algorithm or problem, for one or a small subset of architectures. Even if we only consider the domain of interest for this thesis, there are hundreds of N-body problems, and it is practically impossible to generate hand-optimized code for every single one of them. Furthermore, hand-tuning is not only tedious but also highly machine-specific. Moreover, as the underlying architecture of the machines evolves, these hand-written codes become obsolete.

Domain scientists, who are experts in their particular domain, often lack expertise in parallel programming. In general, scientists prefer to program in high-level languages which allow concise expression of their problem. Matlab and Python are two examples that are widely used in data analytics [25]. However, achieving performance requires computation at a low level and in-depth knowledge of the underlying architecture. These are two examples of the natural tension between the software goals of performance and productivity. In the former, we have *performance programmers* who sacrifice productivity for performance and in the latter, we have *productivity programmers* whose primary goal is rapid prototyping. This motivates the need for an infrastructure to enable *both high performance and high productivity*. While facing the problem of productivity and performance in each domain, one

solution is to have domain specific languages and compilers fix this gap. There is potential for significant impact in this domain, and today general N-body applications are still orders of magnitude from optimal performance. To provide a high performance implementation for general N-body problems, we first present *PASCAL*, an algorithmic framework that brings all the N-body problems under one umbrella, then improve its performance in PASCAL-X, and finally introduce *Portal*, a domain-specific language and compiler embedded in C++ for general N-body problems which also provides productivity.

1.2 Summary of Contributions

In this thesis, we apply the knowledge and expertise gained from optimizing and tuning scientific N-body computations in order to provide high-performance, optimized, and parallel code for general N-body problems. The contributions of this thesis are as follows.

1.2.1 PASCAL

- [**Design of PASCAL Framework**] We design an algorithmic framework for general N-body problems, PASCAL, to automatically generate pruning and approximate conditions from a high-level user specification. PASCAL provides $\mathcal{O}(N \log N)$ and $\mathcal{O}(N)$ algorithms if the operators satisfy the decomposability property over subsets and kernel function monotonically decreases with distance. (Section 3.3)

- [**Optimization and Parallelization**] We apply domain-specific optimizations and then parallelize the algorithms developed in PASCAL. An asymptotically optimal algorithm developed in PASCAL combined with optimizations and parallelization results in $10-230\times$ speedup compared to state-of-the-art libraries and software such as Weka, Scikit-learn, MLPACK, and MATLAB. (Sections 3.5)

- [**Visit Order**] We introduce the concept of *visit order* while traversing the multi-tree.

The key idea is to prioritize visiting branches of the tree that have a higher likelihood of providing the right answer. This could result in better performance due to the increased pruning of sub-trees, depending on the N-body problem and the distribution of the dataset. We analyze the impact of *visit order* for six algorithms and five real-world datasets. (Section 3.9.2)

- [**Design of Nested Prune Generator**] PASCAL is the first framework that generalizes beyond two operators by the design of a *Nested Prune* generator. PASCAL is able to generate a nested prune condition for Hausdorff distance. To the best of our knowledge, this is the first multi-tree algorithm for Hausdorff distance. (Section 3.4)

1.2.2 PASCAL-X

- [**NUMA-aware Parallelization**] We extend the parallelization of PASCAL to provide better load balance for Non-Uniform Memory Access (NUMA) machines. PASCAL-X's NUMA-aware parallelization results in better scalability on systems with multiple NUMA nodes. We use parallel-producer multiple-executer task creation pattern in order to divide the task on different NUMA nodes. (Section 4.2)

- [**Analysis of Tuning Parameters**] We empirically evaluate the impact of algorithmic tuning parameters such as (1) leaf size, which influences the shape and granularity of the nodes in the tree and (2) cut-off level, which influences the number and granularity of tasks created during parallelization. Our results show these two parameters play a critical role in the performance, and by carefully tuning them, one could expect up to 4.6× speedup. (Section 4.3)

1.2.3 Portal

- [**Portal Language**] We design the Portal language, inspired by the mathematical formulation of N-body problems. Our high-level representation enables the compiler

not only to apply transformations and optimizations but also to choose an optimal algorithm. (Section 5.3)

- **[Real Code]** We develop a *domain-specific compiler* that chooses the optimal algorithm and generates optimized and parallel vector code for x86 architectures. Experimental results on six well-known problems show that the programs generated by Portal are within 5% (on average) of expert hand-tuned code in terms of performance. Additionally, we compare the *lines of code* of Portal programs against hand-optimized and library codes. For example, the Portal version of k-nearest neighbors is written in 13 lines of code and achieves approximately similar performance (within $2-5\%$) compared to the hand-tuned code. (Section 5.5)

- **[Portal Validation]** We also validate Portal against three separate N-body problems (namely, 2-point correlation, naïve Bayes classifier, and Barnes-Hut) that are not hand-optimized by us but that are part of state-of-the-art optimized libraries/frameworks. The parallel code generated by Portal using optimal tree-based algorithms outperforms libraries/packages such as scikit-learn [26] and MLPACK [27] by a factor of $15-165\times$ for the computation of 2-point correlation and naïve Bayes classifier. For the Barnes-Hut computation, we compare the performance of Portal against FDPS [28], which is a high-performance hand-optimized particle simulation framework in C++. Portal achieves 70% better performance compared to FDPS on a dual-socket AMD EPYC processor. (Section 5.5)

1.2.4 Case Study

- **[Extending Face Clustering]** In our case study, we consider a face clustering algorithm developed by [29] which includes two N-body problems as (1) computing the k-nearest neighbor for each face, and (2) clustering faces based on a defined metric distance. We extend the distance computation in face clustering algorithm [29] by

removing its unnecessary normalization, resulting in $\sim 2\times$ speedup. (Section 6.3.4)

- [**Extending Portal**] The distance metric used in the face clustering does not satisfy the kernel property (monotonically decreasing with distance). Therefore, we extended Portal to use the cover tree, resulting in tree implementation for face clustering. Also, we extended Portal to include *intersection* and *exist* functionalities for this case study. We option an additional 10% performance improvement compared to using a C++ external function for metric distance. By using this case study we evaluate the ability of Portal to deliver high-performance implementation for N-body problems. (Section 6.3.5)

Chapter 2

Generalized N-body Problems

In this chapter, we describe the class of *generalized N-body problems*, and their common structure. Generally, these are a group of problems which can be solved by considering each pair or N-tuples of points in a metric space. We review these N-body problems in different fields such as physics, machine learning, data mining, and computational geometry.

We also review different space partitioning trees, as the primary data structure used for the computation of general N-body problems in this thesis. These trees are sensitive to the intrinsic dimension of the data rather than their explicit dimension.

2.1 Problem Definition

In physics, the N-body problem can be defined as the problem of predicting the motion of N individual bodies or particles interacting with each other under a force law. The historical motivation comes from understanding the motion of celestial bodies in space, such as various planets of the solar system. Depending on the type of the force there are numerous applications such as the simulation of proteins and cellular assemblies in structural biology, or Coulomb force [30] which has the same form as gravitational force, with positive and negative charges resulting in repulsive as well as attractive forces. One of the most common

forms of these problems, which arises in physical simulations is as follows.

$$f(x_q) = \forall_q \sum_r \quad \mathcal{K}(x_q, x_r) \;\; s(x_r) \tag{2.1}$$

where $f(x_q)$ is the desired potential at query point x_q; $s(x_r)$ is the density at reference point x_r, and $\mathcal{K}(x_q, x_r)$ is an interaction kernel that specifies the physics of the problem. For instance, the single-layer Laplace kernel,

$$\mathcal{K}(x_q, x_r) = \frac{1}{||x_q - x_r||},$$

might model electrostatic or gravitational interactions. There are different algorithms for computing this force exerted on all of the query particles (x_q), and a direct calculation of this force results in $O(N^2)$ computation complexity.

General N-body problems are those in which an update to a single element in the system depends on every other element. The general form applies a set of operators $\{op_1, ..., op_m\}$ to m datasets using a kernel function, $\mathcal{K}$, as follows.

$$op_1, ..., op_m \quad \mathcal{K}(x_1, ..., x_m) \tag{2.2}$$

where $x_1 \in \mathcal{D}_1$,..., $x_m \in \mathcal{D}_m$ and $\mathcal{D}_1 ... \mathcal{D}_m$ are the m datasets. The naïve computation of these problems is asymptotically $O(N^m)$ which is expensive, especially for large datasets. For the rest of this thesis, for any N-body problem, we use the name *operator* for the operators (for example, in Equation 2.1 the set of *operators* are $op_1 = \forall$ and $op_2 = \sum$), and *kernel* function for the main computation function $\mathcal{K}$ between points.

There are computationally less expensive algorithms to reduce the complexity of these problems, which are called tree-methods [31]. Tree-method algorithms use tree data structures to decompose the particles or data points hierarchically. In this work, we will focus on tree-

based algorithms; other N-body methods such as particle mesh methods [32] are outside the scope of this thesis.

2.2 Data Structure: Space-partitioning Trees

Space-partitioning trees are a large class of tree data structure and are used as one of the primary methods in representing geometrical data [33]. These trees are usually built by recursively dividing a metric space into disjoint or non-overlapping subsets.

A powerful class of space-partitioning tree-based algorithms exists that can reduce the complexity of N-body problems from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$ or even $\mathcal{O}(N)$ [34, 2]. These algorithms use techniques such as approximation and pruning to estimate or discard regions of the space. Below we present some of the space-partitioning trees used in different N-body problems. Note the specific criteria of an ideal tree will depend on the type of problem as well as the distribution of the underlying data.

2.2.1 k-d tree

A k-d tree, short for k-dimensional tree, is a binary space-partitioning data structure for organizing data points in k-dimensional space. One of the significant advantages of this data structure is its ability to efficiently handle many types of queries, which are used in data analytics and mining [35].

This binary tree structure maintains bounding boxes for all the nodes in each level. The root node contains all the data points. Children are formed by recursively subdividing the bounding box space based on some splitting criterion. Each non-leaf node has two children, which has been defined by splitting dimension and splitting values presented by the splitting criterion. The splitting criterion results in a hyperplane which divides each space into two

parts. The points in the left side of the hyperplane are represented by the left subtree of that node, and similarly, the points on the right side represent the right child. The partitioning stops when each child node contains no more than l points ($l > 0$).

There are many different splitting criteria for constructing a k-d tree and generating splitting planes such as median of widest dimension and midpoint of widest dimension [36]. In this thesis, at each step, we chose the widest dimension of each node and use the median point as the splitting value. Choosing the median point leads to a more balanced k-d tree, in which each leaf node is approximately the same level distance from the root. This splitting criterion leads to generating k-d trees that are sensitive to the shape of the data density, in contrast to the fixed-size hypercubes provided by other data structures such as grids.

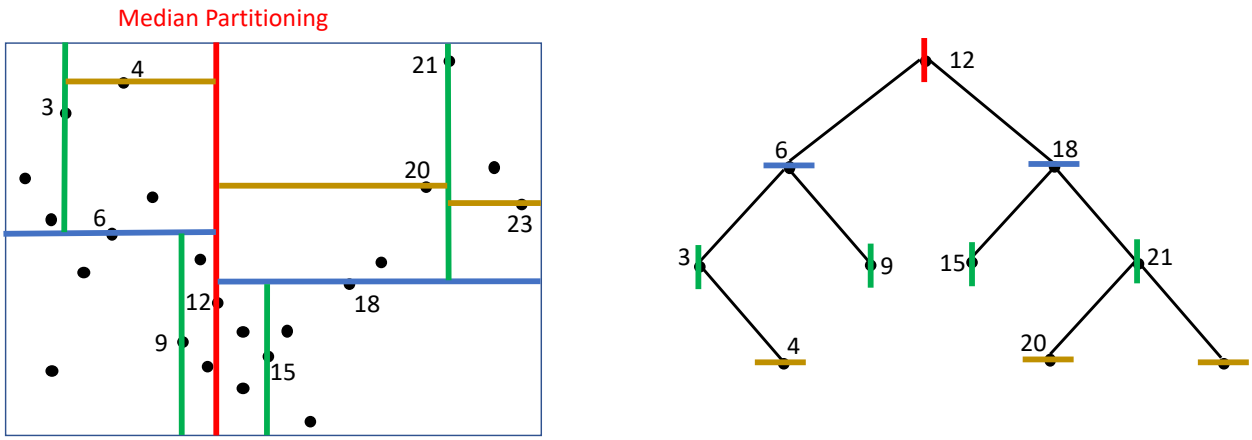


Figure 2.1: A visualization of the k-d tree constructed from a uniform random point distribution. The k-d tree is built by considering median partitioning over the widest dimension in each level for every bounding box.

These simple yet effective space-partitioning trees use bounding boxes, known as hyperrectangles, to define distinct data regions in the metric space. The bounding box information allows us to efficiently compute the minimum and maximum node-to-point or node-to-node distances during evaluation without accessing the actual points in each node, which is critical for performance. For the computation of bounding box information, since the hyperrectangles are axis-aligned, as we are building each node in the k-d tree we can store the minimum

and maximum values for each dimension and hence compute the minimum and maximum distance information. For problems which we need the center point or the number of data points in each node, a post-order traversal is used to compute this information.

An example of k-d tree construction is presented in Figure 2.1 for a uniform random point distribution of two-dimensional data using a median splitting condition. In Figure 2.1, the red vertical line is the median splitting plane (data point 12) on the widest dimension (x-axis), which results in two children. In the next recursion, the blue lines divide each child on their rectangles widest dimension (y-axis), which occur at data points 6 and 18. This recursive process continues until it reaches the stopping criteria (no more than l points in each leaf node). Each level of the tree is denoted by a different color.

2.2.2 Ball-tree

A ball-tree or metric tree is another space-partitioning tree for saving multi-dimensional data, with a wide range of practical applications [37, 38, 39, 40, 41]. In this tree, we recursively partition data points into a nested set of hyperspheres, known as balls. Similar to the k-d tree, ball-tree is a binary tree, and each node is defined as a d-dimensional hypersphere. Each hypersphere partitions the data points into a disjoint set. Even though, the hyperspheres may intersect, each data point is assigned to one or the other ball in the partition according to its distance from the ball's center.

There are several different construction algorithms available for ball-trees [42]. Omohundro [43] represented five different construction methods, which could be useful in different situations such as offline or online constructions. A similar method to k-d tree construction has been developed for ball-trees called the k-d construction algorithm, which is an offline top-down algorithm. This algorithm hierarchically splits each hypersphere into two sets from which ball-trees are recursively built as the right and left children of that node. The splitting

dimension is the one in which the data points are most extended, and the splitting value is the median data point. The recursive partitioning stops when each child node contains no more than l points ($l > 0$).

For computing bounding box information, as we are building the ball-tree, each node stores the centre and radius of its hypersphere. Using the centre and radius of each hypersphere allows us to find the minimum and maximum distance to each node. If a median split method has been used to build the ball-tree, the number of data points in each node is easily known by dividing the parent's node size into half. Otherwise, a post-order traversal is used to compute the number of data points in each node as well as the center. Here, the *centre* refers to a point with equidistant from the surface of hypersphere [44], and the *center* refers to the actual center computed using the data points in each node.

Figure 2.2 represents the construction of a ball-tree for a two-dimensional uniform random point distribution. The red ball represents the root of the ball-tree; the two children of the root, represented with blue balls, are made by splitting on the median of data points in the widest dimension.

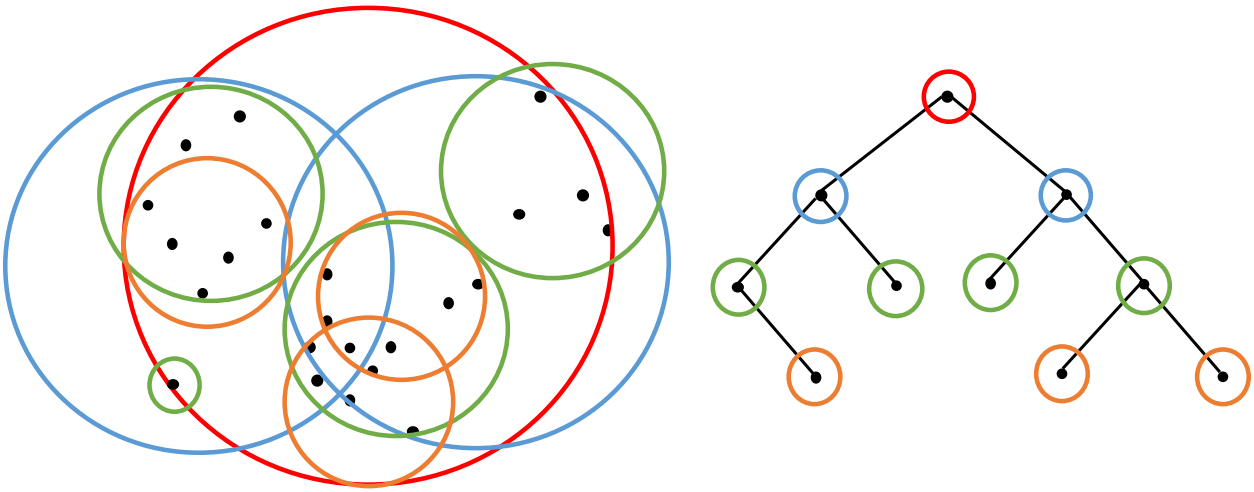


Figure 2.2: A visualization of the ball-tree constructed from a uniform random point distribution. The ball-tree is built by considering median partitioning over the widest dimension of data points in each level for every hypersphere.

2.2.3 Quadtree and Octree

Two well-known algorithms in physics N-body simulations are Fast Moltipole Method [2] and Barnes-Hut [34], which make use of low-dimensional spatial trees such as quadtrees in 2-dimension and octrees in 3-dimension. An octree is based on a similar principle as the k-d tree. The octree most often is used for partitioning 3-dimensional space by recursively subdividing the space into eight octants (or, four quadrants for 2-dimensional quadtree [45]). Similar to a k-d tree, the partitioning stops when each child node contains no more than l points (leaf size, $l > 0$). Octrees and quadtrees are different from k-d trees as they split around a point and into equal subregions, while k-d trees split along a dimension.

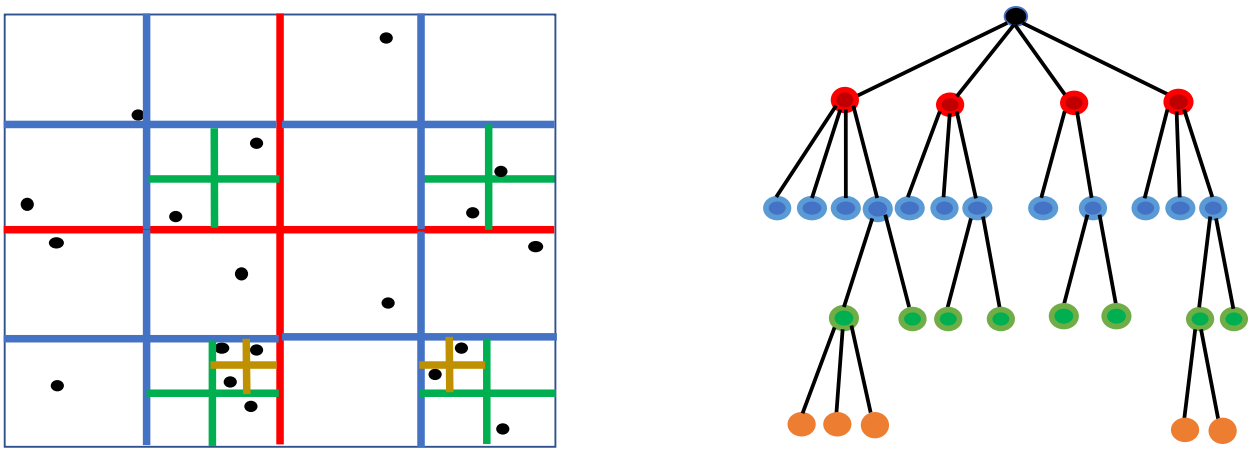


Figure 2.3: A visualization of quadtree constructed from a uniform random point distribution. The quadtree is built by dividing each region into four quadrant recursively.

For computing the bounding box information, as we are building the octree, we compute the side length of each octant node as the half side length of the parent node. We use a post-order traversal on the tree to measure the center and number of data points for each node. Same is true for quadtree in 2-dimension. Figure 2.3 illustrates the visualization of a quadtree (for 2-dimensional data), where each node is subdivided into four children (the octree in 3-dimension is analogous). A dedicated color denotes each level of the tree.

2.2.4 Cover tree

Another tree data structure used for partitioning metric space is cover tree. A cover tree T on a data set S is a leveled tree data structure of which each level is a *cover* for the levels beneath it. Each level is indexed by an integer scale i which decreases as the levels in the tree descend. Each node in the cover tree is associated with one data point in S, and covers a set of data points which are associated with the descendants of the node [46]. Each data point in S may associate with more than one node; however, any data point appears at most once in each level of the tree. Let C_i be the set of data points in S associated with the nodes in the level i of the cover tree, the following three invariants are true for all i in cover tree:

- **Nesting:** This rule implies that when a data point $v \in S$ appears at C_i, then every lower level of the tree has a node associated with data point v, as $C_i \subset C_{i-1}$.
- **Covering:** For every data point $v \in C_{i-1}$, there exist another data point $w \in C_i$ such that $d(v, w) < 2^i$. Also, the node in the level i associated with data point w is a parent of the node in level $i - 1$ associated with data point v.
- **Separation:** For all the distinct data points $v, w \in C_i$, the following rule applies: $d(v, w) > 2^i$

The original explanation of the tree requires an implicit tree representation with an infinite number of levels, but an explicit tree with a limited number of nodes gets implemented [47]. A recursive algorithm could be used to build the explicit cover tree from a data set S. In each recursion, a root node for each sub cover tree is built, then qualified child nodes and their covered data points are found. The recursion stops when there are no more than l points covered by the node (leaf size). The separation invariant is applied by disqualifying data points that are too close to the last root node.

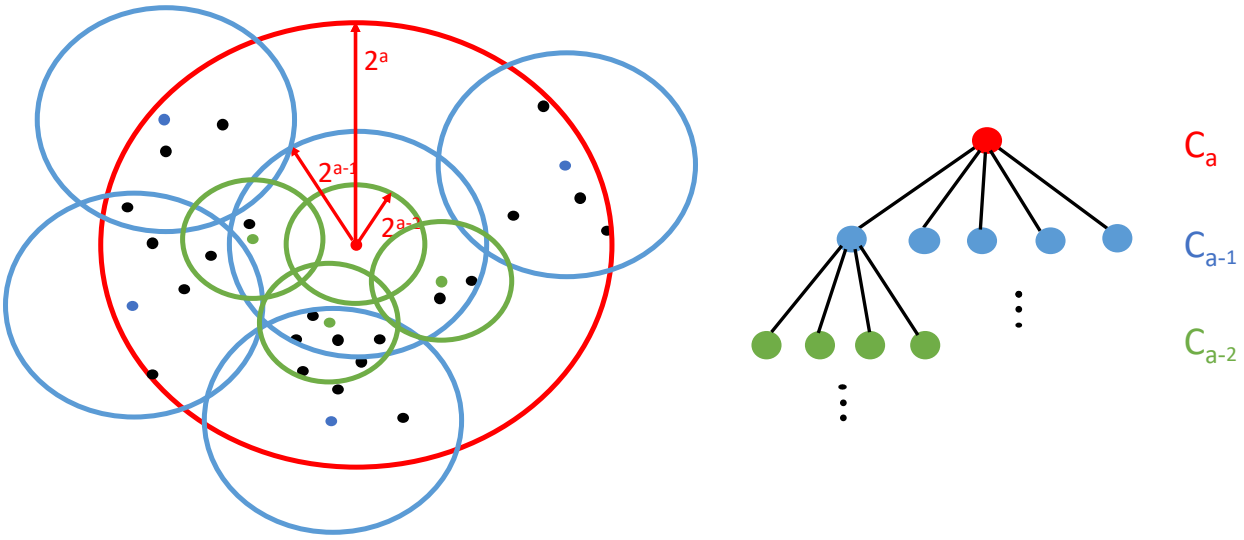


Figure 2.4: A visualization of cover tree constructed from a uniform random point distribution. C_i represents the set of data points in level i, and 2^i shows the radius of level i.

Each node and its covered data points compose a boundary ball, which is similar to hyper-rectangle in the k-d tree or hypersphere in the ball-tree. For computation of bounding box information, during the construction of the cover tree, each node stores the centre and radius of the boundary/cover ball. The radius for each node is half of its parent's radius. Using the centre and radius of each boundary/cover ball allows us to find the minimum and maximum distance to each node in our computation. A post-order traversal is used to compute the number of data points in each node as well as the center. Similar to ball-tree, the *centre* represents the point with equidistant from the surface of boundary/cover ball, and the *center* refers to the actual center computed using the data points in each boundary/cover ball.

Notably, a cover tree only relies on the metrics of a data set, and subsequently, depends on the intrinsic dimensions [48] of the data set and not the explicit dimension. Thus, the cover tree is a useful data structure when dealing with high dimensional data set with lower intrinsic dimensionality. Figure 2.4 shows a visualization of a cover tree (for 2-dimensional data), where each node is subdivided into various number of children (in the cover tree the number of children could be different from node to node). A dedicated color denotes each

level of the tree. The root node has a bounding circle with a large radius, R, in a way that would *cover* all the data points, $R = 2^a$ (first the radius R is measured from the dataset and a is computed as $a = \log_2 R$). The next level of the tree provides a bounding circle with a radius of 2^{a-1}, and so on. In each layer, the three invariants of the cover tree are valid.

2.2.5 Bounding Box Information

Table 2.1 represents the bounding box information for all the trees mentioned in Section 2.2.1 through 2.2.4. The details for computation of each bounding box information is explained in the corresponding section of each tree.

Tree	bounding box information
k-d tree	min, max, center
ball-tree	centre, radius, center
quadtree	side length of quadrant, center
octree	side length of octant, center
cover tree	centre, radius, center

Table 2.1: List of the trees and their bounding box information. Note that centre refers to the point with equidistant from surface of a ball, and center refers to the actual center computed using the data points in each node. Also, all the trees keep information about the number of data points in each node.

2.3 N-body Problems in Physics

N-body algorithms in physics are some of the most well-studied parallel computing problems. The original N-body problem was the problem of predicting the individual motion of celestial objects interacting with each other under the gravitational law, which has been motivated by the desire of understanding the motions of planets and stars. One of popular and widely used fast algorithms for classical gravitational N-body problems is Barnes-Hut [34] developed in 1986. Another well-studied problem in physics is the 2-point correlation function, which

describes the distribution of galaxies. In this section, we overview the Barnes-Hut algorithm as well as the 2-point correlation problem.

This style of N-body problem arises in other significant domains and the common theme that brings these problems under a single umbrella is the insight that their inner-loop computations are analogous and naively require $\mathcal{O}(N^2)$ operations for the all-pairs computation.

2.3.1 Barnes-Hut

The Barnes-Hut algorithm, introduced by J. Barnes and P. Hut in 1986 [34], employs a space-partitioning tree to approximately compute the potential of particles in the gravitational N-body problem. The key idea is to approximate the long-range forces by replacing a group of distant particles/bodies with their center of mass. A particle is considered long-range from a mass (square region as shown in Figure 2.5), if the distance r between the particle and center of mass is larger than a constant times the side length of the node, named E. This ratio is called the *Multipole Acceptance Criterion (MAC)* as $\theta = \frac{E}{r}$.

This algorithm approximates a set of particles in each node by its center of mass (as shown in the Figure 2.5). A smaller θ would result in better accuracy, meaning that the potential evaluated at a point is more accurate when it is far away from the node containing the particles [49]. Detailed analysis on the influence of θ on the accuracy is presented by Blelloch et al. [50].

At a high-level, the Barnes-Hut algorithm has the following main steps:

- Construct the space-partitioning tree for the particles as described in Section 2.2.3 by choosing a leaf size l.
- Compute the center of mass and total mass of particles for each node in the tree.
- For each target particle, traverse the tree to compute the force on it. If a node is far

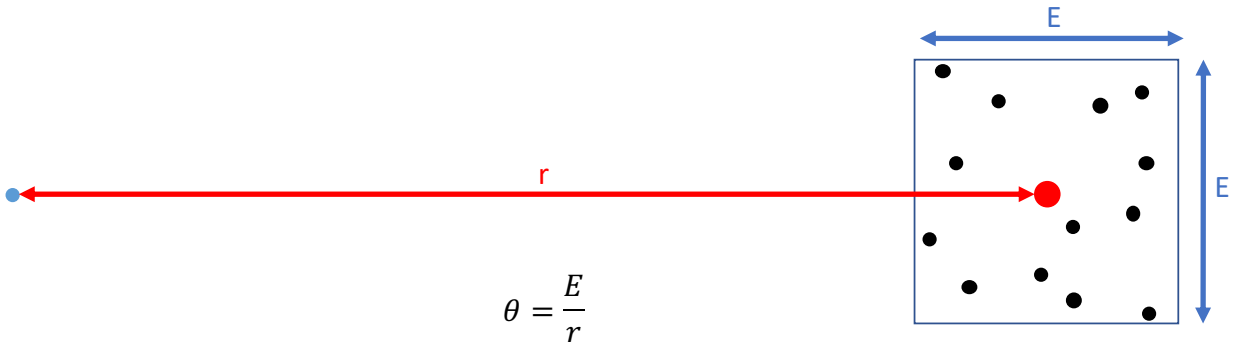


Figure 2.5: The potential of a set of particles (presented with black circles) in a node (square region of size E) from a single target particle (presented with blue circle) at distance r from the target particle could be approximated by the center of mass (presented with red circle).

from the target particle, we compute the potential considering the center of mass and total mass of that node. We decide to approximate if a node's computed MAC is less than a user-defined threshold. Otherwise, we continue to traverse the children of the node for non-leaf nodes or compute a direct evaluation for leaf nodes.

If we consider the depth of the tree in this algorithm as $\mathcal{O}(\log N)$, then generating the tree takes $\mathcal{O}(N \log N)$ in time, and the computation of centers of mass and total mass costs $\mathcal{O}(N \log N)$ time using a post-order traversal of the tree. Finally, the algorithm traverses the tree to compute the potential on each particle, resulting in $\mathcal{O}(N \log N)$ time complexity. Considering these main computations of the Barnes-Hut algorithm, the overall time complexity of this algorithm is $\mathcal{O}(N \log N)$. Hence, this algorithm reduces the computational complexity of the N-body problem from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$.

2.3.2 2-Point Correlation

In astronomy and physical cosmology, a correlation function is used for describing the distribution of galaxies in the universe. For quantifying the clustering of galaxies, we need to both survey the galaxies in clusters as well as the entire galaxy density distribution. The most commonly used quantitative measure of large scale structure is the 2-point correlation

function. The 2-point correlation function traces the amplitude of galaxy clustering as a function of scale.

Using the 2-point correlation function, one can trace the dependence of large scale structure on galaxy properties such as luminosity, color, and stellar mass, and track its evolution [51]. Alternatively, given a random galaxy in a location, the correlation function describes the probability of another galaxy being found within a given distance [52]. Some prominent applications of the correlation function can be found in areas such as galaxy clustering and weak lensing [53]. In general, the 2-point correlation can be considered as roughly the measure of the clumpiness of a set of points, and could be measured as the number of pairs of points in a dataset within a given radius from each other [54, 55].

The 2-point correlation function is defined as,

$$\sum_{q} \sum_{r} I(||x_q - x_r|| < h), \tag{2.3}$$

which counts the pair of points with a distance lower than a defined threshold h.

Both Barnes-Hut and 2-point correlation follow the general N-body problem representation in Equation (2.2).

2.4 N-Body Problems in Machine Learning and Data Mining

There are several significant problems in the machine learning (ML) field which have a similar pattern to physical N-body problems and can be categorized as examples of general N-body problems. It appears that these kinds of problems naturally arise across different fields. Here we address five examples of N-body problems as well their mathematical formulation, which follows the general representation of N-body problems in Equation (2.2).

2.4.1 k-Nearest Neighbors Search

One of the most ubiquitous N-body problems is k-nearest neighbors (k-NN) search, which appears in many fields such as machine learning, computational geometry, pattern recognition, computer vision, database, chemical similarity and many more [56]. It is a form of proximity search, for finding the points in a given set that are closest or most similar to a given point [57].

The mathematical representation of k-NN is defined as,

$$\forall q, \quad \arg\min_r^k \ ||x_q - x_r|| \tag{2.4}$$

where, for each query point x_q we want to find its k nearest neighbors and $\arg\min_r^k$ returns the k reference points x_r whose distance to x_q is minimal. Comparing this to our familiar physical summation in Equation (2.1), we see that the *kernel function* in this case is the Euclidean distance function and the operator `sum` has been replaced by another operator, `arg min`. Various solutions to this problem have been proposed. The usefulness of these approaches are determined by the time complexity of the queries of any search data structures that must be maintained in this search. In this thesis, we use space-partitioning trees in order to optimize the computation of this problem.

2.4.2 Kernel Density Estimation

Another example of sum-based accumulations from statistics is kernel density estimation (KDE), which is a widely used method to estimate the probability density function of data from an unknown distribution. The common approach to density estimation is parametric, meaning that we assume that the data are drawn from one of a known parametric family of

distributions, for example the Gaussian normal distribution. The underlying density of the data could be estimated by using kernel density estimation method [58, 59, 60, 61]. The goal is to estimate the probability density at each x_q, using a kernel function K_σ and the kernel estimator is defined as,

$$\forall q, \quad \frac{1}{|R_{size}|} \sum_r K_\sigma \left(\frac{||x_q - x_r||}{\sigma} \right)$$

where K_σ is a zero-centered probability density function (*e.g.*, a Gaussian); R_{size} is the size of reference dataset, and σ is called the bandwidth of the kernel. When the distance between two points $||x_q - x_r||$ is very large, the contribution of the kernel function to the probability density at x_q is small. Therefore, we can approximate the kernel sum at the expense of reduced precision to achieve a faster algorithm similar to Barnes-Hut.

2.4.3 Minimum Spanning Tree

Minimum Spanning Tree (MST) is one of the oldest and most well-studied problems in machine learning and computational geometry. Given a set of points $S \in \mathbb{R}^d$, the goal is to find the lowest weight spanning tree in the complete graph G, where the edge weights are given by the Euclidean distance between two points. Generally, any edge-weighted connected undirected graph has a minimum spanning tree which has the minimum possible total edge weight. There are different use cases for minimum spanning trees, such as telecommunications companies trying to lay out cables in a new neighborhood while constrained to use cables only on certain paths. There exist different algorithms for finding the MST such as Boruvka's algorithm [62, 63], Prim's algorithm [64], Kruskal's algorithm [65], and reverse-delete algorithm (reverse of Kruskal's algorithm) which are considered to be greedy algorithms. In this thesis, we consider the iterative Boruvka's algorithm for minimum spanning tree (MST), which proceeds as a sequence of stages:

1. Initially all the edges of graph G are uncolored, and we assume each vertex of G is a single-node blue-colored tree.

2. Repeat the following coloring step until we find only one blue tree that includes all the nodes of G.

3. For every blue tree, select the minimum-weight uncolored edge incident to that blue tree. Color all selected edges blue.

Borukva's MST is an iterative algorithm that connects each component to its nearest vertex until only a single component, the MST, remains. The computational bottleneck in MST is finding the nearest neighbor component, which is identical to k-nearest neighbors search. Thus, calculating all neighbor pairs effectively results in an efficient Boruvka's algorithm.

2.4.4 Expectation Maximization

EM is a popular algorithm used in mixture models. Here, we consider a problem where EM is used to learn the parameters of a multivariate Gaussian Mixture Model (GMM). Consider a dataset $D = \{x_1, x_2, .., x_N\}$, where $x_i \in \mathbb{R}^d$ generated independently from an underlying distribution $p(x)$. If $p(x)$ is a Gaussian distribution, we can define a GMM as,

$$p(x|\theta) = \sum_{j=1}^{\mathcal{H}} \pi_j f(x|\mu_j, \Sigma_j), \tag{2.5}$$

$$f(x_i|\theta_j) = \frac{1}{\sqrt{2\pi|\Sigma_j|}} e^{-\frac{1}{2}(x_i-\mu_j)^T \Sigma_j^{-1}(x_i-\mu_j)} \tag{2.6}$$

where $\mathcal{H}$ is the number of Gaussian mixture components, and $\theta_j = \{\mu_j, \Sigma_j\}$ are the parameters of Gaussian component j, with mean vector μ_j and covariance matrix Σ_j. The π_j are called the mixture weights ($\sum_{j=1}^{\mathcal{H}} \pi_j = 1$).

EM starts with an initial estimate of θ, generated randomly, and iteratively updates θ until convergence as follows.

1. Initialize θ randomly.

2. Repeat until convergence (convergence condition is checked by log-likelihood, the third step).

   i. **E-step**: Compute the *responsibility*, r_{nj} (which is representable as the weight factor of data point n for cluster j)

$$r_{nj} = \frac{\pi_j f(x_n|\mu_j, \Sigma_j)}{\sum_{i=1}^{\mathcal{H}} \pi_i f(x_n|\mu_i, \Sigma_i)} \tag{2.7}$$

   ii. **M-step**: Re-estimate θ using the responsibilities computed in the previous step.

$$\pi_j^{\text{new}} = \frac{Z_j}{R_{size}}, \tag{2.8}$$

$$\mu_j^{\text{new}} = \frac{1}{Z_j} \sum_{n=1}^{R_{size}} r_{nj} x_n, \tag{2.9}$$

$$\Sigma_j^{\text{new}} = \frac{1}{Z_j} \sum_{n=1}^{R_{size}} r_{nj} (x_n - \mu_j^{\text{new}})(x_n - \mu_j^{\text{new}})^T, \tag{2.10}$$

   where $Z_j = \sum_{n=1}^{R_{size}} r_{nj}$, R_{size} is the size of reference dataset, π_j^{new} is the new mixture weight for component j, and $\mu_j^{new}, \Sigma_j^{new}$ are the new parameters of Gaussian component j.

   iii. Compute the **log-likelihood** for convergence check.

$$ll(\theta) = \sum_{n=1}^{R_{size}} \log \sum_{j=1}^{\mathcal{H}} \pi_j f(x_n|\mu_j, \Sigma_j) \tag{2.11}$$

The E-step and log-likelihood computation are the two N-body problems in EM.

2.4.5 Naïve Bayes Classifier

The naïve Bayes classier is from the family of probabilistic classifiers, which use Bayes theorem while applying a strong independence assumption over the features provided. Naïve Bayes has been used in different tasks such as text categorization, automatic medical diagnosis, and systems performance management [66, 67, 68].

Bayesian classifiers assign the most likely class to a given example described by its feature vector. Learning such classifiers can be significantly simplified by assuming that features are independent given class, that is

$$P(\mathbf{X}|A) = \prod_{i=1}^{n} P\left(X_i|A\right) \tag{2.12}$$

where $\mathbf{X} = (X_1, \cdots, X_n)$ is a feature vector, n is the size of feature vector, and A is a class. Despite this unrealistic assumption, the resulting classifier known as naïve Bayes is remarkably successful in practice, often competing with much more sophisticated classifiers [69, 70, 71, 72, 73].

The success of naïve Bayes in the presence of feature dependencies can be explained by observing that optimality in terms of zero-one loss (classification error) is not necessarily related to the quality of the fit to a probability distribution (*i.e.*, the appropriateness of the independence assumption). Rather, an optimal classifier is obtained as long as both the actual and estimated distributions agree on the most-probable class [69]. The naïve Bayes classifier is defined as follows.

$$\forall q, \arg\max_{r} \ \mathcal{N}(x_q|\mu_r, \Sigma_r),$$

where $\{\mu_r, \Sigma_r\}$ are the parameters of Gaussian component r, and for each query point x_q we want to classify it to the class r with the highest probability.

2.5 N-Body Problems in Computational Geometry

Computational geometry includes algorithms that solve problems involving the relationship between geometric objects, such as points, lines, and planes. Among the first canonical problems studied in computational geometry is the proximity problems, which involve the relative positions of points in space. Many of the problems in computational geometry follow the same pattern of general N-body problems. Previously we mentioned examples from this category such as k-nearest neighbors and minimum spanning trees, and here we will explain two more examples: range search and Hausdorff distance.

2.5.1 Range Search

A central problem in computational geometry, range searching arises in many applications such as range query which is a common database operation [74]. Another example happens in image analysis for pattern recognition, where after preprocessing an image, one needs to analyze pattern primitives each given by several properties, and range searching in this data is a standard procedure [75]. In this thesis, for the range search (RS) problem, the kernel function is defined as a *delta* function [76]. In this problem we want to find all the reference points that fall within a range $(h_{\min}, h_{\max})$ of a query point, x_q defined as,

$$\forall q, \quad \bigcup \arg_r I(h_{\min} \leq ||x_q - x_r|| \leq h_{\max}), \tag{2.13}$$

where $I(h_{min} \leq ||x_q - x_r|| \leq h_{max})$ is a delta function and the equation above returns the reference points that satisfy the given functionality.

2.5.2 Hausdorff Distance

Our last example N-body problem is Hausdorff distance (HD) calculation, which has many applications in computer vision such as comparing images or finding a given template in target images [77]. Another application of Hausdorff distance is in object matching, which introduces possible distance measures between two point sets of objects [78]. In the field of computer vision, a typical problem is matching a given image with a model, in order to find all the locations in the image which match that model. This is also similar to the problem of matching protein motifs within protein sequences [79].

In general, the Hausdorff distance is used as a shape comparison metric based on binary images. Unlike most shape comparison methods, it is able to build a point-to-point correspondence between a model and a test image and is calculated without explicit point correspondence. Moreover, Hausdorff distance for binary image matching is more tolerant to perturbations in the locations of points than other correlation techniques as it measures proximity rather than exact positions [80].

Two sets are close in the Hausdorff distance if every point of either set is close to some point of the other set. In other words, it is the largest of all the distances from a point in one set to the closest point in the other set. The Hausdorff distance between two subsets is computed as follows.

$$\max_q, \min_r ||x_q - x_r||,$$

where the kernel function $||x_q - x_r||$ is the Euclidean distance between the query data point x_q and the reference data point x_r, and the set of operators includes `max`, `min`.

Chapter 3

PASCAL: A Parallel Algorithmic SCALable Framework for N-body

3.1 Introduction

This chapter proposes PASCAL, a *parallel unified algorithmic framework* for generalized N-body problems. PASCAL utilizes tree data structures and user-controlled pruning or approximations to reduce the asymptotic runtime complexity from being linear in the number of data points to be logarithmic. In PASCAL, the domain scientists express their N-body problem in terms of application-specific operations, and PASCAL generates the pruning and approximation conditions automatically. In order to evaluate PASCAL, we generate solutions for six problems detailed in Chapter 2: k-nearest neighbors, range search, minimum spanning tree, kernel density estimation, expectation maximization, and Hausdorff distance, chosen from various domains.

We show that applying domain-specific optimizations and parallelization to the algorithms generated by PASCAL achieves 10× to 230× speedup compared to state-of-the-art libraries on a dual-socket Intel Xeon processor with 16 cores on real-world datasets. We also obtain a novel out-of-the-box asymptotically optimal algorithm for Hausdorff distance calculation.

PASCAL introduces the concept of *visit order* while traversing the multi-tree. The key idea is to prioritize visiting branches of the tree that have a higher likelihood of resulting in the right answer. This could result in a lower runtime due to the increased pruning of sub-trees depending on the N-body problem and the distribution of the dataset. Analysis of the impact of visit order for the six algorithms and five datasets considered in this chapter is presented in Section 3.9.2.

3.2 Related Work

One of the most popular and widely used fast algorithms for classical N-body problems is Barnes-Hut [34] developed in 1986. It uses trees to approximate distance computations and achieve sub $\mathcal{O}(N^2)$ asymptotic time. There has been a lot of effort in parallelizing tree codes particularly for quad- and octrees [81, 82, 83, 84, 85, 86, 87, 88, 89].

PASCAL differs from the previous work in many ways. First, PASCAL supports algorithms and operations beyond what is usually considered in classical physics, such as machine learning and data mining. This makes PASCAL more general. Second, we consider high-dimensional trees (*e.g.*, kd-trees [35]), which are required to handle high-dimensional datasets. Third, our approach is more portable and easily extensible compared to previous approaches which focus on optimizing a specific algorithm for a specific architecture.

While parallel N-body algorithms in physics have received significant attention, the same is not true in machine learning. There are several freely accessible machine learning libraries; however, they lack one or both of the two following features: (a) efficient optimal algorithms and (b) parallelism and scalability on modern machines. For instance, MLPACK [27], which is a state-of-the-art C++ machine learning library offers a limited set of fast algorithms but is not parallel or distributed. Other popular libraries emphasize ease of use but scale poorly,

such as Weka toolkit [90] and SHOGUN toolbox [91]. Some others implement fast algorithms but in languages such as Python giving poor performance, such as Scitkit-learn [26] and mlpy [92].

The few instances of projects that implement a faster algorithm [93, 94, 95] are also limited to a small subset of N-body problems, and most do not employ parallelism or domain-specific optimizations. PASCAL is developed considering a theory on *generalized N-body* algorithms [1, 54, 96] which are similar in spirit to long-studied physics algorithms. This theory is a stepping stone to our work, but it is limited in two ways: (a) the pruning and approximation conditions are designed manually for each and every problem considered, and (b) the theory is limited to problems with only two operators. Although this is a useful first step, this approach is not scalable.

In PASCAL, we address the above limitations by proposing an algorithm to automate the design of pruning and approximation conditions for two or more operators. To the best of our knowledge, PASCAL is the first truly generalized N-body framework.

3.3 PASCAL Framework

Leveraging the commonalities between N-body problems gives rise to the PASCAL framework shown in Figure 3.1, which consists of space partitioning trees, a prune/approximate condition generator, and a tree-traversal scheme. We then apply domain specific transformations and parallelize the algorithms generated by PASCAL to produce an efficient code for comparison against other state-of-the-art libraries and software. The blue shaded boxes represent the contributions which we will discuss in more detail in the rest of this chapter.

The N-body definition is shown as one of the inputs to the PASCAL framework in Figure 3.1 represents a C++ implementation of the operators and kernel function for each N-body

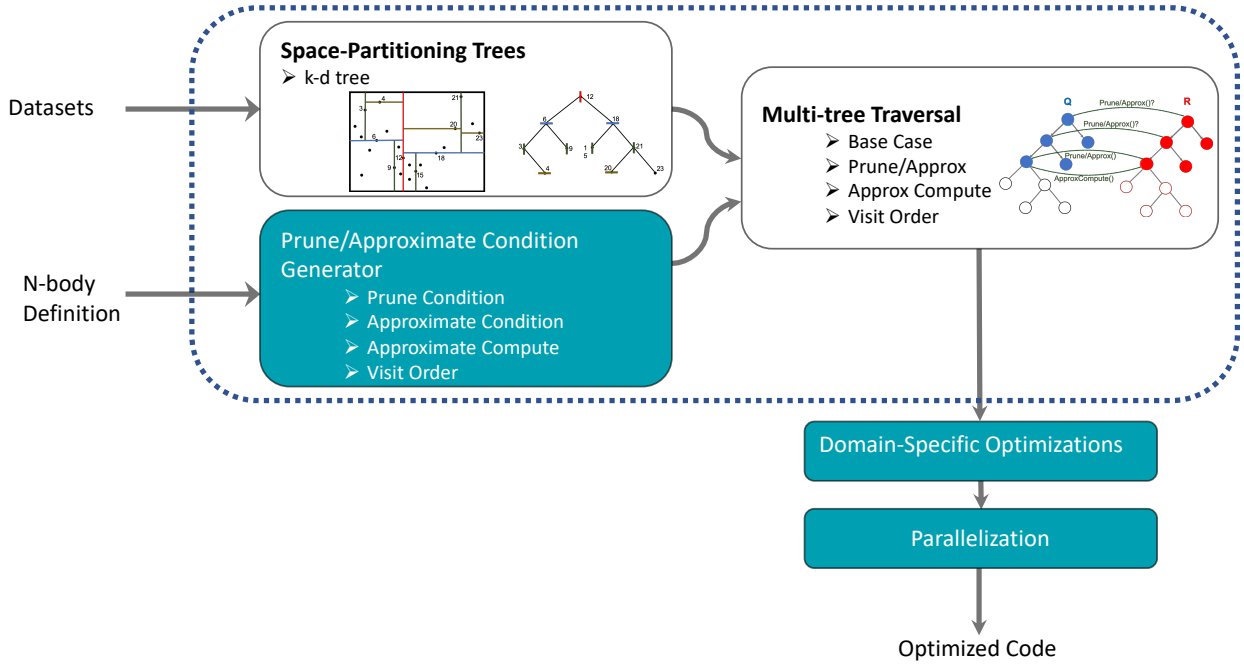


Figure 3.1: Block diagram outlining the overall approach. The dotted box represents the PASCAL framework and the blue shaded boxes are the contributions. The N-body definition represents the definition of the problem by its operators and kernel function written in C++.

problem. These operators are defined by sepearte classes in C++. The implementation includes classes for a set of operators such as *forall*, *min*, *max*, *mink*, *maxk*, *argmin*, *argmax*, *argmink*, *argmaxk*, *sum*, *prod*, *union*, and *unionarg* which are representative of mathematical operators $\forall$, min, max, $\min^k$, $\max^k$, $\arg\min$, $\arg\max$, $\arg\min^k$, $\arg\max^k$, Σ, Π, $\cup$, and $\cup\arg$ respectively. For example, in the case of k-NN, the N-body definition includes the operators $\forall$ and $\arg\min^k$, which are represented using the classes *forall* and *argmink*. The kernel function is passed to PASCAL as a pointer to the C++ function definition of Euclidean distance (kernel function of k-NN as shown in Equation (2.4)) which is shown in Code 3.1.

3.3.1 Classification of N-body Problems

We classify N-body problems into two main categories based on their model of computation and properties: (a) pruning, and (b) approximation problems.

```
float EuclideanDistance(float* s1 , float* s2, int dim)
{
    float sum = 0;
    for (size_t i = 1; i < dim; i++) {
            sum += (s1[i] - s2[i]) * (s1[i] - s2[i])
    }
    return sqrt(sum);
}
```

Code 3.1: Euclidean distance as the kernel function for k-NN as part of N-body definition, inputed to the PASCAL framework

Pruning Problems

Pruning problems are those in which a part of the data and its associated computation can be discarded. One example is nearest neighbors, whose inner operator is `min` and which only returns the minimum value. This provides opportunities to prune parts of the datasets where the minimum value can not be located, without any loss of accuracy. Such pruning opportunities are deduced based on the set of operators and kernel function. Comparative operators such as `min` or `max` result in a pruning problem, as do comparative kernel functions such as $(x_r - x_q < h)$.

Approximation Problems

Approximation problems are those in which the contribution by a subset of the data to the solution can be approximated by a smaller subset. This group of problems only encompass arithmetic operators and non-comparative kernels, such as kernel density estimation. For example, operators such as Σ or Π require the contribution of all the points. In such *approximation* problems, as the name suggests, there is a trade-off between performance and desired accuracy. PASCAL exposes this trade-off to the user as a tuning knob which can be customized based on the target application.

3.3.2 Tree Construction

The space partitioning tree we consider in PASCAL framework is k-d tree, described in Section 2.2.1, with the median point as the splitting value and widest dimension as splitting dimension. As explained in Section 2.2.1, we preserve information for each node such as the high and low boundaries in each dimension as we split the nodes, resulting in fast computation of minimum and maximum distances. In median splitting, the number of data points in each node is easily computed, since each child node contains half of its parent's data points. The center value is calculated by post-order traversal of the tree. This is based on the idea of cached statistics in tree which are efficient in practice [97, 98].

3.3.3 Multi-tree Traversal

After building the space-partitioning trees, we use a multi-tree traversal in order to compute the N-body problem. Algorithm 3.1 describes the multi-tree traversal given two inputs: a set of nodes, $\mathcal{N}^{all}$ and a rule set, $\mathcal{R}$. The rule set provides three main functionalities described in the sequel, and the corresponding functions are highlighted in blue in algorithm 3.1. Note that the traversal is called recursively on the children at each level of the tree as seen by the green function call in line 11. The rule set consists of the following three functions.

Base Case

This function implements the direct point-to-point computation on leaf nodes of the tree, and is called when there is no opportunity to discard the computation on the leaf nodes. The computational complexity of this function for two trees is $\mathcal{O}(l^2)$ which l is the leaf size, and for multiple trees, it is $\mathcal{O}(l^m)$ which m is the number of trees. For instance, for the nearest neighbors problem, this would be equivalent to finding the corresponding point in the reference node which has the minimum distance for every point in the query node.

Algorithm 3.1 MultiTreeTraversal

Input: Nodes set $\{\mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_m\} \approx \mathcal{N}^{all}$, rule set $\mathcal{R}$.

1: **if** $\mathcal{R}$.Prune/Approximate($\mathcal{N}_1$, $\mathcal{N}_2$, ..., $\mathcal{N}_m$) **then**
2: **return** $\mathcal{R}$.ComputeApprox($\mathcal{N}_1$, $\mathcal{N}_2$,..., $\mathcal{N}_m$)
3: **if** ($\forall \mathcal{N}_i \in \mathcal{N}^{all}$ is leaf) **then**
4: $\mathcal{R}$.BaseCase($\mathcal{N}_1$, $\mathcal{N}_2$,..., $\mathcal{N}_m$)
5: **else**
6: **for all** $\mathcal{N}_i \in \mathcal{N}^{all}$ **do**
7: **if** $\mathcal{N}_i$ is leaf **then** $\mathcal{N}_i^{split} = \mathcal{N}_i$
8: **else** $\mathcal{N}_i^{split} = \{\mathcal{N}_i.right, \mathcal{N}_i.left\}$
9: PowerSet-Tuples = $\mathcal{R}$.VisitOrder($\{(\mathcal{N}'_1, ..., \mathcal{N}'_m) | \mathcal{N}'_i \in \mathcal{N}_i^{split}\}$)
10: **for all** $(\mathcal{N}'_1, ..., \mathcal{N}'_m) \in$ PowerSet-Tuples **do**
11: MultiTreeTraversal($(\mathcal{N}'_1, ..., \mathcal{N}'_m), \mathcal{R}$)

Prune/Approximate

This function checks if the computation for a set of nodes can be *approximated* or *pruned* based on the N-body problem definition, using a prune/approximate generator. If so, the node and its descendants will not be visited. For approximating the contribution of a node, we check if the minimum and maximum contribution of that node are very close (*i.e.*, less than a user-defined threshold). If so, we know that all the data points in that node have a similar contribution. In the case of pruning problems, PASCAL first builds a pipeline of pruning opportunities (deduced from comparative operators and kernel) which are then evaluated on the boundary points of each hyper-rectangle node to decide whether to prune that node or not. In some cases, the algorithm prunes entire sub-trees, so the nodes and their descendants will not be visited.

Visit Order

This function returns the order in which the children of a node should be visited while traversing the tree. For instance, *Visit Order* will return whether the left child or the right child should be visited first during k-NN search on k-d trees. To do so, *Visit Order* checks which reference subtree is closest to the query subtree. Visiting the reference subtree with the minimum distance to the query subtree results in a higher probability of finding the nearest neighbors. Additionally, this results in visiting fewer nodes of the tree (additional pruning) which in turn results in lower runtime. n this thesis, the visiting order is only used for pruning problems. For approximation problems, the visiting order function, *Visit Order*, returns an arbitrary default order.

Compute Approximate

This function replaces the computation with the center contribution of each node multiplied by the density of that node which is equivalent to the number of data points in that node. This is only for approximation problems.

While the operations above are not completely orthogonal, they are convenient and powerful to express the range of N-body algorithms. Not only does this representation abstract the actual computation from the traversal, it also abstracts the tree type, which gives us the freedom to plug and play with different trees. Moreover, we are able to express both *pruning* and *approximation* algorithms in the same framework, which enables us to translate our optimizations and parallelization to a much larger class of algorithms.

Note that the N-body problems must satisfy two properties to enable one to choose the tree-based optimal algorithm: (1) operators must satisfy the decomposability property over datasets. Decomposability applies to an operator if the dataset is decomposable across its subsets. For instance, the Σ operator adheres to the decomposability property because the

sum of all interactions over the dataset can be decomposed into the sum of smaller subsets of that data; (2) the kernel function should decrease monotonically with distance. For example, the Gaussian kernel decreases monotonically with distance, while the Cosine kernel doesn't decrease monotonically with the distance.

3.3.4 Prune/Approximate Condition Generator

In order to generate a prune or approximate condition, we extend the classification of N-body problems into 3 categories, namely: (a) approximation, (b) single pruning, and (c) nested pruning. Approximation problems are those in which the contribution by a subset of the data to the solution can be approximated by a smaller subset. Two examples are KDE and EM. Pruning problems are those in which a part of the data and associated computation can be discarded. The main distinction between single and nested pruning is that the former has only one pruning opportunity (*e.g.*, k-NN) while the latter has more than one opportunity for pruning (*e.g.*, Hausdorff distance).

Algorithm 3.2 generates one of three conditions and distinguishes the category of problems by maintaining a queue of possible prune opportunities called `PrunePipeline` (Line 1). We iterate through the operators' set, OP and kernel function, $\mathcal{K}$ and check if there is any pruning opportunity. If so, we push the `reverse` of OP and/or $\mathcal{K}$ into the `PrunePipeline` (Lines 2 - 6). The `reverse` function is defined for operators and kernel function, and defines the reverse of their functionality. For example, the `reverse` of $||x_q - x_r|| < h$ is $||x_q - x_r|| > h$, and the `reverse` of `min` operator is the relational operator *greater than* ($>$).

Algorithm 3.2 Prune/Approximate Condition Generator

Input: Node set $\{\mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_m\} \approx \mathcal{N}^{all}$, kernel function $\mathcal{K}$, operators set OP, threshold σ, tree type ψ.

Output: The prune/approximate condition

queue<Function> PrunePipeline
for all ($op_i \in OP$) **do**
  if (op_i.isComparative()) **then**
    PrunePipeline.push(reverse(op_i))
if $\mathcal{K}$.isComparative() **then**
  PrunePipeline.push(reverse($\mathcal{K}$))
// Approximation category
if (PrunePipline.size == 0) **then**
  $\mathcal{K}_{\min} \leftarrow \min\{\mathcal{K}(\mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_m)\}$
  $\mathcal{K}_{\max} \leftarrow \max\{\mathcal{K}(\mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_m)\}$
  $(\mathcal{N}_1^c, \mathcal{N}_2^c, ..., \mathcal{N}_m^c) \leftarrow$ tuple of node centers
  $\mathcal{K}_{\text{center}} \leftarrow \mathcal{K}(\mathcal{N}_1^c, \mathcal{N}_2^c, ..., \mathcal{N}_m^c)$
  return $\mathcal{K}_{\max} - \mathcal{K}_{\min} < \sigma \times \mathcal{K}_{\text{center}}$
// Single prune category
if (PrunePipline.size == 1) **then**
  $\tau \leftarrow$ threshold based on the operator or kernel
  $\mathcal{N}_m^{\text{border}} \leftarrow$ set of border points based on the tree type ψ
  $op_{\oplus} \leftarrow$ PrunePipeline.pop()
  return $op_{\oplus}(\tau, \mathcal{K}(x_1, ..., x_m))$
  $\{\forall x_i \in \mathcal{N}_i (i = 1, ..., m-1), \forall x_m \in \mathcal{N}_m^{\text{border}}\}$
// Nested Prune category
if (PrunePipline.size > 1) **then**
  return NestedPrune(PrunePipeline)

The problem falls under *approximation* if the size of `PrunePipeline` is zero (Lines 8-13). For approximating the contribution of a node, we check if the minimum and maximum contribution of that node are very close (*i.e.*, less than a threshold). If so, we know that all the data points in that node have a similar contribution and therefore, PASCAL uses the center to approximate the computation of that node. Note that $(\mathcal{N}_1^c, \mathcal{N}_2^c, ..., \mathcal{N}_m^c)$ defined in line 11, represents the centers of nodes $\mathcal{N}_i \in (\mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_m)$ and this is pre-computed as metadata information during tree construction.

The problem falls under the *single prune* category if the size of `PrunePipeline` is one (Lines 15-19). First, we define a threshold for pruning. To do so, we randomly choose points in each set and compute a temporary value using the kernel function. We define $\mathcal{N}_r^{border}$ as the set of border data points which result in the minimum and maximum computations for each node and is defined based on the tree type ψ. Line 18 pops the prune operator and Line 19 generates the prune condition by applying the operator on the tuple of points from the nodes in $\mathcal{N}_1, ..., \mathcal{N}_{m_1}$, and border points of $\mathcal{N}_m$.

Note that in algorithm 3.2, the notation $op_{\oplus}$ is similar to non-member function operators in C++ language. For instance, $op_{\leq}(x_r, x_q)$ is equal to $x_r \leq x_q$. When the size of the `PrunePipeline` is greater than one, the problem belongs to the *nested prune* category and the nested prune algorithm 3.3 is called in line 22. In Line 2 of algorithm 3.3, for each node, we calculate the border data points using the information of templated tree type ψ. Line 3 pops the prune operator from the `PrunePipeline`. Initially, a temporary threshold τ is defined for each prune operator based on its definition. Subsequently, τ is refined as the computation progresses. For example, for the *min* operator, the temporary threshold is set to the highest value for that specific numeric type (*e.g.*, DBL_MAX for double precision data), and each time we find a smaller minimum during computation we refined it with the new minimum. The refining continues till the end of the computation.

Line 5 returns the nested prune condition that we generate. For generating this condition, first, we apply the innermost operator to the border points in the innermost dataset. The result of this is used to call the next innermost operator, and so on. We will continue this process from the innermost operator to the outermost operator in the `PrunePipeline` and apply each operator on the corresponding node borders with the computed thresholds. Each prune operator corresponds to one level in the multi-tree. Note that single prune can be considered as a special case of nested prune with a nesting level of one.

Algorithm 3.3 NestedPrune(PrunePipeline)

Input: Node set $\mathcal{N}_1 ... \mathcal{N}_m$, kernel function $\mathcal{K}$, tree type ψ.

Output: The nested pruning condition

1: **for all** $\mathcal{N}_j \in \mathcal{N}_1 ... \mathcal{N}_m$ **do**

2: $\quad \mathcal{N}_j^{\text{border}} \leftarrow$ set of border points based on the tree type ψ

3: $\quad op_{\oplus_j} \leftarrow$ PrunePipeline.pop()

4: $\quad \tau_j \leftarrow \mathcal{K}(x'_1, ..., x'_m)$ or defined by $\mathcal{K}$

5: **return** $op_{\oplus_1}(\tau_1, \mathcal{K}(x_1, ..., x_m) | op_{\oplus_2}(\tau_2, ...$

$\qquad\qquad | op_{\oplus_m}(\tau_m, \mathcal{K}(x_1, ..., x_m)...)$

s.t. $\{\forall x_1 \in \mathcal{N}_1^{\text{border}}, ..., \forall x_m \in \mathcal{N}_m^{\text{border}}\}$

3.4 Case Studies

In this section, we show how N-body algorithms are generated using PASCAL. Specifically, we consider the six N-body problems discussed with details in Chapter 2 as case studies. The choice of these six problems is because they cover

- Approximation, single pruning, and nested pruning problems

- Both direct and iterative algorithms
- Problems from multiple domains

In all the problems, the `BaseCase` is the direct point-to-point kernel computation at the leaf nodes and `ComputeAprox` is a kernel computation on the center of a subset (for approximate problems). Bellow, we demonstrate how the prune/approximate condition is generated by PASCAL for each of the six N-body problems.

k-Nearest Neighbors Search

Recall that the problem is to find $\forall q,\ \arg\min_r^k ||x_q - x_r||$, meaning for every query point x_q we want to return its k closest neighbors. k-nearest neighbors (k-NN) has only one pruning opportunity, `arg min`, so it is classified as a single prune problem by PASCAL. PASCAL generates the prune condition using a algorithm 3.2. The prune operator that is pushed into `PrunePipeline` is `arg min` and the `reverse` is $\geq$. This `reverse` helps the prune generator in discarding data points that do not contribute in the final answer. The `reverse` of `arg min` is similar to the `reverse` of `min` since they both compute the minimum. The difference is the return value: the latter returns the value of the minimum while the former returns the argument of it. The threshold τ is initialized at the beginning with a temporary computation of the kernel (or a default value such as the maximum value of double precision) and is updated through the algorithm.

PASCAL evaluates $\mathcal{K}$, the kernel function, for each reference point in $\mathcal{N}_r^{\text{border}}$ with respect to the query point x_q. Then, it checks to see if it is greater than or equal to τ. So, the prune condition is

$$op_{\oplus}(\tau, \mathcal{K}(x_q, x_r)) \implies \mathcal{K}(x_q, x_r) \geq \tau, \quad \forall x_r \in \mathcal{N}_r^{\text{border}}, \tag{3.1}$$

where the kernel function $\mathcal{K}$ is Euclidean distance in the case of k-NN and $op_{\oplus}$ is $\geq$ operator.

Range Search

The goal of range search problem is to find all the reference points x_r that fall under a range (h_{min}, h_{max}) of a query point x_q. Range search has only one pruning opportunity via its kernel function, $I(h_{\min} \leq ||x_q - x_r|| \leq h_{\max})$. The `reverse` of this kernel function that is saved in `PrunePipeline` is $h_{\min} > ||x_q - x_r||$ or $||x_q - x_r|| > h_{\max}$ which is used as $op_{\oplus}$ to generate the prune condition ($op_{\oplus_1}$ is $>$, $op_{\oplus_2}$ is $<$). The two thresholds, τ_1 and τ_2 are defined by the kernel function as $h_{\max}$ and $h_{\min}$. We evaluate the kernel function on the points in $\mathcal{N}_r^{\text{border}}$ for each x_q and name it as δ, $\mathcal{K}(x_q, x_r) = \delta$. Then, the prune condition is defined as follows.

$$op_{\oplus_1}(\tau_1, \mathcal{K}(x_q, x_r)) \quad \text{or} \quad op_{\oplus_2}(\tau_2, \mathcal{K}(x_q, x_r)) \implies \delta > \tau_1 | \delta < \tau_2, \forall x_r \in \mathcal{N}_r^{\text{border}} \tag{3.2}$$

Similar to k-NN, the kernel function $\mathcal{K}$ is the Euclidean distance for range search.

Kernel Density Estimation

The goal of kernel density estimation is to estimate the probability density function at each query point x_q using a kernel function K_σ and is defined as $\forall q, \ \frac{1}{|R_{size}|} \sum_r K_\sigma \left(\frac{||x_q - x_r||}{\sigma} \right)$. Here we consider the kernel to be a Gaussian normal distribution. Thus, KDE is an approximation problem since there is no pruning opportunity by considering Gaussian normal distribution as the kernel function, and the `PrunePipeline` queue is empty. `ComputeApprox` will return the probability density at the center of the node, K_{center}, multiplied by the number of data points in that node. We use K_{center} to represent the result of evaluating kernel function K_σ on the `center` of a node. In this problem, τ is a default constant that can be overridden by the user to adjust the overall accuracy. PASCAL uses algorithm 3.2 to generate the approximation condition,

$$(K_{\max} - K_{\min}) < \tau \times K_{\text{center}} \tag{3.3}$$

Minimum Spanning Tree

MST is an iterative algorithm and in each iteration, it uses the same operations as k-NN search. So PASCAL generates exactly the same prune condition and rule set as k-NN. Since iterative algorithms appear in many fields, we consider the minimum spanning tree example as one of our iterative algorithms in the evaluation of PASCAL.

Expectation Maximization

EM is an approximation problem. EM has three steps, the E-step, M-step, and Log-likelihood, where 99% of the time is spent in E-step and Log-likelihood. Moore [99] proposed a powerful space-partitioning tree-based algorithm to reduce the complexity of E-step from $\mathcal{O}(\mathcal{H}N)$ to $\mathcal{O}(\mathcal{H}\log N)$, where N is the number of data points and $\mathcal{H}$ is the number of clusters. We extend Moore's idea and propose a fast algorithm for estimating both the E-step and log-likelihood computation in $\mathcal{O}(\mathcal{H}\log N)$.

The *first* N-body computation in EM is the E-step. In the E-step, if the difference between the maximum and the minimum responsibility of the points i from cluster j, r_{ij} detailed in Equation (2.7), is less than a threshold, we can approximate the influence of these data points. This is because all the data points in that node will approximately have a similar responsibility to the cluster. PASCAL generates the approximation condition

$$(r_j^{\max} - r_j^{\min}) < \tau \times \ r_j^{\text{center}},\ j = 1, .., \mathcal{H} \tag{3.4}$$

where τ is the threshold parameter, r_j^{center} is the responsibility of the center data points

in the node from cluster j, $r_j^{\min}$ and $r_j^{\max}$ are the minimum and maximum responsibilities between all the data point from cluster j.

`ComputeApprox` will return the value of responsibility at the center of the node multiplied by the number of data points in that node. Note that in this algorithm, the distance we compute is the Mahalanobis distance which is defined as $(x-\mu)^T\Sigma^{-1}(x-\mu)$ for a Gaussian with $\theta = (\mu, \Sigma)$.

The *second* N-body computation in EM is the log-likelihood computation. In order to calculate the log-likelihood, we traverse the same tree as in the E-step. The computation pattern is similar in style to E-step albeit with a different approximation condition generated by PASCAL presented below.

$$\log \sum_{i=1}^{\mathcal{H}} \pi_i f(x_{\max}|\theta_i) - \log \sum_{i=1}^{\mathcal{H}} \pi_i f(x_{\min}|\theta_i) < \sigma |\log(\sum_{i=1}^{\mathcal{H}} \pi_i f(x_{\text{center}}|\theta_i))|$$

Hausdorff Distance

One of the N-body problems with more than one pruning opportunity is Hausdorff distance. In this problem, the *kernel function* is the Euclidean distance with the operators set {`max`, `min`} both of which provide pruning opportunities. PASCAL generates the prune condition using the nested prune algorithm, algorithm 3.3.

First, PASCAL constructs the tree data structure and applies each of its operators on one of the levels of the tree. The `PrunePipeline` queue consists of the `reverse` of `max` and `min` which are $\leq$ and $\geq$. Therefore, $op_{\oplus_1}$is $\leq$ and $op_{\oplus_2}$is $\geq$. To form the prune condition, PASCAL creates two nested loops. The inner loop runs over the borders of the inner tree (for example, reference dataset) applying the inner operator which is $\geq$. The outer loop covers the borders of the second tree (for example, the query dataset), applying the $\leq$ operator.

Considering the general definition of the N-body problem in Equation (2.2), each operator that is applied to a dataset is regarded as the operator that is applied to the tree built for that dataset. We define two thresholds, τ_1 and τ_2 and the nested prune condition generated is shown below.

$$\begin{aligned} &op_{\oplus_1}(\tau_1, \mathcal{K}(x_q, x_r) | op_{\oplus_2}(\tau_2, \mathcal{K}(x_q, x_r))) \implies \tau_1 \geq (\mathcal{K}(x_q, x_r) | \tau_2 \leq \mathcal{K}(x_q, x_r)), \\ &s.t. \quad \forall x_q \in \mathcal{N}_q^{\text{border}}, \forall x_r \in \mathcal{N}_r^{\text{border}} \end{aligned} \tag{3.5}$$

This condition is generated by nested prune algorithm 3.3 detailed in Section 3.3.4.

3.5 Domain-Specific Optimizations

In order to achieve an optimized code, we first apply numerous optimizations to both the tree construction and the computational core of the evaluation. Then, we parallelize the tree traversal defined by algorithm 3.1, and finally tune empirically for the associated tuning parameters (*e.g.*, leaf size).

3.5.1 Incremental Bounding Box Calculation

During the tree construction described in Section 2.2, we associate each node with its bounding box data. This is critical for efficient evaluation during traversal. For instance, during range search, we check if the reference node is within a specified range of the query node and if not, the entire node is pruned. This check requires computing the minimum and maximum node-to-point and node-to-node distances. Pre-computing the bounding box information significantly reduces the time to compute these distances since we do not have to access the actual data points each time.

For k-d trees, this is essentially computing the hyper-rectangle boundary information in each

dimension. At the start of the computation, the root bounding box is computed from all the N points. During partitioning, we only incrementally update the bounding box of the dimension that is being split at each node based on the splitting value, resulting in a total complexity of $\mathcal{O}(Nd)$. d is the dimension of data points in vector space.

3.5.2 Optimal Metric Calculation

The evaluation can be performed using a variety of distance metrics. We consider the following metrics intended for real-valued vector spaces:

- Euclidean: $\sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$
- Manhattan: $\sum_{i=1}^{d} |x_i - y_i|$
- Chebyshev: $\max_{i=1}^{d} |x_i - y_i|$
- Mahalanobis: $(\overrightarrow{x} - \overrightarrow{\mu})^T \Sigma^{-1} (\overrightarrow{x} - \overrightarrow{\mu})$, where μ and Σ are distribution's parameters.

Outlined below are two techniques to efficiently compute these metrics which are repeatedly used in all phases of the algorithm. Additionally, we ensure that the compiler generates vectorized code for the metric calculation.

- Each metric defines both a distance and a **reduced distance**, which is often faster to compute and is used whenever possible. For example, in the case of Euclidean distance, the reduced distance is squared Euclidean distance. All the intermediate distances are computed as squared distances, and this eliminates the expensive `sqrt` instruction which has long latencies and not pipelined.
- **Partial** distance between two d-dimensional points x and y is defined as the distance computed on a subset of the d dimensions. For example, when searching for k-nearest

neighbors, we compute the distance between two points and insert the reference point into our neighbor list only if the computed distance is smaller than the k^{th} largest distance in our sorted list. When d is large as in the case for some of our datasets, this optimization offers additional savings in processing time where we can terminate the computation earlier if the computed partial distance exceeds our threshold.

3.5.3 Incremental Distance Calculation

This idea was introduced by Arya and Mount [100] where the node-to-point distance at each node during single-tree traversal is incrementally computed from the parent's distance in constant time independent of dimension. For datasets with large d, this has the potential for significant savings in computation at the cost of minimal additional storage of distance information.

3.6 Tree Construction

There are numerous studies on parallel tree construction [101, 102, 103]. In this thesis, we focus on accelerating the evaluation phase for two main reasons. First, the tree construction time *initially* constituted a small fraction of the total execution time for most of the datasets. Second, in many real applications, the tree once built for a particular dataset can be reused for multiple different queries. One of the algorithms considered in this thesis - *expectation maximization* - iterates over the same tree until convergence, which amortizes the tree construction time. EM also uses the same trees for both its N-body algorithms (E-step and Log-likelihood computation).

3.7 Parallelization

After applying serial optimizations, we parallelize the multi-tree traversal using Cilk [104]. Since there are dependencies across the recursion, we exploit a combination of data and task parallelism. At first, we spawn Cilk tasks recursively until all the threads are saturated, at which point we switch to data parallelism, as seen in Figure 3.2. Note that different fast scheduling methods and frameworks have been introduced in the literature [105, 106, 107, 108]; however, we use Cilk because of its work-stealing scheduler to provide better load balance in our implementation. Since the tree traversal is abstracted from the actual computation, parallelizing the tree traversal leads to parallel implementations of all six algorithms. Moreover, for any new algorithm expressed in PASCAL, we can obtain parallel multi-tree implementations at no additional cost. This greatly accelerates the ability to scale new problems in rapidly growing domains.

Algorithmically, the tree is parametrized by the maximum number of points per leaf node, l. As l increases, the cost of tree construction decreases at the expense of increased cost in performing the `BaseCase`. On the other hand, small l results in a large number of nodes and an increase in the cost of tree traversal. We exhaustively tune l for all implementations.

3.8 Experimental Setup

Here we provide details on the setup we used for our experiments in this chapter, including the libraries, architecture and compilers, and the benchmarks.

3.8.1 Libraries

We compare PASCAL's performance against state-of-the-art software, namely, WEKA [90], Scitkit-learn [26], MLPACK [27], and MATLAB. Table 3.1 summarizes the algorithms sup-

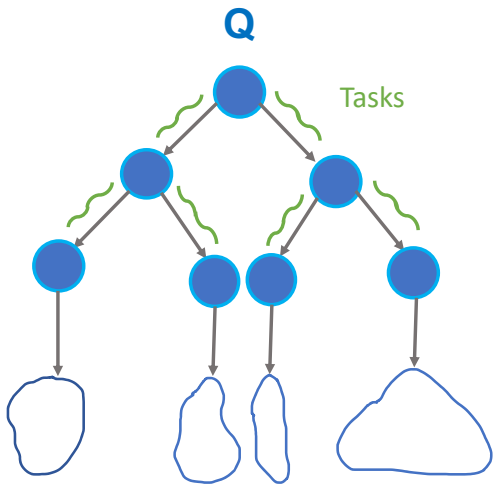


Figure 3.2: The parallelization happening on the query tree. We generate tasks until the number of tasks is equal to the number of the cores on the system.

ported by them.

- **WEKA**: The Waikato Environment for Knowledge Analysis data mining suite is one of the most popular and widely used open-source ML software. It is written in Java and includes tree-based nearest neighbor search.

- **Scikit-learn**: This is a Python module for medium scale supervised and unsupervised algorithms built on top of NumPy and SciPy. It is largely written in Python, with some core algorithms written in Cython for performance. Among its various algorithms are k-d tree and ball-tree based k-nearest neighbors, radius search, and KDE.

- **MLPACK**: This is the state-of-the-art machine learning library with emphasis on scalability, speed, and ease of use. It is written in C++ on top of Armadillo matrix library and makes heavy use of templates for optimization.

- **MATLAB**: This is considered one of the best numerical computing environments. The functions corresponding to the algorithms considered are **knnsearch**, **rangesearch**, **fitdist**, and **graphminspantree** that are part of the Statistics and Bioinformatics toolboxes.

	k-NN	KDE	RS	EMST	EM	HD
Weka	✓	✗	✗	✗	✓	✗
Scikit-learn	✓	✓	✓	✗	✓	✗
MLPACK	✓	✓	✓	✓	✓	✗
MATLAB	✓	✓	✓	✓	✓	✗

Table 3.1: Algorithms considered in our study and their support in other libraries.

3.8.2 Architecture and Compilers

We evaluate our implementations on a dual-socket Intel Xeon E5-2630 v3 processor (Haswell-EP) with a 2.4 GHz clock frequency, 59 GB/s of DRAM bandwidth, and 32/256/20480 KB of L1/L2/L3 cache. Each socket has 8 cores, for a total of 16 cores (32 threads with hyper-threading) and a theoretical double precision peak performance of 614.4 GFlop/s. We use Intel C++ compiler (icpc version 15.0.2) with C++11 feature support. We use Python v2.7.6 for scikit-learn and Java v1.8.0 for Weka.

3.8.3 Benchmarks

We present results on five real-world datasets characterized in Table 3.2. These include *Yahoo! Front Page Module User Click Log Dataset, version 1.0* taken from Yahoo! (Yahoo!), HIGGS bosons signals and background process dataset (HIGGS), Individual Household Electric Power Consumption dataset (IHEPC), US Census data from 1990 (Census), and KDD Cup 1999 dataset (KDD) from UCI machine learning repository [109].

3.9 Results and Discussion

The combined benefits of asymptotically optimal algorithms, optimizations, and parallelization are substantial. In this section, we first compare our performance against state-of-the-

Dataset	Number of data points	d
Yahoo!	41904293	11
IHEPC	2075259	9
HIGGS	11000000	28
Census	2458285	68
KDD	4898431	42

Table 3.2: Description of the datasets. d: dimensionality.

art ML libraries and software. Then, we break down the performance gain step by step and finally, evaluate the scalability of our algorithms.

3.9.1 Performance Summary

Figures 3.3 and 3.4 present the speedup of EM and k-NN on the five datasets characterized in Table 3.2. We compare against Scikit-learn (Sklearn), MLPACK, MATLAB, and Weka. The choice of these two algorithms is because they are the only ones supported by all competing libraries and therefore make good candidates for a comprehensive comparison. Moreover, because the former is a direct pruning algorithm while the latter is an iterative approximation algorithm, together they represent two ends of the spectrum.

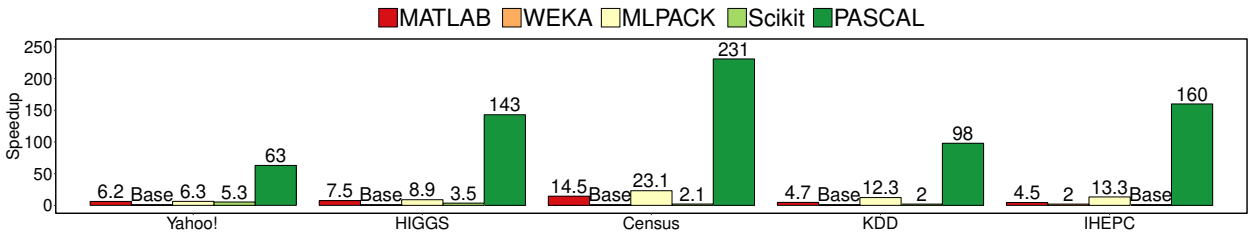


Figure 3.3: Speedup summary of single-tree EM with a threshold value of 0.1. The slowest library is used as the baseline for comparison.

Across the board, our implementation shows significantly better performance compared to Scikit-learn, MLPACK, MATLAB, and Weka. Census is the dataset with the largest di-

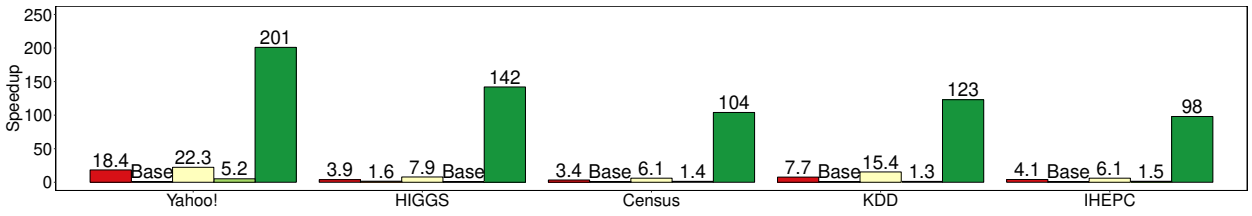


Figure 3.4: Speedup summary of dual-tree k-NN for $k = 3$. The slowest library is used as the baseline for comparison.

mension ($d = 68$) among the ones considered in this chapter. PASCAL shows a staggering 104× speedup against Weka (slowest) and 17.03× speedup compared to MLPACK (fastest) for dual-tree k-nearest neighbors. Note that PASCAL and MLPACK show 104× and 6.1× speedup against Weka respectively, therefore PASCAL shows a 17.03× (104/6.1) speedup against MLPACK. Since the numbers in Figures 3.3 and 3.4 present the speedup numbers against the slowest library, for the rest of this analysis, for calculating the speedup of PASCAL against MLPACK we divide PASCAL's speedup by MLPACK's speedup. Similarly, we observe speedups of 231× and 10× against Weka and MLPACK respectively for single-tree EM. Yahoo! is the dataset with the largest number of data points (41904293) considered in this chapter. For this dataset, we achieve 201× and 9× speedup against Weka (slowest) and MLPACK (fastest) respectively for dual-tree k-nearest neighbors, also 63× and 10× speedup against Weka (slowest) and MLPACK (fastest) respectively for EM. MLPACK is a C++ library that applies many optimizations as well as tree implementation of the N-body problems, resulting in a better speedup compared to other libraries. However, PASCAL is able to provide better speedup compared to MLPACK because it provides parallelization to the tree implementation of the tested N-body problems.

3.9.2 Impact of Visit Order in Tree Traversal

The order of tree traversal or *Visit Order* in the multi-tree traversal algorithm 3.1 helps PASCAL framework make smart decisions while traversing the tree to reduce the total

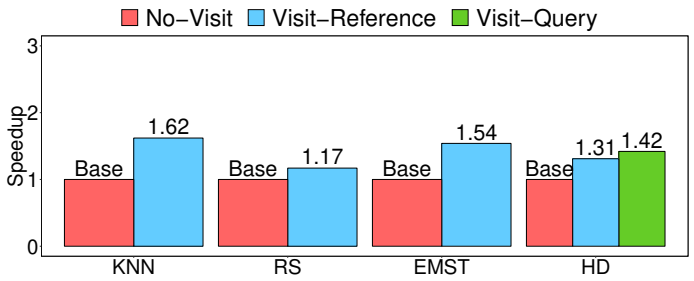
number of nodes visited. For example, in k-NN search, while deciding which child to visit first, *Visit Order* will return the closest child. This ordering in the visit pattern results in finding the nearest neighbors sooner than a random visit pattern, which in turn leads to better performance.

In this section, we analyse the performance of the *Visit Order* function by comparing the results of experiments that do and do not use Visit Order. Each N-body problem, implying the operators' set, defines if the *Visit Order* function can result in better performance. For example, in a problem such as k-NN with operator set $\{\forall, \min\}$, the *VisitOrder* function does not have any impact on the $\forall$ operator, but it will influence the min operator. Therefore, in the k-NN example, the *VisitOrder* function is only beneficial when it is applied to the reference tree (tree with the data for the min operator).

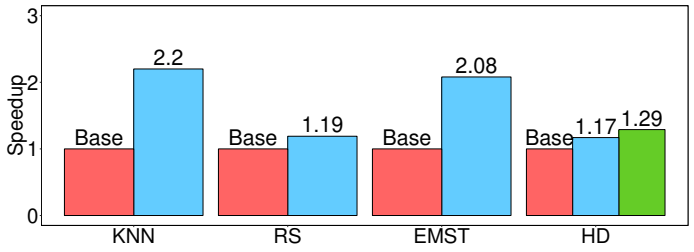
Figure 3.5 shows the additional speedup gain when we apply the *Visit Order* function on different datasets. The blue bar shows the speedup gain for applying the *Visit Order* function on the reference tree. Another example, Hausdorff distance with the operator set $\{\max, \min\}$, can benefit from the *Visit Order* function in both datasets. The green bar in Figure 3.5 shows the additional performance gain by applying *Visit Order* on the query tree as well. Overall, the order of traversal improves the performance of PASCAL. For approximation algorithms such as EM and KDE, the *VisitOrder* function does not have any influence since all nodes of the tree have to be visited, so the order of traversal does not matter.

3.9.3 Performance Breakdown

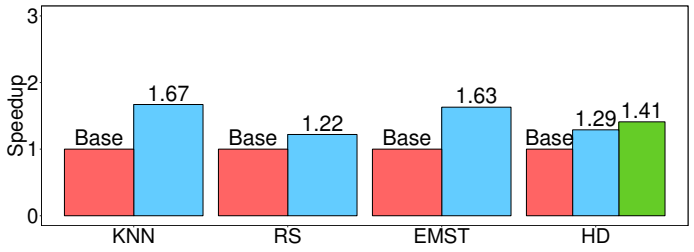
Furthermore, to obtain a better understanding of the factors contributing to the performance improvement, we break down the speedups in Table 3.3 and 3.4. Specifically, this speedup breakdown helps distinguish the improvements that are purely algorithmic (k-d tree algorithm) from improvements via optimization and parallelization. For example, for the Census



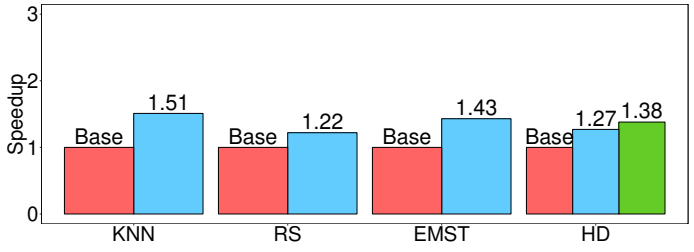
(a) KDD



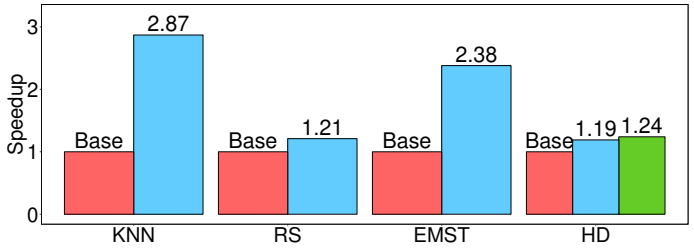
(b) Yahoo!



(c) Census



(d) HIGGS



(e) IHEPC

Figure 3.5: Influence of the *Visit Order* function for the five datasets – KDD, Yahoo!, Census, HIGGS, and IHEPC from top to bottom. The blue bar shows the speedup from applying the *Visit Order* function on the reference tree and the green bar (applicable only to HD) shows the additional performance gain from using *Visit Order* on the query tree as well.

dataset, we observe a 1.4× speedup from an asymptotically faster algorithm, a 6.5× speedup when applying optimizations on top of the tree algorithm, and a combined 90.8× speedup by adding parallelization for k-NN.

Yahoo! is the dataset with the largest number of data points (41904293) considered in this chapter. The performance breakdown for this dataset is 3.1× due to a tree-algorithm, 12.1× due to both optimizations and the asymptotically-optimal tree-algorithm, and 173.1× with adding parallelization for k-NN on top of optimizations and tree-algorithm. Similarly, the breakdown for the EM algorithm is 1.6×, 3.2×, and 53.7× respectively for the same dataset. Note that the speedup numbers presented in Table 3.3 and 3.4 are cumulative, meaning that the numbers for `+Opt` are speedup results for both tree-algorithm and optimization, and `+Par` represents the cumulative speedup of adding parallelization on top. We observe that each dataset benefits differently from algorithmic changes, optimizations, and parallelization based on the number, dimensionality, and distribution of the data points.

	k-NN			EM			KDE		
	Alg	+Opt	+Par	Alg	+Opt	+Par	Alg	+Opt	+Par
Yahoo!	3.1	12.1	173.1	1.6	3.2	53.7	2.1	9.1	92.1
HIGGS	2.1	7.3	108.1	1.5	6.8	117.6	1.7	4.7	50.1
Census	1.4	6.5	90.8	1.3	11.2	190.0	1.4	8.1	75.6
KDD	1.6	6.8	100.7	1.4	4.1	70.9	1.5	3.1	33.5
IHEPC	3.0	4.3	61.5	1.5	7.6	127.6	2.0	5.4	53.6

Table 3.3: Speedup breakdown for k-NN, EM, and KDE algorithms. Alg stands for algorithmic improvement, +Opt refers to optimization on top of Alg, and +Par is parallelization on top of Opt. The baseline for these computations is our implementation of each problem without any tree-algorithm, optimization, and parallelization.

	HD			RS			MST		
	Alg	+Opt	+Par	Alg	+Opt	+Par	Alg	+Opt	+Par
Yahoo!	2.5	11.5	161.1	2.2	9.1	126.8	2.9	11.9	166.7
HIGGS	1.9	6.1	89.6	1.9	6.3	86.5	2.0	6.9	102.8
Census	1.3	10.2	141.8	1.3	10.4	144.9	1.4	10.9	151.6
KDD	1.4	3.8	54.4	1.4	5.1	70.5	1.5	3.8	55.5
IHEPC	2.5	6.8	101.3	2.1	6.3	94.1	2.9	7.1	107.1

Table 3.4: Speedup breakdown for HD, RS, and MST algorithms. Alg stands for algorithmic improvement, +Opt refers to optimization on top of Alg, and +Par is parallelization on top of Opt. The baseline for these computations is our implementation of each problem without any tree-algorithm, optimization, and parallelization.

3.9.4 Scalability

Figure 3.6 shows the scalability of the six algorithms, namely: (1) k-nearest neighbors with $k = 3$, (2) range search with range between 0 and 2, (3) kernel density estimation for Gaussian kernel, K with bandwidth, $\sigma = 0.1$ and relative error tolerance set to 0.1, (4) Expectation Maximization with error tolerance of 0.1, (5) minimum spanning tree, and (6) Hausdorff distance.

We observe that our implementation delivers very good scalability on all the six algorithms. Note that 32 threads is with hyper-threading enabled where we assign two threads per core, in which they share the same hardware. In all cases, hyper-threading further improves the performance resulting in 14×, 16×, 13×, 14×, 10×, and 14× speedup (average 13.5×) for Yahoo! and 15×, 17×, 13×, 15×, 9×, and 15× speedup (average 14×) for IHEPC over the serial optimized code for k-nearest neighbors, expectation maximization, range search, minimum spanning tree, kernel density estimation, and Hausdorff distance respectively.

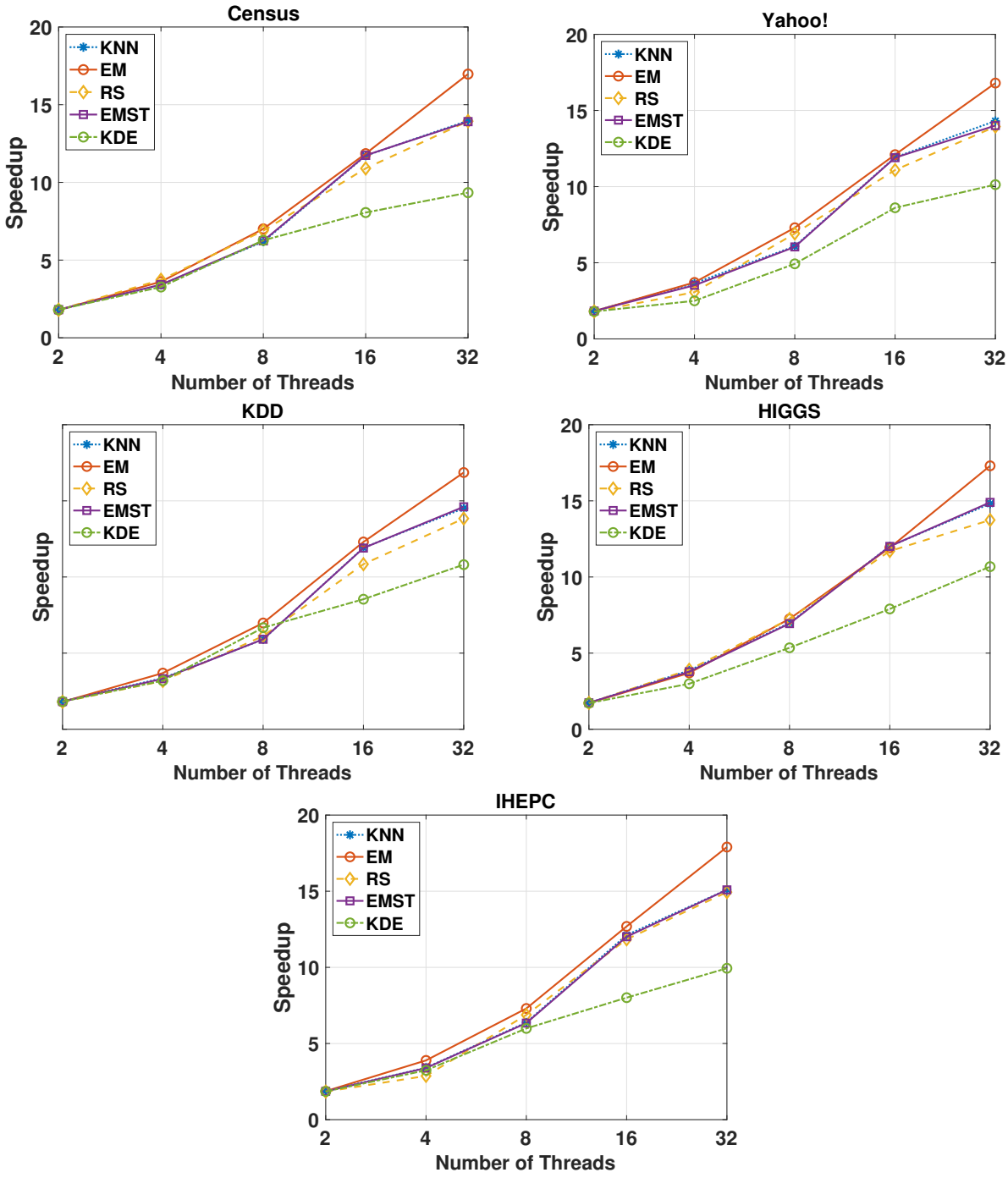


Figure 3.6: Multicore scalability using Cilk for the k-nearest neighbors (k-NN), expectation maximization (EM)range search (RS), Euclidean minimum spanning tree (EMST), and kernel density estimation (KDE) algorithms on a dual-socket Intel Xeon E5-2630 v3 processor for five datasets in Table 3.2.

Scalability Difference in Multi-trees

Tree algorithms used for N-body problems in this thesis are irregular [110]. The dynamic nature of pruning/approximation of sub-trees during tree-traversal makes these problems challenging to parallelize. This load-balancing problem is further exacerbated in dual-tree traversal. As a result, we observe that EM, which uses single-tree traversal with one tree (we use single-tree over a dual-tree traversal for EM because of the small number of clusters), shows better scalability compared to the other five algorithms, which use dual-tree traversals.

We currently defer to Cilk to manage scheduling of tasks using its work-stealing scheduler. In the next chapter, we will explore NUMA-aware parallelization for better load balance, which is critical especially on NUMA machines. In summary, these results show the potential of our approach to achieve orders of magnitude improvement in performance through the use of tree data structures, optimizations, and parallelization.

3.10 Conclusions

In this chapter, we proposed PASCAL, a parallel unified algorithmic framework for N-body problems. PASCAL generates prune and approximation conditions from the N-body definition in C++, which is one the most challenging components in the design of these algorithms. We evaluated PASCAL with six N-body problems from different domains and observe $10 - 230\times$ speedup compared to state-of-the-art libraries/software. The broader impact is to enable scientific discovery not only for N-body problems in scientific computing and machine learning but also to a number of related problems in other unexplored domains that can be expressed in the same style of execution to obtain an out-of-the-box parallel optimized implementation.

Chapter 4

PASCAL-X: Extending PASCAL for Higher Performance

4.1 Introduction

In the previous chapter, we proposed PASCAL as a parallel unified algorithmic framework for generalized N-body problems. In this chapter, we propose PASCAL-X, which is an extension of PASCAL for NUMA architectures. PASCAL-X also introduces additional performance improvement opportunities. Moreover, PASCAL-X provides an analysis of the influences of tuning parameters. We show that by applying NUMA-aware parallelization, and empirically tuning the algorithm parameters, PASCAL-X achieves 10× to 309× speedup compared to state-of-the-art libraries on a dual-socket Intel Xeon processor on various real-world datasets.

Also, we empirically evaluate the impact of algorithmic tuning parameters such as (a) leaf size, which influences the shape and granularity of the nodes in the tree and (b) cut-off level, which influences the number and granularity of tasks created during parallelization. Our results show that tuning these parameters combined could result in up to 4.6× speedup, thus it is essential to tune these parameters in order to achieve high performance.

4.2 NUMA-aware Parallelization

After applying serial optimizations, we parallelize the multi-tree traversal using the OpenMP tasking model. Both the Cilk and OpenMP task model provide similar performance in PASCAL. However, we switch from Cilk used in PASCAL to OpenMP in PASCAL-X, because OpenMP provides opportunities to develop NUMA-aware parallelization for task models [111]. The OpenMP task model has been designed to provide an abstraction for recursive algorithms and irregular problems [111]. Moreover, OpenMP tasking is able to deliver performance comparable to OpenMP work-sharing implementations [112] which makes it a good model for parallelization of multi-tree traversal in PASCAL-X.

Modern SMPs are based on non-uniform memory access. There are two main task creation patterns for NUMA machines [111, 113, 114]: (a) single-producer multiple-executors, and (b) parallel-producer multiple-executors. We employ the parallel-producer multiple-executer pattern on an Intel OpenMP runtime which generates local task queues for each thread.

The producers are the higher levels of recursion in the tree traversal. The number of task producers is equal to the number of NUMA sockets (for instance, on a machine with two NUMA sockets we will have two producers), and each task producer generates local tasks on its own socket which is located on the local queue of each thread. We recursively generate OpenMP tasks until we reach a cut-off threshold in the recursion, at which point we switch to data parallelism. The cut-off level controls the parallel task granularity, which is one of the important tuning parameters.

We also use the first-touch page allocation policy of the operating system to parallelize the data initialization loops. We use the same domain decomposition for both the tree construction and the evaluation. Consequently, each thread initializes the sub-tree that it evaluates on, and the data are initialized on their local DRAM. Because of the irregular nature of

tree traversal, there is load imbalance, which is dynamic and highly input-dependent. In the Intel runtime system, each thread has its local task queue where it enqueues and dequeues tasks. It starts to dequeue from other queues when its queue is empty (commonly known as work-stealing). Threads always steal work from the same socket before stealing from remote sockets, ensuring a NUMA-aware load-balanced parallel implementation.

4.3 Tuning

In this chapter, we consider two tuning parameters – points per leaf node or *leaf size* and parallel task granularity or *cut-off level*. Algorithmically, the tree is parametrized by the maximum number of points per leaf node, l. Similarly, the degree of parallelism and parallel task granularity, which is critical for load balance, is parametrized by the cut-off level. This is the tree level when we switch from task parallelism to data parallelism. We exhaustively tune l and cut-off level for all implementations and datasets.

4.3.1 Impact of Leaf Size and Cut-off Level

In this section, we analyze the impact of two tuning parameters, namely: (a) the leaf size of the space partitioning trees, and (b) the cut-off level to control the number of tasks generated while parallelizing the tree traversal. Both parameters influence performance differently based on the algorithm and the distribution of data points.

To analyse the combined influence of these two tuning parameters, we vary leaf sizes from 32 to 512 in multiples of 2, and cut-off levels from 5 to 13 in increments of 2. Since a k-d tree is a binary tree, each successive level effectively doubles the number of tasks compared to the previous level. We begin our experiments with 5 as the starting cut-off level, which results in 32 tasks (*i.e.*, one per OpenMP thread). Each increment of 2 in cut-off level results in 4× more tasks. Higher cut-off levels result in more fine-grained tasks which help in load

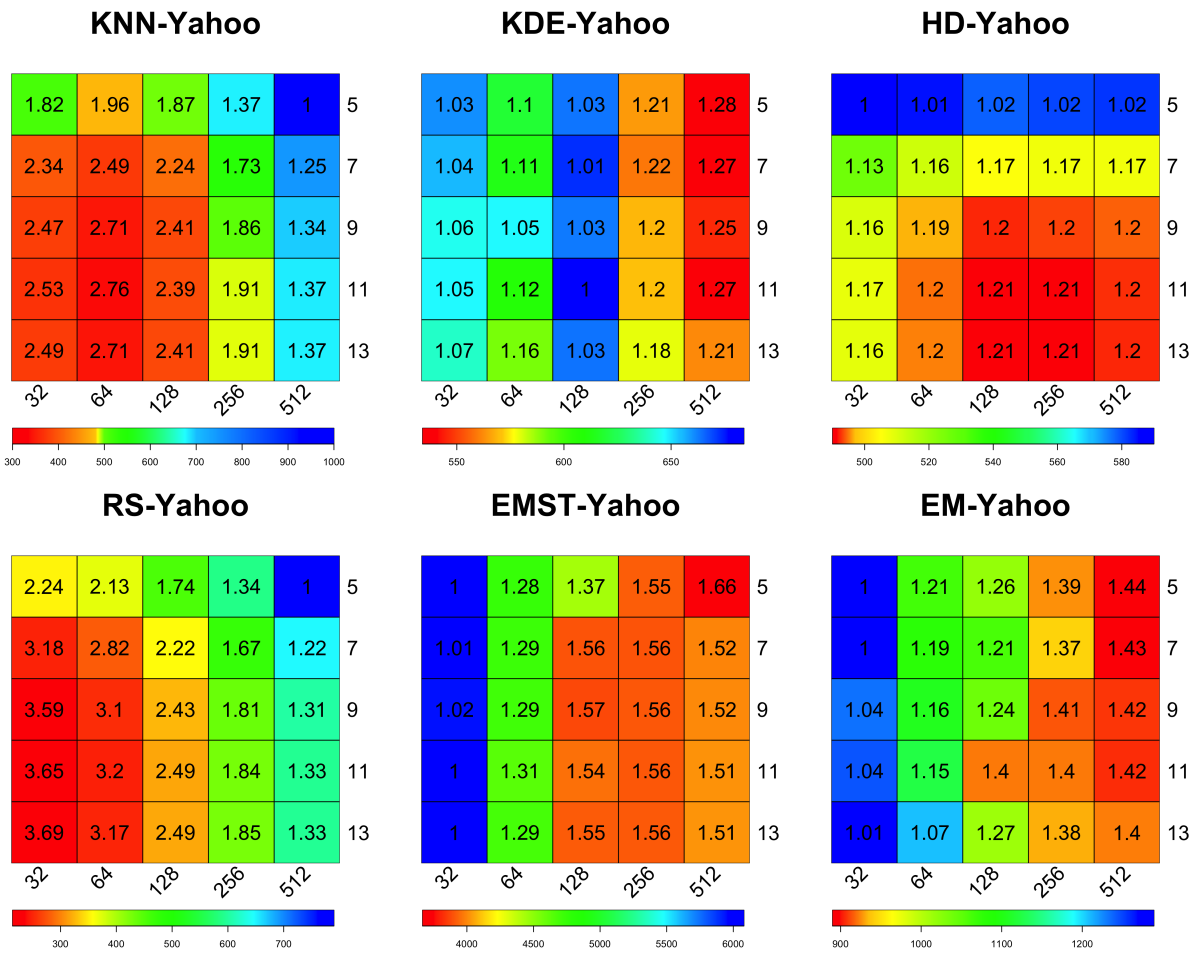


Figure 4.1: Each heatmap presents the influence of leaf size (x-axis) and cut-off level (y-axis) for six algorithms on the Yahoo! dataset on a dual-socket Intel Xeon E5-2630 v3 processor.

balancing highly irregular trees. However, there is a trade-off between this effect and the overhead of creating too many tasks to achieve an optimal load balance.

Figures 4.1 to 4.5 show heat-maps of the six algorithms for five datasets. For example, in the k-NN search on the Yahoo! dataset, we gain 2.7× speedup when changing the leaf size from 512 to 64 (decreasing the size of leaves) and increasing the cut-off level from 5 to 13 (increasing the number of tasks created). The hotter spectrum in the heat-map figures denotes higher performance compared to slower configurations shown by a cooler spectrum.

We observe different configurations of best leaf size and cut-off levels based on the algorithm

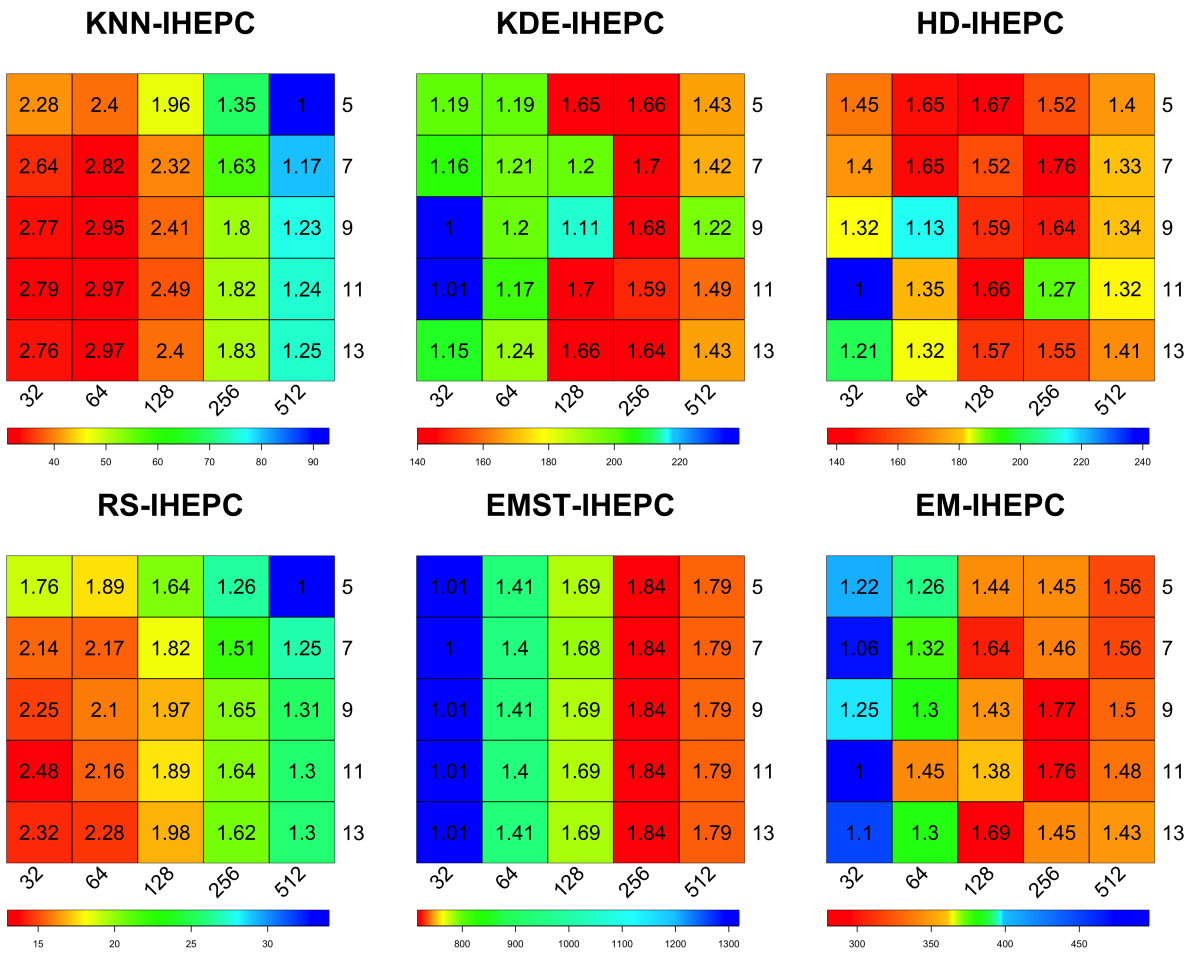


Figure 4.2: Each heatmap presents the influence of leaf size (x-axis) and cut-off level (y-axis) for six algorithms on the IHEPC dataset on a dual-socket Intel Xeon E5-2630 v3 processor.

and dataset, which result in up to 4.6× speedup over a slower configuration. For example for k-NN, a smaller setting of leaf sizes with a large number of tasks results in higher performance across the different datasets, while EM shows better performance for larger leaf sizes. This shows the varying influence of the tuning parameters on the final performance of the algorithm and the potential of using an autotuner to expedite the current tuning process. For these set of experiments, we execute all the combinations, however, ML methods similar to what is used in [115] could be leveraged here to reduce the number of experiments.

In summary, these results show the potential of our approach to achieve improvement in performance by adding NUMA-aware parallelization and tuning. New algorithms can be ex-

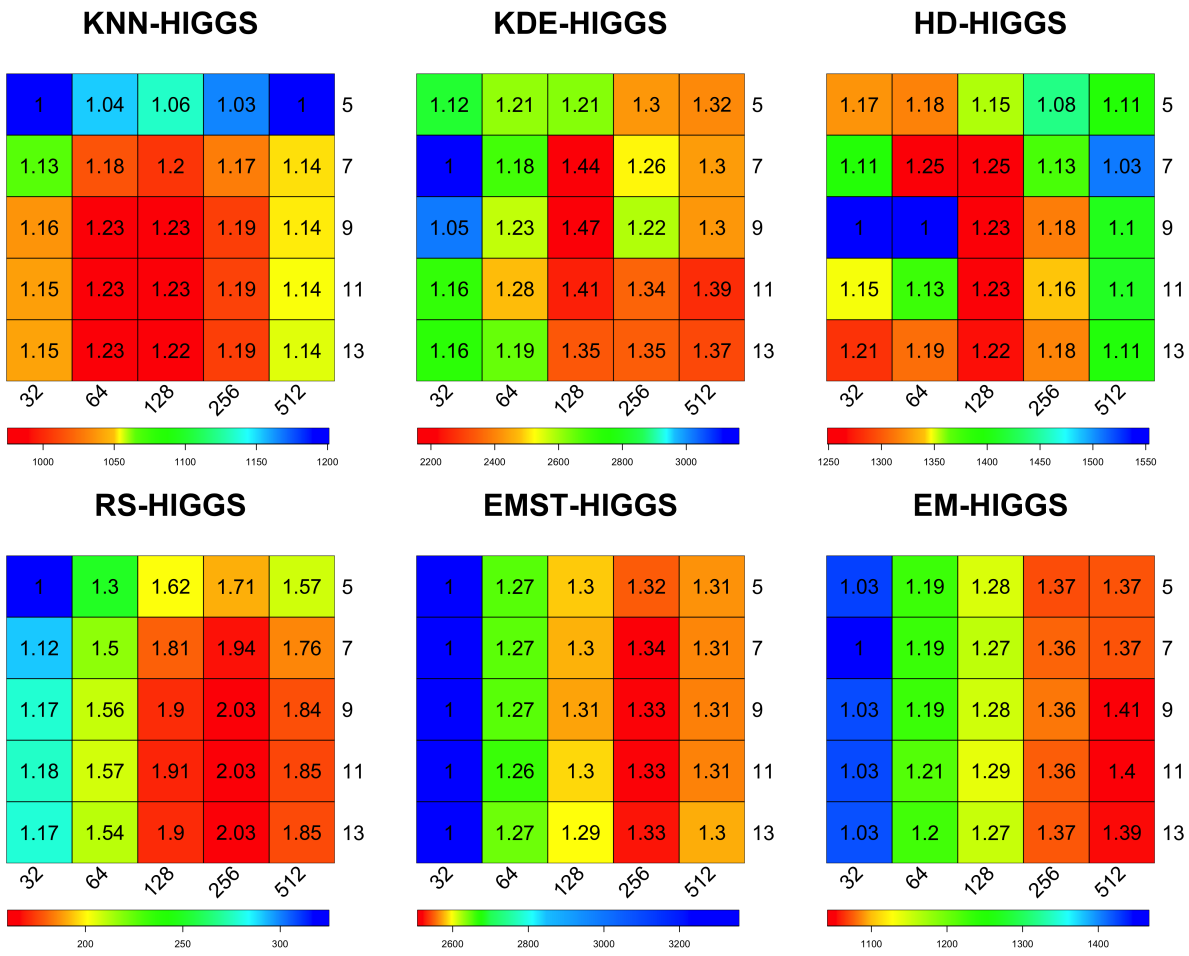


Figure 4.3: Each heatmap presents the influence of leaf size (x-axis) and cut-off level (y-axis) for six algorithms on the HIGGS dataset on a dual-socket Intel Xeon E5-2630 v3 processor.

pressed in PASCAL-X with minimal programming effort, resulting in out-of-the-box NUMA-aware parallel optimized implementations.

4.4 Results and Discussions

The combined benefits of asymptotically optimal algorithms, optimizations, NUMA-aware parallelization, and tuning are substantial. In this section, we analyze the performance of PASCAL-X similar to PASCAL, and finally, we compare PASCAL-X performance against PASCAL.

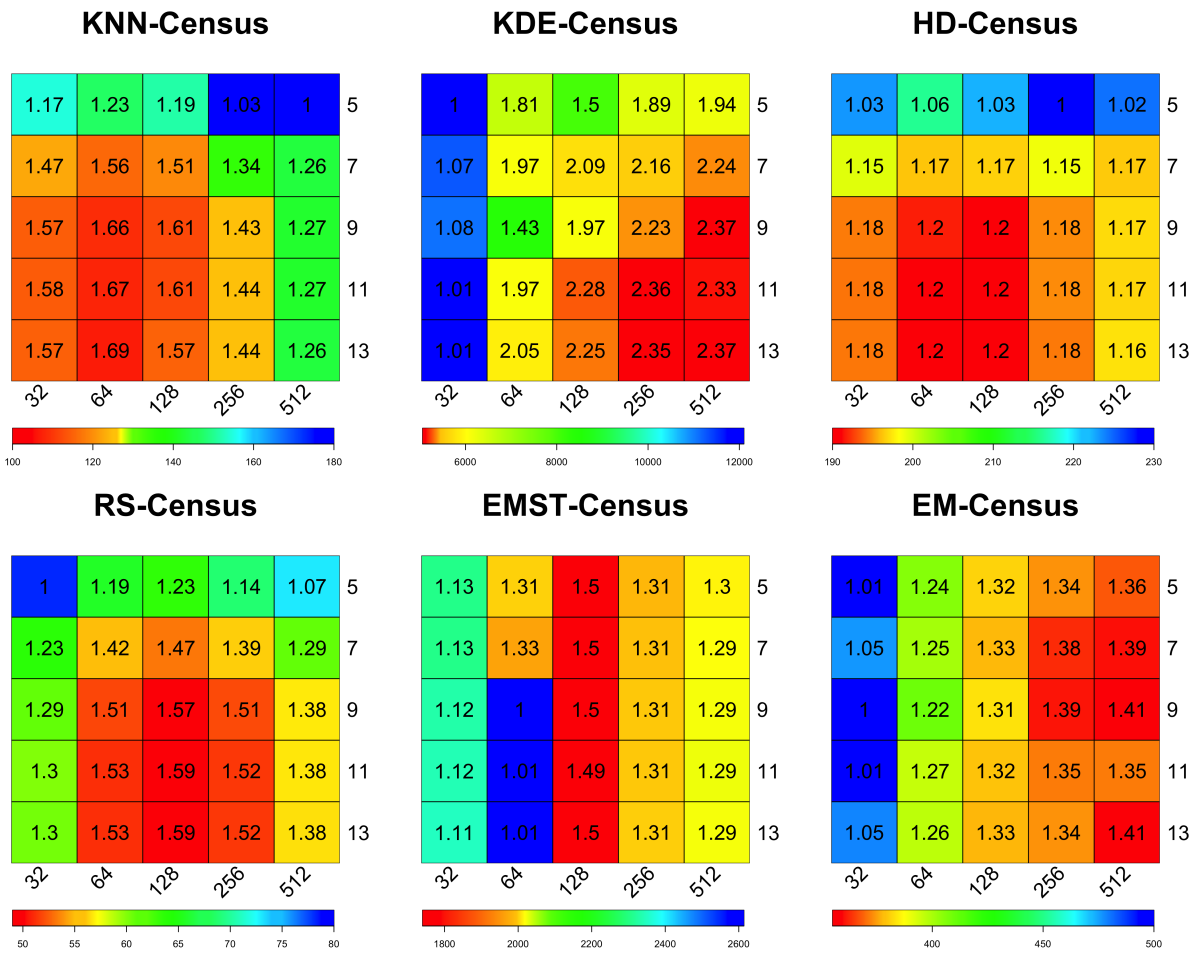


Figure 4.4: Each heatmap presents the influence of leaf size (x-axis) and cut-off level (y-axis) for six algorithms on the Census dataset on a dual-socket Intel Xeon E5-2630 v3 processor.

4.4.1 Comparison with PASCAL

As mentioned in Chapter 3, PASCAL provides state-of-the-art performance against Weka, MATLAB, MLPACK, and scikit-learn. Hence, we compare PASCAL-X performance against PASCAL. Table 4.1 presents the speedup of PASCAL-X over PASCAL for the six algorithms on the five datasets characterized in Table 3.2. The speedup in Table 4.1, shows the combined improvement of NUMA-aware parallelization and tuning, in some cases, PASCAL-X can result in up to $\sim 3\times$ speedup compared to PASCAL.

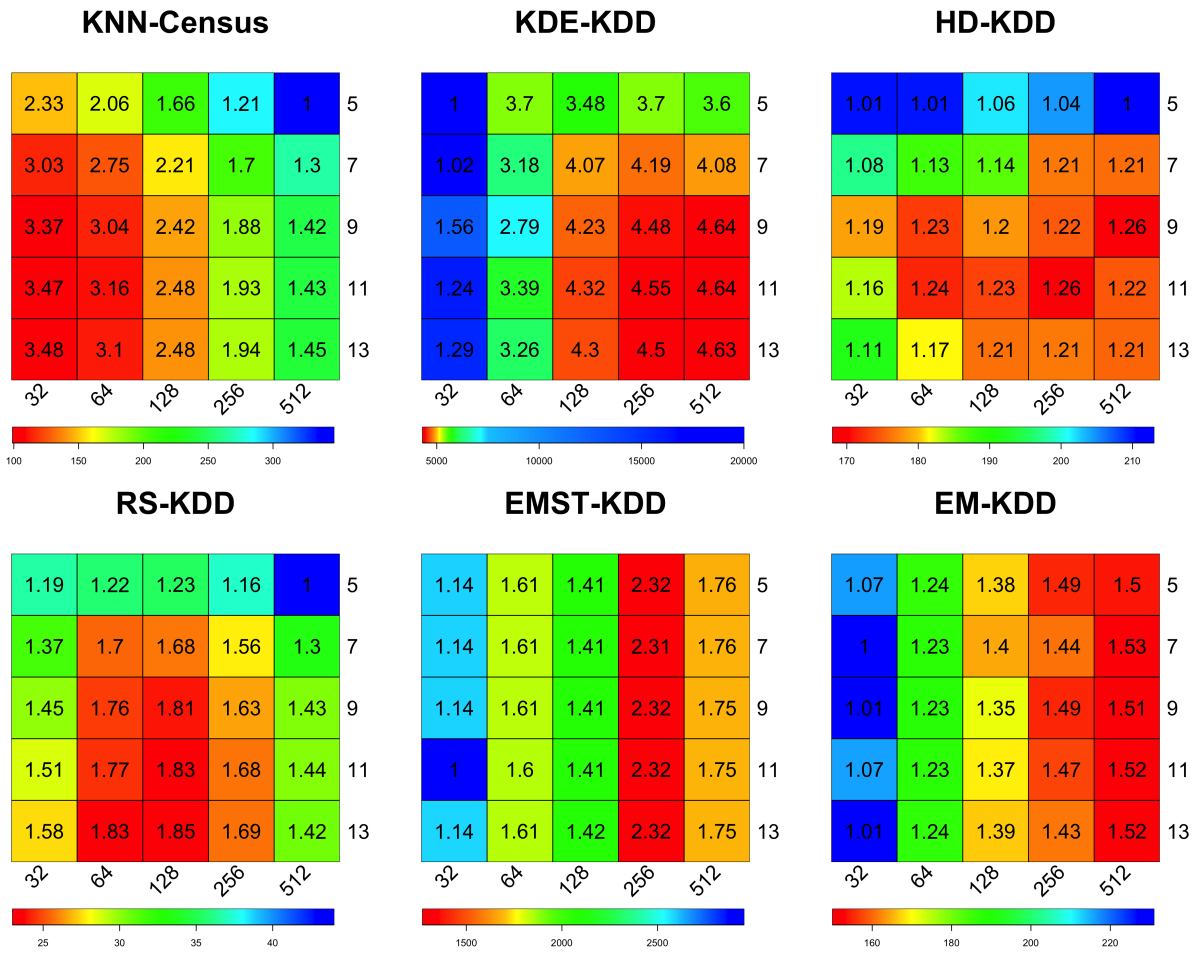


Figure 4.5: Each heatmap presents the influence of leaf size (x-axis) and cut-off level (y-axis) for six algorithms on the KDD dataset on a dual-socket Intel Xeon E5-2630 v3 processor.

	k-NN	EM	KDE	HD	RS	EMST
Yahoo!	1.54	1.19	1.16	1.22	1.75	1.30
HIGGS	1.20	1.18	1.21	1.06	1.56	1.06
Census	1.46	1.14	1.32	1.15	1.38	1.15
KDD	1.50	1.24	1.26	1.25	1.53	1.76
IHEPC	2.94	1.40	1.42	1.07	1.31	1.30

Table 4.1: Speedup of PASCAL-X against PASCAL for the six algorithms on the five datasets intentioned in Table 3.2.

4.4.2 Scalability

Tables 4.2, 4.3, 4.4, 4.5, 4.6 , and 4.7 show the scalability of the six algorithms namely, (i) k-NN with $k = 3$, (ii) RS with range between 0 and 2, (iii) KDE for Gaussian kernel, K with bandwidth, $\sigma = 0.1$ and relative error tolerance set to 0.1, (iv) EM with error tolerance of 0.1, (v) EMST, and (vi) Hausdorff distance with and without NUMA-aware parallelization.

Without NUMA-aware parallelization we gain 14×, 16×, 13×, 14×, 10×, and 14× speedup for Yahoo! over the serial optimized code for k-NN, EM, RS, EMST, KDE, and Hausdorff distance respectively, while NUMA-aware parallelization improves that scaling to 16.23×, 17.1×, 15.23×, 16.11×, 11.94×, and 16.14× speedup respectively. We observe better scaling using NUMA-aware parallelization for all the algorithms as expected, which could be even more beneficial when we use machines with more NUMA nodes.

	2		4		8		16		32	
	noN	+N	noN	+N	noN	+N	noN	+N	noN	+N
Yahoo!	1.83	1.85	3.66	3.82	6.08	6.97	11.92	13.11	14.31	16.23
HIGGS	1.73	1.73	3.86	3.92	6.98	7.17	12.02	13.89	14.81	16.93
Census	1.79	1.79	3.41	3.54	6.19	6.82	11.72	13.02	13.97	16.14
KDD	1.83	1.83	3.36	3.68	5.9	7.01	11.92	13.91	14.51	17.09
IHEPC	1.87	1.87	3.41	3.65	6.38	7.08	12.12	14.01	15.1	17.21

Table 4.2: k-d tree scalability of the k-NN algorithms for five different datasets for both NUMA-aware (+N) and non NUMA-aware (noN) parallelization. The first column shows the number of threads.

4.5 Conclusions

In this chapter, we proposed PASCAL-X as a NUMA-aware extension of PASCAL for N-body problems. We evaluated PASCAL-X experientially in a manner similar to PASCAL, with six N-body problems from different domains, and observe $10 - 309\times$ speedup com-

	2		4		8		16		32	
	noN	+N	noN	+N	noN	+N	noN	+N	noN	+N
Yahoo!	1.82	1.83	3.71	3.75	7.3	7.41	12.1	12.98	16.8	17.41
HIGGS	1.72	1.72	3.7	3.83	7.25	7.42	11.92	13.94	17.3	18.42
Census	1.79	1.79	3.61	3.66	7.01	7.32	11.85	12.35	16.97	17.87
KDD	1.8	1.8	3.69	3.81	6.98	7.12	12.3	14.02	16.86	17.91
IHEPC	1.87	1.87	3.89	3.92	7.3	7.48	12.69	13.97	17.9	18.89

Table 4.3: k-d tree scalability of the EM algorithms for five different datasets for both NUMA-aware (+N) and non NUMA-aware (noN) parallelization. The first column shows the number of threads.

	2		4		8		16		32	
	noN	+N	noN	+N	noN	+N	noN	+N	noN	+N
Yahoo!	1.8	1.8	3.07	3.24	6.92	7.02	11.09	12.92	13.94	15.23
HIGGS	1.73	1.73	3.91	3.95	7.24	7.35	11.7	13.67	13.74	15.59
Census	1.8	1.8	3.73	3.78	6.89	6.99	10.9	12.39	13.94	15.24
KDD	1.8	1.8	3.17	3.48	6.12	6.52	10.84	12.41	13.84	15.97
IHEPC	1.84	1.84	2.87	3.71	6.88	6.92	11.84	13.78	14.94	16.82

Table 4.4: k-d tree scalability of the RS algorithms for five different datasets for both NUMA-aware (+N) and non NUMA-aware (noN) parallelization. The first column shows the number of threads.

	2		4		8		16		32	
	noN	+N	noN	+N	noN	+N	noN	+N	noN	+N
Yahoo!	1.83	1.83	3.52	3.64	6.04	6.73	11.9	13.2	14.01	16.11
HIGGS	1.74	1.74	3.78	3.86	6.94	7.03	12.01	13.91	14.9	16.95
Census	1.81	1.81	3.42	3.48	6.24	6.56	11.74	13.15	13.91	16.02
KDD	1.82	1.82	3.32	3.42	5.91	6.81	11.9	13.64	14.61	16.73
IHEPC	1.86	1.86	3.39	3.68	6.34	6.94	12.01	13.82	15.09	16.91

Table 4.5: k-d tree scalability of the EMST algorithms for five different datasets for both NUMA-aware (+N) and non NUMA-aware (noN) parallelization. The first column shows the number of threads.

	2		4		8		16		32	
	noN	+N	noN	+N	noN	+N	noN	+N	noN	+N
Yahoo!	1.8	1.81	2.5	2.75	4.93	5.13	8.61	9.86	10.13	11.94
HIGGS	1.72	1.72	2.99	3.19	5.35	6.21	7.9	9.31	10.68	12.87
Census	1.81	1.81	3.27	3.32	6.27	6.48	8.06	9.23	9.34	11.34
KDD	1.82	1.82	3.19	3.33	6.67	6.92	8.54	9.46	10.81	12.91
IHEPC	1.85	1.85	3.24	3.47	5.99	6.99	8.01	10.11	9.94	12.24

Table 4.6: k-d tree scalability of the KDE algorithms for five different datasets for both NUMA-aware (+N) and non NUMA-aware (noN) parallelization. The first column shows the number of threads.

	2		4		8		16		32	
	noN	+N	noN	+N	noN	+N	noN	+N	noN	+N
Yahoo!	1.81	1.81	3.59	3.61	6.3	6.58	11.51	13.14	14.01	16.14
HIGGS	1.69	1.69	3.74	3.83	6.89	7.01	12.1	13.82	14.69	16.98
Census	1.75	1.75	3.43	3.52	6.15	6.83	11.65	13.52	13.91	16.22
KDD	1.81	1.81	3.29	3.42	5.87	6.97	11.84	13.74	14.32	16.85
IHEPC	1.86	1.86	3.39	3.47	6.29	7.19	12.1	13.91	14.9	17.13

Table 4.7: k-d tree scalability of the HD algorithms for five different datasets for both NUMA-aware (+N) and non NUMA-aware (noN) parallelization. The first column shows the number of threads.

pared to state-of-the-art software libraries. This improvement in the performance and scalability stems from implementing the NUMA-aware parallelization as well as tuning. We also analyzed the influence of tuning parameters (such as leaf-size and the cut-off level of parallelization) in the final performance.

Chapter 5

Portal: A High-Performance Language and Compiler for Parallel N-body Problems

In this chapter, we present *Portal* [116], a domain-specific language and compiler built on top of PASCAL-X that is designed to enable high-performance implementations of N-body problems on modern multicore systems. Our goal in the development of Portal is three-fold: (a) to implement *scalable, fast* algorithms that have $\mathcal{O}(N \log N)$ and $\mathcal{O}(N)$ complexity, (b) to design an *intuitive language* to enable rapid implementations of a *variety of problems*, and (c) to enable *parallel large-scale* problems to run on multicore systems. We target N-body problems in various domains from machine learning to scientific computing that can be expressed in Portal to obtain an out-of-the-box optimized parallel implementation. Experimental results on six N-body problems show that Portal is within a factor of 5% on average of expert hand-optimized C++ code on a dual-socket AMD EPYC processor. To our knowledge, there are no known libraries or frameworks that implement *parallel asymptotically optimal algorithms* for the class of *generalized N-body problems* and Portal aims to fill this gap. Moreover, the Portal language and intermediate algorithm representation are portable and easily extensible to different platforms.

5.1 Introduction

Modern machines are becoming increasingly more complex resulting in even the most advanced compilers failing to generate the best-optimized code despite many efforts in optimizing performance for compilers [117, 118, 119, 120]. Moreover, Proebsting's Law [24] states that improvements in compiler technology double the performance of typical programs every 18 years, showing the slow progress of general compilers in improving performance. This has resulted in the trend of expert high-performance programmers who desire the best possible performance writing hand-tuned and hand-optimized code that outperforms compiler generated code. Unfortunately, this typically results in implementing a single algorithm or problem, for one or a small subset of architectures. Even if we just consider the domain of interest in this thesis, there are hundreds of N-body problems and it is practically impossible to generate hand-optimized code for every single one of them. Furthermore, hand-tuning is not only tedious but also highly machine-specific. Given the trend that architectures are constantly evolving, these hand-written codes quickly become obsolete.

Scientists who are experts in their particular domain often lack expertise in parallel programming. In general, scientists prefer to program in high-level languages which allow concise expression of their problem. MATLAB and Python are two examples that are widely used in data analytics [25]. However, achieving performance requires computation at a low level and in-depth knowledge of the underlying architecture.

These are two examples of the natural tension between the software goals of performance and productivity. In the former, we have *performance programmers* who sacrifice productivity for performance and in the latter, we have *productivity programmers* whose main goal is rapid prototyping. This motivates the need for an infrastructure to enable *both high-performance and high productivity*. To that end, we present *Portal*, a domain-specific language and compiler embedded in C++ for domain of N-body problems. There is potential for significant

impact in this domain and general N-body implementations today are still orders of magnitude from optimal. The first goal in the development of Portal is to choose optimal algorithms with time/accuracy guarantees. Big data motivates fast approximate algorithms. Portal is built on the top of the PASCAL-X which utilizes tree data-structures and user-controlled pruning or approximations to reduce the asymptotic runtime complexity from being linear in the number of data points to be logarithmic.

5.2 Related Work

In the past few years many domain-specific languages (DSLs) such as DeepDSL [121] for deep learning, Diesel [122] for linear algebra and neural nets, Saiph [123] for computational fluid dynamics, Indigo [124] for image reconstruction, and Halide [125] for image processing have demonstrated that DSLs not only produce terse, extensible, and composable programs but also achieve state-of-the-art performance across different hardware. This is due to a representation where the choices for *how* to execute a program are separated from the definition of *what* to compute. This distinction shows in part the separation between the computation from the problem specification, which gives the compiler flexibility to do the computation in the most efficient manner. Portal, a DSL for domain of N-body problems is inspired by the same philosophy.

DSLs can be stand-alone or embedded. For instance, OptiML [126], DeepDSL [121], and Saiph [123] are embedded in a host language, Scala, while SCOPE [127] is a stand-alone DSL. Embedded DSLs inherit the language constructs of their host language and add domain specific primitives that allow programmers to express their problem at a higher level of abstraction. Embedding also enables ease-of-adoption. For these reasons we choose to develop Portal as an embedded DSL in C++.

5.3 Portal DSL

The design of the Portal DSL is inspired by the mathematical formulation in Equation 2.2. It aims to describe N-body problems in a well-structured high-level form that allows domain experts to focus on the specification of the problem rather than the algorithm or its associated implementation on a target platform.

Recall that an N-body problem is defined using a set of m operators applied to m datasets $(\mathcal{D}_1 ... \mathcal{D}_m)$ using a kernel function, $\mathcal{K}$, as seen in Equation (2.2) in Chapter 2. Each operator, dataset, and/or kernel function can be associated with a layer. Problems are built up by chaining multiple layers to specify a query. Changing the operator, dataset, or the order of layers can change the specification and meaning of the problem. Note that the same dataset may be reused in multiple layers. We explain the structure and semantics of the Portal language using *nearest neighbor*, which has the following form.

$$\forall_q \arg\min_r \ ||x_q - x_r|| \tag{5.1}$$

Code 5.1 shows the Portal specification of the nearest neighbor problem *. `PortalExpr` in line 3 is the main object that holds the problem definition. The first or outer-most layer applies the ∀ `(FORALL)` operator over the query (q) dataset. The second or inner-most layer applies the `argmin (ARGMIN)` operator over the reference (r) dataset, as well as the Euclidean distance kernel function, which calculates the distance between two points. A layer is added to the `PortalExpr` using the `addLayer()` method, which allows a user to build the structure of the problem (lines 4-5). The `execute()` function in line 6 generates and runs the tree-based N-body algorithm, and `getOutput()` in line 7 returns the result of the computation in the format of a `Storage` object.

*Appendix 5.6 describes the grammar of the Portal language

```
Storage query("query_file.csv");
Storage reference("reference_file.csv");
PortalExpr expr;
expr.addLayer(PortalOp::FORALL, query);
expr.addLayer(PortalOp::ARGMIN, reference, PortalFunc::EUCLIDEAN);
expr.execute();
Storage output = expr.getOutput();
```

Portal code 5.1: Portal language spec of the nearest neighbor problem using pre-defined Euclidean distance metric.

As seen from code 5.1, each layer can consist of three components: (a) portal operator, (b) storage object, and (c) kernel/modifying function. We explain each component in detail in the subsequent sections.

5.3.1 Portal Operators

Portal operators are a set of distinct operators that filter the results of their layer and pass it to the next outer layer or the output. A list of these operators is defined in Table 5.1. Each layer requires an operator to specify what sort of filtering happens on the data that layer computes on. We categorize operators into three groups:

- Single variable reduction operators
- Multi variable reduction operators
- All operator

This categorization is helpful in deciding the type of intermediate storage required for each layer. It allows Portal to allocate just the right amount of storage to propagate to outer

layers.

Category	Mathematical Operator	Portal Operator
All	$\forall$	FORALL
Single	$\sum$	SUM
Single	$\prod$	PROD
Single	argmin	ARGMIN
Single	argmax	ARGMAX
Single	min	MIN
Single	max	MAX
Multi	$\cup$	UNION
Multi	$\cup$ arg	UNIONARG
Multi	argmin^k	KARGMIN
Multi	argmax^k	KARGMAX
Multi	min^k	KMIN
Multi	max^k	KMAX

Table 5.1: Mathematical operators supported in Portal along with their categorization into *Single*, *Multi*, and *All* operators.

Single variable reduction operators

These operators take a set of values as input and reduce the set to a single output. These operators are listed in the *Single* category in Table 5.1. For example, when the min operator is applied to a reference dataset, it returns the minimum value which is a single output.

Multi variable reduction operators

These operators take a set of values as input and reduce them to a smaller set of values, typically with a specified length, k. These operators include argmin^k , argmax^k, min^k , max^k, $\cup$, and $\cup$ arg. Each multi variable reduction operator requires us to specify a number k, which limits the number of values that get filtered, except $\cup$ and $\cup$ arg operators. In our

nearest neighbor example, using `argmin` is the same as using $\texttt{argmin}^k$ when $k = 1$. If one desires to compute the k closest reference points to each query point, the Portal operator in the inner `addLayer()` method in line 5 of code 5.1 has to be modified to apply a multi variable reduction operator as shown below.

```
expr.addLayer((PortalOp::KARGMIN, k), reference, PortalFunc::EUCLIDEAN);
```

All operator

The $\forall$ operator does not filter any values and given an input, it returns all the data. In the nearest neighbor example, the $\forall$ operator is applied to the outermost layer to compute the nearest neighbors of **all** query points.

5.3.2 Storage

Each layer includes a Storage object, which corresponds to a dataset. Storage objects are designed to be the primary user-facing data structure. Storage objects can be created from C++ vectors (as shown below) or from CSV files (as shown in lines 1-2 of code 5.1).

```
/* Construct Storage from C++ data-structure */
std::vector<std::vector<float>> input;
// fill in the input data structure with values
Storage query(input);
```

Portal determines the data layout of the Storage objects based on the dimensionality of the dataset to further optimize performance. Portal can choose between column- or row-major data layout. When the dataset has a smaller dimensionality (less than or equal to 4), Portal chooses a column-major layout. Datasets with a larger dimensionality default to a row-major

layout. Portal makes this decision to enable efficient compiler auto-vectorization which is discussed in more detail in Section 5.4.

Portal generates a space-partitioning tree for each input Storage object. The output Storage object will be returned by calling the `execute()` function. Both input and output storage objects are available after `execute()` function. The programmer can delete the input/output storage objects using the `clear()` function as shown below.

```
PortalExpr expr;
// do some computation for expr
expr.clear();
```

5.3.3 Kernel/Modifying Function

The last component constituting a layer is the kernel/modifying function. This function is required for the inner-most layer and is commonly known as the *kernel function* in the literature [54]. Other layers can also specify functions, however; we call these *modifying functions*. The kernel function specifies the *science* of the problem which depends on the distance between pairs of points in the chosen parameter space. Portal implements a set of commonly used distance metrics (for ease of use), which can be used to compose kernel functions.

```
1 expr.addLayer(PortalOp::FORALL, reference, PortalFunc::MANAHATTAN));
2 expr.addLayer(PortalOp::FORALL, reference, PortalFunc::CHEBYSHEV));
3 expr.addLayer(PortalOp::FORALL, reference, PortalFunc::MAHALANOBIS));
4 expr.addLayer(PortalOp::FORALL, reference, PortalFunc::SQREUCDIST));
```

Portal code 5.2: Examples of pre-defined distance metrics.

For instance, code 5.1 uses the pre-defined Euclidean distance metric to specify the kernel function for nearest neighbors which in this case is equal to the distance between points in sets q and r in Euclidean space. Code 5.2 shows examples of other pre-defined distance metrics such as the Manhattan, Chebyshev, Mahalanobis, and Square Euclidean distances.

Portal also allows the user to define custom kernel functions. Code 5.3 shows how Euclidean distance, the same function used in our nearest neighbor example, can be defined using the `Expr` object in line 5. `Expr` can be specified using `Var` objects as input variables. These `Var`s can be used in a wide variety of calculations using a specified set of operations that can be compiled and optimized by Portal. The `Var` objects are then linked to specific layers using the `addLayer` method as shown in code 5.3.

```
1  Storage query("query_file.csv");
2  Storage reference("reference_file.csv");
3  Var q;
4  Var r;
5  Expr EuclidDist = sqrt(pow((q-r), 2));
6  PortalExpr expr;
7  expr.addLayer(PortalOp::FORALL, q, query);
8  expr.addLayer(PortalOp::ARGMIN, r, reference, EuclidDist);
9  expr.execute();
10 Storage output = expr.getOutput();
```

Portal code 5.3: Portal spec for user-defined distance metric and kernel function for nearest neighbors.

To allow for greater flexibility, users can also define their own external C++ functions. For example, if the user wants to use another library to specify a kernel function, the `addLayer` method accepts C++ functions. Since Portal is an embedded DSL, we can take advantages of these libraries and infrastructures provided by our host language. Although external

C++ functions will not be optimized in the same way the internally specified Portal kernel functions are, they allow for greater flexibility on what problems Portal can solve. Additional bindings to other languages, such as MATLAB or Python, can be added in the future.

5.4 Portal Compiler

To synthesize optimal machine code from the high-level Portal language, the Portal compiler proceeds through the major stages as illustrated in Figure 5.1. First, Portal builds space-partitioning trees for the input datasets. The next step is to synthesize code for the three key functions in the multi-tree traversal (Algorithm 3.1). The *Prune/Approximate* and *ComputeApprox* functions in the tree traversal are generated using a modified version of the prune generator supported by the PASCAL algorithmic framework [128]. To use the prune generator in Portal, we modified it to take the Portal operators and kernel function as input and emit the *Prune/Approximate* and *ComputeApprox* functionality in Portal IR.

The Portal core compiler lowers and synthesizes loops for the *BaseCase* given a `PortalExpr` and automatically manages the intra-layer storage injection. Then, it flattens multi-dimensional store, load, and allocations. After flattening, we perform numerical and strength reduction optimizations specific to N-body problems. Figures 5.2 and 5.3 illustrates the IR for the key functions in the tree-traversal (Algorithm 3.1) and walk through the different stages of the compiler transformations for two N-body problems, namely nearest neighbor and kernel density estimation respectively. The reason for choosing these two problems is to illustrate the IR and Portal stages for both prune (nearest neighbor) and approximation (kernel density estimation) examples from the two categories of N-body problems.

Finally, back-end code generation emits x86 machine code using LLVM. The parallelization is applied in the tree traversal which applies to all the N-body problems expressed in Portal.

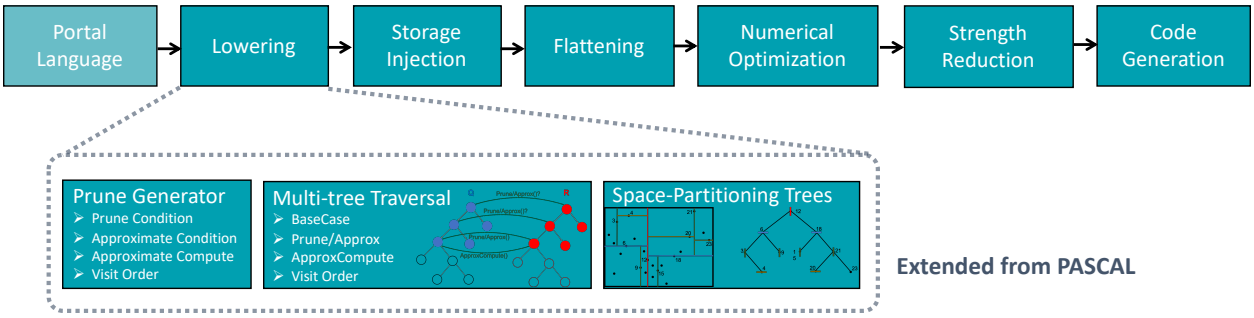


Figure 5.1: Portal block diagram. The core of the compiler lowers the N-body problem defined in the Portal language to imperative code. Portal constructs a loop nest and injects storage for each layer of the loop according to their corresponding operators. Portals back end code generator emits machine code via LLVM.

Note that in addition to the asymptotically optimal tree algorithm, Portal also generates the code for the brute-force algorithm. This is currently used for correctness checks. In the rest of this section, we describe each of these major compiler steps in detail.

5.4.1 Lowering

The first step of our compiler is the lowering process that synthesizes a set of nested loops given a `PortalExpr` object. The order of the loops follows the same sequence as the language specification (*e.g.*, the outermost layer mapping to the outermost loop), since the ordering defines the structure of the N-body computation. Loops are defined by their minimum and maximum values, and all loops implicitly stride by 1.

Next, Portal builds an argument list for each layer and assigns the initial values for each operator and reduction filter. For all inner layer operators, intermediate values of operations are stored in the intermediate storage objects, which are initially assigned with their operators default values. For example, for the `min` operator, the initial value of the intermediate storage is set to the highest value for that specific numeric type (*e.g.*, `DBL_MAX` for double precision data).

After that, the kernel/modifying functions are lowered into Portal IR. Finally, Portal lowers

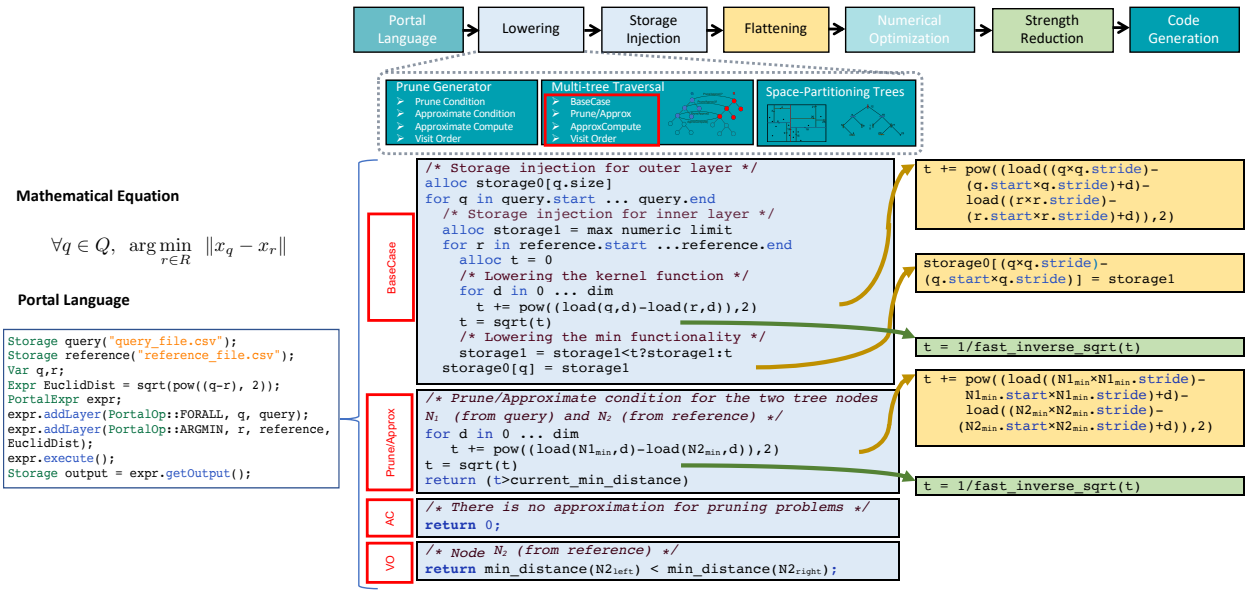


Figure 5.2: The IR representation of the nearest neighbor problem illustrating the IR of the three main functions in the tree traversal (BaseCase, Prune/Approximate, and ApproxCompute) along with the different transformation applied to it. The nearest neighbor returns zero for ApproxCompute since it is a prune N-body problem (shown by the AC red box). Note that there is no numerical optimization applied for this problem since nearest neighbor doesn't use Mahalanobis distance. Portal uses metadata information from the tree, such as min, max, and center of the nodes of the tree for computing the Prune/Approximate condition efficiently; see Chapter 3 for more details on the generation of the Prune/Approximate conditions. The blue colored rectangles (middle) show the IR after lowering and storage injection (Sections 5.4.1 and 5.4.2), while yellow and green colored rectangles (right) present the IR after flattening and strength reduction respectively (Sections 5.4.3 and 5.4.5). The VO red box shows the *Visit Order* function.

the mathematical functionality of each operator at the end of the corresponding synthesized loop. For example, for the `min` operator, Portal generates a comparison imperative code at the end of loop synthesis, to update the minimum computation of that layer.

5.4.2 Storage Injection

Storage injection creates the output and all intermediate data storage for each layer. The intermediate storage is necessary to pass data between layers; each inner layer filters and passes its results to the next outer layer as an intermediate storage. As Portal recursively

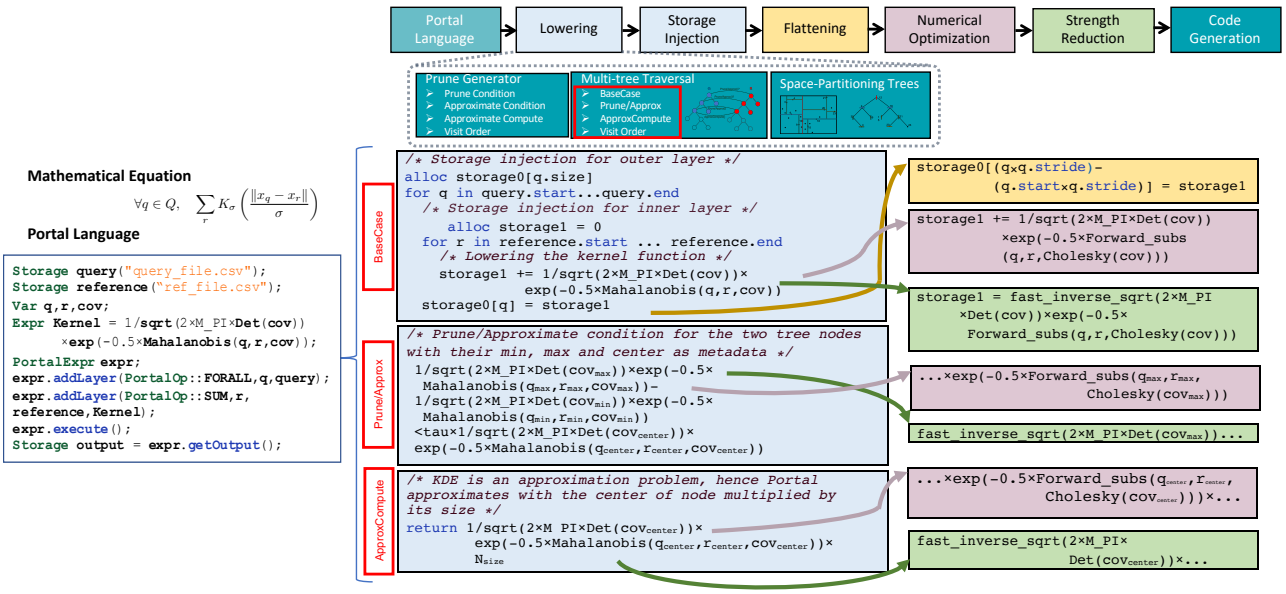


Figure 5.3: The IR representation of kernel density estimation (KDE) problem illustrating the IR of the three main functionalities in the tree traversal (BaseCase, Prune/Approximate, and ApproxCompute) along with the different transformation applied to it. The Kernel ($\mathcal{K}$) in the mathematical equation is the Gaussian kernel. KDE also benefits from the numerical optimizations provided for the Mahalanobis distance computation. Portal uses metadata information from the trees such as min, max, center, and size of the node for computing Prune/Approximate condition and the associated approximation, *ApproxCompute*. This metadata is denoted with the subscript notation. For example, the metadata q_{center} shows the center of a node of the tree built on the dataset corresponding to the variable q in the Portal code for KDE. The blue colored rectangles (middle) show the IR after lowering and storage injection level (subsections 5.4.1 and 5.4.2), while yellow, purple, and green colored rectangles (right) present the IR after flattening, numerical optimization, and strength reduction respectively (Section 5.4.3, 5.4.4, and 5.4.5). Since KDE is an approximation problem, the *Visit Order* returns an arbitrary output.

moves across layers in order to synthesize nested loops, it dedicates storage for each layer depending on the layer's operator and category listed in Table 5.1. The storage injected for *single variable reduction operators* is only one unit of data, as the output of that layer is only a single value, while the storage for *multi-variable reduction operators* is equal to the size of the multi-variable defined by the operator, such as k for `KARGMIN` ($\texttt{argmin}^k$) multi-variable reduction operator. As a special case, the $\forall$ operator injects a storage object equal to the size of that layer's dataset. This is because $\forall$ operator outputs all the calculations.

For example in the nearest neighbor problem, the inner layer searches for the `argmin` on the reference dataset. Since `argmin` is a single reduction variable, we inject one memory location per output of this layer. The outer layer applies a $\forall$ operator on the query data, which results in a storage injection as large as the query set.

5.4.3 Flattening

The Portal compiler flattens multi-dimensional loads and stores into one-dimensional load and store operations. Similarly, for nested loop arguments, Portal flattens the arguments using the base offset and strides from each loop to compute a one-dimensional version of its arguments.

5.4.4 Numerical Optimization

After flattening, Portal performs two optimization passes that exploit domain (N-body) specific knowledge. The first such pass is numerical optimization, which implements a fast calculation between a point x and a distribution Γ, for the Mahalanobis distance. It generalizes the measure of how many standard deviations away x is from the mean of distribution Γ. About 60% of basic statistical inference N-body problems presented in a list of N-body problems mentioned in Grey's Ph.D. thesis [1] have a form of Mahalanobis distance calculation. Therefore, optimizing this distance metric can benefit a large subset of N-body problems.

The Mahalanobis distance between two points i and j is defined as,

$$(x_i - \mu_j)^T \Sigma^{-1} (x_i - \mu_j), \tag{5.2}$$

where μ and Σ are the mean and covariance respectively. Naïvely evaluating this distance

requires computing the inverse of the covariance matrix which is an expensive linear algebra operation that can take on the order of d^3 operations where d is the matrix dimension. However, we can exploit the fact that the covariance matrix Σ is symmetric positive semi-definite.

That is, we rewrite the covariance matrix with a combination of Cholesky decomposition and forward substitution which reduces the complexity to $d^2/2$. First, we factorize the covariance matrix using Cholesky decomposition to obtain a lower triangular matrix as follows.

$$(x_q - \mu_r)^T \Sigma^{-1} (x_q - \mu_r) = (x_q - \mu_r)^T (LL^T)^{-1} (x_q - \mu_r)$$

where L is a lower triangular matrix. If $Y = x_q - \mu_r$, we can further re-write the above equation as,

$$Y^T (LL^T)^{-1} Y = (L^{-1}Y)^T L^{-1} Y$$

Since L is a lower triangular matrix, we can use forward substitution to compute $X = L^{-1}Y$, in turn, reducing the original computation to $X^T X$, a simple and cheap inner product. Moreover, we can now compute the determinant of the covariance matrix, $|\Sigma|$, by simply multiplying the diagonal entries of the lower triangular matrix L, which is another expensive operation that arises in such computations.

5.4.5 Strength Reduction

Operations such as power (`pow`), square-root (`sqrt`), and reciprocal square-root ($1/\sqrt{x}$) that arise in many N-body problems [1] have long latencies. The second optimization pass is *strength reduction*, which replaces such expensive operations with faster, albeit less accurate versions. If the `pow` operation has an exponent less than 4, Portal replaces this with a chained

multiplication. For computing $1/\sqrt{x}$, we use the fast inverse square root that is provided by LLVM, which can result in up to 4× faster performance compared to the naïve version with an error of 0.17% [129]. For approximation problems with an inherent time/accuracy trade-off, this optimization pass can provide an additional tuning knob.

There are two potential ways of calculating $\sqrt{x}$: (1) to multiply x with its fast inverse square root ($x{*}1/\sqrt{x} = \sqrt{x}$), or (2) take the inverse of the fast inverse square root ($1/(1/\sqrt{x}) = \sqrt{x}$). The former choice is faster; however when $x = 0$, it returns NaN, while the latter returns 0 as desired. So, we used the latter in this pass.

5.4.6 Code Generation

Finally, we perform low-level optimizations and emit machine code for the defined N-body problem. The Portal back-end uses LLVM for low-level code generation. We first perform a set of standard passes, such as constant-folding and dead-code elimination. The Portal IR is then lowered to LLVM IR. For the most part, there is a one-to-one mapping between Portal and the LLVM IR. However, for some filters, we provide additional data-structures. For example in the case of multi-variable reduction filters such as min^k, we implement an ordered array of size k to provide this functionality. After generating LLVM IR, the compiler links external functions, including user-defined external kernel functions. Before finishing compilation, this function is finally wrapped around another function that the user and our methods can easily call upon.

The parallelization happens in the tree traversal using OpenMP. Portal uses task parallelization provided by OpenMP and supported by the clang compiler in order to provide scalable N-body computations. The generated code is also auto-vectorized by the compiler. To enable efficient auto-vectorization, Portal chooses between column- and row-major data layout based on the dimensionality of the dataset.

To explain Portal choice for column- and row-major data layout, we break down our nearest neighbor example. The *BaseCase* computation between two leaf nodes consists of three nested loops; the outermost loop iterates over all the points in the query set, the middle loop iterates over all the points in the reference set, and the innermost loop iterates through all the elements within the two chosen query and reference data points to calculate the distance. The loop extent of the innermost loop is determined by the dimensionality of the datasets.

For low dimensional data, the compiler can unroll the innermost loop, thereby exposing vectorization opportunities at the level of the middle loop. To exploit this, we store the data in a column-major layout such that every row stores values from the same dimension for different points. In other words, each data point is stored as a column while in row-major layout, each data point is stored as a row. When using column-major layout, each cache line loads data from different data points. Since the middle loop is the target of vectorization for low dimensional data, this results in less wait for memory loads and consequently better vectorization performance. On the other hand, for high dimensional data, the compiler does not unroll the innermost loop due to large loop counts. Therefore, to exploit vectorization in the innermost loop, we use a row-major data layout where every row stores one data point.

5.5 Evaluation and Discussion

In this section, we first evaluate the code generated by Portal against hand-optimized PASCAL-X implementations of six N-body problems. We then demonstrate Portal's ability to generate optimal code on three additional N-body problems not implemented in PASCAL-X. For the ML problems, we compare their performance against widely used open-source libraries and frameworks such as MLPACK [27] and scikit-learn [26]. For Barnes-Hut computation, we compare against hand-optimized C++ code from the FDPS framework [28]. All evaluations are performed on the state-of-the-art AMD EPYC 7551 multicore processor

with a total of 128 cores.

5.5.1 Experimental Setup

Architecture and Compiler. For our evaluation, we choose a dual-socket AMD EPYC 7551 processor. Each socket has 64 cores, for a total of 128 cores (256 threads with hyper-threading) and a theoretical double precision peak performance of 2611.2 GFlops/s. We use clang compiler version 6.0.0 and LLVM version 6.0.0. We use Python v3.7.0 for scikit-learn v0.20.0 and MLPACK 3.0.3.

Benchmarks. We present results on six real-world datasets characterized in Table 5.2 which is extending the Table 3.2. These include Yahoo! front page module user click log dataset, v1.0 (Yahoo!), Higgs boson's signals and background process dataset (HIGGS), Individual Household Electric Power Consumption dataset (IHEPC), US Census data from 1990 (Census), and KDD Cup 1999 dataset (KDD) from the UCI ML repository [109]. The 3-dimensional dataset (Elliptical) is generated for evaluating the Barnes-Hut algorithm, where particles are angularly uniformly (in spherical coordinates) distributed on the surface of an ellipsoid with an aspect ratio 1:1:4. The elliptical dataset generates an adaptively refined octree.

Dataset	Number of data points	d
Yahoo!	41904293	11
IHEPC	2075259	9
HIGGS	11000000	28
Census	2458285	68
KDD	4898431	42
Elliptical	10000000	3

Table 5.2: Description of the datasets. d: dimensionality.

N-body Problems	Operators	Kernel Function	Pruning/Approximation Condition
k-Nearest Neighbors	$\forall, \arg\min$	$\|\|x_q - x_r\|\|$	Prune $\Leftrightarrow \|\|x_q - x_r\|\| \geq \tau, \quad \forall x_r \in \mathcal{N}_r^{border}$
Range Search	$\forall, \bigcup \arg$	$I(h_{\min} < \|\|x_q - x_r\|\| < h_{\max})$	Prune $\Leftrightarrow \|\|x_q - x_r\|\| > h_{\max}$ or $\|\|x_q - x_r\|\| < h_{\min}$ $\forall x_r \in \mathcal{N}_r^{border}$
Hausdorff Distance	$\max, \min$	$\|\|x_q - x_r\|\|$	Prune $\Leftrightarrow \tau_1 \geq (\mathcal{K}(x_q, x_r) \| \tau_2 \leq \mathcal{K}(x_q, x_r))$, $\forall x_q \in \mathcal{N}_q^{border}, \forall x_r \in \mathcal{N}_r^{border}$
Kernel Density Estimation	$\forall, \sum$	$\mathcal{K}(\frac{\|\|x_q - x_r\|\|}{\sigma})$	Approximate $\Leftrightarrow \mathcal{K}_{\max} - \mathcal{K}_{\min} < \tau \times \mathcal{K}_{\text{center}}$
Minimum Spanning Tree*	$\forall, \arg\min$	$\|\|x_q - x_r\|\|$	Prune $\Leftrightarrow \|\|x_q - x_r\|\| \geq \tau, \quad \forall x_r \in \mathcal{N}_r^{border}$
E-step in EM*	$\forall, \forall$	$r_{nj} = \frac{\pi_j \mathcal{N}(x_n \| \mu_j, \Sigma_j)}{\sum_{i=1}^{\mathcal{H}} \pi_i \mathcal{N}(x_n \| \mu_i, \Sigma_i)}$	Approximate $\Leftrightarrow (r_j^{max} - r_j^{min}) < \tau \times \ r_j^{center}$, $j = 1, .., \mathcal{H}$
Log-likelihood in EM*	$\sum, \log \sum$	$\pi_j \mathcal{N}(x_n \| \mu_j, \Sigma_j)$	Approximate $\Leftrightarrow$ $log \sum_{j=1}^{\mathcal{H}} \pi_j \mathcal{N}(x_{max} \| \theta_j) - log \sum_{j=1}^{\mathcal{H}} \pi_j \mathcal{N}(x_{min} \| \theta_j)$ $< \tau \times \| log(\sum_{j=1}^{\mathcal{H}} \pi_j \mathcal{N}(x_{center} \| \theta_j)) \|$
2-Point Correlation	$\sum, \sum$	$I(\|\|x_q - x_r\|\| < h)$	Prune $\Leftrightarrow \|\|x_q - x_r\|\| \geq h, \quad \forall x_r \in \mathcal{N}_r^{border}$
Naïve Bayes Classifier	$\forall, \arg\max$	$\mathcal{N}(x_n \| \mu_k, \Sigma_k)$	Prune $\Leftrightarrow \mathcal{N}(x_n \| \mu_k, \Sigma_k) > \tau$
Barnes-Hut	$\forall, \sum$	$f = \frac{GM_1M_2}{(\|\|x_q - x_r\|\|)^2}$	Approximate $\Leftrightarrow$ $E < \tau \times \|\|x_q - x_r\|\|, \ x_r : \mathcal{N}_r^{center}, E : \mathcal{N}_r^{diameter}$

Table 5.3: Summary of the characteristics of the eight N-body problems chosen for evaluation (EM consists of two N-body sub-problems: E-step and log-likelihood). The kernel functions are evaluated on the border points given by $\mathcal{N}_r^{border}$ and $\mathcal{N}_q^{border}$. τ, τ_1, τ_2, and σ represent threshold values defined based on the N-body problem or controlled by the user, E represents the side length of a node. * denotes iterative algorithms.

5.5.2 Comparison with PASCAL-X

Table 5.3 shows a summary of key characteristics of the six N-body problems chosen for comparison namely, k-nearest neighbor (k-NN), range search (RS), kernel density estimation (KDE), Hausdorff distance (HD), minimum spanning tree (MST), and expectation maximization (EM). These six problems cover a variety of different N-body problems including pruning vs. approximation and direct vs. iterative problems. They also include problems from different metric spaces (*e.g.*, Euclidean vs. Mahanalabois space). PASCAL-X improves the performance of PASCAL, which achieves orders of magnitude higher performance compared to widely used libraries and frameworks such as MLPACK, scikit-learn, MATLAB, and Weka [128] for all six problems. Therefore, we compare against PASCAL-X's hand-optimized implementations.

		k-NN	KDE	RS
Census	Expert	22.83	1087.42	42.4
	Portal	23.94	1129.31	44.3
	% Difference	4	3	5
Yahoo!	Expert	84.63	133.75	214.52
	Portal	85.24	139.52	223.17
	% Difference	2	4	4
IHEPC	Expert	8.72	39.24	15.01
	Portal	9.11	41.72	16.11
	% Difference	4	6	7
HIGGS	Expert	186.02	411.82	122.32
	Portal	191.01	430.91	130.12
	% Difference	3	4	6
KDD	Expert	21.42	926.53	20.12
	Portal	22.63	945.98	21.13
	% Difference	5	2	4
Line of Code	Expert	867	626	673
	Portal	13	15	13
	× Shorter	67	42	52

Table 5.4: Comparison of the Portal running time (in seconds) against PASCAL for k-Nearest Neighbor(k-NN), Kernel Density Estimation(KDE), and Range Search (RS) across 5 datasets. Each dataset spans 3 rows that report their respective performance and % difference. The last 3 rows show Lines of Code(LOC) as an indicator for user productivity. The last row (× shorter) shows the factor that Portal code is shorter than PASCAL.

Table 5.4 and 5.5 present the running time across five datasets for the code generated by Portal and hand-optimized PASCAL-X implementations for the six problems. Both implementations use the same k-d tree data-structure (built using median partitioning strategy by splitting along the widest dimension) and the same multi-tree traversal template (Algorithm 3.1). For both Portal and PASCAL-X, we use the number of cores that deliver the highest performance for each dataset and problem combination. We also empirically tune the algorithmic parameters, *leaf size* and *cut-off level* in parallelization, which control the

		MST	EM	HD
Census	Expert	374.11	76.32	40.94
	Portal	391.74	82.61	43.13
	% Difference	4	8	5
Yahoo	Expert	918.47	224.53	122.84
	Portal	946.13	242.82	129.76
	% Difference	3	8	5
IHEPC	Expert	200.87	78.61	38.42
	Portal	211.02	85.35	40.14
	% Difference	5	8	4
HIGGS	Expert	478.37	198.81	236.65
	Portal	486.26	216.71	243.83
	% Difference	2	9	3
KDD	Expert	273.65	32.44	36.26
	Portal	281.07	35.32	38.34
	% Difference	3	8	5
Line of Code	Expert	956	1681	689
	Portal	54	104	13
	× Shorter	17	16	53

Table 5.5: Comparison of the Portal running time (in seconds) against PASCAL for Minimum Spanning Tree (MST), Expectation Maximization(EM), and Hausdorf Distance(HD) across 5 datasets. Each dataset spans 3 rows that report their respective performance and % difference. The last 3 rows show Line of Code (LOC) as an indicator for user productivity. Note that MST and EM in Portal only require 12 and 30 lines respectively for specifying the N-body problem. The rest of the code implements the iterative logic which is written in native C++ code. The last row (× shorter) shows the factor that Portal code is shorter than PASCAL.

size and number of tasks created. Since we rely on the OpenMP work-stealing scheduler to balance the work across all 256 threads (at the highest concurrently) across eight NUMA domains, it is critical to tune these parameters to achieve scalability, especially at this scale on a multicore system.

Across a range of problems and datasets, our Portal language is able to *express* and our Portal compiler is able to *generate* implementations using optimal tree-based algorithms that are within 5% (on average) of state-of-the-art performance. We observe the largest deviation of $8-9\%$ for EM because of external function calls of matrix calculations of covariance that are required for evaluating the kernel function.

In addition to running time, we also report the number of lines of code (LOC) in the last three columns of Table 5.4 and 5.5. Portal programs require significantly fewer code lines than hand-written hand-optimized code. For example, the Portal version of k-nearest neighbors was written in 13 lines of code and achieves within $2-5\%$ of PASCAL-X performance. EM, a soft clustering algorithm composed of two N-body sub-problems is the longest Portal program, is written in 104 LOC, consisting of 30 lines of Portal code and 74 lines of native C++ code; this is $16\times$ fewer LOC compared to PASCAL-X. Note that we do not report the tree construction, tree traversal, and prune generator LOC for PASCAL-X since these modules can be reused when implementing a new problem. In summary, these results show the potential of Portal to express a wide range of N-body problems while achieving competitive performance with reduced programming effort.

5.5.3 Validation

We also validate Portal with three other N-body problems, namely 2-point correlation, naïve Bayes classifier and Barnes-Hut, that are not implemented in PASCAL-X. We refer the reader to the last three rows of Table 5.3 for the operators and the kernel functions defining

		2-PC	NBC	BH
Census	State-of-the-art	3529.23	1337.14	–
	Portal	53.12	87.22	–
	Speedup	66	15	–
Yahoo!	State-of-the-art	37043.52	3629.73	–
	Portal	250.28	198.49	–
	Speedup	4	3	–
IHEPC	State-of-the-art	4281.84	1699.92	–
	Portal	26.36	88.07	–
	Speedup	162	19	–
HIGGS	State-of-the-art	17823.58	5231.31	–
	Portal	151.39	261.14	–
	Speedup	117	20	–
KDD	State-of-the-art	5134.82	981.61	–
	Portal	31.14	47.22	–
	Speedup	165	21	–
Elliptical	State-of-the-art	5412.42	1026.81	473.51
	Portal	94.01	194.52	278.13
	Speedup	57	5	1.7

Table 5.6: Comparison of the Portal performance (in seconds) against state-of-the-art libraries/packages. state-of-the-art for 2-point correlation (2-PC), naive Bayes classifier (NBC), and Barnes-Hut (BH) are scikit-learn[26], MLPACK [27], and FDPS [28] respectively. Note that Barnes-Hut is limited to 3-dimensional data, hence there is a dash(-) for higher dimension datasets. The elliptical dataset is a 3-dimensional dataset specifically generated for Barnes-Hut.

these three problems. Since no framework like Portal exists at this time for generalized N-body problems (to the best of the author's knowledge), we compare against open-source ML libraries/packages and an optimized Barnes-Hut implementation. Specifically, we compare against scikit-learn [26] for 2-point correlation, and MLPACK library [27] for naïve Bayes classifier. This is due to the fact that MLPACK does not implement 2-point correlation, but both MLPACK and scikit-learn implement the naïve Bayes classifier. However, MLPACK delivers the best performance among the two, so we only report MLPACK performance.

Scikit-learn is an open source project with more than 170,000 downloads, with 85 releases by 913 contributors. MLPACK is a C++ machine learning library with emphasis on speed and ease of use with more than 118 contributors. Both scikit-learn and MLPACK implement a tree-based algorithm for 2-point correlation and naïve Bayes respectively. For Barnes-Hut computation, we compare against FDPS, a high-performance framework for particle simulations which provides a hand-optimized C++ implementation [28]. The FDPS framework is designed specifically for parallel particle simulations.

We choose k-d tree as the tree type for fair comparison across all implementations, and octree for Barnes-Hut. Table 5.6 shows that Portal achieves $66 - 165\times$ better performance than scikit-learn for 2-point correlation and $15 - 47\times$ faster than MLPACK for the naïve Bayes classifier across different datasets. Portal achieves $1.7\times$ better performance compared to FDPS for a Barnes-Hut computation on 10 million particles.

In summary, these results validate the potential of our approach resulting in performance that is orders of magnitude faster than state-of-the-art libraries. Additional algorithms can be expressed in this style with minimal programming effort resulting in out-of-the-box parallel optimized implementations.

5.6 Appendix

The Portal grammar is presented in listing 5.4. The <*name*> in the grammar is the same as variable names in the C++ language. The <*call*> function in the grammar is for pre-defined functions defined in Portal such as Mahalanobis, Cholesky, etc. as well as user-defined function calls (similar to C++ function calls).

```
<PortalProgram> → <StorageDef>+ <VarDef>*
                                    <PortalExprDef>
<StorageDef> → "Storage" <name> "("<file_name> ")"
              |"Storage" <name> "("<std_vector>")"
<VarDef> →  "Var" <name>
<PortalExprDef> → <ExprDef> <AddLayer>+ <Execute>
<ExprDef> → "PortalExpr" <name>
<AddLayer> → <name>".addlayer("<OP>"," <name> ","
                      <name> "," <Kernel>? ")"
<Kernel> →  sqrt(Kernel) | pow(Kernel) |
            <expression> | <call> (a*) |...
<Execute> →  <name>".execute()"
<OP> →  "FORALL" | "SUM" | "PROD" | "ARGMIN" |
        "ARGMAX" | "MIN" | "MAX" | "UNION" |
        "UNIONARG" | "KARGMIN" | "KARGMAX" |
        "KMIN" | "KMAX"
```

Portal code 5.4: Grammar specification for Portal.

5.7 Conclusions

Portal is a high-performance domain-specific language and compiler for generalized N-body problems. We show how a DSL with an appropriately high-level mathematical formulation leads directly to both asymptotically fast algorithms and their efficient parallel implementations on x86 platforms. Moreover, Portal enables terse expression of the problem, thereby reducing the line of code written by experts up to 67× while obtaining performance comparable to expert tuned code. Portal DSL and intermediate representation are independent of underlying architecture which makes it easily extensible to different platforms.

We hope Portal will enable scientific discovery not only for N-body problems in scientific

computing and machine learning but to a number of related problems in domains such as computer graphics, computational geometry, and applied mathematics that can be expressed in Portal to obtain an out-of-the-box parallel optimized implementation.

Chapter 6

Case Study

In this chapter, we present a real-world example of using Portal for implementing a face clustering algorithm. In other words, the face clustering example is used as a case study to evaluate the ability of Portal to deliver high-performance solutions for N-body problems in different domains.

6.1 Introduction

In this chapter, we consider the face clustering problem, which categorizes a large number of unlabelled faces into individual identities presented in a data set. This problem occurs in many applications such as social media, forensics, and law enforcement. In many of these applications, the number of faces could be in orders of thousands and millions. For examples, Facebook revealed in a white paper [130] that its users are uploading 350 million images per day on average, and a large portion of these images are face images. However, some identity information has been provided by tagging, but many of the images are missing identity information. Clustering algorithms provide opportunities to organize these face images into identities and auto-tagging. For instance, Facebook's tag-suggestion method suggests possible name tags for identity considering similar faces with tagging information.

In law enforcement, finding the suspects or perpetrators requires an investigation in large media collections. Meanwhile, FBI is including over 50 million face photographs in the Next Generation Identification (NGI) dataset, in order to enable investigator leads the searching ability for images similar to a suspect's photo*. The large scale of data in different domains rises the need for high-performance scalable face clustering implementations. Some of the images in these datasets do not have the desired quality in terms of pose, illumination, and occlusion. In order to mitigate the mentioned issues, recent progress in face recognition leverage a state-of-the-art convolutional neural network for face representation [29, 131]. Using this state-of-the-art face representation, the face clustering accuracy is still not perfect, especially with difficult datasets which contain faces with pose variation, occlusion, mask, expression, and change of illumination. [132].

Zhu et al. [133] developed a rank-order distance-based clustering algorithm for face tagging, which introduces a distance measure based on shared nearest neighbors of face images. Another challenge in face clustering is scalability and performance, considering the high volume of data. Otto et al. [29] developed a version of the rank-order clustering which improves scalability and simplifies the actual clustering procedure by considering a smaller set of nearest neighbors in their distance metric. We extend the Otto et al. rank-order clustering method to a fast simplified version and implement it using Portal. We evaluate our implementation in Portal using a well-known LFW dataset [134]. By simplifying the Otto et al. method we get even faster results preserving similar accuracy.

6.2 Related Work

Clustering is well known in data analysis and has been used in many different fields such as statistics, machine learning, and pattern recognition [135, 136, 137]. Clustering for images

*http://goo.gl/UYlT8p

and faces becomes more challenging when the number of images and clusters are very large. In clustering faces, there have been different face representation methods such as scale invariant feature transform (SIFT) [138] and convolutional neural networks (CNN) [139], as well as various distance metrics such as cosine similarity [140] and chi-squared distances [141]. Hence, the quality of clustering not only depends on the choice of the clustering algorithm but also on the metric distance and face representation method.

6.2.1 General Image Clustering

A substantial amount of research work has been done for clustering images in general [142, 143, 144, 145]. Foo et al. [146] address the detection of near-duplicate images. Their algorithm identifies similarities between images considering different image processing operations including cropping, rotation, and conversions. For image representation, they applied a visual word approach to the local PCA-SIFT descriptors and indexed it with a Locality Sensitive Hashing scheme. Gong et al. [147] design a version of k-means clustering which encodes images feature vectors to binary vectors, and use an indexing method to support constant-time lookup for cluster centers of k-means. Moëllic et al. [148] present a method for extracting meaningful and representative clusters based on a shared nearest neighbors approach, which considers both content-based features and textual descriptions (tags). By using the nearest neighbor method, their approach is able to build representative clusters.

6.2.2 Face Clustering

Different methods have been proposed for clustering faces specifically. Cui et al. [149] extract features from the detected faces and apply spectral clustering in order to develop a semi-automatic tool for annotating photographs. They use a human operator for manually adjusting the clustering results. This algorithm is further improved by Tian et al. [150] via using a probabilistic clustering model, including an additional class to discard clusters

with not enough tightly distributed data. Zhao et al. [151] combined a variety of contextual information such as time and the probability of faces of certain people to appear in one image, as well as identity estimates to cluster personal photograph collection using hierarchical clustering. Zhu et al. [133] designed a similarity measure based on the ranking of faces in each other's nearest neighbor lists. These nearest neighbors lists are formed using a simple distance metric such as Euclidean distance. They applied a hierarchical clustering on the calculated rank-order distance function.

6.3 Case Definition

As mentioned in the previous section, many different clustering methods have been developed in the literature. Considering the evaluation presented by Otto et al. [152] for different face clustering approaches we chose the approximate version of rank-order clustering for Portal's case study due to its high performance and high accuracy. After explaining the face representation method (Section 6.3.1), we present the original rank-order clustering (Section 6.3.2), then approximate rank-order clustering (Section 6.3.3) and finally its symmetric version implemented in Portal (Section 6.3.4 & 6.3.5).

6.3.1 Face Representation

Following the success of deep convolutional neural networks for face representation by various researchers[†], we consider a deep learning approach for our face representation in clustering. In this chapter, we use the architecture described in [131]. For extracting the feature, given an input image, 68 facial landmarks are detected using the DLIB library implementation method by Kazemi and Sullivan [153]. Each image is normalized based on the detected keypoints, by rotating based on the angle between the eyes and locating the eye lines at 45

[†]http://vis-www.cs.umass.edu/lfw/results.html

percent of image height from the top of the image; similarly the mouth line is located at 25 percent of the image height from the bottom of the image, and the midpoint of all the key points is centered in the X dimension. The final aligned image is scaled to 110×110, and the center 100×100 is the normalized image.

In the next step, the normalized image is inputted to a deep convolutional neural network [154] with a total of 10 convolutional layers and small 3×3 filters. The neural network architecture consists of pairs of convolutional layers followed by max-pooling layers, repeated four times and finally two convolutional layers followed by an average pooling layer, with ReLU neurons following all convolutional layers, except for the last one. The output of the final average pooling layer is a 320-dimensional vector used as our feature vector, which during training is fed into a fully connected layer followed by a softmax loss. The final 320-dimensional output of the average-pooling layer is used as a feature vector in our clustering experiment.

6.3.2 Rank-Order Clustering

Zhu et al. [133] developed the rank-order clustering algorithm as a form of agglomerative hierarchical clustering [155], which uses the nearest neighbor based distance metric. The general approach in this algorithm is to initialize all samples to be separate clusters, then compute distances between pairs of clusters, and merge clusters with a distance below a predefined threshold. It iteratively computes the cluster-to-cluster distance in each iteration and merges the clusters based on the recomputed distances. They defined the distance between two clusters to be the minimum distance between any two samples in the pair of clusters. The distance defined in rank-order clustering between two images a and b is as follows.

$$dist(a, b) = \sum_{i=1}^{\mathcal{O}_a(b)} \mathcal{O}_b(f_a(i)), \tag{6.1}$$

where $f_a(i)$ is the ith face in a's neighbor list; $\mathcal{O}_a(b)$ represents the ranking of face b in a's neighbor list, and $\mathcal{O}_b(f_a(i))$ represents the ranking of face $f_a(i)$ in b's neighbor list. These nearest neighbors lists are generated using Euclidean distance. The distance in Equation (6.1) is asymmetric, therefore a symmetric distance measure is build upon it, which is considered as the final distance between a and b, as

$$dist_{sym}(a,b) = \frac{dist(a,b) + dist(b,a)}{min(\mathcal{O}_a(b), \mathcal{O}_b(a))} \tag{6.2}$$

The symmetric rank order distance in Equation (6.2) generates low value if the two faces are close to each other and have neighbors in common. As mentioned before, the clustering is performed by initializing each face image as its own cluster, computing the symmetric distance between each cluster, and merging clusters with distance below the specified threshold. The iterations continue until no further cluster can be merged considering the threshold. In this clustering algorithm, rather than specifying the desired number of clusters, the threshold specifies the final number of clusters.

6.3.3 Approximate Rank-Order Clustering

Otto et al. [29] developed approximate rank-order clustering, which uses a short list of nearest neighbors for each image instead of computing all neighbors for nearest neighbor list of every sample in the dataset. Applying their method requires some modification of the original rank-order clustering algorithm. They use the k top nearest neighbors under the assumption that cluster formation relies on local neighborhoods. Moreover, they enforce locality by only computing the approximate rank-order distance between pairs of samples for which both are within each other sample's top nearest neighbors.

Also, they note that considering a short list of top-k neighbors, the presence or absence of

a particular example on the short nearest neighbors list may be more significant than the sample's numerical rank. Hence, they consider a distance measure based on summing the absence/presence of shared nearest neighbors, rather than their ranking, which results in a distance function between images a and b as follows.

$$dist'(a,b) = \sum_{i=1}^{k} I_a(\mathcal{O}_b(f_a(i),k)), \tag{6.3}$$

where $I_b(x,k)$ represents an indicator function resulting in 0 if face x is in face b's top-k nearest neighbors list, and 1 otherwise. This distance function implies that the presence or absence of shared neighbors towards the top of the nearest neighbors list is important, while the numerical value or the ranks are not adding additional information (based on their empirical experiments). The same normalization process as in the original rank-order clustering is applied as given below.

$$dist'_{sym}(a,b) = \frac{dist'(a,b) + dist'(b,a)}{min(\mathcal{O}_a(b), \mathcal{O}_b(a))} \tag{6.4}$$

Furthermore, in order to improve the runtime of the clustering step, they only compute distances between samples which appear in each other's nearest neighbors list and only perform one round of merges of individual faces into clusters, rather than performing multiple merge iterations as in the original rank-order algorithm.

The final approximate rank-order clustering algorithm is summarized as:

- Extract deep features for each face
- Compute the k-nearest neighbors for each face
- Compute the pairwise distance between each face and the faces in its top k-nearest

neighbors list for which the face is also on the neighbor's nearest neighbors list

- Merge all the pairs of faces with distances below a pre-defined threshold

The selected threshold determines the number of clusters and is one of the challenging tasks in data clustering. We consider different thresholds in our experiments and present the best result for the chosen dataset. For building the k-nearest neighbors list in the computation of approximate rank-order, we use a space-partitioning tree named randomized k-d tree similar to [29] for direct comparison with state-of-the-art. The randomized k-d tree is built similar to the k-d tree algorithm explained in Section 2.2.1. The only difference is that the randomized k-d tree improves efficiency by generating multiple k-d trees by choosing the first splitting dimension randomly, and simultaneously searches among the different generated k-d trees. The search will stop when a specified number of tree nodes has been visited [156].

6.3.4 Symmetric Approximate Rank-Order Clustering

The normalization in the original rank-order distance measure in Equation (6.2) was designed to eliminate the asymmetry of the distance measure in Equation (6.1). This normalization is also applied to the approximate rank-order in [29]. However, in approximate rank-order distance measure (Equation (6.3)), we are essentially counting the number of shared neighbors in each other's nearest neighbors list (equal to the size of the intersection of the two lists). This count of shared neighbors in the two lists is symmetric, which essentially eliminates the need for the normalization. Hence, in our experiment, we use the Equation (6.3) which is symmetric. Note that, we are changing the approximate rank-order algorithm, however, we are able to gain similar accuracy.

Portal Implementation

In order to implement this face clustering algorithm in Portal, we observe that there are two N-body problems embedded in the approximate rank-order clustering outlined in Section 6.3.3 as depicted in Figure 6.1.

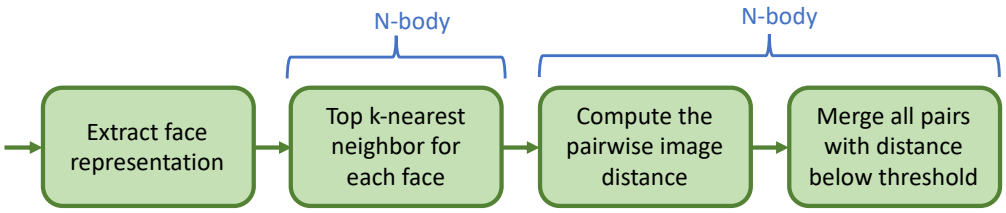
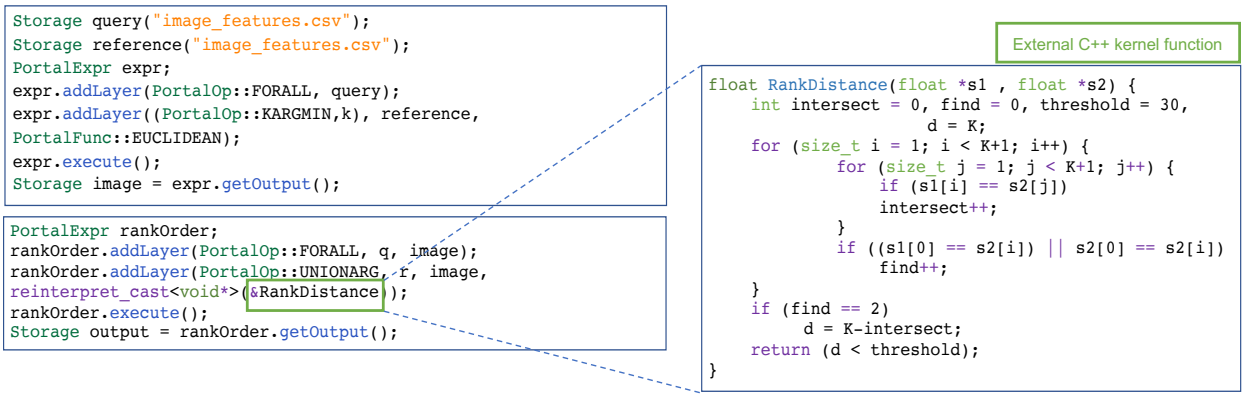


Figure 6.1: Steps for approximate rank-order algorithm

The first N-body problem is calculating the top k-nearest neighbors for all the images in the dataset, which has been detailed in Chapter 5. The second N-body problem is defined with $\forall$ and $\cup$arg as the operators, and the kernel function is the intersection of the two lists of k-nearest neighbors for each face image. The intersection kernel function does not satisfy the requirement for Portal's kernel function (monotonically decreasing with distance); hence, Portal uses a brute force solution for the second N-body problem instead of the tree method. Since Portal is embedded in C++ we are still able to provide this kernel function as a external C++ function as presented in Figure 6.1.

6.3.5 Improved Symmetric Approximate Rank-Order Clustering

For supporting the kernel function mentioned in Section 6.3.4, we can extend Portal to further improve the face clustering algorithm. In order to extend Portal for implementing face clustering algorithm we need to consider two main steps: (1) adding the functionalities needed for face clustering kernel function and (2) adding a space-partitioning tree such as cover tree which supports different distance metrics including the distance metric used in the face clustering algorithm.



Portal code 6.1: Face clustering algorithm in Portal using an external function as the kernel. The left side shows portal implementation for the two N-body problems in Figure 6.1: k-nearest neighbor (top) and final clustering (bottom). The right side shows the external function used as the kernel function for computing the approximate rank distance.

For adding the functionality we extend Portal to include the *intersection* and *exist* functionalities. The *intersection* functionality will count the size of the intersection between two lists of numbers similar to the mathematical operation intersection ($\cap$). In the face clustering example intersection is used to count the number of shared nearest neighbors in each other's list. The *exist* functionality checks if a value exists in a list of numbers, and in the face clustering example it is used to check if one image exists in the list of nearest neighbors of another image. So we can write the kernel function for face clustering outlined in Figure 6.2.

```
Var r,q;
Expr RankDistance = exist(r,q) && exist(q,r) && (intersection(r,q) < threshold);
PortalExpr rankOrder;
rankOrder.addLayer(PortalOp::FORALL, q, image);
rankOrder.addLayer(PortalOp::UNIONARG, r, image, RankDistance);
rankOrder.execute();
Storage output = rankOrder.getOutput();
```

Portal code 6.2: Face clustering algorithm in Portal using Portal's functionalities.This code represents the second N-body problem in Figure 6.1.

We also extend Portal to use the cover tree for the computation of face clustering algorithm. The cover tree's ability to support different metric spaces [46] allows utilization of Portal

for algorithms with different distance metrics such as approximate rank order distance. Portal's prune generator algorithm is developed in a modular templated way, meaning that it gets the type of the tree as a template parameter. This design helps to modularized and separate the functionalities which are specific to each tree from prune generator. The prune generator is calling functionalities for computing the minimum and maximum distances on the inputted tree template (the tree type that has been used as the template parameter of prune generator). The implementation of these functionalities is provided for each tree type in their corresponding class. As a result, for face clustering extension, we develop our cover tree considering the approximate rank order distance as the metric distance. In the cover tree, the center and radius of each node are used to compute the minimum and maximum distances as mentioned in Section 2.2.4.

6.4 Face Clustering Evaluation

For Portal implementation of face clustering, we use a similar evaluation of clustering performance as Otto et al. [29]. They use a predefined definition of correct clustering, which considers the identity, and evaluates accuracy in terms of clusters corresponding to known identity labels. They evaluate the clustering quality using *pairwise precision/recall*.

Pairwise precision is defined as the fraction of pairs of samples within a cluster which are of the same class, over the total number of same-cluster pairs in the datasets. *Pairwise recall* is defined as the fraction of pairs of samples within a class which are placed in the same cluster, over the total number of same-class pairs in the dataset. The *pairwise precision/recall* are able to address two types of errors; as a method that clusters all the samples as individual clusters will have high precision, but low recall, while a method that clusters all samples in the same cluster will have high recall but low precision. These two measures are combined

to define the F-measure as follows.

$$F = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{6.5}$$

We use F-measure to evaluate the accuracy of the clustering implementation. We use LFW [134] dataset, which contains 13,233 face images of 5,749 individuals. The dataset was compiled by searching for images of celebrities and public figures while retaining only images with an automatically detectable face. For our evaluation, we choose a dual-socket Intel Xeon E5-2630 processor. Each socket has 8 cores, and a theoretical peak performance of 614.4 Gflops/s. We use clang compiler version 6.0.0 and LLVM version 6.0.0. For the experiments, we use our implementation of the original and approximate rank-order clustering algorithm.

Clustering method	Image distance	k-NN	Clustering	Performance (Sec)	Accuracy
Rank-order [133]	$dist(a,b) = \sum_{i=1}^{\mathcal{O}_a(b)} \mathcal{O}_b(f_a(i))$	Naive	Naive	21134	0.66
Approximate rank-order [29]	$dist'(a,b) = \sum_{i=1}^{k} I_a(\mathcal{O}_b(f_a(i), k))$	R-kdtree	Naive	655	0.87
Symmetric approximate rank-order	$dist'(a,b) = \sum_{i=1}^{k} I_a(\mathcal{O}_b(f_a(i), k))$	R-kdtree	Naive	343	0.86
Improved symmetric approximate rank-order	$dist'(a,b) = \sum_{i=1}^{k} I_a(\mathcal{O}_b(f_a(i), k))$	R-kdtree	Cover tree	308	0.86

Table 6.1: The evaluation of the three clustering algorithm mentioned in this chapter in terms of performance and accuracy. k-NN column refers to the underlying algorithm used for the computation of k-NN. the R-kdtree refers to randomized k-d tree.

Table 6.1 shows the performance and cluster accuracy (F-measure) for each of the clustering algorithms mentioned in this chapter. As we can see the Portal implementation achieves similar accuracy to the approximate rank-order distance. The performance improvement of symmetric approximate rank-order clustering (about 2× speedup) in Portal implementation stems from eliminating the normalization part, as mentioned in section 6.3.4. The additional performance improvement in the improved symmetric approximate rank-order clustering stems from the use of the cover tree for the clustering N-body computation.

Note that in the previous chapters we represented results for k-d tree which is built very

well-balanced by using median splitting, resulting in a better opportunity for parallelization and consequently higher performance. In some of the N-body problems, the pruning causes a load imbalance in the tree which is mitigated using the work-stealing. However, for Barnes-Hut, we use the octree, which doesn't have a well-balanced structure due to its splitting method (details in Section 2.2.3), gaining only 70% improvement against the state-of-the-art framework FDPS. Using unbalanced trees could influence the final performance. Another factor influencing the performance is the underlying dataset and the distribution of data in the metric space. Factors such as the nature of the cover tree, which could be very imbalanced because of its structure to satisfy the three invariants mentioned in Section 2.2.4, and the underlying distribution of data influence the final performance gain for clustering algorithm which is 10% for the LFW dataset.

6.5 Conclusion

We have shown the feasibility of using Portal in a real-world problem namely, face clustering. The face clustering problem is of practical interest for organizing large collections of unlabelled face images, especially in forensic investigation and social media. Being able to express this problem in only 14 lines of code in Portal while maintaining high-performance and accuracy results in better productivity for scientists.

Chapter 7

Conclusions and Future directions

7.1 Conclusion

In this thesis, we proposed high-performance parallel solutions for the domain of N-body problems by developing an algorithmic framework as well as a domain-specific compiler. The main contributions and results of this thesis can be summarized as follows.

- **PASCAL**: In Chapter 3, we addressed the design and development of PASCAL, a unified Parallel Algorithmic SCALable framework for general N-body problems. PASCAL provides space partitioning trees as well as user-controlled pruning or approximations in order to reduce the runtime complexity of N-body problems from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$ and even $\mathcal{O}(N)$. The automatic generation of pruning and approximation conditions by PASCAL for each N-body problem makes this framework a suitable choice for fast prototyping of new N-body problems. For example, we produced the first pruning condition for the Hausdorff distance using the PASCAL prune generator algorithm, and hence achieve an high-performance solution for its calculation. Also, PASCAL introduced the visit order functionality. Adding the visit order results in more opportunities to prune and consequently, better performance.

- **PASCAL-X**: In Chapter 4, we address some of the opportunities for improving the

performance of the PASCAL framework. PASCAL-X is an extension of PASCAL which provides better scalability of tree traversal on NUMA systems. Moreover, we analyse the influence of tuning parameters such as leaf size and cut-off level in generating tasks. We show that by exploring tuning parameters, we can improve the performance up to 4.5×.

- **Portal**: In Chapter 5, we take a step further in providing high-performance code for N-body problems by developing a domain specific language and code generator named Portal. We show how a DSL with an appropriately high-level mathematical formulation leads directly to both asymptotically fast algorithms and their efficient parallel implementations on x86 platforms. Portal enables terse expression of an N-body problem, thereby reducing the lines of code written by experts up to 67× while obtaining performance comparable to expert hand-tuned code. The Portal DSL and intermediate representation are independent of the underlying architecture, which makes it easily extensible to different platforms.

- **Case Study**: In Chapter 6, we use a face clustering example as a case study for evaluation of Portal to deliver high-performance solutions for real-world N-body problems. In order to support the face clustering case study, we extend Portal by adding cover tree as well as functionalities such as intersection and exist. We generate a high-performance solution for face clustering in only 14 lines of Portal code while attaining an accuracy similar to the state-of-the-art.

The code for this thesis is open-source* and available to enable domain scientists for harnessing the performance power of parallel computing.

*https://gitlab.com/Nbody-Portal/Code

7.2 Future Directions

Currently, our work assists domain scientists to generate high-performance, optimized, and scalable codes for a large group of problems. However, we believe there are still opportunities for improving performance and increasing coverage of our framework to support a larger class of problems. For instance, one could extend Portal to support many types of trees, or add more splitting criteria for each tree in order to gain more pruning and better performance. Additionally, it is possible to add an auto-tuner to either systematically explore the parameter spaces or intelligently tune the parameters in Portal.

7.2.1 Extending Tree Data Structures

Portal is easily extensible to support different types of trees and their various splitting criteria. By plugging different space-partitioning trees with different splitting criteria in Portal, we could obtain new opportunities for higher performance and better scalability. For instance, Figures 7.1 and 7.2 represent the k-d tree for the KDD dataset (shown in Table 3.2) with two different splitting criteria: (1) median splitting, which divides the widest dimension on the median data point, and (2) mid-point splitting, which divides the data in the middle of the widest dimension (similar to the concept of quadtree and octree). We can see that the median splitting results in a more balanced tree which could be more beneficial for load balancing, while mid-point splitting results in a less balanced tree but could provide different pruning opportunities. Their influence on performance depends on the dataset, the distribution of the data, and the N-body computation to be performed. Tuning and analysis of different trees and their splitting criterion represent an interesting future direction of this work.

In the Figures 7.1 and 7.2, each level of tree is shown with a different color, and each node is divided into two, since k-d tree is a binary tree. Each rectangle represents a hyper-rectangle

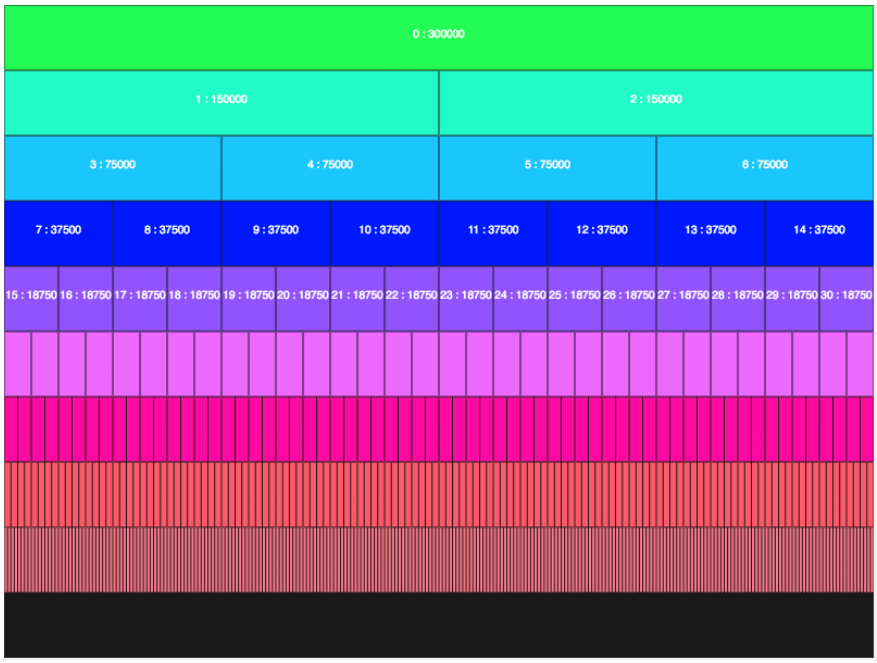


Figure 7.1: Visualization of k-d tree for KDD dataset with median splitting criterion on the widest dimension.

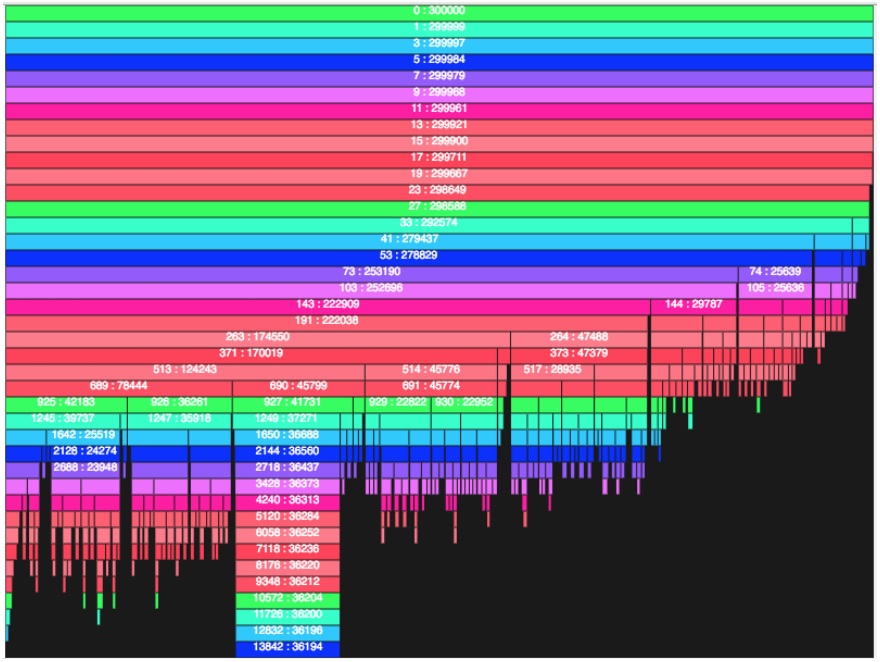


Figure 7.2: Visualization of k-d tree for KDD dataset with mid-point splitting criterion on the widest dimension.

in the k-d tree showing the node's ID followed by a colon and the size of that node.

7.2.2 Autotuner

By supporting more tree data structures and alternative splitting criteria, the tuning parameters will expand even further. This reinforces the need to select the optimal tuning parameters such as the tree structure, splitting criterion, leaf size, and cut-off level. An autotuner could be designed or included off-the-shelf in Portal in order to select the optimal tuning parameters for the execution. Each of the tuning knobs has its own set of distinct parameters that have to be adjusted for a specific architecture. The autotuner could explore the tuning space and the interaction between high-level algorithm and low-level architecture.

7.2.3 Back End

Portal only generates code for the x86 architecture. As a future direction, the back end of Portal could be extended to generate code for GPUs. In order to make the best use of this extension, there is a need for expanding the autotuner, to select the best performing tree on a given architecture (CPU or GPU) or a hybrid system. Furthermore, GPU-specific optimizations and transformations could be added to the back end of the Portal compiler. In extending Portal for GPU back end, there is a need to address the challenges of running the irregular tree traversal algorithm on the GPU. There are several interesting bodies of work for implementing the tree traversal on GPUs [110, 157, 158, 159] and Portal's back end could be extended to support GPU execution as well.

7.2.4 Wrapper for Other Languages

We can extend Portal's front-end to support other languages such as Python and MATLAB. This would be useful because many domain scientists write their code in such high-level

languages and extending the Portal's front-end to support these languages would increase the adoptability of this work.

Chapter 8

Appendix

The table below represents all the variables used in this thesis, accompanied by their related sections, Equations, and definitions.

Variable	Section	Equation	Explanation
x_r	2.1	2.1	the reference point
x_q	2.1	2.1	the query point
$f(x_q)$	2.1	2.1	the desired potential at the query point x_q
$s(x_r)$	2.1	2.1	the density at the reference point x_r
$\mathcal{K}(x_q, x_r)$	2.1	2.1	a kernel function with two inputs x_r and x_q
op_i	2.1	2.2	the ith operator in N-body definition
$\mathcal{D}_i$	2.1	2.2	the ith dataset in N-body definition
m	2.1	2.2	the number of datasets
$\mathcal{K}(x_1, ..., x_m)$	2.1	2.2	a kernel function $\mathcal{K}$ with m inputs
l	2.2	–	the number of data points in the leaf node
T	2.2.4	–	the cover tree, as a leveled tree data structure

S	2.2.4	–	the data set used in building the cover tree T
C_i	2.2.4	–	the set of data points in S associated with the nodes in the level i of the cover tree T
v	2.2.4	–	a data point in dataset S
w	2.2.4	–	a data point in dataset S
$d(v, w)$	2.2.4	–	distance between two data points v and w in cover tree
θ	2.3.1	–	Multipole Acceptance Criterion (MAC) ratio
r	2.3.1	–	the distance between the interested particle and center of mass
E	2.3.1	–	the side length of the node
$I(\|\|x_q - x_r\|\| < h)$	2.3.2	2.3	the step function used for 2-point correlation with threshold h
R_{size}	2.4.2	2.4.2	the size of the reference dataset
K_σ	2.4.2	2.4.2	a zero-centered probability density function
π_k	2.4.4	2.6	the mixture weight for cluster k
$\mathcal{H}$	2.4.4	2.6	the number of Gaussian mixture components or clusters
$\theta_j = \{\mu_j, \Sigma_j\}$	2.4.4	2.6	the parameters of Gaussian component j, with mean vector μ_j and covariance matrix Σ_j

r_{nj}	2.4.4	2.7	the responsibility, as the weight factor of data point n for cluster j
Z_j	2.4.4	2.10	the sum of all the weight factors for cluster j
π_j^{new}	2.4.4	2.10	the new mixture weight for component j
$\mu_j^{new}, \Sigma_j^{new}$	2.4.4	2.10	the new parameters of Gaussian component j
$ll(\theta)$	2.4.4	2.11	the log-likelihood computation
$\mathbf{X}$	2.4.5	2.12	a feature vector
n	2.4.5	2.12	the size of the feature vector
A	2.4.5	2.12	a class in naïve Bayes classifier
$(h_{\min}, h_{\max})$	2.5.1	2.13	the range used in the range search
$\mathcal{N}^{all}$	3.3.1	–	a set of nodes
$\mathcal{R}$	3.3.1	–	a rule set
$\mathcal{N}_i$	3.3.1	–	a node from the tree build on the dataset i
$\mathcal{N}_i^{split}$	3.3.1	–	a node generated by splitting node $\mathcal{N}_i$
$\mathcal{N}_i^c$	3.3.1	–	the center of node $\mathcal{N}_i$
$\mathcal{N}_i^{border}$	3.3.1	–	the set of border data points for the node $\mathcal{N}_i$
τ	3.4	–	a threshold
$op_{\oplus}$	3.4	–	a general representation for an operator
δ	3.4	3.2	the evaluation of the kernel function on the points in $\mathcal{N}_r^{\text{border}}$ for each x_q

K_{center}	3.4	3.3	calling the kernel function on the center of a node
r_i^{center}	3.4	3.4	the responsibility of the center data point from cluster i
$r_i^{\max}$	3.4	3.4	the maximum responsibility between all the data points from cluster i
$r_i^{\min}$	3.4	3.4	the minimum responsibility between all the data points from cluster i
τ_i	3.4	3.5	the nested prune threshold for level i
d	3.5.2	–	the dimension of data points in vector space
Y	5.4.4	–	the difference between a data point and center of distribution
L	5.4.4	–	the lower triangular matrix in Cholesky decomposition
G	5.5	–	the gravitational constant in Newton's law of universal gravitation
M_i	5.5	–	the mass of object i
a, b	6.3.2	6.1	variables used for representing image indexes
$dist(a, b)$	6.3.2	6.1	distance defined in rank-order clustering between two images a and b
$dist_{sym}(a, b)$	6.3.2	6.2	symmetric distance defined in rank-order clustering between two images a and b

$dist'(a, b)$	6.3.2	6.3	approximate distance defined in rank-order clustering between two images a and b
$dist'_{sym}(a, b)$	6.3.2	6.4	approximate symmetric distance defined in rank-order clustering between two images a and b

Table 8.1: List of variables in this thesis.

Bibliography

[1] Alexander Gray. *Bringing tractability to generalized N-body problems in statistical and scientific computation*. PhD thesis, Carnegie Mellon University, 2003.

[2] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–348, 1987.

[3] John Salmon and Michael Warren. Fast parallel tree codes for gravitational and fluid dynamical N-body problems. *International Journal of High Performance Computing Applications*, 8(2):129–142, 1994.

[4] Yasmeen Farouk and Sherine Rady. Supervised classification techniques for identifying Alzheimer's disease. In *International Conference on Advanced Intelligent Systems and Informatics*, pages 189–197. Springer, 2018.

[5] Sami Dhahbi, Walid Barhoumi, and Ezzeddine Zagrouba. Breast cancer diagnosis in digitized mammograms using curvelet moments. *Computers in Biology and Medicine*, 64:79–90, 2015.

[6] Xindong Wu and Vipin Kumar. *The top ten algorithms in data mining*. CRC press, 2009.

[7] Yongyue Zhang, Michael Brady, and Stephen Smith. Segmentation of brain MR images through a hidden Markov random field model and the expectation-maximization algorithm. *IEEE Transactions on Medical Imaging*, 20(1):45–57, 2001.

[8] Raman Vijay and Jagu Subhashini. An efficient brain tumor detection methodology using k-means clustering algorithm. In *International Conference on Communication and Signal Processing*, pages 653–657. IEEE, 2013.

[9] Matthew Gerber. Predicting crime using twitter and kernel density estimation. *Decision Support Systems*, 61:115–125, 2014.

[10] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. Trends in big data analytics. *Journal of Parallel and Distributed Computing*, 74(7):2561–2573, 2014.

[11] Ayşe Demirhan. Classification of structural MRI for detecting Alzheimer's disease. *International Journal of Intelligent Systems and Applications in Engineering*, pages 195–198, 2016.

[12] Ahsan Bin Tufail, Ali Abidi, Adil Masood Siddiqui, and Muhammad Shahzad Younis. Automatic classification of initial categories of Alzheimer's disease from structural MRI phase images: a comparison of PSVM, KNN and ANN methods. *International Journal of Biomedical and Biological Engineering*, page 1731, 2012.

[13] Seyyid Ahmed Medjahed, Tamazouzt Ait Saadi, and Abdelkader Benyettou. Breast cancer diagnosis by using k-nearest neighbor with different distances and classification rules. *International Journal of Computer Applications*, 62(1), 2013.

[14] Moh'd Rasoul Al-Hadidi, Abdulsalam Alarabeyyat, and Mohannad Alhanahnah. Breast cancer detection using k-nearest neighbor machine learning algorithm. In *9th International Conference on Developments in eSystems Engineering (DeSE)*, pages 35–39. IEEE, 2016.

[15] Mourad Mouhamed, Mona Soliman, Ashraf Darwish, and Aboul Ella Hassanien. Interest points detection of 3D mesh model using k-means and shape curvature. In *International Conference on Advanced Intelligent Systems and Informatics*, pages 415–424. Springer, 2018.

[16] Chinki Chandhok, Soni Chaturvedi, and Aleefia Khurshid. An approach to image segmentation using k-means clustering algorithm. *International Journal of Information Technology (IJIT)*, 1(1):11–17, 2012.

[17] Vitorino Ramos and Fernando Muge. Map segmentation by colour cube genetic k-mean clustering. In *International Conference on Theory and Practice of Digital Libraries*, pages 319–323. Springer, 2000.

[18] Pan Ng and Chi-Man Pun. Skin color segmentation by texture feature extraction and k-mean clustering. In *Third International Conference on Computational Intelligence, Communication Systems and Networks*, pages 213–218. IEEE, 2011.

[19] Najeed Ahmed Khan, Hassan Pervaz, Arsalan Khalid Latif, and Ayesha Musharraf. Unsupervised identification of Malaria parasites using computer vision. In *11th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 263–267. IEEE, 2014.

[20] Enrique Bermejo Nievas, Oscar Deniz Suarez, Gloria Bueno García, and Rahul Sukthankar. Violence detection in video using computer vision techniques. In *International Conference on Computer Analysis of Images and Patterns*, pages 332–339. Springer, 2011.

[21] Tessa Anderson. Kernel density estimation and k-means clustering to profile road accident hotspots. *Accident Analysis & Prevention*, 41(3):359–364, 2009.

[22] Xindong Wu, Vipin Kumar, Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey McLachlan, Angus Ng, Bing Liu, Yu Philip, Zhi-Hua Zhou, Michael Steinbach, David Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, 2008.

[23] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing

research: A view from Berkeley. Technical Report, UCB/EECS-2006-183, University of California, Berkeley, 2006.

[24] Kevin Scott. On Proebsting's law. Technical report, CS-2001-12, Department of Computer Science, University of Virginia, 2001.

[25] Ceyhun Ozgur, Taylor Colliau, Grace Rogers, Zachariah Hughes, and Myer Tyson. MATLAB vs. Python vs. R. *Journal of Data Science*, 15(3):355–372, 2017.

[26] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[27] Ryan Curtin, James Cline, Neil Slagle, William March, Parikshit Ram, Nishant Mehta, and Alexander Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 14:801–805, 2013.

[28] Masaki Iwasawa, Ataru Tanikawa, Natsuki Hosono, Keigo Nitadori, Takayuki Muranushi, and Junichiro Makino. Implementation and performance of FDPS: a framework for developing parallel particle simulation codes. *Publications of the Astronomical Society of Japan*, 68(4), 2016.

[29] Charles Otto, Dayong Wang, and Anil Jain. Clustering millions of faces by identity. *IEEE transactions on pattern analysis and machine intelligence*, 40(2):289–303, 2017.

[30] David Halliday, Jearl Walker, and Robert Resnick. *Fundamentals of Physics*. John Wiley & Sons, 2013.

[31] Susanne Pfalzner and Paul Gibbon. *Many-body tree-methods in physics*. Cambridge University Press, 2005.

[32] Hugh Couchman. Mesh-refined P3M-A fast adaptive N-body algorithm. *The Astrophysical Journal*, 368:L23–L26, 1991.

[33] Bruce Naylor. Constructing good partitioning trees. In *Graphics Interface*, pages 181–181. Canadian Information Processing Society, 1993.

[34] Josh Barnes and Piet Hut. A hierarchical $\mathcal{O}(n \log n)$ force calculation algorithm. *Nature*, 324(446-449), December 1986.

[35] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM (CACM)*, 18(9):509–517, September 1975.

[36] Ashraf Masood Kibriya. *Fast algorithms for nearest neighbour search*. PhD thesis, The University of Waikato, 2007.

[37] Tomás Skopal, Jaroslav Pokornỳ, and Vaclav Snasel. PM-tree: Pivoting metric tree for similarity search in multimedia databases. In *European Conference on Advances in Databases and Information Systems (ADBIS)*, 2004.

[38] Matthias Kunze and Mathias Weske. Metric trees for efficient similarity search in large process model repositories. In *International Conference on Business Process Management*, pages 535–546. Springer, 2010.

[39] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-time rendering*. AK Peters/CRC Press, 2019.

[40] Christer Ericson. *Real-time collision detection*. CRC Press, 2004.

[41] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[42] Ting Liu, Andrew Moore, and Alexander Gray. New algorithms for efficient high-dimensional nonparametric classification. *Journal of Machine Learning Research*, 7(Jun):1135–1158, 2006.

[43] Stephen Omohundro. *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[44] John Hammersley. The distribution of distance in a hypersphere. *The Annals of Mathematical Statistics*, pages 447–452, 1950.

[45] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[46] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *International Conference on Machine Learning*. ACM, 2006.

[47] Mike Izbicki and Christian Shelton. Faster cover trees. In *International Conference on Machine Learning*, pages 1162–1170, 2015.

[48] Harry Strange and Reyer Zwiggelaar. Intrinsic dimensionality. In *Open Problems in Spectral Dimensionality Reduction*, pages 41–52. Springer, 2014.

[49] Aparna Chandramowlishwaran. *The fast multipole method at exascale*. PhD thesis, Georgia Institute of Technology, 2013.

[50] Guy Blelloch and Girija Narlikar. A Practical Comparison of N-Body Algorithms. *Parallel Algorithms: Third DIMACS Implementation Challenge, October 17-19, 1994*, 30:81, 1997.

[51] Alison Coil. *The large-scale structure of the universe*. Planets, Stars and Stellar Systems, Springer, 2013.

[52] Phillip Peebles and James Edwin. *The large-scale structure of the universe*. Princeton University Press, 1980.

[53] Lucy Liuxuan Zhang and Ue-Li Pen. Fast N-point correlation functions and three-point lensing application. *New Astronomy*, 10(7):569–590, 2005.

[54] Alexander Gray and Andrew Moore. *N*-Body Problems in Statistical Learning. In *Advances in Neural Information Processing Systems*. MIT Press, 2000.

[55] Andrew Moore, Andy Connolly, Chris Genovese, Alex Gray, Larry Grone, Nick Kanidoris II, Robert Nichol, Jeff Schneider, Alex Szalay, and Istvan Szapudi. Fast algorithms and efficient statistics: N-point correlation functions. In *Mining the Sky*, pages 71–82. Springer, 2001.

[56] Sunil Arya. *Nearest Neighbor Searching and Applications*. PhD thesis, College Park, MD, USA, 1995. UMI Order No. GAX95-39606.

[57] Kenneth Clarkson. Nearest neighbor searching and metric space dimensions. *Nearest neighbor methods for learning and vision: theory and practice*, pages 15–59, 2006.

[58] Bernard Silverman. *Density estimation for statistics and data analysis*. Taylor & Francis Group, Routledge, 2018.

[59] Khosrow Dehnad. *Density estimation for statistics and data analysis*. Taylor & Francis Group, 1987.

[60] Emanuel Parzen. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076, 1962.

[61] Vassiliy Epanechnikov. Non-parametric estimation of a multivariate probability density. *Theory of Probability & Its Applications*, 14(1):153–158, 1969.

[62] Jaroslav Nešetřil and Helena Nešetřilová. The origins of minimal spanning tree algorithms–Boruvka and Jarník. *Documenta Mathematica*, pages 127–141, 2012.

[63] Otakar Boruvka. Contribution to the solution of a problem of economical construction of electrical networks. *Elektronickỳ obzor*, 15:153–154, 1926.

[64] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[65] Joseph Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[66] Jason Rennie, Lawrence Shih, Jaime Teevan, and David Karger. Tackling the poor assumptions of naïve Bayes text classifiers. *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 616–623, 2003.

[67] Joseph Hellerstein, Thathachar Jayram, and Irina Rish. *Recognizing end-user transactions in performance management.* IBM Thomas J. Watson Research Division Hawthorne, NY, 2000.

[68] Tom Mitchell. Machine learning and data mining. *Communications of the ACM*, 42(11), 1999.

[69] Pedro Domingos and Michael Pazzani. On the optimality of the simple Bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997.

[70] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29(2-3):131–163, 1997.

[71] Jørgen Hilden. Statistical diagnosis based on conditional independence does not require it. *Computers in Biology and Medicine*, 14(4):429–435, 1984.

[72] Pat Langley, Wayne Iba, and Kevin Thompson. An analysis of Bayesian classifiers. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, volume 90, pages 223–228. MIT Press, 1992.

[73] Irina Rish. An empirical study of the naïve Bayes classifier. In *Workshop on Empirical Methods in Artificial Intelligence*, volume 3, pages 41–46, 2001.

[74] Jing Han, Ee Haihong, Guan Le, and Jian Du. Survey on NoSQL database. In *International Conference on Pervasive Computing and Applications*, pages 363–366. IEEE, 2011.

[75] Herbert Tropf and Helmut Herzog. Multidimensional range search in dynamically balanced trees. *Applied Informatics*, (2):71–77, 1981.

[76] Jiří Matoušek. Geometric range searching. *ACM Computing Surveys (CSUR)*, 26(4):422–461, 1994.

[77] Daniel Huttenlocher, Gregory Klanderman, and William Rucklidge. Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863, 1993.

[78] Marie-Pierre Dubuisson and Anil Jain. A modified Hausdorff distance for object matching. In *International Conference on Pattern Recognition*, volume 1, pages 566–568. IEEE, 1994.

[79] Yue Lu, Chew Lim Tan, Weihua Huang, and Liying Fan. An approach to word image matching based on weighted Hausdorff distance. In *Proceedings of Sixth International Conference on Document Analysis and Recognition*, pages 921–925. IEEE, 2001.

[80] Yongsheng Gao. Efficiently comparing face images using a modified Hausdorff distance. *IEE Proceedings-Vision, Image and Signal Processing*, 150(6):346–350, 2003.

[81] John Salmon. *Parallel Hierarchical N-Body Methods*. PhD thesis, California Institute of Technology, 1990.

[82] Jaswinder Pal Singh. *Parallel Hierarchical N-body Methods and Their Implications for Multiprocessors*. PhD thesis, Stanford University, 1993. UMI Order No. GAX93-17818.

[83] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John Hennessy. Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, Fast Multipole, and Radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, June 1995.

[84] Michael Warren and John Salmon. A portable parallel particle program. *Computer Physics Communications*, 87:266–290, 1995.

[85] David Blackston and Torsten Suel. Highly portable and efficient implementations of parallel adaptive N-body methods. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, pages 1–20, 1997.

[86] David Blackston. *Pbody: A parallel N-body library*. PhD thesis, University of California, Berkeley, CA, USA, May 2000.

[87] Pangfeng Liu and Sandeep Bhatt. Experiences with Parallel N-Body Simulation. *IEEE Transactions on Parallel and Distributed Systems*, 11(12):1306–1323, December 2000.

[88] Shuji Ogata, Timothy Campbell, Rajiv Kalia, Aiichiro Nakano, Priya Vashishta, and Satyavani Vemparala. Scalable and portable implementation of the fast multipole method on parallel comptuers. *Computer Physics Communications*, 153(3):445–461, July 2003.

[89] Jakub Kurzak and Bernard Montgomery Pettitt. Massively parallel implementation of a fast multipole method for distributed memory machines. *Journal of Parallel and Distributed Computing*, 65:870–881, July 2005.

[90] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian Witten. The WEKA data mining software. *ACM SIGKDD explorations newsletter*, 11(1):10–18, November 2009.

[91] Sören Sonnenburg, Gunnar Rätsch, Sebastian Henschel, Christian Widmer, Jonas Behr, Alexander Zien, Fabio de Bona, Alexander Binder, Christian Gehl, and Vojtch Franc. The SHOGUN machine learning toolbox. *Journal of Machine Learning Research*, 11:1799–1802, August 2010.

[92] Davide Albanese, Roberto Visintainer, Stefano Merler, Samantha Riccadonna, Giuseppe Jurman, and Cesare Furlanello. Mlpy: Machine Learning Python. arXiv preprint arXiv:1202.6548, 2012.

[93] Dongryeol Lee, Andrew Moore, and Alexander Gray. Dual-Tree Fast Gauss Transforms. In *Advances in Neural Information Processing Systems*, pages 747–754. MIT Press, 2006.

[94] William March, Andrew Connolly, and Alexander Gray. Fast algorithms for comprehensive N-point correlation estimates. In *Proceedings of International Conference on Knowledge Discovery and Data Mining*, pages 1478–1486, 2012.

[95] Dongryeol Lee, Piyush Sao, Richard Vuduc, and Alexander Gray. A distributed kernel summation framework for general-dimension machine learning. *Statistical Analysis and Data Mining*, 7(1):1–13, 2014.

[96] Ryan Curtin, William March, Parikshit Ram, David Anderson, Alexander Gray, and Charles Isbell. Tree-independent dual-tree algorithms. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 1435–1443, May 2013.

[97] Andrew Moore and Mary Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, 8:67–91, 1998.

[98] Kan Deng and Andrew Moore. Multiresolution instance-based learning. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, volume 95, pages 1233–1239. Citeseer, 1995.

[99] Andrew Moore. Very fast EM-based mixture model clustering using multiresolution kd-trees. *Advances in Neural Information Processing Systems*, pages 543–549, 1999.

[100] Sunil Arya and David Mount. Algorithms for fast vector quantization. In *Proceeding of Data Compression Conference*, pages 381–390. IEEE Press, 1993.

[101] Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert Bocchino, Sarita Adve, and John Hart. Parallel SAH k-d tree construction. In *Proceedings of the Conference on High Performance Graphics*, pages 77–86. Eurographics Association, 2010.

[102] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time k-d tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)*, 27(5):126, 2008.

[103] Michael Warren and John Salmon. A parallel hashed octree N-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 12–21. ACM, 1993.

[104] Robert Blumofe, Christopher Joerg, Bradley Kuszmaul, Charles Leiserson, Keith Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[105] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *IEEE Real-Time Systems Symposium*, pages 259–268. IEEE, 2010.

[106] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013.

[107] Yizhuo Wang, Laleh Aghababaie Beni, Alexandru Nicolau, Alexander V Veidenbaum, and Rosario Cammarota. A compilation and run-time framework for maximizing performance of self-scheduling algorithms. In *IFIP International Conference on Network and Parallel Computing*, pages 459–470. Springer, 2014.

[108] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of openmp task scheduling strategies. In *International Workshop on OpenMP*, pages 100–110. Springer, 2008.

[109] Kevin Bache and Moshe Lichman. UCI Machine Learning Repository. University of California, Irvine, School of Information and Computer Sciences, 2013.

[110] Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. General transformations for GPU execution of tree traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 10. ACM, 2013.

[111] Christian Terboven, Dirk Schmidl, Tim Cramer, and Dieter Mey. Task-Parallel Programming on NUMA Architectures. In *European Conference on Parallel Processing*, pages 638–649. Springer, 2012.

[112] Eduard Ayguadé, Alejandro Duran, Jay Hoeflinger, Federico Massaioli, and Xavier Teruel. An experimental evaluation of the new OpenMP tasking model. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 63–77. Springer, 2007.

[113] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. ForestGOMP: an efficient OpenMP environment for NUMA architectures. *International Journal of Parallel Programming*, 38(5):418–439, 2010.

[114] Stephen Olivier, Allan Porterfield, Kyle Wheeler, and Jan Prins. Scheduling task parallelism on multi-socket multicore systems. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, pages 49–56. ACM, 2011.

[115] Rosario Cammarota, Laleh Aghababaie Beni, Alexandru Nicolau, and Alexander V Veidenbaum. Effective evaluation of multi-core based systems. In *International Symposium on Parallel and Distributed Computing*, pages 19–25. IEEE, 2013.

[116] Laleh Aghababaie Beni, Saikiran Ramanan, and Aparna Chandramowlishwaran. Portal: A high-performance language and compiler for parallel n-body problems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 984–995. IEEE, 2019.

[117] David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[118] Ken Kennedy and John R Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[119] Rosario Cammarota, Laleh Aghababaie Beni, Alexandru Nicolau, and Alexander V Veidenbaum. Optimizing program performance via similarity, using a feature-agnostic approach. In *International Workshop on Advanced Parallel Processing Technologies*, pages 199–213. Springer, 2013.

[120] Mary W Hall, Jennifer M Anderson, Saman P. Amarasinghe, Brian R Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.

[121] Tian Zhao and Xiaobing Huang. Design and implementation of DeepDSL: A DSL for deep learning. *Computer Languages, Systems & Structures*, 54:39–70, 2018.

[122] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. Diesel: DSL for linear algebra and neural net computations on GPUs. In *Proceedings of the 2nd International Workshop on Machine Learning and Programming Languages*, pages 42–51. ACM, 2018.

[123] Sandra Macià, Sergi Mateo, Pedro J Martínez-Ferrer, Vicenç Beltran, Daniel Mira, and Eduard Ayguadé. Saiph: Towards a DSL for High-Performance Computational Fluid Dynamics. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, page 6. ACM, 2018.

[124] Michael Driscoll, Benjamin Brock, Frank Ong, Jonathan Tamir, Hsiou-Yuan Liu, Michael Lustig, Armando Fox, and Katherine Yelick. Indigo: A Domain-Specific Language for Fast, Portable Image Reconstruction. In *International Parallel and Distributed Processing Symposium*, pages 495–504. IEEE, 2018.

[125] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th Conference on Programming Language Design and Implementation*, pages 519–530, 2013.

[126] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, pages 609–616, 2011.

[127] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[128] Laleh Aghababaie Beni and Aparna Chandramowlishwaran. PASCAL: A Parallel Algorithmic SCALable Framework for N-body Problems. In *European Conference on Parallel Processing*, pages 482–496. Springer, 2017.

[129] Chris Lomont. Fast inverse square root. Technical report, Indiana: Purdue University, 2003.

[130] Measuring and improving network performance, an analysis of network and application research, testing and optimization. Technical report, Ericsson, Facebook and XL Axiata, October 13, 2014.

[131] Dayong Wang, Charles Otto, and Anil Jain. Face search at scale. *IEEE transactions on pattern analysis and machine intelligence*, 39(6):1122–1136, 2016.

[132] Shiv Ram Dubey and Snehasis Mukherjee. A multi-face challenging dataset for robust face recognition. In *International Conference on Control, Automation, Robotics and Vision*, pages 168–173. IEEE, 2018.

[133] Chunhui Zhu, Fang Wen, and Jian Sun. A rank-order distance based clustering algorithm for face tagging. In *Computer Vision and Pattern Recognition*, pages 481–488. IEEE, 2011.

[134] Gary Huang, Marwan Mattar, Tamara Berg, and Eric Learned-Miller. Labeled faces in the wild: A database forstudying face recognition in unconstrained environments. 2008.

[135] Jeffrey Ho, Ming-Hsuan Yang, Jongwoo Lim, Kuang-Chih Lee, and David Kriegman. Clustering appearances of objects under varying illumination conditions. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 11–18, 2003.

[136] Anil Jain, Narasimha Murty, and Patrick Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

[137] Anil Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[138] Jun Luo, Yong Ma, Erina Takikawa, Shihong Lao, Masato Kawade, and Bao-Liang Lu. Person-specific SIFT features for face recognition. In *International Conference on Acoustics, Speech and Signal Processing*, volume 2, pages II–593. IEEE, 2007.

[139] Yi Sun, Xiaogang Wang, and Xiaoou Tang. Deep learning face representation from predicting 10,000 classes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1891–1898, 2014.

[140] Hieu Nguyen and Li Bai. Cosine similarity metric learning for face verification. In *Asian conference on computer vision*, pages 709–720. Springer, 2010.

[141] Manesh Kokare, Biswa Chatterji, and Prabir Biswas. Comparison of similarity metrics for texture image retrieval. In *Conference on convergent technologies for Asia-Pacific region*, volume 2, pages 571–575. IEEE, 2003.

[142] Jun Yu, Richang Hong, Meng Wang, and Jane You. Image clustering based on sparse patch alignment framework. *Pattern Recognition*, 47(11):3512–3519, 2014.

[143] Arputharaj Kannan, Krishna Mohan, and Neelamegam Anbazhagan. Image clustering and retrieval using image mining techniques. In *international conference on computational intelligence and computing research*, volume 2, 2010.

[144] Roberto Caldelli, Irene Amerini, Francesco Picchioni, and Matteo Innocenti. Fast image clustering of unknown source images. In *International Workshop on Information Forensics and Security*, pages 1–5. IEEE, 2010.

[145] Edward Kim, Hongsheng Li, and Xiaolei Huang. A hierarchical image clustering cosegmentation framework. In *Conference on Computer Vision and Pattern Recognition*, pages 686–693. IEEE, 2012.

[146] Jun Jie Foo, Justin Zobel, and Ranjan Sinha. Clustering near-duplicate images in large collections. In *Proceedings of the international workshop on Workshop on multimedia information retrieval*, pages 21–30. ACM, 2007.

[147] Yunchao Gong, Marcin Pawlowski, Fei Yang, Louis Brandy, Lubomir Bourdev, and Rob Fergus. Web scale photo hash clustering on a single machine. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 19–27, 2015.

[148] Pierre-Alain Moëllic, Jean-Emmanuel Haugeard, and Guillaume Pitel. Image clustering based on a shared nearest neighbors approach for tagged collections. In *Proceedings of the international conference on Content-based image and video retrieval*, pages 269–278. ACM, 2008.

[149] Jingyu Cui, Fang Wen, Rong Xiao, Yuandong Tian, and Xiaoou Tang. Easyalbum: an interactive photo annotation system based on face clustering and re-ranking. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 367–376. ACM, 2007.

[150] Yuandong Tian, Wei Liu, Rong Xiao, Fang Wen, and Xiaoou Tang. A face annotation framework with partial clustering and interactive labeling. In *Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2007.

[151] Ming Zhao, Yong Wei Teo, Siliang Liu, Tat-Seng Chua, and Ramesh Jain. Automatic person annotation of family photo album. In *International Conference on Image and Video Retrieval*, pages 163–172. Springer, 2006.

[152] Charles Otto, Brendan Klare, and Anil Jain. An efficient approach for clustering face images. In *International Conference on Biometrics (ICB)*, pages 243–250. IEEE, 2015.

[153] Vahid Kazemi and Josephine Sullivan. One millisecond face alignment with an ensemble of regression trees. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1867–1874, 2014.

[154] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[155] William Day and Herbert Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of classification*, 1(1):7–24, 1984.

[156] Chanop Silpa-Anan and Richard Hartley. Optimised k-d trees for fast image descriptor matching. In *Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2008.

[157] Martin Burtscher and Keshav Pingali. An efficient CUDA implementation of the tree-based Barnes Hut N-body algorithm. In *GPU computing Gems Emerald edition*, pages 75–92. Elsevier, 2011.

[158] Feng Zhang, Peng Di, Hao Zhou, Xiangke Liao, and Jingling Xue. RegTT: Accelerating tree traversals on GPUs by exploiting regularities. In *45th International Conference on Parallel Processing (ICPP)*, pages 562–571. IEEE, 2016.

[159] Jianqiao Liu, Nikhil Hegde, and Milind Kulkarni. Hybrid CPU-GPU scheduling and execution of tree traversals. In *Proceedings of the International Conference on Supercomputing*, page 2. ACM, 2016.