A Distributed Kernel Summation Framework for General-Dimension Machine Learning

Dongryeol Lee* Richard Vuduc† Alexander G. Gray‡

Abstract

Kernel summations are a ubiquitous key computational bottleneck in many data analysis methods. In this paper, we attempt to marry, for the first time, the best relevant techniques in parallel computing, where kernel summations are in low dimensions, with the best general-dimension algorithms from the machine learning literature. We provide the first distributed implementation of kernel summation framework that can utilize: 1) various types of deterministic and probabilistic approximations that may be suitable for low and high-dimensional problems with a large number of data points; 2) any multi-dimensional binary tree using both distributed memory and shared memory parallelism; 3) a dynamic load balancing scheme to adjust work imbalances during the computation. Our hybrid MPI/OpenMP codebase has wide applicability in providing a general framework to accelerate the computation of many popular machine learning methods. Our experiments show scalability results for kernel density estimation on a synthetic ten-dimensional dataset containing over one billion points and a subset of the Sloan Digital Sky Survey Data up to 6,144 cores.

1 Introduction

Kernel summations occur ubiquitously in both old and new machine learning algorithms, including kernel density estimation [31], kernel regression [26], Gaussian process regression [33], kernel PCA [39], and kernel support vector machines (SVM) [38]. In these methods, we are given a set of reference/training points $r_i \in \mathbb{R}^D$, $R = \left[r_1, \cdots, r_{|R|}\right]$ and their weights $W = \left[w_1, \cdots, w_{|R|}\right]$ and a set of query/test points $q_j \in \mathbb{R}^D$, $Q = \left[q_1, \cdots, q_{|Q|}\right]$ (analogous to the source points and the target points in FMM literature). We consider the problem of rapidly evaluating, for each $q \in Q$, sums of the form :

$$f(q; R) = \sum_{i=1}^{|R|} w_i k(q, r_i) \tag{1.1}$$

where $k(\cdot, \cdot) : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ is the given kernel.

Method	$k(\cdot, \cdot)$	Train/Batch test
KDE [31]/NWR [26]	PDFs	✔ / ✔
KSVM [38]/GPR [33]	PD kernels	✘ / ✔
KPCA [39]	CPD kernels	✘ / ✔

Table 1: Methods that can be sped up using our framework. Although the parts marked with ✘ can be sped up in some cases by sparsifying the kernel matrix and applying Krylov-subspace methods, computed results are usually numerically unstable.

In this paper, we consider the setting of evaluating $f(q; R)$ on a distributed set of training points/test points. Data may be distributed because: 1) it is more cost-effective to distribute data on a network of less powerful nodes than storing everything on one powerful node; 2) it allows distributed query processing for high scalability. Each process (which may/may not be on the same node) owns a subset of R and Q and needs to initiate communications (i.e. MPI, memory-mapped files) when it needs a remote piece of data owned by another process. Cross-validation in all of the methods above require evaluating Equation 1.1 for multiple parameter values, yielding $\mathcal{O}(D|Q||R|)$ cost. Especially, $|Q|$ and $|R|$ can be prohibitively large so that one CPU cannot handle the computation in a tractable amount of time. Unlike the usual $3-D$ setting in N-body simulations, D may be as high as 1000 in many kernel methods. This paper attempts to provide a general framework that encompasses acceleration techniques for a wide range of both low-dimensional and high-dimensional problems with a large number of data points.

Shared/distributed memory parallelism. Achieving scalability in distributed setting requires: 1) minimizing inherently serial portions of the algorithm (Amdahl's law); 2) minimizing the time spent in critical sections; 3) overlapping communication and computation as much as possible. To achieve this goal, we utilize OpenMP for shared-memory parallelism and MPI for distributed-memory parallelism in a hybrid MPI/OpenMP framework. Kernel summation can be

*Georgia Institute of Technology. **dongryel@cc.gatech.edu**
†Georgia Institute of Technology. **richie@cc.gatech.edu**
‡Georgia Institute of Technology. **agray@cc.gatech.edu**

Approximation	Type	Basis functions	Applicability
Series expansion [21, 19]	Deterministic	Taylor basis	General
Reduced set [38]	Deterministic	Pseudo-particles	Low-rank PD/CPD kernels
Monte Carlo [16, 20]	Probabilistic	None	General smooth kernels
Random feature extraction [32]	Probabilistic	Fourier basis	Low-rank PD/CPD kernels

Table 2: Examples of approximation schemes that can be utilized in our framework.

Tree type	Bound type	RULE(x)
kd-trees [5]	hyper-rectangle $\{b_{d,\min}, b_{d,\max}\}_{d=1}^{D}$	$x_i \leq s_i$ for $1 \leq i \leq D$, $b_{d,\min} \leq s_i \leq b_{d,\max}$
metric trees [27]	hyper-sphere $B(c,r)$, $c \in \mathbb{R}^D, r > 0$	$\|\|x - p_{left}\|\| < \|\|x - p_{right}\|\|$ for $p_{left}, p_{right} \in \mathbb{R}^D$
vp-trees [46]	$B(c,r_1) \cap B(c,r_2)$ for $0 \leq r_1 < r_2$	$\|\|x - p\|\| < t$ for $t > 0, p \in \mathbb{R}^D$
RP-trees [12]	Hyperplane $a^T x = b$	$x^T v \leq \text{MEDIAN}(z^T v : z \in S)$

Table 3: Examples of multi-dimensional binary trees that can be utilized in our framework. If RULE(x) returns true, then x is assigned to the left child (as defined in [12]).

parallelized because each $f(q;R)$ can be computed in parallel. In practice, Q is partitioned into a pairwise disjoint set of points $Q = \bigcup_{i=1}^{t} Q_i$ and a set of batch sums for each Q_i proceeds in parallel. We use a query subtree as Q_i (see Figure 6) since their spatial proximity makes it more efficient to be processed as a group.

1.1 Our Contributions In this paper, we attempt to marry, for the first time, the best relevant techniques in parallel computing, where kernel summations are in low dimensions, with the best general-dimension algorithms from the machine learning literature. We provide a unified, efficient parallel kernel summation framework that can utilize: 1) various types of deterministic and probabilistic approximations (Table 2) that may be suitable for both low and high-dimensional problems with a large number of data points; 2) any multi-dimensional binary tree using both distributed memory (MPI) and shared memory (OpenMP) parallelism (Table 3 lists some examples); 3) a dynamic load balancing scheme to adjust work imbalances during the computation. Our framework provides a general approach for accelerating the computation of many popular machine learning methods (see Table 1). Our motivation is similar to that of [22], where a general framework was developed to support various types of scientific simulations, and is based on parallelization of the dual-tree method [13].

Outline of this paper. In Section 3, we show how to exploit distributed/shared memory parallelism in building distributed multidimensional trees. In Section 4, we describe the overall algorithm and the parallelism involved. In Section 4.2, we describe how we exchange messages among different processes using the recursive doubling scheme; during this process, we touch briefly upon a problem of distributed termination detection. In Section 4.3, we discuss our static and dynamic load balancing schemes. In Section 5, we demonstrate the scalability of our framework on kernel density estimation on both synthetically generated dataset and a subset of SDSS dataset [48]. In Section 6, we discuss some limitations and planned extensions.

Terminology. An *MPI communicator* connects a set of MPI processes, each of which is given a unique identifier called an *MPI rank*, in an ordered *topology*. Commonly used topologies include: the ring topology, the star topology, and the hypercube topology. We denote C_{world} as the MPI communicator over all MPI processes, and D_P the portion of the data D owned by the P-th process. In this paper, we assume that: 1) the nodes are connected using a hypercube topology since it is the most commonly used one; 2) there are p_{thread} threads associated with each MPI process; 3) the number of MPI processes p is a power of two, though our approach can be easily extended for arbitrary positive integers p; 4) the query set equals the reference set ($Q = R$, and we denote D as the common dataset and $N = |D|$ the size of the dataset), and D is equidistributed across all MPI processes. Particularly the monochromatic case of $Q = R$ occurs often in cross-validating for optimal parameters in many non-parametric methods.

2 Related Work

2.1 Error Bounds Many algorithms approximate the kernel sums at the expense of reduced precision. The following error bounding criteria are variously used in the literature:

DEFINITION 2.1. τ **absolute error bound:** *For each* $f(q_i;R)$ *for* $q_i \in Q$, *it computes* $\widetilde{f}(q_i;R)$ *such that* $\left|\widetilde{f}(q_i;R) - f(q_i;R)\right| \leq \tau$.

DEFINITION 2.2. ϵ **relative error bound:** *For each* $f(q_i;R)$ *for* $q_i \in Q$, *compute* $\widetilde{f}(q_i;R)$ *such that* $\left|\widetilde{f}(q_i;R) - f(q_i;R)\right| \leq \epsilon\,|f(q_i;R)|$.

Bounding the relative error is much harder because the error bound criterion is in terms of the initially unknown exact quantity. As a result, many previous methods [15, 45] have focused on bounding the absolute error. The relative error bound criterion is preferred to the absolute error bound criterion in statistical applications in which high accuracy is desired. Our framework can enforce the following error form:

DEFINITION 2.3. $(1-\alpha)$ **probabilistic** ϵ **relative/**τ **absolute error:** *For each* $f(q_i;R)$ *for* $q_i \in Q$, *compute* $\widetilde{f}(q_i;R)$, *such that with at least probability* $0 < 1-\alpha \leq 1$, $\left|\widetilde{f}(q_i;R) - f(q_i;R)\right| \leq \epsilon\,|f(q_i;R)| + \tau$.

2.2 Serial Approaches Fast algorithms for evaluating Equation (1.1) can be divided into two types: 1) reduced set methods from the physics/machine learning communities [38]; 2) hierarchical methods which employ spatial partitioning structures such as octrees, *kd*-trees [5], and cover-trees [6].

Reduced set methods. Reduced set methods express each data point as a linear combination of points (so called dictionary points each of which gives arise to the function $b : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$):

$$f(q;R) \approx f_{reduced}(q;R_{reduced}) = \sum_{d_k \in S} u_k b(q,d_k)$$

where $|R_{reduced}| \ll |R|$ and the resulting kernel sum can be evaluated more quickly. In the physics community, uniform grid points are chosen and points are projected on Fourier bases (i.e. $b(\cdot,\cdot)$ is the Fourier basis). Depending on how the particle-particle interactions are treated, a FFT-based summation method belongs to the category of Particle-Particle-Particle Mesh (P^3M) method or Particle-Mesh (PM) method. However, these methods do not scale beyond three dimensions due to uniform grids. Recently, machine learning practitioners have employed a variant of reduced set method that utilize positive-definiteness (or conditionally positive-definiteness) of the kernel function and successfully scaled many kernel methods such as SVM and GPR [44, 43, 28, 40]. However, these methods require optimizing the basis points given a pre-selected error criterion (i.e. on reconstruction error in the reproducing kernel Hilbert space or generalization error with/without regularization) and the resulting dictionary $R_{reduced}$ can be quite large in some cases.

Hierarchical methods. Most hierarchical methods

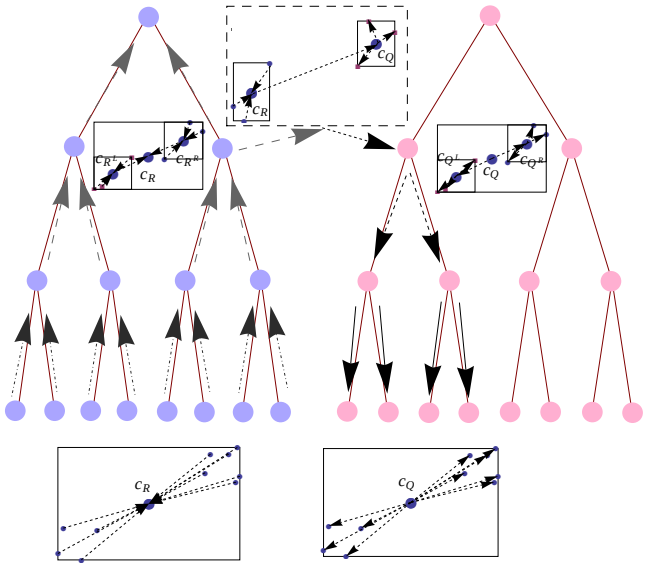

Figure 1: The reference points (the left tree) are hierarchically compressed and uncompressed when a pair of query (from the right tree)/reference nodes is approximated within an error tolerance.

ALGORITHM 2.1. DUALTREE(Q,R)
if CANSUMMARIZE(Q,R) **then**
  SUMMARIZE(Q,R)
else
  if Q is a leaf node and R is a leaf node **then**
    DUALTREEBASE(Q,R)
  else
    DUALTREE$\left(Q^L,R^L\right)$, DUALTREE$\left(Q^L,R^R\right)$
    DUALTREE$\left(Q^R,R^L\right)$, DUALTREE$\left(Q^R,R^R\right)$
  end if
end if

using trees utilize series expansions (see Figure 1). The pseudocode for a *dual-tree* method [13] that subsumes most of hierarchical methods is shown in Algorithm 2.1. The first expansion called the far-field expansion summarizes the contribution of R_{sub} for a given query q:

$$\begin{aligned} f(q;R_{sub}) &= \sum_{r_{j_n} \in R_{sub}} w_{j_n} k(q,r_{j_n}) \\ &= \sum_{r_{j_n} \in R_{sub}} w_{j_n} \sum_{m=1}^{\infty} b_m \phi_m(q,R_{sub}) \psi_m(r_{j_n},R_{sub}) \\ &= \sum_{m=1}^{\infty} \phi_m(q,R_{sub}) \left(\sum_{r_{j_n} \in R_{sub}} b_m w_{j_n} \psi_m(r_{j_n},R_{sub}) \right) \\ &= \sum_{m=1}^{\infty} \phi_m(q,R_{sub}) M_m(R_{sub}) \end{aligned}$$

where ϕ_m's and ψ_m's show dependence on the subset R_{sub}. The second type called the local expansion for $q \in Q_{sub} \subset Q$ expresses the contribution of R_{sub} near q:

$$
\begin{aligned}
f(q; R_{sub}) &= \sum_{r_{j_n} \in R_{sub}} w_{j_n} k(q, r_{j_n}) \\
&= \sum_{r_{j_n} \in R_{sub}} w_{j_n} \sum_{m=1}^{\infty} g_m \phi_m(r_{j_n}, Q_{sub}) \psi_m(q, Q_{sub}) \\
&= \sum_{m=1}^{\infty} \psi_m(q, Q_{sub}) \left(\sum_{r_{j_n} \in R_{sub}} g_m w_{j_n} \phi_m(r_{j_n}, Q_{sub}) \right) \\
&= \sum_{m=1}^{\infty} \psi_m(q, Q_{sub}) L_m(R, Q_{sub})
\end{aligned}
$$

Both representation are truncated at a finite number of terms depending on the level of prescribed accuracy, achieving $\mathcal{O}(|Q| \log |R|)$ runtime in most cases. To achieve $\mathcal{O}(|Q| + |R|)$ runtime, we require an efficient linear operator that converts $M_m(R)$ into $L_m(R, Q)$'s. Depending on the basis representations of ϕ's and ψ's, the far-to-local linear operator is diagonal and the translation is linear in the number of coefficients. There are many serial algorithms [3, 4, 14, 15, 8, 13, 47] that use different series expansions forms to bound error deterministically. [16] proposes a probabilistic approximation scheme based on the central limit theorem, and [20] used both deterministic and probabilistic approximations. Especially, probabilistic approximations can help overcome the *curse of dimensionality* at the expense of indeterminism in approximated kernel sums.

In this paper, we focus on hierarchical methods because: 1) it is a natural framework to control approximation in a varying degree of resolution; 2) the specialized acceleration techniques for positive-definite kernels can be plugged in as a special case. We would like to point out that the code base can also be used in scientific N-body simulations [22] but we will defer its applications in a future paper.

2.3 Parallelizations Hierarchical N-body methods present an interesting challenge in parallelization: 1) both data distribution and work distribution are highly non-uniform across MPI processes; 2) often involves long-range communication due to the kernel function $k(\cdot, \cdot)$. In the worst case, every process will need almost every piece of data owned by the other processes. Here we discuss the three main important issues in a scalable distributed hierarchical N-body code:

Parallel tree building: [18] proposed a novel distributed octree construction algorithm and a new reduction algorithm for evaluation to scale up to over 65K cores. [2] describes a parallel *kd*-tree construction on a distributed memory setting, while [9] works on a shared-memory setting. [23] discuss building spill-trees, a variant of metric trees that permit overlapping of data between two branches, using the map-reduce framework.

Load balancing: Most common static load balancing algorithms include: 1) the costzone [42] which partitions a pre-built query tree and assigns each query particle to a zone. A common approach employs a graph partitioner [11]; 2) the ORB (orthogonal recursive bisection) which directly partitions each dimension of the space containing the query points in a cyclic fashion. Dynamic load balancing [24] strategies adjust the imbalance between the work loads during the computation.

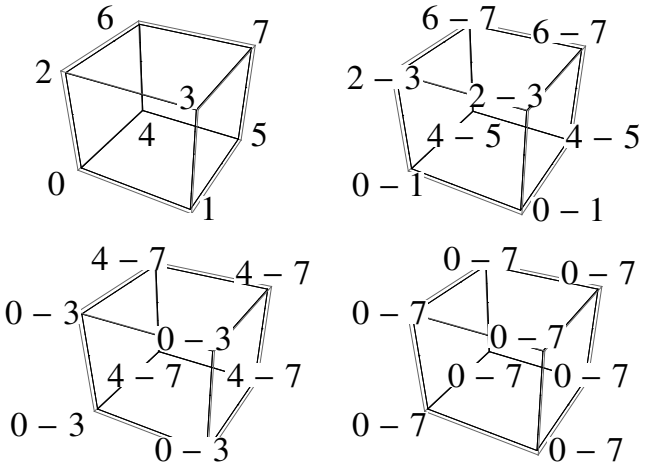

Figure 2: Recursive doubling on the hypercube topology. Initially, each node begins with its own message (top left). The exchanges proceed in: the top right, the bottom left, then bottom right in order. Note that the amount of data exchanged in each stage doubles.

Interprocess communication: The local essential trees approach [35] (which involves few large-grained communication) is a sender-initiated communication approach. Using the ORB, each process sends out essential data that may be needed by the other processes using the recursive doubling scheme (see Figure 2). An alternative approach has the receiver initiate communication; this approach involves many fine-grained communication, and is preferable if interprocess communication overheads are small. For more details, see [41].

3 Distributed Multidimensional Tree

Our approach for building a *general-dimension* distributed tree closely follows [2]. Following the ORB (orthogonal recursive bisection) in [35], we define the *global tree*, which is a hierarchical decomposition of the data points on the process level. The *local tree* of each process is built on its own local data D_P.

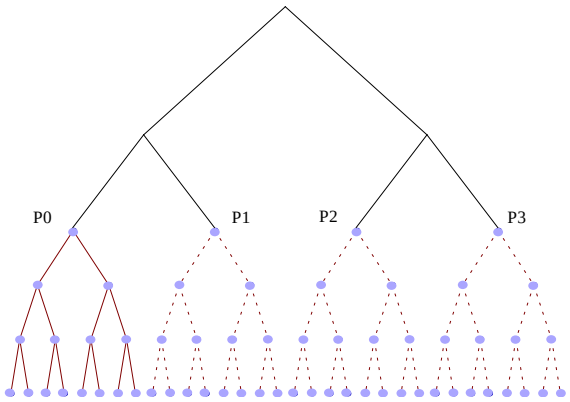

Figure 3: Each process owns the global tree of processes (the top part) and its own local tree (the bottom part).

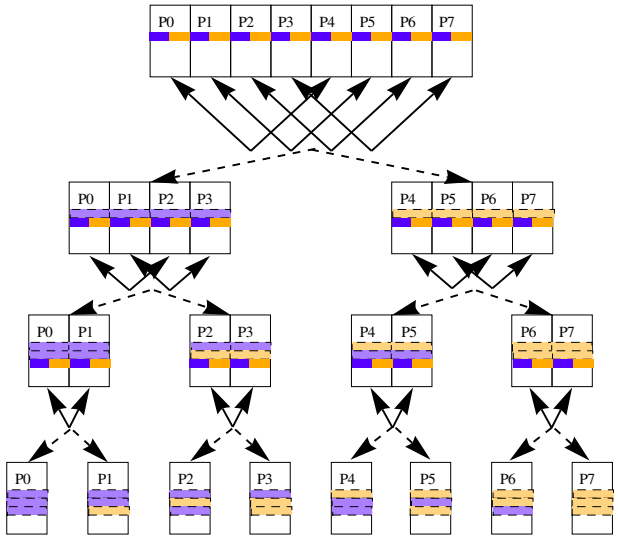

Figure 4: Distributed memory parallelism in building the global tree (the first $\log p$ levels of the entire tree). Each solid arrow indicates a data exchange between two given processes. After exchanges on each level, the MPI communicator is split (shown as a dashed arrow) and the construction works in parallel subsequently.

Building the distributed tree. Initially, all MPI processes in a common MPI communicator agree on a rule for partitioning each of its data into two parts (see Figure 1). The MPI communicator is then split in two depending on the MPI process rank. This process is recursively repeated until there are $\log p$ levels in the global tree. Shared-memory parallelism can be utilized in the (independent) reduction step in each MPI process in generating the split rule (see Figure 1). Depending on a split rule and using C++ meta-programming, we can auto-generate any binary tree (see Table 3) utilizing an associative reduction operator for constructing bounding primitives. Generalizing to multidimensional trees with an arbitrary number of child nodes (such as cover-trees [6]) is left as a future work.

Building the local tree. Here we closely follow the approach in [9]. The first few levels of the tree are built in a breadth-first manner with the assigned number of OpenMP threads proportional to the number of points participating in a reduction to form the bounding primitive (see Figure 5). The number of participating OpenMP threads per task halves as we descend each level. Each independent task with only one assigned OpenMP thread proceeds with the construction in a depth-first manner. We utilized the nested loop parallelization feature in OpenMP for this part.

Overall runtime complexity. All-reduce operation on the hypercube topology takes $\mathcal{O}\left(t_s \log p + t_w m(p-1)\right)$ where t_s, t_w, and m are the latency constant, the bandwidth constant, and the message size respectively. Assume that each process starts with the same number of points $\frac{N}{p}$ and each split on a global/local level results in equidistribution of points and only distributed memory parallelism is used (i.e. $p_{thread} = 1$). Let m_{bound} be the message size of the bounding primitive divided by D. The overall runtime for each MPI process is:

- The reduction cost and the split cost at each level $0 \leq i < \log p$: $\mathcal{O}\left(\frac{2N(D+t_w)}{p}\right)$
- The all-reduce cost on each level $0 \leq i < \log p$: $\mathcal{O}(t_w D m_{bound}\left(\frac{p}{2^i} - 1\right))$
- The total latency cost at each level $0 \leq i < \log p$: $\mathcal{O}\left(t_s\left(\log \frac{p}{2^i} + 1\right)\right)$.
- The base case at the level $\log p$ (the depth-first build of local tree): $\mathcal{O}\left(\frac{DN}{p} \log\left(\frac{N}{p}\right)\right)$

Therefore, the overall complexity is: $\mathcal{O}\left(\frac{DN}{p} \log\left(\frac{N}{p}\right)\right) + \mathcal{O}\left(Dt_w m_{bound}\left(2p - \log 4p\right)\right) + \mathcal{O}\left(\frac{2N(D+t_w)}{p} \log p\right) + \mathcal{O}\left(\frac{t_s}{2} \log p\left(\log p + 3\right)\right)$. This implies that the growth of the number of data points must be $N \log N \sim \mathcal{O}(p^2)$ to achieve the same level of parallel efficiency. Note that the last terms have zero contribution if $p = 1$.

4 Overall Algorithm

Algorithm 4.1 shows the overall algorithm. Initially, each MPI process initializes its distributed task queue by dividing its own local query subtree into a set of T query grain subtrees where $T > p_{thread}$ is more than the number of threads p_{thread} running on each MPI process;

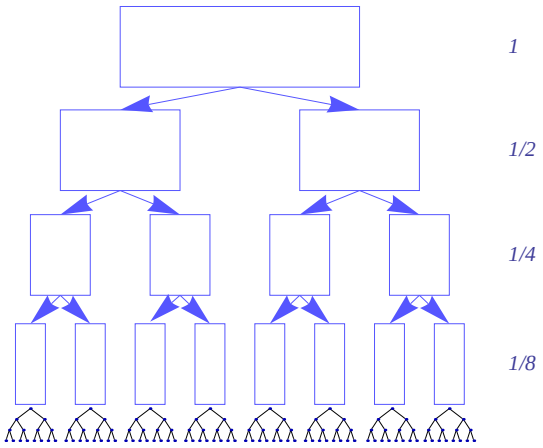

Figure 5: Shared-memory parallelism in building the local tree for each MPI process. The first top levels are built in a breadth-first manner with the number of threads proportional to the amount of performed reduction. Any task with one assigned thread proceeds in a depth-first manner.

initially each of these trees has no tasks. The tree walker object maintains a stack of pairs of Q and R_P that must be considered. It is first initialized with the following tuple: the root node of Q, the root node of the local reference tree R_P, and the probability guarantee α; the relative error tolerance/absolute error tolerance are global constants ϵ and τ respectively. Threads not involved with the tree walk or exchanging data can dequeue tasks from the local task queue.

4.1 Walking the Trees Each MPI process takes the root node of the global query tree (the left tree) and the root node of its local reference tree (the right tree) and

ALGORITHM 3.1. BUILDDISTTREE(C_{world}, D_P): (**MPI**)
$C \leftarrow C_{world}$
while $C.size() > 1$ **do**
  $rule \leftarrow$ CHOOSESPLITRULE(C, D_P)
  for each P-th MPI process in C in parallel **do**
    Divide $D_P = L_P \cup R_P$ using $rule$.
    if $P < \frac{|C|}{2}$ **then**
      $P_{comp} \leftarrow P + \frac{|C|}{2}$, SEND($P_{comp}, R_P$)
      $L_{comp} \leftarrow$ RECEIVE(P_{comp}), $D_p \leftarrow L_p \cup L_{comp}$
    else
      $P_{comp} \leftarrow P - \frac{|C|}{2}$, $R_{comp} \leftarrow$ RECEIVE(P_{comp})
      SEND(P_{comp}, L_P), $D_p \leftarrow R_p \cup R_{comp}$
    end if
    $C \leftarrow$ SPLITCOMM($P >= \frac{|C|}{2}$)
  end for
end while
BUILDLOCALTREE($\mathcal{D}_P$)

ALGORITHM 3.2. CHOOSESPLITRULE(W, D_P): (**OpenMP**)
$b_{local} \leftarrow$ an empty bound
for each data point $r \in D_P$ in parallel **do**
  Expand b_{local} to include r.
end for
$b_{common} \leftarrow$ COMBINE(W, b_{local})
return RULE(x) using b_{common}.

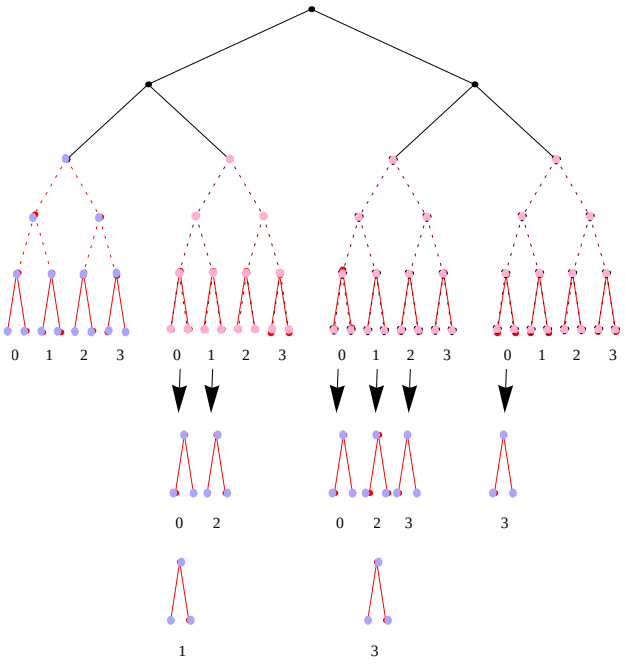

Figure 6: The global query tree is divided into a set of query subtrees each of which can queue up a set of reference subset to compute (shown vertically below each query subtree). The kernel summations for each query subtree can proceed in parallel.

performs a dual-tree recursion (see Algorithm 4.2). For simplicity, we show the case where the reference side is descended first then the query side. Any of the running threads can walk by dequeuing from the stack of frontier nodes, generate local tasks, and queue up reference subtrees to send to other processes. The expansion can be prioritized using the HEURISTIC function that takes a pair of query/reference nodes. It would be possible to extend the walking procedure to include fancier expansion patterns described in [34].

4.2 Message Passing Inspired by the local essential trees approach, we develop a message passing system utilizing the *recursive doubling scheme*. We assume that the master thread is the only thread that may initiate MPI calls. The key differences from the vanilla local essential tree approach are two-fold: 1)

ALGORITHM 4.1. Overall algorithm.

Each MPI process initializes its distributed task queue with a set of query grain subtrees and the tree walker with $(Q, R_P, \epsilon, \tau, \alpha)$.
OpenMP parallel region start (threads spawned)
while there are remaining tasks globally **do**
  if I am the master thread **then**
    Route messages via recursive doubling.
  end if
  if If the task queue is nearly empty **then**
    WALK() (Algorithm 4.2,Figure7).
  end if
  Choose a query subtree and lock it. Dequeue a set of task from it and call the serial algorithm (Algorithm 2.1) on each (Q_{sub}, R_{sub}) pair. For each completed task, queue up completed work quantity.

  Unlock the query subtree. If the checked out query subtree is imported from another process and has no more tasks, queue up a flush request to write back the query subtree.
end while
OpenMP parallel region end (threads synchronized)

ALGORITHM 4.2. WALK()

while there is a MPI process asking for work **do**
  $(Q_{sub}, R_{sub}, \alpha_{sub}) \leftarrow$ POP()
  if CANSUMMARIZE$(Q_{sub}, R_{sub}, \epsilon, \tau, \alpha_{sub})$ **then**
    SUMMARIZE$(Q_{sub}, R_{sub}, \epsilon, \tau, \alpha_{sub})$, Queue the work-complete message $(|Q_{sub}||R_{sub}|, \{1, \cdots, P-1, P+1, \cdots, p\})$.
  else
    if Q_{sub} is a root node of a query grain subtree **then**
      if R_{sub} is a leaf node **then**
        if Q_{sub} belongs to the self, **then**
          Add (Q_{sub}, R_{sub}) to the task list of Q_{sub}.
        else
          Add the MPI rank of Q_{sub} to a list of R_{sub}'s destinations.
        end if
      else
        $(R_1, R_2) \leftarrow$ HEURISTIC$(Q_{sub}, R^L_{sub}, R^R_{sub})$
        PUSH $\left(Q_{sub}, R_2, \frac{\alpha_{sub}}{2}\right)$, PUSH $\left(Q_{sub}, R_1, \frac{\alpha_{sub}}{2}\right)$
      end if
    else
      if R_{sub} is a leaf node **then**
        PUSH$(Q^L_{sub}, R_{sub}, \alpha_{sub})$, PUSH$(Q^R_{sub}, R_{sub}, \alpha_{sub})$
      else
        $(R_{L,1}, R_{L,2}) \leftarrow$ HEURISTIC$(Q^L_{sub}, R^L_{sub}, R^R_{sub})$
        $(R_{R,1}, R_{R,2}) \leftarrow$ HEURISTIC$(Q^R_{sub}, R^L_{sub}, R^R_{sub})$
        PUSH $\left(Q^R_{sub}, R_{R,2}, \frac{\alpha_{sub}}{2}\right)$, PUSH $\left(Q^L_{sub}, R_{L,2}, \frac{\alpha_{sub}}{2}\right)$
        PUSH $\left(Q^R_{sub}, R_{R,1}, \frac{\alpha_{sub}}{2}\right)$, PUSH $\left(Q^L_{sub}, R_{L,1}, \frac{\alpha_{sub}}{2}\right)$
      end if
    end if
  end if
end while

our framework can support computations that have dynamic work requirement, unlike FMM; 2) our framework does not require each MPI process to accommodate all of the non-local data in its essential tree. Algorithm 4.3 shows the message passing routine called by the master threshold on each MPI process. Any message from a pair of processes in a hypercube topology needs at most $\log p$ rounds of routing. At each stage i, the process P with binary representation $P = (b_{\log p-1}, \cdots, b_{i+1}, 0, b_{i-1}, \cdots, b_0)_2$ sends messages to process $P_{neighbor} = (b_{\log p-1}, \cdots, b_{i+1}, 1, b_{i-1}, \cdots, b_0)_2$ (and vice versa). Here are the types of messages exchanged between a pair of processes:

1. Reference subtrees: each MPI process sends out a reference subtree with the tag $(R_{sub}, \{Q_{sub}\})$ where $\{Q_{sub}\}$ is the list of remote query subtrees that needs R_{sub}.
2. Work-complete message: whenever each thread finishes computing a task (Q_{sub}, R_{sub}), it queues up a pair of completed work quantity and the list of all MPI ranks excluding the self. The form of the message is: $(|Q_{sub}||R_{sub}|, \{0, \cdots, P-1, P+1, p-1\}))$.
3. Extra tasks: one of the paired MPI processes can donate some of its tasks to the other (Section 4.3). This has a form of $(Q_{sub}, \{R_{sub}\})$ where $\{R_{sub}\}$ is a list of reference subsets that must be computed for Q_{sub}.
4. Imported query subtree flushes: during load balancing, query subtrees with several reference tasks may be imported from another process. These must be synchronized with the original query subtree on its originating process before tasks associated with it are dequeued.
5. The current load: the load is defined as the sum of $|Q_{sub}R_{sub}|$ associated with all query subtrees (both native and imported) on a given process.

Distributed termination detection. We follow a similar idea discussed in Section 14.7.4 of [30], Initially, all MPI processes collectively have to complete $|Q||R|$ amount of work. Each thread dequeues a work and completes a portion of its assigned local work (see Figure 6); the completed work quantity is then broadcast using the recursive doubling message passing to all the other processes. The completed and uncompleted work is conserved at any given point of time. When every process thinks all of $|Q||R|$ work have been completed and it has sent out all of its queued up work-complete messages, it can safely terminate.

4.3 Load Balancing Our framework employs both static load balancing and dynamic load balancing.

Static load balancing. Each MPI task is initially in charge of computing the kernel sums for all of its grain query subtrees. This approach is similar to the ORB

ALGORITHM 4.3. RouteMessage

$P_{neighbor} \leftarrow P$ **XOR** *stage*.

Asynchronously send to $P_{neighbor}$:

1. A set of query subtree flushes
2. A set of query subtrees with tasks
3. The work-complete messages
4. The recently received load estimates of other processes.

From $P_{neighbor}$, receive:

1. A set of query subtree flushes from $P_{neighbor}$. Synchronize those that belong to P.
2. Query subtrees with tasks from $P_{neighbor}$ and have the local task queue import them.
3. Load estimates of other processes from $P_{neighbor}$.
4. Work complete messages from $P_{neighbor}$ and update the global work count.

Wait until all sends are complete.

$stage \leftarrow (stage + 1)$ **mod** $\log p$

approach where the distributed tree determines the task distribution.

Dynamic load balancing. It is likely that the initial query subtree assignments will cause imbalance among processes. During the computation, we allow each query task to migrate from the current P-th process to a neighboring $P_{neighbor}$-th process. We use a very simple scheme in which two processes that are paired up during each stage of the repeated recursive doubling stages attempt to load balance. Each process keeps sending out a snapshot of its computation load in the recursive doubling scheme, and maintains a table of estimated remaining amount of computation on the other processes. Therefore, load estimates could be outdated by the time a given process considers transferring extra tasks. Therefore, we employ a simple heuristic of initiating the load balance for a pair of imbalanced processes: if the estimated load on the process $P_{neighbor}$ is below $0 < \beta_{threshold} < 1$ of the current load on the process P, transfer $0.5(1-\beta_{threshold})$ amount of tasks from P to $P_{neighbor}$.

5 Experimental Results

We developed our code base in C++ called MLPACK [7] and utilized open-source libraries such as Boost library [17], Armadillo linear algebra library [37], and the GNU Scientific Library [10]. We have tested on the Hopper cluster at NERSC. Each node on the Hopper cluster has 24 cores, and we used the recommended setting of 6 OpenMP threads/node ($p_{thread} = 6$) and a maximum 4 MPI tasks/node and compiled using GNU C++ compiler version 4.6.1 under the $-O3$ optimization flag. The configuration details are available at [1].

We chose to evaluate the scalability of our framework in the context of computing kernel density estimates [31]. We used the Epanechnikov kernel $k(q,r) = I\left(1 - \frac{||q-r||^2}{h^2}\right)$ since it is the most asymptotically optimal kernel. For the first part of our experiments, we considered uniformly distributed data points in the 10-dimensional hypercube $[0,1]^{10}$ since non-parametric methods such as KDE and NWR require an exorbitant number of samples in the uniform distribution case. Applying non-parametric methods for higher dimensional datasets requires exploiting correlations between dimensions [29]. For the second part, we measured the strong scalability of our implementation on the SDSS dataset. All timings are maximum ones across all processes.

5.1 Scalability of Distributed Tree Building

We have compared the strong scalability of building two main tree structures: *kd*-trees and metric-trees on an uniformly distributed 10-dimensional dataset containing 20,029,440 points (Figure 8). In all cases, building a metric-tree is more expensive than building a *kd*-tree; a reduction operation in Algorithm 3.2 for metric-trees involves distance computations whereas the reduction operator for *kd*-trees is the computation of minimum/maximum. For the weak-scaling result (shown in Figure 9), we added 166,912 ten-dimensional data points per core up to 1,025,507,328 points. Our analysis in Section 3 has shown that the exact distributed tree building algorithm require the growth of the data points to be $N \log N \sim O(p^2)$, and this is reflected in our experimental results.

However, readers should note that: 1) the depth of the trees built in our setting is much deeper than the ones in other papers [18]. Each leaf in our tree contains 40 points; 2) the tree building is empirically fast. On 6,144 cores, we were able to build a *kd*-tree on over one billion 10-dimensional data points under 30 seconds; 3) the one-time cost of building the distributed tree can be amortized over many queries.

[23] took a simple map-reduce approach in building a multidimensional binary tree (hybrid spill-trees specifically). We conjecture that this approach may be faster to build but result in slower query times due to generating suboptimal partitions. Future experiments will reveal its strengths and the weaknesses.

5.2 Scalability of Kernel Summation

In this experiment, we measure the scalability of the overall kernel summation. Our algorithm has three main parts: building the distributed tree (Algorithm 1), walking the tree to generate the tasks (Algorithm 4.2,Figure 7),

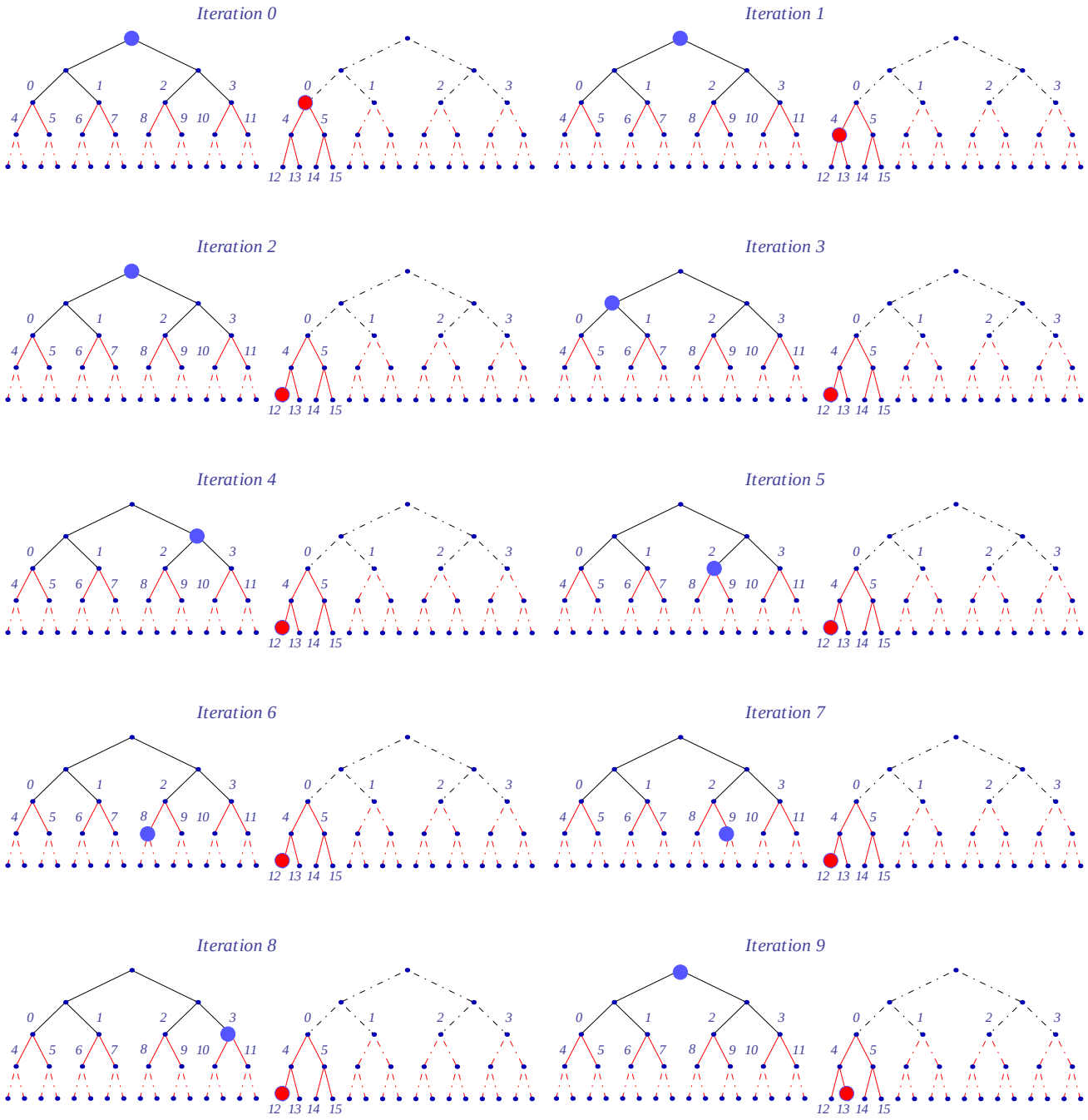

Figure 7: Illustration of the tree walk performed by the 0-th MPI process in a group of 4 MPI processes. Iteration 0: starting with the global query tree root and the root node of the local reference tree owned by the 0-th MPI process; Iteration 1-2: descend the reference side before expanding the query side; Iteration 3: the reference subtree 12 is pruned for the 0-th and 1st MPI processes; Iteration 6-7: the reference subtree 12 is hashed to the list of subtrees to be considered for the query subtrees 8 and 9 (owned by the 2nd MPI process); Iteration 8: the reference subtree 12 is pruned for the 3rd MPI process. Iteration 9: the reference subtree 13 is considered subsequently after the reference subtree 12. At this point, the hashed reference subtree list includes (12, {8, 9}).

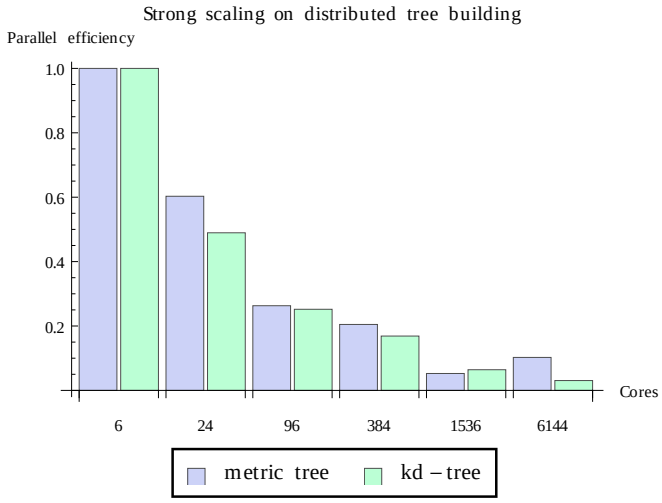

Figure 8: Strong scaling result for distributed *kd*-tree building on an uniform point distribution in the 10-dimensional unit hypercube $[0,1]^{10}$. The dataset has 20,029,440 points. The base timings for 6 cores are 105 seconds and 52.9 seconds for metric-tree and kd-tree respectively.

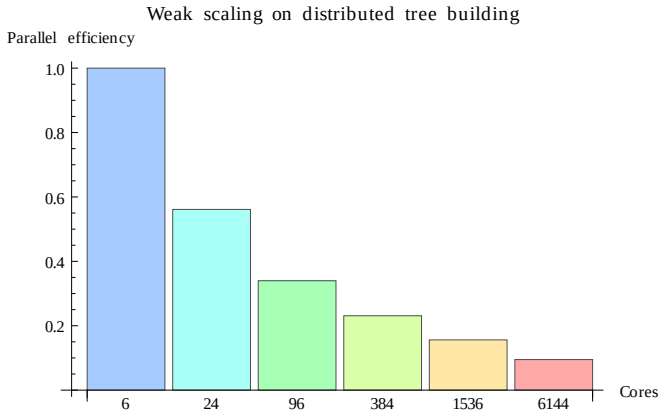

Figure 9: Weak scaling result for distributed *kd*-tree building on an uniform point distribution in 10 dimensions. We used 166,912 points / core. The base timing for 6 cores is 2.81 seconds.

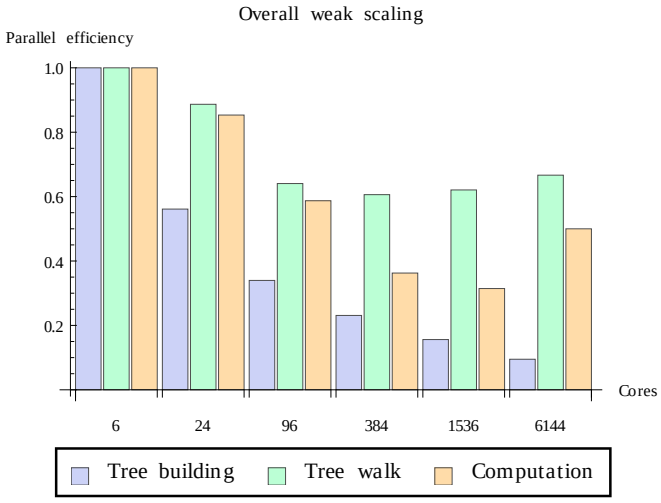

Figure 10: Weak scaling result for overall kernel summation computation on an uniform point distribution in 10 dimensions. We used 166,912 points / core and $\epsilon = 0.1$ and $h = \frac{1}{1024}$, halving h for every 4-fold increase in the number of cores. The base-timings for 6 cores are: 2.84 seconds for tree building, 1.8 seconds for the tree walk, and 128 seconds for the computation.

and performing reductions on the generated tasks (Figure 6). The kernel summation algorithm tested here employs only the deterministic approximations [21, 19]. We used $\epsilon = 0.1$, $\tau = 0$, and $\alpha = 1$ (see Definition 2.3).

Weak scaling. We measured the weak scalability of all phases of computation (the distributed tree building, the tree walk, and the computation). The data distribution we consider is a set of uniformly distributed 10-dimensional points. We vary the number of cores from 96 to 6144, adding 166,912 points per core. We used $\epsilon = 0.1$ and decreased the bandwidth parameter h as more cores are added to keep the number of distance computations constant per core; a similar experiment setup was used in [36], though we plan to perform more thorough evaluations. The timings for the computation maintains around 60 % parallel efficiency above 96 cores.

Strong scaling. Figure 11 presents strong scaling results on a 10 million/4-dimensional subset of the SDSS dataset. We used the Epanechnikov kernel with $h = 0.000030518$ (chosen by the plug-in rule) with $\epsilon = 0.1$.

6 Conclusion

In this paper, we proposed a hybrid MPI/OpenMP kernel summation framework for scaling many popular data analysis methods. Our approach has advantages including: 1) the platform-independent C++ code base that utilize standard protocols such as MPI and

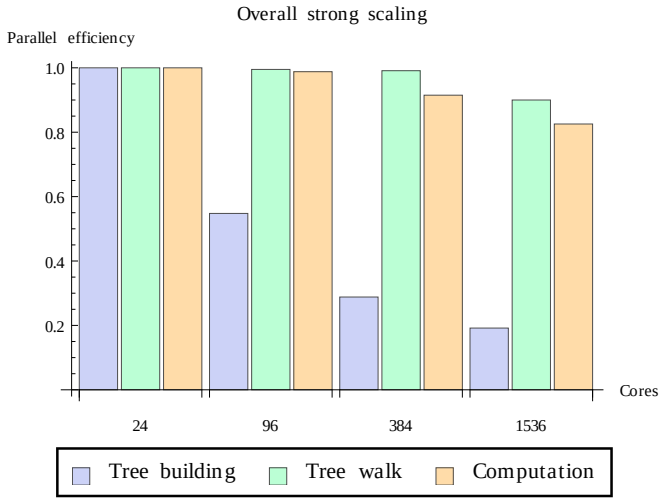

Figure 11: Strong scaling result for overall kernel summation computation on the 10 million subset of SDSS Data Release 6. The base timings for 24 cores are: 13.5 seconds, 340 seconds, 2370 seconds for tree building, tree walk, and computation respectively.

OpenMP; 2) the template code structure that uses any multidimensional binary trees and any approximation schemes that may be suitable for high-dimensional problems; 3) extendibility to a large class of problems that require fast evaluations of kernel sums. Our future work will address: 1) distributed computation on unreliable network connections; 2) extending to take advantage of heterogeneous architectures including GPGPUs for a hybrid MPI/OpenMP/CUDA framework; 3) extension of the parallel engine to handle problems with more than pair-wise interactions, such as the computation of n-point correlation functions [13, 25].

Acknowledgement

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] NERSC computational systems. `http://www.nersc.gov/users/computational-systems/`.

[2] I. Al-Furajh, S. Aluru, S. Goil, and S. Ranka. Parallel construction of multidimensional binary search trees. *Parallel and Distributed Systems, IEEE Transactions on*, 11(2):136–148, 2002.

[3] A. Appel. An efficient program for many-body simulation. *SIAM Journal on Scientific and Statistical Computing*, 6:85, 1985.

[4] J. Barnes and P. Hut. A Hierarchical $O(NlogN)$ Force-Calculation Algorithm. *Nature*, 324, 1986.

[5] J. L. Bentley. Multidimensional Binary Search Trees used for Associative Searching. *Communications of the ACM*, 18:509–517, 1975.

[6] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*, pages 97–104. ACM New York, NY, USA, 2006.

[7] G. Boyer, R. Riegel, N. Vasiloglou, D. Lee, L. Poorman, C. Mappus, N. Mehta, H. Ouyang, P. Ram, L. Tran, W. C. Wong, and A. Gray. MLPACK. `http://mloss.org/software/view/152`, 2009.

[8] P. Callahan and S. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM (JACM)*, 42(1):67–90, 1995.

[9] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. Bocchino, S. Adve, and J. Hart. Parallel sah kd tree construction. In *Proceedings of the Conference on High Performance Graphics*, pages 77–86. Eurographics Association, 2010.

[10] G. P. Contributors. GSL - GNU scientific library - GNU project - free software foundation (FSF). http://www.gnu.org/software/gsl/, 2010.

[11] F. Cruz, M. Knepley, and L. Barba. PetFMM–A dynamically load-balancing parallel fast multipole library. *Arxiv preprint arXiv:0905.2637*, 2009.

[12] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 537–546. ACM, 2008.

[13] A. Gray and A. W. Moore. N-Body Problems in Statistical Learning. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13 (December 2000)*. MIT Press, 2001.

[14] L. Greengard and V. Rokhlin. A Fast Algorithm for Particle Simulations. *Journal of Computational Physics*, 73, 1987.

[15] L. Greengard and J. Strain. The Fast Gauss Transform. *SIAM Journal of Scientific and Statistical Computing*, 12(1):79–94, 1991.

[16] M. Holmes, A. Gray, and C. Isbell Jr. Ultrafast Monte Carlo for kernel estimators and generalized statistical summations. *Advances in Neural Information Processing Systems (NIPS)*, 21, 2008.

[17] S. Koranne. Boost c++ libraries. *Handbook of Open Source Tools*, pages 127–143, 2011.

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

[19] D. Lee and A. Gray. Faster gaussian summation: Theory and experiment. In *Proceedings of the Twenty-*

second Conference on Uncertainty in Artificial Intelligence. 2006.

[20] D. Lee and A. Gray. Fast high-dimensional kernel summations using the monte carlo multipole method. In *In Advances in Neural Information Processing Systems 21*. 2009.

[21] D. Lee, A. Gray, and A. Moore. Dual-tree fast gauss transforms. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 747–754. MIT Press, Cambridge, MA, 2006.

[22] P. Liu and J. jan Wu. A framework for parallel tree-based scientific simulations. In *Proceedings of 26 th International Conference on Parallel Processing*, pages 137–144, 1997.

[23] T. Liu, C. Rosenberg, and H. Rowley. Clustering Billions of Images with Large Scale Nearest Neighbor Search. In *Proceedings of the Eighth IEEE Workshop on Applications of Computer Vision*, page 28. IEEE Computer Society, 2007.

[24] P. Loh, W. Hsu, C. Wentong, and N. Sriskanthan. How network topology affects dynamic loading balancing. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 4(3):25–35, 1996.

[25] A. Moore, A. Connolly, C. Genovese, A. Gray, L. Grone, N. Kanidoris, R. Nichol, J. Schneider, A. Szalay, I. Szapudi, and L. Wasserman. Fast algorithms and efficient statistics: N-point correlation functions. In *Proceedings of MPA/MPE/ESO Conference Mining the Sky, July 31–August 4, Garching, Germany*, 2000.

[26] E. Nadaraya. On estimating regression. *Theory of Prob. and Appl.*, 9:141–142, 1964.

[27] S. M. Omohundro. Five Balltree Construction Algorithms. Technical Report TR-89-063, International Computer Science Institute, 1989.

[28] M. Ouimet and Y. Bengio. Greedy spectral embedding. In *Proceedings of the 10th International Workshop on Artificial Intelligence and Statistics*, pages 253–260. Citeseer, 2005.

[29] A. Ozakin and A. Gray. Submanifold density estimation. *Advances in Neural Information Processing Systems*, 22, 2009.

[30] P. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann, 1997.

[31] E. Parzen. On estimation of a probability density function and mode. *The annals of mathematical statistics*, 33(3):1065–1076, 1962.

[32] A. Rahimi and B. Recht. Random features for large-scale kernel machines. *Advances in neural information processing systems*, 20:1177–1184, 2008.

[33] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[34] R. Riegel, A. Gray, and G. Richards. Massive-scale kernel discriminant analysis: Mining for quasars. In *SIAM International Conference on Data Mining*. Citeseer, 2008.

[35] J. K. Salmon. *Parallel Hierarchical N-body Methods*. PhD thesis, Ph. D. Thesis, California Institute of Technology, 1990.

[36] R. Sampath, H. Sundar, and S. Veerapaneni. Parallel Fast Gauss Transform. In *Supercomputing*, 2010.

[37] C. Sanderson. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, Australia, 2010.

[38] B. Scholkopf and A. Smola. Learning with Kernels: Support Vector Machines, Regularization. *Optimization, and Beyond. MIT Press*, 1:2, 2002.

[39] B. Scholkopf, A. Smola, and K. Muller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation*, 10(5):1299–1319, 1998.

[40] M. Seeger, C. Williams, N. Lawrence, and S. Dp. Fast forward selection to speed up sparse gaussian process regression. In *in Workshop on AI and Statistics 9*, 2003.

[41] J. Singh, J. Hennessy, and A. Gupta. Implications of hierarchical n-body methods for multiprocessor architectures. *ACM Transactions on Computer Systems (TOCS)*, 13(2):141–202, 1995.

[42] J. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, 1995.

[43] A. Smola and P. Bartlett. Sparse greedy Gaussian process regression. *Advances in Neural Information Processing Systems 13*, 2001.

[44] A. Smola and B. Scholkopf. Sparse greedy matrix approximation for machine learning. *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 911–918, 2000.

[45] C. Yang, R. Duraiswami, N. A. Gumerov, and L. Davis. Improved fast gauss transform and efficient kernel density estimation. *International Conference on Computer Vision*, 2003.

[46] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 311–321. Society for Industrial and Applied Mathematics, 1993.

[47] L. Ying, G. Biros, and D. Zorin. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics*, 196(2):591–626, 2004.

[48] D. York, J. Adelman, J. Anderson Jr, S. Anderson, J. Annis, N. Bahcall, J. Bakken, R. Barkhouser, S. Bastian, E. Berman, et al. The sloan digital sky survey: Technical summary. *The Astronomical Journal*, 120:1579, 2000.